**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ**
**ім. Ю.М. Потебні**
**ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ**

**КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**

# Кваліфікаційна робота

перший (бакалаврський)
(рівень вищої освіти)

на тему **Створення фреймворку для односпрямованої синхронізації даних між web-сервером і web-клієнтом в реальному часі**

Виконав: студент 4 курсу, групи 6.1210 - ПЗС
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Інженерія програмного забезпечення
(код і назва освітньої програми)
В. В. Одайський
(ініціали та прізвище )
Керівник доцент, к.т.н., доцент
О. М. Міхайлуца
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дісітел»
П.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2024

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**
**ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ**
**ім. Ю.М. Потебні**
**ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ**

Кафедра _____ програмного забезпечення автоматизованих систем
Рівень вищої освіти _____четвертий (бакалавр)_____
Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)
Освітня програма ___Інженерія програмного забезпечення
(код та назва)

**ЗАТВЕРДЖУЮ**

Завідувач кафедри_____Т. В. Критська
“__” _____ 2024_року

**З А В Д А Н Н Я**
**НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ**

Одайського Володимира Вячеславовича
(прізвище, ім’я, по батькові)

1.Тема роботи __Створення фреймворку для односпрямованої синхронізації даних між web-сервером і web-клієнтом в реальному часі

керівник роботи ___Міхайлуца Олена Миколаївна, доцент, к.т.н._____
( прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від 26.12.2023 р. № 2215-с

2. Строк подання студентом кваліфікаційної роботи_____09.06.2024_____
3. Вихідні дані бакалаврської роботи
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- дослідження проблеми синхронізації даних у реальному часі
- створення програмного продукту та його опис;
- дослідження поставленої проблеми та розробка висновків.

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

6. Консультанти розділів бакалаврської роботи

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата Завдання прийняв |
|---|---|---|
| | | |

7. Дата видачі завдання_____

## КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів бакалаврської роботи | Строк виконання етапів бакалаврської роботи | Примітка |
|---|---|---|---|
| 1 | Аналіз предметної області | 24.04.2024 | виконано |
| 2 | Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником | 25.04.2024 | виконано |
| 3 | Аналіз існуючих рішень | 30.04.2024 | виконано |
| 4 | Аналіз методів розробки баз даних реального часу | 05.05.2024 | виконано |
| 5 | Аналіз сучасних технологій для розробки HTTP сервісів | 10.05.2024 | виконано |
| 6 | Узгодження подальших дій з науковим керівником | 14.05.2024 | виконано |
| 7 | Програмна реалізація сервеної частини | 15.05.2024 | виконано |
| 8 | Програмна реалізація клієнтської частини | 28.05.2024 | виконано |
| 9 | Представлення отриманих результатів науковому керівнику та узгодження плану подальшого дослідження | 29.05.2024 | виконано |
| 10 | Реалізація користувацького інтерфейсу | 04.06.2024 | виконано |
| 11 | Перевірка роботоздатності проекту | 06.06.2024 | виконано |
| 12 | Оформлення звіту | 07.06.2024 | виконано |
| 13 | Оформлення презентації. | 14.06.2024 | виконано |

Студент _____ Одайський В.В.
( підпис ) (прізвище та ініціали)

Керівник роботи _____ Міхайлуца О.М.
( підпис ) (прізвище та ініціали)

**Нормоконтроль пройдено**

Нормоконтролер _____ Скрипник І.А.
( підпис ) (прізвище та ініціал

# АНОТАЦІЯ

Сторінок: 84

Рисунків: 11

Джерел: 16

Одайсьий В. В. Створення фреймворку для односпрямованої синхронізації даних між web-сервером і web-клієнтом в реальному часі: кваліфікаційна робота бакалавра спеціальності 121 “Інженерія програмного забезпечення” / наук. керівник О.М. Міхайлуца. Запоріжжя : ЗНУ, 2024.84 с.

Мета і завдання роботи полягає у дослідженні теми синхронізації даних у реальному часі між клієнтами та сервером. На основі цього дослідження необхідно розробити фреймворк, що складатиметься з сервера та клієнта, і надаватиме можливість зручної автоматичної синхронізації даних, відображених на клієнті у відповідності до змін, що відбуваються у базі даних.

У процесі вивчення теми були порушені такі проблеми як: недостатня гнучкість існуючих рішень, неможливість користування перевагами реляційних баз даних у контексті більшості баз даних реального часу, недостатня зрчучність у використанні для програміста. Як результат був створений фреймворк на мові програмування Python, та клієнтська бібліотека на мові програмування TypeScript, що разом дозволяють вирішити поставлену проблему шляхом надання інструментів для автоманичної синхронізації між клієнтом та сервером, та підтримки реактивності, наданої фреймворком Vue.js. Для демонстрації функцій створеного рішення було розроблено простий веб-додаток, що функціонує у режимі реального часу.

Ключові слова: *веб-сокет, синхронізація даних, реальний час, фреймворк, логічна реплікація, база даних.*

# SUMMARY

Pages: 84

Drawings: 11

Sources: 16

Odaiskyi V. V. Creation of Framework for One-Way Data Synchronization between a Web-Server and Web-Client in Real Time: bachelor's thesis in specialty 121 “Software engineering” / Supervisor O.M. Mikhailutsa. Zaporizhzhia : ZNU, 2024. 84 p.

Goal of the work is to study the topic of real-time data synchronization between clients and the server. Based on this study, a framework should be developed consisting of a server and a client, providing the possibility of convenient automatic synchronization of data displayed on the client in accordance to changes occurring in the database.

During the study, such problems were raised as: insufficient flexibility of existing solutions, impossibility of using advantages of relational databases in the context of most real-time databases, insufficient usability for developers. As a result, a framework was developed using Python programming language as well as a client library in TypeScript programming language, which together allow solving the problem by providing tools for automatic synchronization between client and server, while also supporting reactivity provided by Vue.js framework. A simple real-time web application was built to demonstrate functions of developed solution.

Keywords: *web- socket, data synchronization, real time, framework, logical replication, database.*

**ЗМІСТ**

# ВСТУП

**Актуальність теми**

У світі, де веб-технології та мобільні пристрої стають неодмінною частиною нашого повсякдення, проблема ефективної синхронізації даних у клієнт-серверній архітектурі стає надзвичайно актуальною. Користувачі більше не задовольняються обмеженим доступом до своїх даних лише з одного пристрою чи місця — вони очікують безперервного доступу незалежно від контексту.

Зі зростанням різноманітності пристроїв та платформ користувачів, виникає потреба в універсальному підході до синхронізації даних. Це вимагає від розробників зусиль у створенні ефективних механізмів синхронізації, які забезпечать доступність даних у будь-який час і з будь-якого пристрою.

У такому контексті, швидкість розробки програмного забезпечення стає ключовим фактором успіху. Стандартизація підходів до синхронізації даних може значно полегшити цей процес і сприяти прискоренню розробки, забезпечуючи при цьому високу якість продукту.

Сучасні програмні застосунки вимагають реактивного підходу до обробки даних, щоб реагувати на їх зміни у реальному часі. Автоматизовані механізми синхронізації даних стають невід'ємною частиною цього процесу, забезпечуючи безперервну актуальність і консистентність інформації.

Такий підхід до синхронізації даних має широкі застосування в різних сферах, від веб-застосунків до мобільних додатків. Створення стандартизованих рішень у цій області може відкрити нові можливості для універсального використання та сприяти подальшому розвитку цифрової екосистеми.

Отже, створення стандартизованих рішень для синхронізації даних у клієнт-серверній архітектурі є не лише актуальним, але й стратегічно важливим завданням. Це допоможе покращити ефективність розробки

програмного забезпечення, забезпечити задоволення потреб користувачів та сприяти подальшому розвитку сучасного цифрового середовища.

**Мета і завдання дослідження**

Вивчити та проаналізувати сучасні підходи та технології синхронізації даних у клієнт-серверній архітектурі з метою розробки стандартизованого рішення, що сприятиме покращенню ефективності розробки програмного забезпечення та задоволенню потреб користувачів у сучасному цифровому середовищі.

**Об’єкт дослідження**

Синхронізація даних у реальному часі між web-клієнтами та сервером.

**Предмет дослідження**

Розробка фреймворку для односпрямованої синхронізації даних між web-сервером і web-клієнтом в реальному часі.

**Практичне значення одержаних результатів**

Практичне значення одержаних результатів дослідження полягає у можливості створення стандартизованого рішення для синхронізації даних у клієнт-серверній архітектурі. Це рішення сприятиме покращенню ефективності розробки програмного забезпечення, зниженню часу на пошук та впровадження відповідних технологій синхронізації даних. Результати дослідження можуть бути корисними для розробників програмного забезпечення, що працюють у сфері веб-розробки та мобільних додатків. Вони зможуть скористатися створеним стандартизованим рішенням для швидкої та ефективної синхронізації даних між різними платформами та пристроями.

# РОЗДІЛ 1 ДОСЛІДЖЕННЯ ПРОБЛЕМИ СИНХРОНІЗАЦІЇ ДАНИХ У РЕАЛЬНОМУ ЧАСІ

## 1.1 Огляд проблеми

Синхронізація даних в реальному часі у клієнт-серверній архітектурі, де база даних є джерелом інформації не є простою задачею, але вона є необхідною для все більшого числа додатків.

Одна з основних проблем цього процесу полягає в тому, що традиційні реляційні бази даних не були спроектовіні для такого використання. Вони розроблені під так звану “pull” модель, при якій клієнт, щоб отримати дані, має спочатку ініціювати запит. Це призводить до необхідності шукати альтернативні шляхи реалізації, або повністю відмовлятись від реляційної моделі.

Ще одна проблема синхронізації даних полягає у необхідності точно визначати, на кого з підключених клієнтів нещодавні зміни мали вплив. Мінімізація трафіку безперечно є важливою умовою ефективної синхронізації, задля забезпечення маштабованості системи. Клієнт має отримувати тільки ті оновлення, що є актуальними для нього. Визначення конкретних залежностей конкретного клієнта є складною задачею, враховуючи те, що запити, ініційовані клієнтом можуть бути досить заплутаними, з багатьма умовами і залежати від декількох таблиць.

Іншою проблемою є необхідність у способах маштабування на рівні архітектури проекту. Зазвичай, великі системи маштабуються горизонтально за допомогою збільшення кількості реплік, на яких запущене і працює програмне забезпечння.

Загалом, проблема синхронізації даних в реальному часі включає в себе потребу у високому рівні оптимізації алгоритмів для достатньої швидкодії системи, складність коректного визначення залежностей клієнтських запитів

для забезпечення мінімальної кількості трафіку гарантуючи високу ефективність, і можливість маштабування системи під час розгортання.

### 1.2 Огляд літературних джерел

Інтернет став необхідною складовою людського життя в епоху інформації. У відповідь на цей попит кількість веб-додатків постійно зростає. Веб-додатки складаються з двох основних частин: клієнтської і серверної. Зазвичай роль клієнта виконує веб-браузер, який ініціює запити до сервера, а сервер відповідає на ці запити, передаючи необхідні дані. Під час веб-зв'язку клієнти повинні встановлювати підключення за допомогою HTTP (Hyper Text Transfer Protocol), щоб з'єднатися з сервером. Однак HTTP є протоколом односпрямованого зв'язку, де спілкування відбувається лише за бажанням клієнта. Велика кількість непотрібних запитів виникає через постійну перевірку актуальності даних для клієнта. Існує кілька методів для усунення цих недоліків, серед яких найважливішими є довге опитування на основі AJAX і технологія Комети. У статті “The Comparison of New Web Communication Method WebSocket with Traditional Methods” [1] описано та досліджено можливі способи дуплексної передачі даних у веб-браузері.

Розширення Інтернету та прогрес у апаратному забезпеченні протягом останнього десятиліття призвели до виникнення парадигми реактивного програмування в інформатиці. Сучасний веб-сайт обробляє стільки трафіку, скільки весь Інтернет всього кілька десятів років тому [2]. Це викликало зміни в промисловості розробки програмного забезпечення, оскільки користувачі все більше покладаються на програми та очікують миттєвого відгуку та надійності.

“Реактивний маніфест” визначає підхід до ефективного управління взаємодією користувачів, враховуючи нові вимоги щодо швидкої реакції та постійного доступу та оновлення. Однак сучасні бази даних, які мають справлятися з цими вимогами, повинні бути адаптовані для забезпечення

очікуваної ємності зберігання, що може сягати петабайт. Реактивні програми вимагають миттєвого доступу до даних, які вони використовують, і можливості їх публікації та оновлення негайно після створення.

У новому сценарії бази даних повинні вирішувати два основні складні аспекти: масштабованість та забезпечення доступу до даних, відомого як push-based. У відповідь на нові потреби з'явилася концепція баз даних реального часу, що мають зменшити прогалину між класичним підходом до баз даних і потребою в більш ефективній логіці зберігання даних, зокрема для роботи з колекціями, що постійно розвиваються.

Традиційні системи управління базами даних (СУБД) працюють на основі витягування даних, отримуючи інформацію при кожному запиті клієнта. Ця архітектура відповідає різноманітним доменам, але не ефективна для додатків, які вимагають одночасної обробки для підтримки актуальних даних. Щоб підтримувати ці вимоги, виникає необхідність використання баз даних реального часу.

Основна складність для традиційних баз даних полягає у здатності ефективно відслідковувати результати будь-якого запиту в реальному часі для отримання нової інформації та оновлення даних. Це важке завдання через те, що необхідні результати не представляють собою простого списку ідентифікаторів, які можна легко відстежити. Крім того, кожен раз при виконанні операції в режимі реального часу база даних повинна перевірити кожен окремий задіяний об'єкт. Для кожного режиму реального часу постійно стоїть два ключові питання щодо кожного запиту: перше — чи збігається записаний об'єкт із запитом, і друге — чи збігався цей об'єкт раніше. Якщо відповідь на обидва питання негативна, то оновлення, запропоноване запитом, ігнорується. В іншому випадку результат оновлюється, і кінцевий користувач отримує оновлення. Для реалізації цього в реальному світі Волфрам Вінгерат пояснює: “Припустимо, у вас є програма з 1000 одночасних користувачів і середньою пропускною здатністю 1000 операцій в секунду. Враховуючи, що кожен користувач має лише один активний запит в реальному часі, база даних

у реальному часі вже повинна виконати 1 мільйон відповідей кожну секунду. І це лише для звичайної фільтрації та не враховує більш складних запитів з ще більшими витратами. Хоча можливі певні оптимізації, які обмінюють пропускну здатність на збільшену затримку, для миттєвих сповіщенн це єдиний варінт.” [2]

## 1.3 Приклади баз даних реального часу

### 1.3.1 Meteor

Meteor — це JavaScript платформа, призначена для розробки інтерактивних програм і веб-сайтів [3]. Він побудований на основі документно-орієнтованої бази даних MongoDB з відкритим вихідним кодом, що дозволяє оновлювати дані в реальному часі, використовувати запити у вигляді виразів, та забезпечує гнучкість для зберігання різноманітних типів даних та адаптації до різних додатків.

