

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**  
**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ**  
**імені ІГОРЯ СІКОРСЬКОГО»**  
**Факультет прикладної математики**  
**Кафедра системного програмування**  
**і спеціалізованих комп'ютерних систем**

«На правах рукопису»  
УДК 004.62, 004.65

До захисту допущено:  
Завідувач кафедри СПСКС

\_\_\_\_\_  
Віталій РОМАНКЕВИЧ  
(підпис) (ім'я, прізвище)  
“ ” \_\_\_\_\_ 2022р.

**Магістерська дисертація**  
**на здобуття ступеня магістра**  
**за освітньо-професійною програмою «Системне програмування та спеціалізовані**  
**комп'ютерні системи»**  
**зі спеціальності 123 Комп'ютерна інженерія**

**на тему: «Комп'ютерна система об'єктно-реляційного відображення даних»**

Виконав:  
студент II курсу, групи КВ-12мп  
Пушняк Дмитро Сергійович

Науковий керівник:  
доцент, с.н.с., к.т.н.,  
Боярінова Ю.Є.

Консультант з нормоконтролю:  
доцент, с.н.с., к.т.н.,  
Боярінова Ю.Є.

Рецензент:

)

Засвідчую, що у цій магістерській дисертації  
немає запозичень з праць інших авторів без  
відповідних посилань.

Студент \_\_\_\_\_  
(підпис)

Київ – 2022 року

**Національний технічний університет України**  
**«Київський політехнічний інститут імені Ігоря Сікорського»**  
**Факультет прикладної математики**  
**Кафедра системного програмування**  
**і спеціалізованих комп'ютерних систем**

Рівень вищої освіти – другий (магістерський)

за освітньо-професійною програмою

Спеціальність 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Системне програмування та спеціалізовані комп'ютерні системи»

ЗАТВЕРДЖУЮ

Завідувач кафедри СПіСКС

Віталій РОМАНКЕВИЧ

(підпис)

\_\_\_\_\_ 202\_р.

**ЗАВДАННЯ**  
**на магістерську дисертацію студенту**  
**Пушняку Дмитру Сергійовичу**

1. Тема дисертації «Комп'ютерна система об'єктно-реляційного відображення даних», науковий керівник дисертації доцент кафедри СПіСКС, к.т.н, Боярінова Ю.Є., затверджені наказом по університету від «25» 11.2022 р. №4317-С
2. Термін подання студентом дисертації «12 » грудня 2022 р.
3. Об'єкт дослідження: є взаємодія між об'єктно-орієнтованим середовищем (сервісом) та реляційною базою даних.
4. Вихідні дані: прототип комп'ютерної системи об'єктно-реляційного відображення даних, яка автоматично генерує міграції на основі внесених змін в моделі даних.
5. Перелік завдань:
  - 1) Проаналізувати існуючі комп'ютерні системи об'єктно-реляційної проєкції, особливості їх синтезу та принципи їх роботи.
  - 2) Розробити комп'ютерну систему об'єктно-реляційного відображення даних, яка автоматично генеруватиме міграції на основі внесених змін в моделі даних в джерельному коді.
6. Перелік ілюстративного матеріалу – презентація.

7. Перелік публікацій: тези до доповіді на XV конференції молодих вчених ПМК-2022 року за темою «Комп'ютерна система об'єктно-реляційного відображення із автоматичною генерацією міграцій на основі змін в моделях даних» (Київ, 16-18 листопада 2022 р.); тези доповіді на IX Міжнародній науково-технічній Internet-конференції «Сучасні методи, інформаційне, програмне та технічне забезпечення систем керування організаційно-технічними