Живі запити (live queries) в Meteor виконуються на серверах бази даних додатків, які реєструють оновлення даних і надсилають повідомлення зацікавленим кінцевим користувачам. Архітектура Meteor може включати один або кілька серверів додатків в залежності від рівня масштабування. Коли існує кілька серверів додатків, вони підключені до сервера MongoDB, але не взаємодіють напряму між собою. Це означає, що сервер додатку взаємодіє з іншим сервером, читаючи записи бази даних. Така синхронізація відбувається кожні 10 секунд, коли кожен сервер повторює кожен синхронізований запит, щоб отримати реальний стан системи майже в реальному часі. Цей режим виконання відомий як poll-and-diff і є ефективним способом управління оновленням даних, але водночас створює проблему застарілих даних через 10-секундні інтервали, що може бути неприйнятним для деяких програм, і обмежує масштабованість, особливо в умовах великої кількості користувачів та запитів.

Ця проблема перетворилася у новий спосіб взаємодії з живими запитами у Meteor. Meteor використовує можливості реплікації MongoDB, щоб працювати в режимі tail oplog запитів. В цьому сценарії кожен сервер додатків Meteor функціонує як допоміжний кластер для кожного фрагмента основного кластера MongoDB. Кожне оновлення даних спочатку досягає серверів програм, які негайно надсилають оновлення до своїх відповідних первинних кластерів, а ці кластери передають цю операцію у формі oplog всім серверам додатків, що потребують синхронізації. Таким чином, жоден з серверів додатків не потребує повторного запиту інформації, оскільки вони вже проінформовані про зміни.

Проте іноді може статися так, що oplog не містить повної інформації. Тому кожен сервер додатків Meteor повинен виконати процес моніторингу oplog, який періодично запитує оновлення даних, які можуть бути не готові для сповіщення. Однак проблема з цим недоліком полягає в тому, що при необхідності повторного запиту численних письмових об'єктів це може стати слабкою ланкою на серверах програм.

### 1.3.2 RethinkDB

RethinkDB є базою даних з відкритим вихідним кодом на основі JSON, розробленою на C++ і опублікованою за ліцензією Apache 2.0 [4]. Її концепція схожа на ту, що використовується в Meteor, але відзначається покращеними запитами на основі push-моделі і об'єднаннями (join). Крім того, так як у Meteor, RethinkDB має свій власний JavaScript інтерфейс керування, Horizon.

RethinkDB використовує архітектуру changefeed, яка дозволяє реалізувати свого роду oplog. Однак важливо відзначити, що кластер зберігання у RethinkDB не є аналогічним MongoDB, але є власноруч розробленим рішенням для цієї бази даних, де сервери додатків частіше відомі як проксі. Загалом, з точки зору робочої моделі, виникають ті ж самі проблеми масштабованості для великої кількості користувачів, що були помічені і в Meteor.

### 1.3.3 Parse

Це один з найпопулярніших інструментів Backend-as-a-Service для розробки мобільних додатків [5]. Він має графічний інтерфейс (див. Рис. 1), і доволі зручний для використання. Крім того, він є відкритим і базується на MongoDB, що призвело до одного з найширших розгортань цієї бази даних у всьому світі.

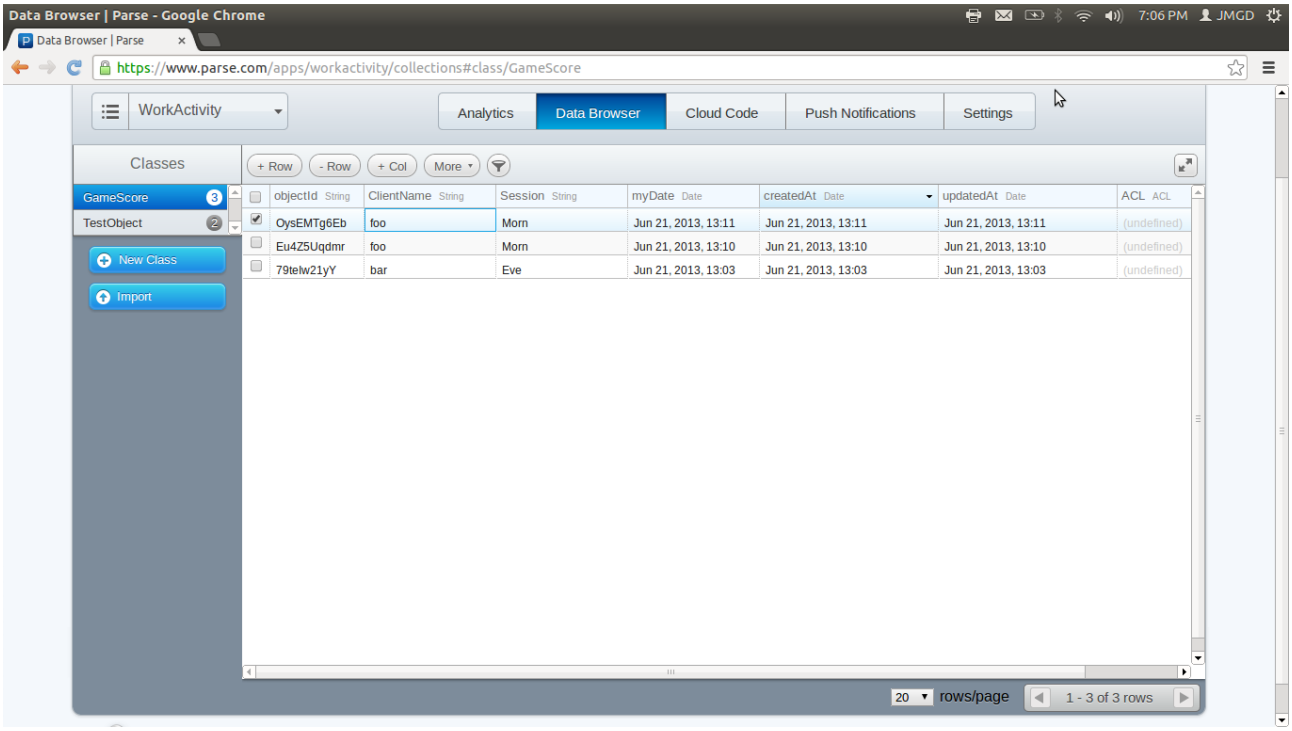


Рисунок *1 — Панель керування Parse*

Parse підтримує функції автентифікації користувачів, надсилання push-повідомлень та має надійну документацію з цих можливостей.

Зараз, навіть при тому, що Parse припинив своє існування як сервіс, він залишається широко використовуваним в ряді мобільних додатків. Це пояснюється тим, що різні постачальники, такі як Back4app, Stamplay і AWS, продовжують надавати хостинг для програм, пропонуючи альтернативи для міграції.

### 1.3.4 Firebase

База даних Firebase від Google (див. Рис. 2) — це хмарна база даних, де дані зберігаються у форматі об'єктів JSON та синхронізуються в режимі реального часу з кожним підключеним клієнтом (веб, iOS, Android і т. д.) [6]. Доступ до цієї бази даних можливий напряму з клієнтів, і не вимагає використання проміжного серверу.

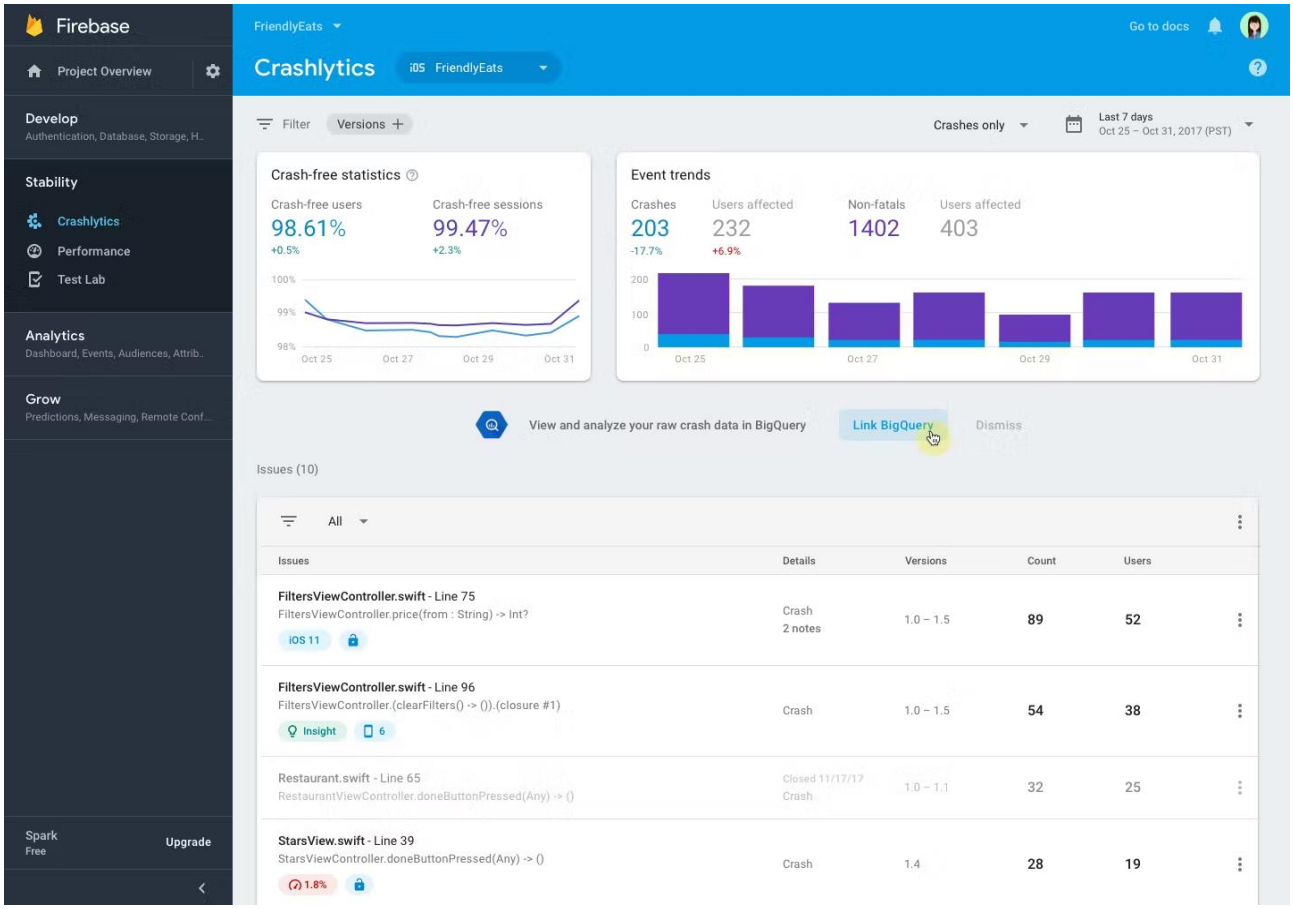


Рисунок *2 — Панень керування Firebase*

Якщо клієнтська програма працює офлайн, дані можна також зберігати локально. Коли пристрій відновлює підключення, Firebase синхронізує локальні зміни даних із віддаленими оновленнями, які відбувалися, коли клієнт був офлайн, автоматично вирішуючи конфлікти

Створення структурованої бази даних у Firebase, яка є базою даних NoSQL, вимагає виваженого планування того, як дані будуть зберігатися та

отримуватися пізніше, з метою використання переваг оптимізації, які Firebase пропонує для неструктурованих даних. Firebase надає API бази даних у реальному часі, яке дозволяє виконувати лише ті операції, які можна виконати швидко.

Усі рішення, запропоновані вище, мають однакові недоліки: По-перше, всі вони залежать від неструктурованих, документо-орієнтованих NoSQL баз даних (див. Рис. 3), що зазвичай накладають значні обмеження на можливості створення складних запитів і підтримання цілісності [3].

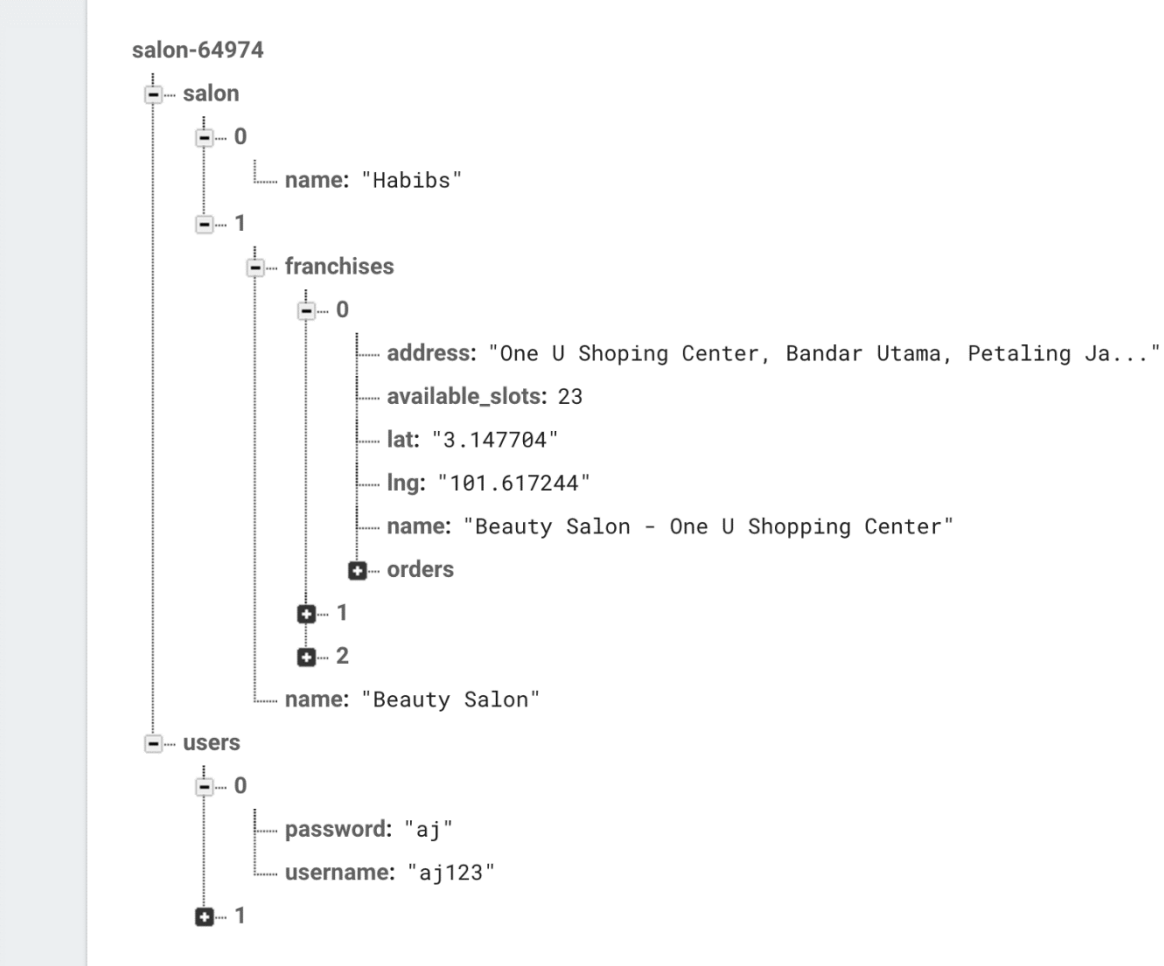


Рисунок *3 — Приклад представлення даних у NoSQL базі даних*

По-друге, всі вони є або розподіленими сервісами з закритим кодом, або повноцінними фреймворками, що повністю диктують спосіб розробки, не лишаючи вибору для розробника, на відміну від запропонованого рішення у

ційній роботі, що має за мету стати модульною надбудовою над СКБД Postgres, і може бути легко інтегроване у існуючі проекти.

# РОЗДІЛ 2 ДОСЛІДЖЕННЯ ПРОГРАМНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ

## 2.1 Вибір мови програмування. Python

Python — це високорівнева, інтерпретована мова програмування, що була створена Гвідо ван Россумом. Вона відзначається простотою синтаксису та читабельністю коду, що дозволяє розробникам висловлювати ідеї більш зрозуміло і ефективно [7].

Мова підтримує об'єктно-орієнтоване, функціональне та процедурне програмування. Вона має широкий спектр вбудованих бібліотек для різних завдань, що робить її дуже потужним та гнучким інструментом для розробки різних застосунків.

Інтерпретований характер Python дозволяє розробникам виконувати код без необхідності компіляції, що полегшує тестування та експериментування. Велика активна спільнота і широка екосистема пакетів сприяють постійному розвитку та вдосконаленню мови.

Python часто використовується для розробки веб-додатків, штучного інтелекту, обробки даних, наукових обчислень та багатьох інших сфер, завдяки своїй універсальності та простоті використання.

У проекті мова Python використовується для розробки серверної частини фреймворку, що відповідає за надання доступу до інформації, взаємодію з базою даних, авторизацію та розсилку повідомлень клієнтам.

## 2.2 Формати запиту і передачі даних. GraphQL

GraphQL — це мова запитів для взаємодії з API, яка була розроблена компанією Facebook і вперше представлена у 2012 році. Однак у 2015 році вона була відкрита для громадськості та здобула широку популярність в розробці програмного забезпечення [8]. GraphQL надає зручний спосіб для клієнтів

отримувати тільки ту інформацію, яка їм потрібна, що відрізняє його від традиційних REST API.

Принципова ідея GraphQL полягає в тому, щоб дозволити клієнтам визначати структуру даних, яку вони хочуть отримати, і отримувати її у одному запиті. У порівнянні з REST, де кожен ендпоінт має фіксовану структуру відповіді, GraphQL дає можливість точно визначати потрібні дані, що дозволяє уникнути надлишкової передачі інформації та покращити ефективність комунікації між клієнтом і сервером.

Крім того, GraphQL підтримує автодокументацію, що полегшує розробникам розуміння доступних API і робить процес розробки більш ефективним.

У проекті GraphQL використовується з метою забезпечення доступу до полів таблиць у індивідуальному порядку, означаючи що клієнт не зобов’язаний отримувати всю доступну інформацію роблячи запит будь-якої сутності. Всі схеми даних генеруються автоматично фреймворком на основі визначених моделей. Обчислення значень полів відбувається так само автоматично, виходячи з полів таблиці що запитується, або виходячи з визначень обчислюваних полів у випадку з віртуальними таблицями.

### 2.3 In-memory бази даних. Кешування. Redis

Redis — це in-memory база даних, яка використовується для зберігання та обробки даних в оперативній пам'яті комп'ютера. Ця технологія виникла як ключ-значення (key-value) система зберігання даних, і її головною особливістю є використання оперативної пам'яті для швидкого доступу до даних. Redis дозволяє ефективно зберігати, видаляти та оновлювати дані за допомогою ключів, які можуть мати різні типи значень, такі як рядки, хеші, списки, множини та інші.

Центральною концепцією Redis є принцип “дані в пам'яті”. Це означає, що дані зберігаються в оперативній пам'яті сервера, що дозволяє значно

зменшити час доступу до даних порівняно з традиційними дисковими базами даних. Швидкий доступ до даних робить Redis ідеальним вибором для сценаріїв, де важлива висока продуктивність та миттєвий відгук.

Однією з ключових переваг Redis є його простота та висока продуктивність. Він надає можливість використовувати різні структури даних та операції для ефективного вирішення різних задач. Redis також має можливості реплікації та кластеризації, що дозволяє створювати резервні копії даних та забезпечувати високу доступність. Усе це робить Redis популярним рішенням для сучасних додатків, особливо тих, де важливий швидкий та надійний доступ до даних.

Redis було обрано як одну з найрозповсюдженіших систем, що не має зайвих функцій і ефективно кешує дані у пам’яті для швидкого доступу.

У проекті redis використовується з метою реалізації кешування (деякі запити робляться частіше за інші, вимагають більше ресурсів для обчислення). Завдяки механізму залежностей, фреймворк знатиме, коли той чи інший запит має бути оновлений, і в залежності від типу запиту таке оновлення може відбутися навіть без звернення до бази даних.

### 2.4 Веб-фреймворки. FastAPI

FastAPI — це сучасний та високопродуктивний фреймворк для створення веб-додатків на мові програмування Python. Розроблений як альтернатива Flask та Django, FastAPI надає зручний інтерфейс для створення RESTful API, а також підтримує асинхронне програмування, що дозволяє оптимізувати обробку багатьох запитів одночасно.

Однією з ключових особливостей FastAPI є використання типізації даних за допомогою стандарту Python 3.7+ анотацій типів. Це дозволяє автоматично генерувати документацію API, спрощує роботу з валідацією вхідних та вихідних даних, а також робить код більш надійним і підтримуваним. FastAPI використовує вбудований генератор OpenAPI та

інтегрується з іншими інструментами, такими як Swagger UI, що полегшує взаємодію з API.

Однією з сильних сторін FastAPI є висока швидкодія. Фреймворк побудований на основі Starlette та Pydantic, що забезпечує високу продуктивність завдяки використанню асинхронності та компіляції в C. Це дозволяє FastAPI конкурувати з іншими швидкими фреймворками та відмінно підходить для високонавантажених додатків.

FastAPI має велику спільноту та активно розвивається. Регулярні оновлення та підтримка з боку розробників дозволяють тримати фреймворк у найсучаснішому стані та надавати користувачам доступ до новітніх можливостей мови програмування Python.

FastAPI — це потужний та ефективний фреймворк, який поєднує в собі зручний синтаксис та високу продуктивність. Він ідеально підходить для розробки сучасних веб-додатків та API, забезпечуючи високу якість коду, зручність в роботі та високу швидкодію.

Одним з популярних аналогів FastAPI є Flask. Порівняно з ним, FastAPI є зручнішим для виконання поставленої задачі через наявність зручних інструментів валідації даних, хорошу підтримку веб-сокетів, свою сучасність і асинхронність.

Іншим популярним фреймворком для розробки серверної частини є Django. Однак цей фреймворк є full-stack фреймворком, що означає велику кількість зайвого функціоналу, виходячи з того, що для поставленої задачі від фреймворка потребується виключно можливість розробки REST API, тому мініатюрність FastAPI є великим плюсом.

У проекті FastAPI використовується як web-додаток, що надаватиме доступ до даних через мову запитів GraphQL, а також керуватиме підключеними веб-сокетами.

## 2.5 Веб-сервери. ASGI

ASGI, або Asynchronous Server Gateway Interface, є протоколом взаємодії python програми та веб-серверів, що на відміну від WSGI (Web Server Gateway Interface) використовують асинхронне програмування в мові Python [9]. Основна ідея ASGI полягає в створенні стандартного інтерфейсу взаємодії, що може підтримуватись багатьма веб-серверами, надаючи засоби для обробки багатьох одночасних з'єднань без блокування, що поліпшує масштабованість та продуктивність веб-додатків.

ASGI використовує концепцію асинхронності для обробки багатьох запитів паралельно, дозволяючи веб-серверам ефективно взаємодіяти з асинхронними фреймворками, такими як Django Channels або FastAPI. Замість того, щоб блокувати виконання програми при очікуванні відповіді від бази даних чи іншого сервісу, асинхронні фреймворки можуть переключатися між задачами та обробляти інші запити, що покращує швидкість відгуку додатка.

Переваги ASGI включають здатність обробляти багато з'єднань одночасно без необхідності створювати велику кількість потоків чи процесів. ASGI також полегшує інтеграцію з WebSocket, що є актуальним для реалізації поставлених задач.

Очевидною альтернативою до ASGI є WSGI, однак проект активно використовує асинхронне програмування. тож переваги ASGI неможливо недооцінити. Іншою альтернативою могло стати використання повноцінного веб-сервера для запуску програми, однак такий підхід унеможливлює включення серверної частини фреймворку у інший, існуючий додаток.

У проекті ASGI використовується для експорту веб-додатку що має змогу обробляти GraphQL запити і веб-сокет підключення у форматі, що може бути використаний багатьма веб-серверами.

## 2.6 UI-фреймворки. Vue3

Vue.js 3 (або просто Vue 3) — це прогресивний JavaScript фреймворк для створення веб-інтерфейсів, який зосереджений на реактивному програмуванні та легкості використання [10]. Розроблений Еваном Ю (Evan You), Vue 3 є наступником популярної версії Vue.js 2 і містить ряд інновацій та поліпшень.

Однією з ключових особливостей Vue 3 є нова система реактивності, яка є основою для створення динамічних інтерфейсів. Вона дозволяє встановлювати зв'язки між даними та їх відображенням в реальному часі, автоматично оновлюючи відображення при зміні даних. Це робить код більш чистим та зручним для розробників, оскільки вони можуть уникнути безлічі ручних маніпуляцій з DOM.

Vue 3 також має новий синтаксис для опису компонентів, що дозволяє використовувати більше декларативного підходу до роботи з інтерфейсом. Крім того, нововведений Compositions API розширює можливості розробників у плані організації та повторного використання коду компонентів.

Завдяки оптимізаціям та новим можливостям, Vue 3 демонструє покращену продуктивність та ефективність у порівнянні з попередніми версіями. Взагалі, Vue 3 визначається своєю простотою, ефективністю та інноваційністю, які роблять його привабливим вибором для веб-розробників. На сьогоднішній день існує багато альтернатив, коли мова заходить про JavaScript фреймворки для створення веб-інтерфейсів. Популярними є Svelte, React, Backbone, Amber та інші. Однак саме Vue було обрано для реалізації прикладу інтеграції з розробленим фреймворком через його простоту у використанні та зручну систему реактивності, з якою можна досить легко інтегрувати код розробленого програмного рішення.

У проекті Vue використано з метою створення невеликого додатку, що демонструє інтеграцію фреймворку в код клієнта, і отримані переваги автоматичного реактивного оновлення даних, спираючись на отриманий код

## 2.7 Черги повідомлень. RabbitMQ

RabbitMQ є однією з найпопулярніших систем обробки повідомлень, яка використовується для побудови розподілених застосунків, що працюють в архітектурі клієнт-сервер [11]. Це програмне забезпечення надає механізм для ефективної асинхронної комунікації між компонентами системи, дозволяючи реалізовувати шаблон “Шина подій” у системі, що складається з декількох сервісів.

RabbitMQ використовує протокол AMQP (Advanced Message Queuing Protocol), який дозволяє створювати надійний і масштабований обмін повідомленнями між різними частинами системи. Принцип роботи полягає в тому, що відправник (продюсер) розсилає повідомлення до “обмінника” (exchange), а звідти вони розподіляються до “черг” (queues), з яких вже читають або обробляють їх призначені “споживачі” (consumers).

Однією з ключових переваг RabbitMQ є його здатність забезпечити гнучку та надійну асинхронну комунікацію між компонентами системи. Він дозволяє створювати розподілені системи, де різні частини можуть взаємодіяти безпосередньо, не чекаючи одна на одну. Крім того, RabbitMQ забезпечує масштабованість і високу доступність, що робить його ефективним інструментом для побудови великих і надійних систем. Він також підтримує різні типи обмінників і стратегії розподілу повідомлень, що дозволяє налаштовувати систему під конкретні вимоги додатку.

Альтернативами до RabbitMQ можуть виступати Google pub-sub або Apache Kafka. Всі три рішення дозволяють досить гнучке масштабування. У цьому питання google pub-sub спирається на хмарну інфраструктуру Google, у той час як RabbitMQ підтримує можливість встановлення та налаштування на власному сервері, що дає більше контролю над ресурсами. У плані гнучкості налаштування, однак, pub-sub програє, не в змозі конкурувати з широким спектром можливостей налаштувань та взаємодії з різними протоколами, що надає RabbitMQ.

Apache Kafka, в свою чергу, є досить схожим на RabbitMQ. Kafka так само використовує publish-subscribe модель, що є log-based (на основі журналу), де повідомлення надсилаються до тем (topic) та зберігаються в журналах. Kafka підходить для потужних аналітичних сценаріїв, обробки великих обсягів даних, та стійкого збереження журналів, в той час як RabbitMQ зручний для розподіленої обробки завдань і обміну повідомленнями в реальному часі у меншому масштабі. RabbitMQ на відміну від Kafka зазвичай не зберігає повідомлення після їхнього оброблення.

У результаті, RabbitMQ є кращою альтернативною, завдяки своїй ефективності при обробці великої кількості малих повідомлень, та можливості розгортання на основі власних ресурсів, на відміну від хмарної pub-sub. Менша надійність порівняно з kafka уе становить великої загрози, з огляду на те, що повідомлення має бути отримане незалежно від інших повідомлень, і будь-який стан системи буде скинуто після перезавантаження клієнта, тим самим позбавляючи нас необхідності персистентного сховища повідомлень. Навіть у разі, якщо окреме повідомлення не було отримане, це не призведе до жодних серйозних наслідків.

У проекті RabbitMQ використовуються для обміну повідомленнями між сервісами, що складають серверну частину фреймворку.

### 2.8 Дуплексні протоколи передачі даних. Websocket

WebSocket є протоколом зв'язку між клієнтом і сервером через Інтернет, призначеним для реального часу обміну даними. Основною його відмінністю від традиційних протоколів, таких як HTTP, є постійне підключення між сторонами, що дозволяє обмінюватися даними в обох напрямках без необхідності постійно встановлювати нові з'єднання.

Процес взаємодії між клієнтом і сервером починається з встановлення WebSocket-з'єднання через процес “рукостискання”, під час якого дві сторони, клієнт і сервер, домовляються про зміну протоколу. Після цього сторони

можуть надсилати повідомлення одна одній в реальному часі, використовуючи для цього одне активне підключення. Однією з ключових переваг WebSocket є зменшення затримок і покращення ефективності передачі даних порівняно з традиційним HTTP-з'єднанням. Крім того, WebSocket дозволяє економити ресурси сервера, позбуваючись необхідності постійно створювати нові з'єднання для кожного запиту. Цей протокол добре підходить для сценаріїв реального часу, таких як онлайн-гри, чати, фінансові системи тощо, де швидкість і ефективність відіграють важливу роль.

У проекті WebSocket використовується з метою обміну даних з клієнтом у реальному часі, оскільки з усіх наявних можливих рішень саме WebSocket є найефективнішим.

### 2.9 Бази даних. Postgres

PostgreSQL (або просто Postgres) є потужною системою управління базами даних (СУБД), яка визначається високою надійністю, гнучкістю та здатністю до розширення [12]. PostgreSQL є програмним забезпеченням з відкритим кодом, що означає, що воно може бути вільно використовуване, змінене та розповсюджуване.

PostgreSQL використовує мову SQL для взаємодії з даними та підтримує різні типи даних. Ця СУБД підтримує транзакції з високим рівнем ізоляції, що забезпечує консистентність та надійність даних.

Однією з ключових переваг PostgreSQL є його розширюваність. Він дозволяє користувачам визначати власні функції, типи та індекси, що робить його ідеальним вибором для проектів різного розміру та складності. Postgres також підтримує реплікацію для забезпечення високої доступності та шкальованості, що дозволяє розподілено обробляти завдання та підвищує продуктивність в умовах великої робочої навантаженості.

Крім того, PostgreSQL активно підтримується та оновлюється спільнотою розробників, що робить його стабільним та безпечним рішенням для потреб бізнесу та розробки програмного забезпечення.

Існує велика кількість альтернативних СУБД, таких як MySQL, SQL Server, Oracle, однак для інтеграції з фреймворком було обрано саме Postgres через його популярність, корисні функції і наявність механізму логічної реплікації.

У проекті Postgres використовується як основна база даних, що містить всі дані, на які клієнт буде підписуватись.

Іншою значною перевагою Postgres є наявність механізму логічної реплікації. Логічна реплікація в PostgreSQL — це механізм, який дозволяє передавати дані між базами даних PostgreSQL для подальшого відтворення. На відміну від фізичної реплікації, де дані передаються у ідентичних блоках і записуються у однакових адресах, логічна реплікація працює на рівні об'єктів бази даних, таких як таблиці та записи.

Механізм логічної реплікації включає в себе використання спеціальних модулів та компонентів, які перетворюють транзакції, що відбуваються на джерелі, у логічні записи, які потім передаються до приймача. Це дозволяє ефективно відслідковувати зміни в структурі бази даних та реплікувати їх без прив'язки до фізичного формату даних.

Однією з ключових переваг логічної реплікації є можливість виборів стосовно того, які дані реплікувати та яким чином це робити. Користувач може вказати, які таблиці чи колонки реплікувати, виключити певні дані, або навіть трансформувати дані перед їх передачею до приймача. Це робить логічну реплікацію дуже гнучкою і придатною для різних сценаріїв використання, включаючи міграції даних, резервне копіювання та розподілені системи.

Ще однією вагомою перевагою логічної реплікації є можливість використовувати її для реплікації між різними версіями PostgreSQL, оскільки вона не прив'язана до формату фізичних блоків. Це полегшує процес оновлення та підтримки розподілених середовищ баз даних.

У проекті логічна реплікація використовується з метою отримання інформації про транзакції. За допомогою цих даних фреймворк отримує інформацію про всі зміни у базі даних у реальному часі, надаючи змогу надсилати оновлення до клієнтів.

# РОЗДІЛ 3 РОЗРОБКА ФРЕЙМВОРКУ СИНХРОНІЗАЦІЇ ДАНИХ РЕАЛЬНОГО ЧАСУ В КЛІЄНТ-СЕРВЕРНІЙ АРХІТЕКТУРІ

## 3.1 Архітектура системи

Оскільки фреймворк регулює комунікацію між сервером та клієнтом, він складається з двох частин, що працюють на відповідних сторонах. Серверна частина відповідає за відслідковування змін у базі даних, керування підписками клієнтів, та надання доступу до даних. Клієнтська сторона в свою чергу зберігає поточний стан представлень даних, робить автоматизовані запити на сервер у разі необхідності оновлення існуючих представлень, або завантаження нових, ініціює підписки на необхідні ресурси, та обробляє повідомлення, що надходять. Серверна частина поділяється далі на три сервіси (див. Рис. 4).

- Сервіс представлень, що обчислює та повертає дані за допомогою мови запитів GraphQL. Види запитів, що підтримуються системою визначаються розробником заздалегідь у декларативному вигляді.
- Сервіс підписок, що контролює веб-сокет підключення клієнтів, реєструє їх підписки і надсилає оновлення, на які вони підписались.
- Сервіс сигналів, що прослуховує повідомлення логічної реплікації і записує ті, що необхідні, у чергу повідомлень, звідки їх зчитує сервіс підписок.

Всі сервіси можна масштабувати горизонтально. Сервіс представлень не містить жодного внутрішнього стану, оскільки його відповідальності обмежені обчисленнями та запитами до бази даних. Сервіс підписок, хоч і містить внутрішній стан у вигляді відкритого веб-сокет підключення та списку підписок, але все одно може бути легко масштабований горизонтально, оскільки вся комунікація, на яку він покладається проходить через централізовану чергу. Сервіс сигналів, в свою чергу, не потребує

масштабування в залежності від кількості відкритих підключень клієнтів, оскільки єдина його залежність — це база даних.

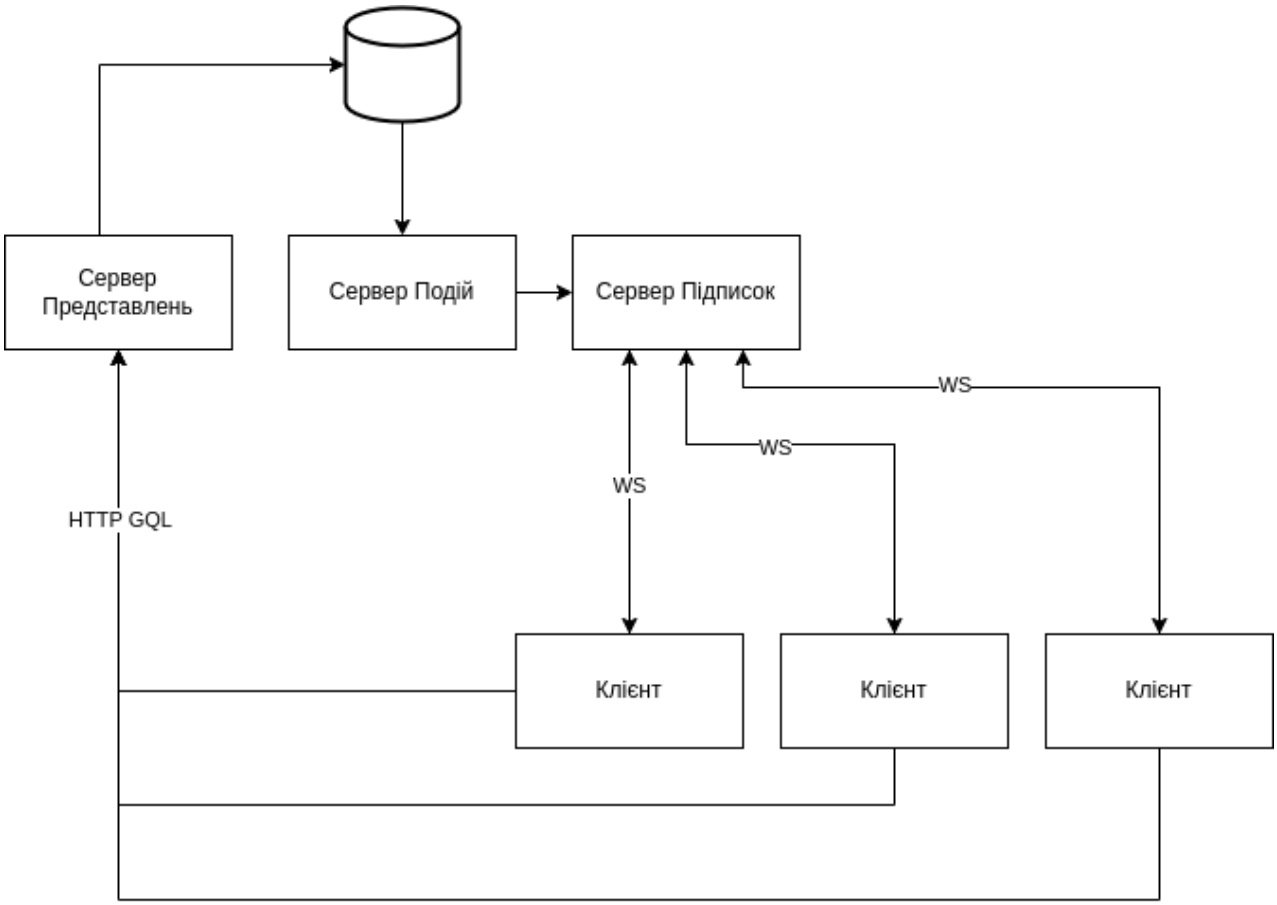


Рисунок *4 — Схема взаємодії сервісів*

### 3.2 Проєктування програмної системи

Центральною ідеєю фреймворку є декларативний опис представлень даних на стороні сервера, динамічне визначення залежностей цих представлень стороною клієнта після завантаження та подальша синхронізація з можливою оптимізацією у вигляді переобчислення виключно на стороні клієнта після отриманого оновлення (див. Рис. 5).

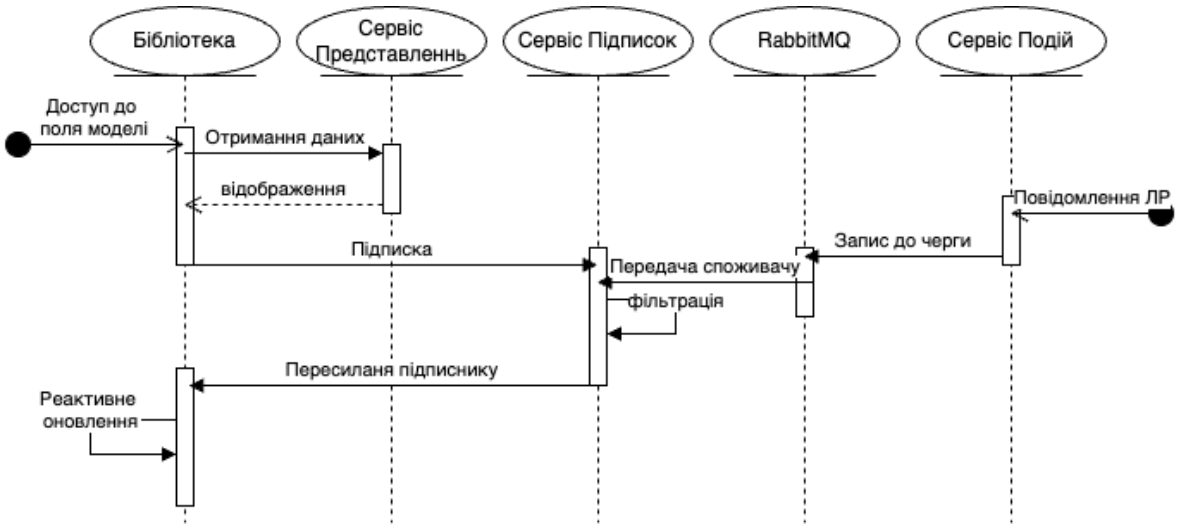


Рисунок *5 — Діаграма послідовностей для процесу синхронізації*

Оновленнями у системі називаються короткі повідомлення одного з трьох видів — INSERT, UPDATE або DELETE. Ці повідомлення містять в собі виключно інформацію про рядок бази даних, що було змінено. Залежностями у системі називаються конкретні рядки бази даних, при зміні яких змінюється значення залежного представлення.

Виходячи з описаної у попередньому розділі архітектури, фреймворк має складатись з трьох сервісів та клієнтської бібліотеки. Сервіс представлень має експортувати ASGI додаток з двома ендпоінтами — один для запитів GraphQL, та інший для завантаження списку визначених представлень даних. Для забезпечення роботи першого з ендпоінтів використовується бібліотека ariadne, що є реалізацією GraphQL та надає весь необхідний функціонал. Для ініціалізації GraphQL сервера бібліотека потребує визначення всіх типів і запитів у відповідному форматі. Фреймворк генерує їх автоматично, завдяки заздалегідь визначеним розробником моделям. Приклад визначення типів даних у мові GraphQL наведено у лістингу 1.

Лістинг *1 Приклад визначень моделей у мові GraphQL*

```
type ObjectId {
  id: Int
  entity: String
```

```
}
type Player {
  best_weapon: String
  total_deaths: Int
  total_kills: Int
  id: Int
  uuid: String
  username: String
  created_at: String
  expressions: String
}
type PlayersView {
  players(page: Int, size: Int):
GenericHelper__Page__PlayerTable
  expressions: String
}
type GamePlayerEvent {
  id: Int
  game_id: Int
  player_id: Int
  type: String
  round_id: Int
  created_at: String
  team_id: Int
  expressions: String
}
type GameTeam {
  players: [ObjectId]
  id: Int
  created_at: String
  expressions: String
}
type Round {
```

```
  id: Int
  game_id: Int
  number: Int
  winner_id: Int
  created_at: String
  expressions: String
}
type PlayerKills {
  player: ObjectId
  kills: Int
  expressions: String
}
type Game {
  rounds: [ObjectId]
  team_a: ObjectId
  team_a_stats: [PlayerKills]
  team_b: ObjectId
  team_b_stats: [PlayerKills]
  id: Int
  map: String
  team_a_id: Int
  team_b_id: Int
  created_at: String
  expressions: String
}
type GamesView {
  games(page: Int, size: Int):
GenericHelper__Page__GameTable
  expressions: String
}
type GamePlayer {
  id: Int
  game_id: Int
```

```
  player_id: Int
  team_id: Int
  created_at: String
  expressions: String
}
type GenericHelper__Page__PlayerTable {
  items: [ObjectId]
  count: Int
}
type GenericHelper__Page__GameTable {
  items: [ObjectId]
  count: Int
}
input ObjectIdInput {
  entity: String
  id: Int
  modifiers: String
  dependencies: [ObjectIdInput]
}
type Query {
  player(identifier: ObjectIdInput): Player
  gamePlayerEvent(identifier: ObjectIdInput):
GamePlayerEvent
  game(identifier: ObjectIdInput): Game
  gamesView(identifier: ObjectIdInput): GamesView
  gamePlayer(identifier: ObjectIdInput): GamePlayer
  blogPostComment(identifier: ObjectIdInput):
BlogPostComment
  blogPost(identifier: ObjectIdInput): BlogPost
  blogPostsView(identifier: ObjectIdInput): BlogPostsView
}
```

Резолвери для полів назначаються так само автоматично, виходячи з визначень моделей фреймворка. Другий ендпоінт є необхідним для надання повної інформації про наявні представлення даних та типи клієнтам, що використовують їх для коректного парсингу даних, отриманих від сервіса представлень. Приклад даних, що повертає цей ендпоінт наведено у лістінгу 2.

Лістінг *2 Приклад експорту схем моделей*

```
[
  {
    "type":"struct",
    "name":"ObjectId",
    "fields":[
      {
        "name":"id",
        "type":"int",
        "config":{}
      },
      {
        "name":"entity",
        "type":"str",
        "config":{}
      },
    ]
  },
  {
    "type":"model",
    "name":"Player",
    "fields":[
      {
        "name":"best_weapon",
        "type":"str",
        "config":{}
```

```
},
{
 "name":"total_deaths",
 "type":"int",
 "config":{}
},
{
 "name":"total_kills",
 "type":"int",
 "config":{}
},
{
 "name":"id",
 "type":"int",
 "config":{}
},
{
 "name":"uuid",
 "type":"str",
 "config":{}
},
{
 "name":"username",
 "type":"str",
 "config":{}
},
{
 "name":"created_at",
 "type":"datetime",
 "config":{}
},
{
 "name":"expressions",
```

```
      "type":"str",
      "config":{}
     }
   ]
  },
  {
   "type":"model",
   "name":"PlayersView",
   "fields":[
     {
      "name":"players",
      "type":"Player",
      "config":{"is_page":true}
     },
     {
      "name":"expressions",
      "type":"str",
      "config":{}
     }
   ]
  },
 ]
```

Сервіс має надати змогу розробнику визначити необхідну кількість представлень даних, що можуть бути таблицями, віртуальними таблицями або структурами. Приклад визначення представлень даних наведено у лістінгу 3.

Лістінг *3 Приклад визначення представлень даних*

```
@table_manager.table
class PlayerTable(Table[Player]):
  id: int
  uuid: str
```

```
  username: str
  created_at: datetime

  def __init__(self, model, context: BlazeContext, *args,
**kwargs):
    super().__init__(model, context)

  @computed()
  def total_deaths(self) -> int:
    return (
      Select(count(SQLTable(GamePlayerEvent).col("id")))
      .where(
        SQLTable(GamePlayerEvent).col("type") == "death",
        SQLTable(GamePlayerEvent).col("player_id") ==
self.get_model().id
      )
      .one()
    )

  @computed()
  def total_kills(self) -> int:
    return (
      Select(count(SQLTable(GamePlayerEvent).col("id")))
      .where(
        SQLTable(GamePlayerEvent).col("type") == "kill",
        SQLTable(GamePlayerEvent).col("player_id") ==
self.get_model().id
      )
      .one()
    )

  @computed()
```

```
  def best_weapon(self) -> str:
    return (

Select(SQLTable(GamePlayerEvent).col("extra")["weapon"])
      .where(
        SQLTable(GamePlayerEvent).col("type") == "kill",
        SQLTable(GamePlayerEvent).col("player_id") ==
self.get_model().id
      )
      .order_by(count(SQLTable(GamePlayerEvent).col("extra")
["weapon"]).desc())
      .group_by(SQLTable(GamePlayerEvent).col("extra")["weap
on"])
      .one()
    )
```

Фреймворк надає можливість розширювати рядки таблиць або віртуальних таблиць додатковими обчислюваними полями. Ці поля представлені у вигляді методів класу з декоратором “computed”. Обчислювані поля можуть приймати додаткові аргументи, для цього метод має оголосити їх як свої параметри і додати до них відповідні анотації типів. Також анотація типу, що повертається є необхідною. Обчислювані поля можуть виконувати операції одразу, або повертати спеціальний об'єкт, що представляє SQL запит. У другому випадку виконання запиту так само відбудеться, але об’єкт запиту буде серіалізовано і передано на сторону клієнта. Код класів-репрезентацій SQL запитів наведено у лістінгу 4.

Лістінг *4 Класи-репрезентації SQL запитів*

```
class Select:
  def __init__(self, *args):
```

```
    self.args = args
    self.where_expr = []
    self.group_by_expr = []
    self._one = False
    self.order_by_expr = []
    self.join_expr = []
  def where(self, *args):
    self.where_expr.extend(args)
    return self
  def join(self, table):
    self.join_expr.append(table)
    return self
  def group_by(self, *args):
    self.group_by_expr.extend(args)
    return self
  def order_by(self, *args):
    self.order_by_expr.extend(args)
    return self
  def one(self):
    self._one = True
    return self
  def dict(self, tm):
    return {
      "select": [arg.dict(tm) for arg in self.args],
      "where": [arg.dict(tm) for arg in self.where_expr],
      "group_by": [arg.dict(tm) for arg in
self.group_by_expr],
      "order_by": [arg.dict(tm) for arg in
self.order_by_expr],
      "join": [table.dict(tm) for table in self.join_expr],
      "one": self._one
    }
  def compile(self):
```

```
    stmt = select(
      *[arg.compile() for arg in self.args]
    ).where(
      *[arg.compile() for arg in self.where_expr]
    ).group_by(
      *[arg.compile() for arg in self.group_by_expr]
    ).order_by(
      *[arg.compile() for arg in self.order_by_expr]
    )
    if self.join_expr:
      stmt = join(
        self.join_expr[0].compile()
      )
    return stmt
class Selectable:
  def __init__(self):
    self._label = None
    self._asc = True
  def label(self, name):
    self._label = name
    return self
  def asc(self):
    self._asc = True
    return self
  def desc(self):
    self._asc = False
    return self
  def __eq__(self, other):
    if isinstance(other, str | int):
      other = Literal(other)
    return BinaryOperation(self, other, "==")
  def dict(self):
    return {
```

```
      "label": self._label,
      "asc": self._asc
    }
class Literal(Selectable):
  def __init__(self, value):
    super().__init__()
    self.value = value
  def dict(self, tm):
    return {
      "value": self.value,
      "label": self._label,
      "object": "literal"
    }
  def compile(self):
    return self.value
class Func(Selectable):
  def __init__(self, name, args):
    super().__init__()
    self.name = name
    self.args = args
  def dict(self, tm):
    return {
      **super().dict(),
      "function": self.name,
      "clauses": [arg.dict(tm) for arg in self.args],
      "object": "function"
    }
  def compile(self):
    fn = getattr(func, self.name)(*[arg.compile() for arg in
self.args])
    if self._label:
      return fn.label(self._label)
    return fn
```

```
def count(arg):
  if isinstance(arg, str | int):
    arg = Literal(arg)
  return Func("count", [arg])
class JSONProperty(Selectable):
  def __init__(self, column, path):
    super().__init__()
    self.column = column
    self.path = path
  def dict(self, tm):
    return {
      **super().dict(),
      "column": self.column.dict(tm),
      "path": self.path,
      "object": "json"
    }
  def compile(self):
    return self.column.compile()[self.path]
class TableColumn(Selectable):
  def __init__(self, table, name):
    super().__init__()
    self.table = table
    self.name = name
  def __getitem__(self, item):
    return JSONProperty(self, item)
  def dict(self, tm):
    return {
      **super().dict(),
      "table": self.table.dict(tm)["table"],
      "column": self.name,
      "object": "column"
    }
  def compile(self):
```

```
    col = getattr(self.table.sqlalchemy_table, self.name)
    if self._label:
      return col.label(self._label)
    return col
class BinaryOperation:
  def __init__(self, left, right, operator):
    self.left = left
    self.right = right
    self.operator = operator
  def dict(self, tm):
    return {
      "object": "biexpr",
      "left": self.left.dict(tm),
      "right": self.right.dict(tm),
      "operator": self.operator
    }
  def compile(self):
    if self.operator == "==":
      return self.left.compile() == self.right.compile()
    raise ValueError(f"Unknown operator {self.operator}")
class SQLTable:
  def __init__(self, sqlalchemy_table):
    self.sqlalchemy_table = sqlalchemy_table
    self.name = sqlalchemy_table.__table__.name
  def col(self, name):
    return TableColumn(self, name)
  def dict(self, tm):
    model = tm.table_name_to_model.get(self.name)
    if not model:
      raise ValueError(f"Unknown table {self.name}")
    return {
      "object": "table",
      "table": model.get_type_name()
```

```
    }
  def compile(self):
    return self.sqlalchemy_table
```

Завдяки цьому клієнт матиме інформацію щодо того, від яких рядків залежить це представлення, що дозволяє автоматично створити всі необхідні підписки-вирази, та в деяких випадках додатково надає можливість переобчислити значення представлення при надходженні оновлення без необхідності повторного виконання запиту на сервері.

Таблиці у фреймворку репрезентують таблицю бази даних. Ця сутність за замовчуванням має єдину залежність — рядок бази даних, який вони представляють. Віртуальними таблицями називаються представлення даних, що не мають за собою конкретного рядка у базі даних. Такі таблиці скоріш за все представлятимуть на конкретний рядок, а цілий запит до бази даних, що повертає певний результат, та містить свої залежності. Третій тип даних, структури, не являють собою тип даних, що може бути отриманий прямим запитом GraphQL. Структури це допоміжні типи даних, що можуть представляти вкладену структуру. Для визначення схеми представлень фреймворк активно використовує можливості анотацій мови програмування Python. Поля таблиць фреймворку визначаються за допомогою анотацій полів класів, обчислювані поля так само активно використовують анотації для опису вхідних параметрів і результату.

Сервер представлень можна запустити з використанням бібліотеки uvicorn у ролі веб-сервера, що продемонстровано у лістінгу 5.

Лістинг *5 Запуск сервіса представлень*

```
app = FastAPI()
app.add_middleware(
  CORSMiddleware,
  allow_origins=["*"],
  allow_credentials=True,
```

```
  allow_methods=["*"],
  allow_headers=["*"],
)
asgi_app = GraphQL(
  create_schema(table_manager),
  http_handler=GraphQLHTTPHandler(
    extensions=[
      partial(SessionExtension, session_maker)
    ]
  )
)
app.mount(
  "/graphql",
  asgi_app
)

if __name__ == '__main__':
  import uvicorn
  uvicorn.run(app, host="0.0.0.0", port=8000)
```

Таким чином сервер буде запущено на порті 8000 і він буде готовий приймати GraphQL запити за шляхом /graphql.

Сервіс підписок, в свою чергу, відповідає за комунікацію з підключеними клієнтами в режимі реального часу. Він маршрутизує події з внутрішньої черги повідомлень до клієнтів, базуючись на підписках-виразах, що були отримані та збережені для кожного з них. Підпискою-виразом у цьому контексті є невеликий JSON об’єкт, що у вигляді обчислюваного виразу дає опис залежностям представлення. У певному сенсі підписки-вирази схожі на WHERE сегмент SQL запиту. Вони містять поле `entity`, що відповідає таблиці, зареєстрованій у фреймворку, поле `event`, що відповідає дії, яка була застосована до рядка таблиці, та поле `where`, що містить умову, яка

складається з двох частин: і-умови та або-умови. Ці умови націлені на визначення чи є будь-яке оновлення релевантним для даного клієнта. Якщо умови виконуються, оновлення буде маршрутизоване до клієнта, у іншому ж випадку — ні. Для простоти і швидкості обчислення умови максимально спрощуються до двох списків — і-та-або умови. Обидві групи мають прийняти значення істини, щоб оновлення було надіслане до клієнта, що створив цю підписку, це означає, що якщо хоча б одна умова у “і” списку приймає значення хибності, обчислення зупиняється і оновлення ігнорується. Визначення структури залежностей наведено у лістінгу 6.

Лістінг *6 Визначення структури залежності*

```
class Value(BaseModel):
 class Source(str, Enum):
  CONSTANT = "CONSTANT",
  DEPENDENCY_OLD = "DEPENDENCY.OLD",
  DEPENDENCY_NEW = "DEPENDENCY.NEW",
  DEPENDENCY_NEW_OR_ANY = "DEPENDENCY.NEW_OR_ANY",
  DEPENDENCY_OLD_OR_ANY = "DEPENDENCY.OLD_OR_ANY",
  DEPENDENCY_OLD_OR_TRUE = "DEPENDENCY.OLD_OR_TRUE",
  DEPENDENCY_NEW_OR_TRUE = "DEPENDENCY.NEW_OR_TRUE",
 source: Source
 expression: str | int

class Condition(BaseModel):
  left: Value
  right: Value
  operator: Literal["==", "!=", ">", "<"]

class Where(BaseModel):
  and_: list[Condition] = Field(alias="and")
  or_: list[Condition] = Field(alias="or")
```

```
class Subscription(BaseModel):
  entity: str
  where: Where
  event: Literal["insert", "update", "delete"]
  subscription_id: int
```

Умови, в свою чергу, визначаються як логчні бінарні операції. Вони складаються з лівостороннього аргумента, правостороннього аргумента, та оператора. Аргументи можуть приймати різний вигляд. Основними типами виразів (поле `source`) є константи — аргументи, що завжди прийматимуть одне значення, та посилання на значення рядка, нове або старе, що оновлюється. Однак нові та старі значення рядка не завжди є доступними. При створенні рядка (insert) існуватимуть нові значення рядка, однак старих значеннь, очевидно, існувати не може. У разі видалення рядка (delete) виникає аналогічна, але протилежна ситуація, існуватимуть старі значення, але не нові. Єдина подія рядка, що матиме і старі і нові значення, це update. По цій причині, крім типів виразів `DEPENDENCY_NEW` (посилання на нове значення), та `DEPENDENCY_OLD` (посилання на попередні значення) існують допоміжні типи значень, такі як `DEPENDENCY_NEW_OR_ANY`, що отримує доступ до будь-якого значення, оскільки або нове, або старе значення гарантовано існує, але надаючи перевагу новому. Поле `expression` представляє безпосередньо вираз, що має бути обчислений у значення, що пізніше буде застосоване у обчисленні бінарного логічного виразу — умови. Для типу виразу константа (`CONSTANT`), значення цього поля і буде результатом обчислення. Для всіх інших посилальних типів виразу, поле прийматиме значення назви атрибуту таблиці, що була оновлена. Наприклад, якщо існує таблиця гравців `Players`, і існує представлення даних `PlayersWithHighScore`, що містить тільки гравців, рахунок яких перевищує 100 очків, клієнт, що відображає таке представлення матиме створити залежність, що відслідковуватиме рядки

таблиці `Players`, що мають підходящий рахунок. Приклад такої залежності продемонстровано у лістінгу 7.

Лістінг *7 Приклад виразу-підписки*

```
{
"entity": "Player",
"event": "insert",
"where": {
 "and": [
  {
   "left": {
    "expression": "score",
    "source": "DEPENDENCY.NEW"
   },
   "right": {
    "expression": 100,
    "source": "CONSTANT"
   },
   "operator": ">"
  }
 ],
 "or": []
}
}
```

Як можна побачити, підписка створюється на таблицю `Players`, і відслідковує вона лише рядки, що мають рахунок вищий за 100. Однак нескладно помітити, що ця підписка стосується лише insert подій, тож виникає питання, чи можна використати цю ж підписку для двох інших подій? Відповідь — ні. Причина в тому, що якщо замінити лише селектор події на update, система отримуватиме оновлення навіть у тому випадку, якщо поле, що є релевантним до запиту, у цьому випадку `score`, не змінилось, іншими словами, якщо воно підходило раніше, і все ще підходить після. Тож для update

події рядка можна змінити підписку-вираз, щоб вона виглядала як показано у лістінгу 8.

Лістінг *8 Вираз-підписка, що не відслідковує зайві оновлення*

```
{
 "entity": "Player",
 "event": "update",
 "where": {
  "and": [
   {
    "left": {
     "expression": "score",
     "source": "DEPENDENCY.NEW"
    },
    "right": {
     "expression": 100,
     "source": "CONSTANT"
    },
    "operator": ">"
   },
   {
    "left": {
     "expression": "score",
     "source": "DEPENDENCY.OLD"
    },
    "right": {
     "expression": 100,
     "source": "CONSTANT"
    },
    "operator": "<="
   }
```

```
  ],
  "or": []
 }
}
```

Тепер клієнт отримуватиме тільки ті оновлення, під час яких рядок перейшов з стану “не підходить” до стану “підходить”. Однак, нескладно помітити що випадок зміни станів навпаки не обробляється. У разі, якщо рядок, що підходив до виразу-підписки було змінено таким чином, що він більше не підходить, клієнт так само має отримати оновлення. Для цього можна створити ще одну підписку, що буде відслідковувати нові та старі значення подібно до існуючої, але навпаки, як продемонстровано у лістінгу 9.

Лістінг *9 Вираз-підписка, що відслідковує рядки, які більше не підходять*

```
{
 "entity": "Player",
 "event": "update",
 "where": {
  "and": [
   {
    "left": {
     "expression": "score",
     "source": "DEPENDENCY.OLD"
    },
    "right": {
     "expression": 100,
     "source": "CONSTANT"
    },
    "operator": ">"
   },
   {
    "left": {
```

```
     "expression": "score",
     "source": "DEPENDENCY.NEW"
    },
    "right": {
     "expression": 100,
     "source": "CONSTANT"
    },
    "operator": "<="
   }

  ],
  "or": []
 }
}
```

Ця підписка буде відфільтровувати тільки ті оновлення, при яких старе значення рядка підходило під вираз, а нове вже ні.

Delete оновлення може бути опрацьоване схожим чином. Воно має відфільтрувати ті рядки, старі значення котрих підходити до виразу.

Ситуація ускладнюється у випадках, коли потрібно відслідковувати декілька полів одразу. Якщо розширити приклад з представленням `PlayersWithHighScore`, і сказатищо це представлення додатково залежить від рядка гри (`Game`), у якій приймали участь гравці. вираз-підписка для insert події рядка може бути легко розширена, як продемонстровано у лістінгу 10.

Лістінг *10 Вираз-підписка, що має декілька умов*

```
{
 "entity": "Player",
 "event": "insert",
 "where": {
  "and": [
   {
```

```
   "left": {
    "expression": "score",
    "source": "DEPENDENCY.NEW"
   },
   "right": {
    "expression": 100,
    "source": "CONSTANT"
   },
   "operator": ">"
  },
  {
   "left": {
    "expression": "game_id",
    "source": "DEPENDENCY.NEW"
   },
   "right": {
    "expression": 1,
    "source": "CONSTANT"
   },
   "operator": "=="
  }
 ],
 "or": []
}
}
```

Подібним чином можна переформатувати і першу підписку-вираз для update події рядка, що тестує позитивний сценарій. Однак тестування негативного сценарію вже має виглядати інакше. Якщо хоча б одне з полів `score` або `game_id` перестає підходти під вираз, весь рядок перестає підходити. виходячи з цього, можна отримати підписку-вираз, що наведена у лістінгу 11.

Лістінг *11 Вираз-підписка для update події*

```
{
"entity": "Player",
"event": "update",
"where": {
 "and": [],
 "or": [
  {
   "left": {
    "expression": "score",
    "source": "DEPENDENCY.NEW"
   },
   "right": {
    "expression": 100,
    "source": "CONSTANT"
   },
   "operator": "<"
  },
  {
   "left": {
    "expression": "game_id",
    "source": "DEPENDENCY.NEW"
   },
   "right": {
    "expression": 1,
    "source": "CONSTANT"
   },
   "operator": "!="
  }
 ]
}
}
```

Варто звернути увагу на те, що аби перетворити (A && B) на !(A && B) необхідно замінити операцію “і” на “або”, та інвертувати оператор, що і відбулося у прикладі.

Для запуску сервіса подій можна використати веб-сервер uvicorn, як продемонстровано у лістінгу 12.

Лістінг *12 Запуск сервіса подій*

```
@app.on_event("startup")
async def startup_event():
  await msgqueue.start()

if __name__ == '__main__':
  import uvicorn
  uvicorn.run(app, host="0.0.0.0", port=9000)
```

Сервіс сигналів використовує інструмнт логічної реплікації PostgreSQL для отримання змін у базі даних у реальному часі. Для того, щоб використовувати логічну реплікацію, потрібно спершу виконати певні дії для підготовки бази даних, а саме створити слот для логічної реплікації і налаштувати ідентичність реплікації для кожної таблиці як FULL, дозволяючи отримати доступ до всього рядка повністю при отримані повідомлення реплікації. Серверна частина фреймворку, не дивлячись на розподіл на сервіси, побудована за принципом monorepository, що означає, що всі сервіси використовують один і той самий код. Це дозволяє сервісу сигналів імпортувати менеджер таблиць, що містить визначення таблиць, задекларованих як частина сервісу представлень. Маючи цей доступ, сервіс сигналів може налаштувати режим реплікації для всіх таблиць, що використовуються фреймворком, в той час як всі інші таблиці залишаються незмінними. Іншим використанням цього доступу є фільтрація подій. Публічно доступними є тільки події, що стосуються таблиць бази даних, для

яких явним чином було визначено відповідне представлення, і тільки ті поля будуть присутні в оновленні, які були так само явно визначені на таблиці. Наприклад, якщо у таблиці користувачів існує поле “пароль”, то воно не буде фігурувати у змісті оновлення якщо воно явно не було визначено на відповідній таблиці фреймворку. Логіку вибіркової обробки повідомлень реплікації наведено у лістінгу 13.

Лістінг *13 Обробка повідомлень реплікації*

```
async def start(self):
    await msgqueue.start()
    message: ReplicationMessage
    async for message in read_events(
self.db_host, self.db_port, self.db_name, self.db_user,
self.db_password, self.slot_name):
      data = json.loads(message.payload)
      changes = data['change']
      for change in changes:
        if change['kind'] not in {'insert', 'update',
'delete'}:
          continue
        model =
self.table_name_to_model.get(change['table'])
        # model is not registered, so we should not
propagate it's changes
        # (could be sensitive)
        if not model:
          continue
        try:
          # sanitize data by only including fields present
on the model
          fields =
model.get_model_props(self.table_manager.models)
```

```
          explicit_field_set = set([f.name for f in fields])
          change = self.parse_change(change, model,
explicit_field_set)
          await carpet.produce(change)
        except Exception as e:
          import traceback
          traceback.print_exc()
```

Для зчитування повідомлень, отриманих від PostgeSQL, сервер відкриває нове підключення під час запуску, після чого очікує, щоб дескриптор підключення був готовий до читання або запису, залежно від ситуації. Функції контролю стану підключення наведено у лістінгу 14.

Лістінг *14 Допоміжні функції для отримання стану підключення*

```
async def connection_ready(conn: connection) -> None:
  while True:
    state: int = conn.poll()
    if state == psycopg2.extensions.POLL_OK:
      break
    elif state == psycopg2.extensions.POLL_WRITE:
      await wait_write_ready(conn.fileno())
    elif state == psycopg2.extensions.POLL_READ:
      await wait_read_ready(conn.fileno())
    elif state == psycopg2.extensions.POLL_ERROR:
      raise psycopg2.OperationalError("POLL_ERROR")
async def wait_read_ready(descriptor: int):
  """Wait until the descriptor is ready for reading.
  Args:
    descriptor: The file descriptor to wait for.
  """
  future = Future()
  def callback():
```

```
    future.set_result(None)
    asyncio.get_event_loop().remove_reader(descriptor)
  asyncio.get_event_loop().add_reader(descriptor, callback)
  await future
async def wait_write_ready(descriptor):
  """Wait until the descriptor is ready for writing.
  Args:
    descriptor: The file descriptor to wait for.
  """
  future = Future()
  def callback():
    future.set_result(None)
    asyncio.get_event_loop().remove_writer(descriptor)
  asyncio.get_event_loop().add_writer(descriptor, callback)
  await future
```

Після того, як підключення відкрите і готове до запису, сервіс ініціює початок реплікації. Логіку зчитування повідомлень наведено у лістінгу 15.

Лістінг *15 Зчитування повідомлень реплікації*

```
async def read_events(host, port, database, user, password,
slot_name) -> AsyncGenerator[str, None]:
  conn: connection = psycopg2.connect(
    host=host,
    port=port,
    database=database,
    user=user,
    password=password,
```

```
connection_factory=psycopg2.extras.LogicalReplicationConnec
tion,
    async_=True,
  )
  await connection_ready(conn)
  _cursor: ReplicationCursor
  with conn.cursor() as _cursor:
    await ensure_slot_exists(_cursor, slot_name, database)
    _cursor.start_replication(
      slot_name=slot_name,
      decode=True,
      options=REPLICATION_OPTIONS,
    )
    await connection_ready(conn)
    while True:
      await wait_read_ready(_cursor.fileno())
      message: Optional[ReplicationMessage] =
_cursor.read_message()
      if message is None:
        continue

message.cursor.send_feedback(flush_lsn=message.data_start)
      yield message
```

Повідомлення, отримані через реплікацію надсилаються до черги повідомлень, звідки вони будуть зчитані іншими сервісами.

Запуск сервіса описано у лістінгу 16.

Лістінг *16 Запуск сервісу подій*

```
signaler = Signaler(
  db_host=...,
```

```
  db_port=...,
  db_name='blzlk_demo',
  db_user='...',
  db_password='...',
  slot_name="blzlk_demo",
  table_manager=table_manager
)
if __name__ == '__main__':
  asyncio.run(signaler.start())
```

Клієнтська частина фреймворку складається з бібліотеки, що містить всі необхідні інструменти для зручної комунікації з сервером та реалізації логіки збереження та оновлення даних. Розроблена бібліотека націлена на інтеграцію з додатками, написаними на Vue.js, але подібна бібліотека може бути розроблена для майже будь-якого середовища, не обмежуючись лише браузерами. До функцій бібліотеки входить:

- Автоматичне виконання GraphQL запитів до сервера для отримання початкових представлень даних.
- Підтримка веб-сокет комунікації з сервером.
- Автоматична генерація виразів-підписок і подальше їх надсилання на сервер.
- Обробка оновлень, отриманих від сервера, включаючи можливе переобчислення значень на клієнті, для запобігання зайвих звернень до бази даних.

Підтримка реактивності.

Представлення даних на клієнті реалізовані у вигляді спеціальних об’єкетів — моделей. Класи цих моделей генеруються автоматично, завдяки схемам даних, що отримуються від сервера. Код завантаження визначень наведено у лістінгу 17. Логіку динамічної генерації моделей наведено у лістінгу 18.

Лістінг *17 Автоматичне завантаження моделей*

```
async loadSchema() {
    const response = await fetch(SCHEMAS_URL, {
      headers: {
        "Content-Type": "application/json",
        "session_id": this.sessionId,
        "Authorization": this.getAuthKey(),
      },
    });
    this.__schemaCache = await response.json();
  }

  loadModelDefFromSchema(schema: any) {
    const modelType = modelFromSchema(schema);
    this.registerModelType(modelType as any);
    return modelType;
  }

  async loadModels() {
    await this.loadSchema();
    for (let schema of this.__schemaCache) {
      this.loadModelDefFromSchema(schema);
    }
    this.resolveTypes();
    for (const callback of this.__onModelLoad) {
      callback();
    }
  }
```

Лістинг *18 Динамічне створення класу моделі*

```
export function modelFromSchema(schema: ModelSchema):
typeof Model | typeof Struct {

  const Base = schema.type == "model" ? Model : Struct;

  class DynamicModel extends Base {
    static [key: string]: any
  }

  Object.defineProperty(
    DynamicModel, "name", {
      writable: true,
      value: schema.name
    }
  )

  for (let fieldSchema of schema.fields) {
    let type;

    switch (fieldSchema.type) {
      case "str": case "datetime":
        type = String;
        break;
      case "int":
        type = Int;
        break;
      default:
        type = fieldSchema.type;
    }

    if (fieldSchema.config.is_page) {
      DynamicModel[fieldSchema.name] = Page(
```

```
        type as any,
        fieldSchema.config
      );
    } else {
      DynamicModel[fieldSchema.name] = Field(
        type as any,
        fieldSchema.config
      );
    }
  }

  return DynamicModel as typeof Model | typeof Struct;
}
```

Автоматичне виконання GraphQL запитів відбувається у так званому “лінивому” режимі, що означає, що модель залишатиметься у “неініціалізованому стані” до тих пір, поки не буде зроблено спробу доступу до її полів. Таке динамічне завантаження стає можливим завдяки використанню об’єкта проксі, що були додані до JavaScript у стандарті ES6. Код визначення proxy обгортки наведено у лістінгу 19.

Лістінг *19 Створення proxy об’єкта моделі*

```
this.__myProxy = new Proxy(this, {
      ownKeys: (target) => {
        const Constr = this.constructor as typeof Loadable;
        const fields = Constr.getDeclaredFields();
        return Object.keys(fields);
      },
      get: (target: any, name: string | Symbol) => {
        // we don't intercept symbols
        if (typeof name === "symbol") {
          return target[name];
```

```
        }
        name = name as string;
        const Constr = this.constructor as typeof Loadable;
        const fields = Constr.getDeclaredFields();
        if (!Object.prototype.hasOwnProperty.call(fields,
name)) {
          return target[name];
        }
        const field = fields[name as string];
        if (!field) {
          return target[name as string] as any;
        }
        return this.getOrLoadValue(name);
      },
      set: (target, name, value) => {
        target[name] = value;
        return true;
      },
    });
```

Важливо помітити, що сервіс представлень не повертає глибоко вкладених даних. Якщо модель має іншу модель як своє поле, то фреймворк поверне лише ідентифікатор на вкладену модель. На стороні клієнта цього ідентифікатора буде достатньо аби створити нову, неініціалізовану модель, або у разі якщо ця модель вже була створена і синхронізується іншим представленням, перевикористати її. Код функції пошуку моделі наведено у лістінгу 20.

Лістінг *20 Функція пошуку моделі за ідентифікатором*

```
static Find<T>(
 this: typeof Model & {new(o: any): T},
 id: number,
```

```
 {
  dependencies=[],
  unique=true
 }: {dependencies: ObjID[], unique: boolean} = {
 unique: true, dependencies: []
 }
): T {
    const objId = {id: id, entity: this.name};
    if (!unique) {
      return this.__modelManager.ModelFactory(
       objId, {unique: false}
      )?.getReactiveModel() as T;
    }
    return this.getOrCreate(
      objId, "", null as any, null, {}, () => {}
    )
  }
```

Автоматична генерація виразів-підписок відбувається завдяки тому, що сервер разом із даними представлення надсилає список виразів у JSON форматі, що представляють собою репрезентацію SQL запиту, що було використано при обчисленні кожного з полів. Наприклад, зробивши наступний запит:

```
query {
 blogPostsView {
  posts(size: 2) { items { id } count }
  expressions
 }
}
```

Отримаємо такий результат:

```
{
 "data": {
```

```
  "blogPostsView": {
   "posts": {
    "items": [
     {"id": 1},
     {"id": 2}
    ],
    "count": 11
   },
   "expressions": "{\"posts\": {\"select\": [{\"object\":
\"table\", \"table\": \"BlogPost\"}], \"where\": [],
\"group_by\": [], \"order_by\": [{\"label\": null, \"asc\":
true, \"table\": \"BlogPost\", \"column\": \"id\",
\"object\": \"column\"}], \"join\": [], \"one\": false}}"
  }
 }
}
```


Це представлення є дуже примітивним, воно містить лише одне обчислюване поле, що визначається простим запитом:


```
{
 "posts": {
  "select": [
   {
    "object": "table",
    "table": "BlogPost"
   }
  ],
  "where": [],
  "group_by": [],
  "order_by": [
   {
    "label": null,
    "asc": true,
    "table": "BlogPost",
```

```
    "column": "id",
    "object": "column"
   }
  ],
  "join": [],
  "one": false
 }
}
```

Як можна побачити, це звичайний `select` запит з упорядкуванням по id.

Якщо ж спробувати запитати складніше представлення, як наприклад:

```
query {
 game(identifier: {id: 1, entity: "Game"}) {
  team_a_stats {
   kills
   player {
    id
   }
  }
  expressions
 }
}
```

То можна побачити, що і відповідний SQL запит стає значно складнішим:

```
{
 "team_a_stats": {
  "select": [
   {
    "label": "kills",
    "asc": true,
    "function": "count",
    "clauses": [
     {
      "value": "*",
```

```
    "label": null,
    "object": "literal"
   }
  ],
  "object": "function"
 },
 {
  "label": "player",
  "asc": true,
  "table": "GamePlayerEvent",
  "column": "player_id",
  "object": "column"
 }
],
"where": [
 {
  "object": "biexpr",
  "left": {
   "label": null,
   "asc": true,
   "table": "GamePlayerEvent",
   "column": "game_id",
   "object": "column"
  },
  "right": {
   "value": 1,
   "label": null,
   "object": "literal"
  },
  "operator": "=="
 },
 {
  "object": "biexpr",
```

```
  "left": {
   "label": null,
   "asc": true,
   "table": "GamePlayerEvent",
   "column": "team_id",
   "object": "column"
  },
  "right": {
   "label": null,
   "asc": true,
   "table": "Game",
   "column": "team_a_id",
   "object": "column"
  },
  "operator": "=="
 },
 {
  "object": "biexpr",
  "left": {
   "label": null,
   "asc": true,
   "table": "GamePlayerEvent",
   "column": "type",
   "object": "column"
  },
  "right": {
   "value": "kill",
   "label": null,
   "object": "literal"
  },
  "operator": "=="
 }
],
```

```
  "group_by": [
   {
    "label": null,
    "asc": true,
    "table": "GamePlayerEvent",
    "column": "player_id",
    "object": "column"
   }
  ],
  "order_by": [],
  "join": [],
  "one": false
 }
}
```

Однак складність запиту не відіграє великого значення для генерації виразів-підписок. Лише три секції мають вплив на залежності представлення — where секція, що може бути перетворена у вирази-підписки один до одного, group by, де будь-яка зміна значення матиме вплив на результат, та order by, де знову таки будь-яка зміна матиме вплив.

Оновлення даних представлень після отримання оновлення може відбуватись двома шляхами: методом повторного запиту представлення, або обчисленням нового значення на стороні клієнта, користуючись отриманим від сервера оновленням залежності. Фреймворк зберігає зв’язок між виразом-підпискою та моделлю. Таким чином при отриманні оновлення через вираз-підписку клієнт знатиме, який саме SQL запит породив цю підписку, тим самим дозволяючи провести обчислення нового значення на стороні клієнта, взагалі без додаткових звернень до сервера. Прикладом може стати обчислення наведеного вище поля “`team_a_stats`”. Воно має залежність від однієї таблиці, тож при створенні, оновленні, або видаленні рядків, що підходить під

встановлені умови, значення поля буде обчислюватись заново. Логіка перетворення SQL запита на список залежностей наведена у лістінгу 21.

Лістінг *21 Функція претворення специфікації SQL запита на залежності*

```
translateQuerySpecToDependencies(fieldName: string, spec:
FieldQuerySpec) {
    console.log("SQL SPEC of", fieldName, spec)
    let tablesAffected: Set<string> = new Set<string>();
    for (let col of spec.select) {
      if (col.object == "function") {
        for (let clause of col.clauses) {
          if (clause.object == "column") {
            tablesAffected.add(clause.table);
          }
        }
      } else {
        tablesAffected.add(col.table);
      }
    }
    if (tablesAffected.size > 1) {
      console.warn("More than one table affected by query
spec", tablesAffected);
      return
    }
    let onlyTable: string =
tablesAffected[Symbol.iterator]().next().value
    // validate that all of where clauses reference single
table
    for (let where of spec.where) {
      if (where.left.object == 'column') {
        if (where.left.table != onlyTable) {
```

```
          console.warn(`Where clause references second table
- ${where.left.table}, != ${onlyTable}`)
          return;
        }
      }
    }
    const __this = this;
    function $_registerListeners(events: ModelEventType[],
confirming: boolean) {
      // subscribe to relevant updates based on where clause
of spec
      // and update model values on change according to
select section of spec
      events.forEach(
        event => __this.subscribe(
          onlyTable,
          event as ModelEventType,
          spec.where.reduce(
            (acc, obj) => {
              const {and, or} = SQLWhereToCondition(obj,
__this.getModelClass().name, event as ModelEventType,
confirming);
              acc.or.push(...or);
              acc.and.push(...and);
              return acc;
            },
            {or: [], and: []}
          ),
          (model, payload: ModelUpdatePayload) => {
            return SQLRecomputeModelValues(fieldName, spec,
__this, payload, model, confirming);
          }
        )
```

```
      );
      return this;
    }
    // weird a$$ indentation for no reason
    $_registerListeners
    // delete event can't turn model from deviant to
conforming
    (['insert', 'update'], true)
    // insert event can't turn model from conforming to
deviant
    (['update', 'delete'], false)
    ;
  }
```

Однак у деяких випадках неможливо обчислити нове значення без додаткових запитів до сервера. Такі ситуації виникають, коли збережений стан представлення на клієнті не містить достатньої інформації про те, з яких рядків його було обчислено. Взяти, наприклад, `select count(distinct team_a_id) from game`. При створенні нового рядка у таблиці `game` на клієнті неможливо визначити, чи було його вже підраховано до загального рахунку чи ні. У подібних ситуаціях, коли бібліотека не в змозі зрозуміти або переобчислити значення, буде зроблено повторний запит до сервіса представлень з метою отримати оновлені дані.

Підтримка реактивності бібліотекою розрахована в першу чергу саме на Vue.js 3. У цій версії Vue покладається на систему рхоху схожим чином до клієнтської бібліотеки. Всі об’єкти, що використовуються для відображення компонента у Vue мають бути реактивними, що означає що вони обгорнуті у proxy об’єкт, створений самим Vue. Таким чином UI фреймворк знатиме, на які з об’єктів стану компонент покладається під час відображення, та у разі їх зміни він може заново відобразити компонент з використанням нових значень. На щастя, Vue надає API для роботи з реактивністю, що використовується

клієнтською бібліотекою для перетворення новостворених моделей на реактивні. Код перетворення фреймворком моделі на реактивну наведено у лістінгу 22. Код головного класу інтеграції бібліотеки та UI-фреймворка наведено у лістінгу 23.

Лістінг *22 Метод перетворення моделі на реактивну*

```
getReactiveModel(): this {
    if (!this.__reactive) {
      const modelMgr = this.getClass().__modelManager;
      if (!modelMgr) {
        throw new Error("getReactiveModel called before
model manager was set.");
      }
      this.__reactive = modelMgr.makeReactive("" +
this.uniqId, this.__myProxy as any);
    }
    return this.__reactive;
  }
```

Лістінг *23 Методи менеджера моделей для підтримки реактивності*

```
export class VueModelManager extends ModelManager {
  private readonly vueReactive: (x: any) => any;
  constructor(websocketUrl: string, graphQlUrl: string,
_vueReactive: (x: any) => any) {
    super(websocketUrl, graphQlUrl);
    this.vueReactive = _vueReactive;
  }
  makeReactive<T extends Model | Struct>(modelKey: string,
model: T): T {
    this.models[modelKey] = this.vueReactive(model) as any;
    return this.models[modelKey] as T;
  }
```

```
  reactivelySet(model: any, name: string, value: any, stage:
string) {
    if (Array.isArray(value)) {
      if (stage == 'post') {
        model[name] = value;
        model[name].push(null);
        model[name].pop();
      }
      return;
    }
    model.__reactive[name] = value;
  }
  getAuthKey() { return ""; }
}
```

За допомогою цього рішення, коли моделі оновлюють свої значення, зміни автоматично пропагуються до інтерфейсу, тим самим роблячи можливою наскрізну реактивність напряму від бази даних до інтерфейсу на екрані, без необхідності втручання зі сторони розробника.

Інтеграція бібліотеки у Vue відбувається швидко. Спочатку необхідно ініціалізувати менеджер моделей, передавши URL адреси сервісів представлень та підписок, та функцію реактивності Vue. Приклад ініціалізації менеджера моделей наведено у лістінгу 24.

Лістінг *24 Ініціалізація менеджера моделей*

```
export const modelMgr = new VueModelManager(
  "ws://localhost:9000/subscribe-to-updates",
  "http://localhost:8000/graphql/",
  reactive
);
```

Після цього у головному компоненті програми необхідно завантажити визначення моделей з сервіса представлень. Опційно можна запобігти відображенню сторінки до моменту коли моделі будуть завантажені, задля уникнення помилок. Завантаження моделей продемонстроване у лістінгу 25. Відтерміноване відображення додатку до моменту завантаження відображене у лістінгу 26.

Лістінг *25 Ініціація завантаження моделей*

```
export default {
   data() {
      return {
         isLoaded: false,
      }
   },
   created() {
      const self = this;
      modelMgr.loadModels().then(() => self.isLoaded = true);
   },
}
```

Лістінг *26 Відображення додатку після завершення завантаження*

```
<router-view v-if="isLoaded" />
<v-progress-linear v-else indeterminate></v-progress-
linear>
```

На цьому інтеграція бібліотеки завершена, за бажанням розробника можливо додати TypeScript визначення для автодоповнення в редакторі, як продемонстровано у лістінгу 27.

Лістінг *27 Визначення типів TypeScript*

```
export interface Player extends Model {
```

```
 id: number;
 uuid: string;
 username: string;
 total_kills: number;
}
export class GameTeam extends Model {
 id: number;
 players: Player[];
}
export interface PlayerKills extends Struct {
 player: Player;
 kills: number;
}
export interface Game extends Model {
 id: number;
 map: string;
 team_a: GameTeam;
 team_b: GameTeam;
 rounds: Round[];
 team_a_stats: PlayerKills[];
 team_b_stats: PlayerKills[];
}
export interface GamesView extends Model {
 games: PageType<Game>;
}
export interface PlayersView extends Model {
 players: PageType<Player>;
}
// @ts-ignore
export let models: {
 Player: ModelType<Player>,
 Game: ModelType<Game>,
 GameTeam: ModelType<GameTeam>,
```

```
 GamesView: ModelType<GamesView>,
 PlayersView: ModelType<PlayersView>,
} = modelMgr.modelTypes;
```

### 3.3 Розробка веб-додатку для демонстрації використання фреймворка

З метою демонстрації роботи фреймворку було розроблено невеликий веб-додаток на Vue.js. додаток містить сторінку блогу (див. Рис. 6), на якій можна побачити список доступних постів та коротку інформацію про них. На сторінці є пагінація для послідовного завантаження постів, у порядку їх створення.


Рисунок *6 — Сторінка блогу*

Частина компоненту, що містить сам скрипт, дуже невелика. Бібліотека самостійно вирішує питання запиту і парсингу даних, а також пагінації, що продемонстровано у лістінгу 28.

Лістінг *28 Скрипт компоненту сторінки блогу*

```
export default {
    data() {
        return {
            blogPostsView: models.BlogPostsView.createView(),
            pageNumber: 0,
        }
    },
    watch: {
        pageNumber(newVal) {
            this.blogPostsView.setFieldArg("posts", "page", newVal - 1);
        },
    }
}
```

При переході за посиланням на сторінку конкретного поста, можна побачити список коментарів, що так само як і все інше на сторінці оновлюється у реальному часі (див. Рис. 7).

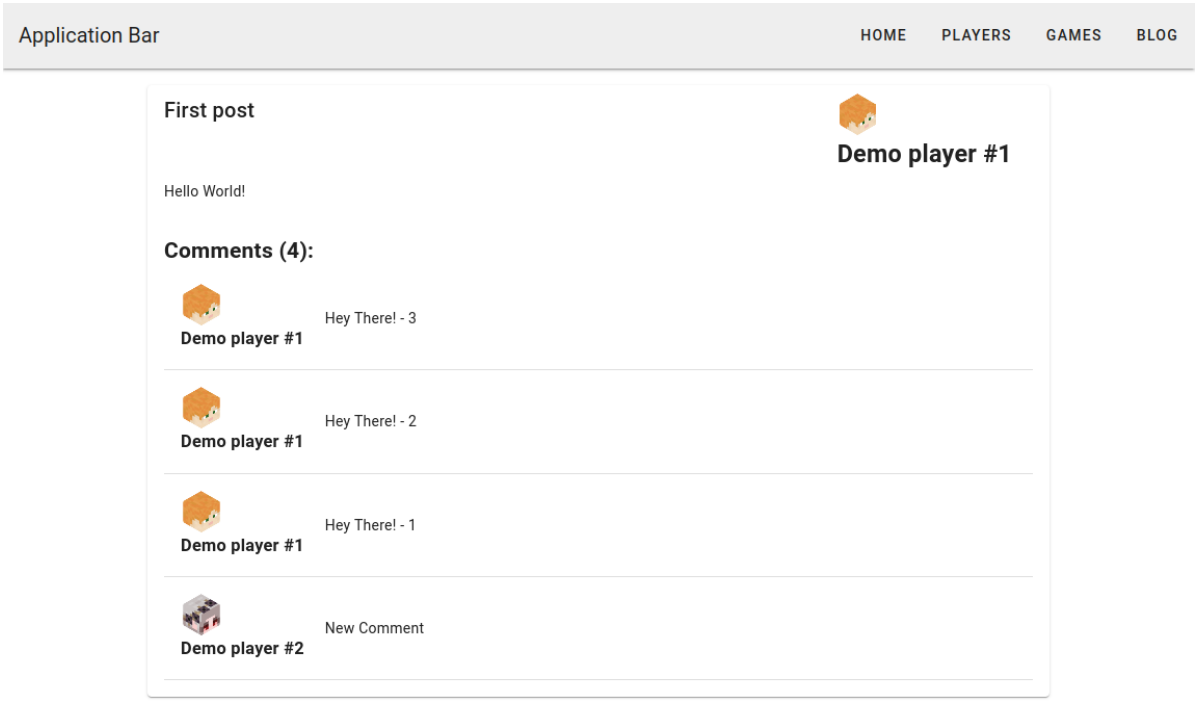


Рисунок *7 — Сторінка коментарів*

Скрипт компонента складається лише з єдиного виклику метода Find(), щоб отримати об’єкт моделі за ідентифікатором, що показано у лістінгу 29.

Лістінг *29 Скрипт сторінки поста*

```
export default {
  data() {
    return {
      blogPostView:
models.BlogPost.Find(this.$route.params.id as any),
    }
  },
}
```

Ідентично побудовані і всі інші сторінки, такі як список гравців (див. Рис. 8).

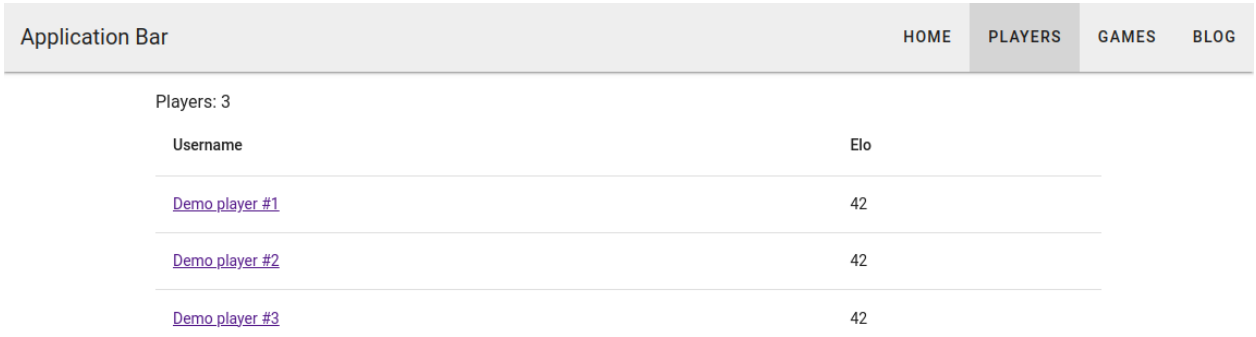


Рисунок *8 — Сторінка гравців*

На рисунках 9, 10  представлено сторінки гравця та ігор відповідно.


Рисунок *9 — Сторінка гравця*

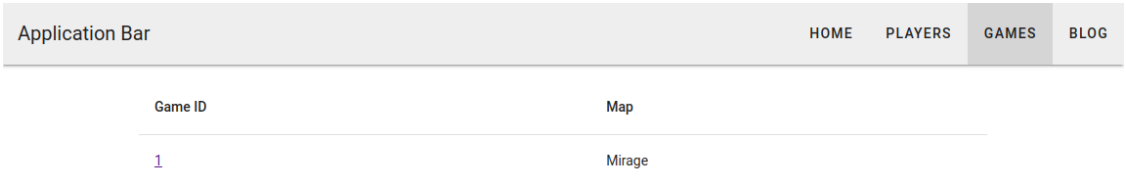


Рисунок *10 — Сторінка ігор*

Сторінка конкретної гри зі статистикою учасників наведена на рисунку 11.

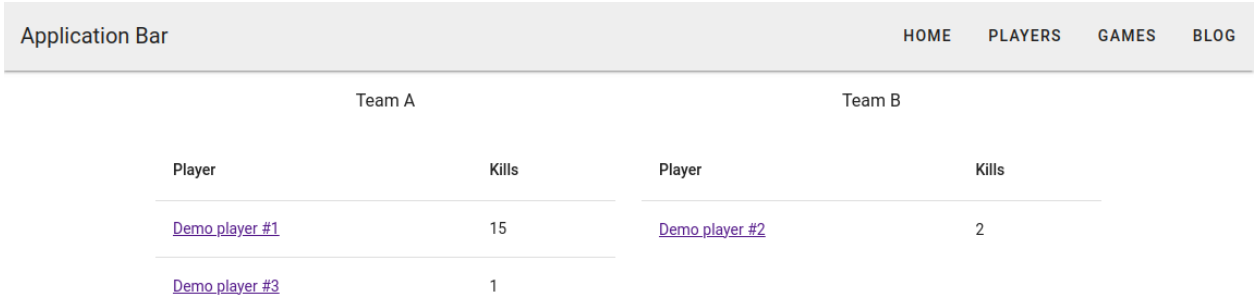


Рисунок *11 — Сторінка гри*

Всі дані, видимі на екрані синхронізуються з базою даних у реальному часі.

# ВИСНОВКИ

1. У ході виконання кваліфікаційної роботи бакалавра мною було вивчено існуючі методи синхронізації даних у реальному часі, основні принципи роботи баз даних реального часу та їх недоліки.
2. Розроблене рішення для синхронізації даних у реальному часі між сервером та клієнтами. Запропоноване рішення значно спрощує процес розробки програм та зменшує кількість шаблонного коду, надаючи розробнику можливість автоматично синхронізувати дані без необхідності втручання для визначення умов синхронізації або ручного переліку залежностей.
3. Запропоноване рішення дозволяє користуватися всіма перевагами реляційних баз даних, що зустрічається досить рідко серед аналогічних баз даних реального часу. І
4. Перевагою запропонованого рішення є можливість оптимізації деяких запитів шляхом їх виконання на стороні клієнта, запобігаючи додаткового навантаження на сервер.
5. Розроблено веб-додаток з метою демонстрації сценаріїв використання та можливостей створеного фреймворка, проаналізовано отриманий результат.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. The Comparison of New Web Communication Method WebSocket with Traditional Methods: веб-сайт. URL: https://dergipark.org.tr/tr/download/article-file/270677 (дата звернення 22.03.2024).
2. InvaliDB: Scalable Push-Based Real-Time Queries on Top of Pull-Based Databases (Extended): веб-сайт. URL: https://www.baqend.com/files/invalidb_vldb_2020.pdf (дата звернення 22.03.2024).
3. Документація Meteor : веб-сайт. URL: https://docs.meteor.com (дата звернення 22.03.2024).
4. Документація RethinkDB : веб-сайт. URL: https://rethinkdb.com/docs com (дата звернення 22.03.2024).
5. Документація Parse : веб-сайт. URL: https://parseplatform.org/parse-server/api/7.0.0/ (дата звернення 22.03.2024).
6. Документація Firebase : веб-сайт. URL: https://firebase.google.com/docs (дата звернення 22.03.2024).
7. Python’s standard documentation : веб-сайт. URL: https://www.python.org/doc/ (дата звернення 22.03.2024).
8. Специфікація GraphQL : веб-сайт. URL: https://spec.graphql.org/October2021/ (дата звернення 22.03.2024).
9. Специфікація ASGI : веб-сайт. URL: https://asgi.readthedocs.io/en/latest/specs/main.html (дата звернення 22.03.2024).
10. Введення до Vue.js : веб-сайт. URL: https://vuejs.org/guide/introduction.html (дата звернення 22.03.2024).
11. Документація RabbitMQ : веб-сайт. URL: https://www.rabbitmq.com/docs (дата звернення 22.03.2024).
12. Документкація PostgreSQL : веб-сайт. URL: https://www.postgresql.org/files/documentation/pdf/12/postgresql-12-A4.pdf (дата звернення 22.03.2024).

13. User Preference and Search Engine Latency : веб-сайт. URL: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/34439.pdf (дата звернення 22.03.2024).

14. The Three Musketeers By Alexandre Dumas : веб-сайт. URL: https://www.gutenberg.org/cache/epub/1257/pg1257-images.html (дата звернення 22.03.2024).

15. Advantages and challenges of nosql compared to sql databases : веб-сайт. URL: https://zbornik.efb.ues.rs.ba/dokumenta/Zbornik_radova_16-2022/02%20ADVANTAGES%20AND%20CHALLENGES%20OF%20NOSQL%20COMPARED%20TO%20SQL.pdf (дата звернення 22.03.2024).

16. WebSocket Adoption and the Landscape of the Real-Time Web : веб-сайт. URL: https://faculty.cc.gatech.edu/~mbailey/publications/www21_websocket.pdf (дата звернення 22.03.2024).

**Декларація**
**академічної доброчесності**
**здобувача ступеня вищої освіти ЗНУ**

Я, Одайський Володимир Вячеславович, студент 4 курсу, форми навчання денної, Інженерного навчально-наукового інституту, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти ipz20bd-206@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему « **Створення фреймворку для односпрямованої синхронізації даних між web-сервером і web-клієнтом в реальному часі»** відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

- згоден на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 14.06.2024 Одайський Володимир Вячеславович
(підпис) (прізвище та ініціали) (студент)

Дата 15.06.2024 _____ Міхайлуца Олена Миколаївна
(підпис) (прізвище та ініціали) (керівник)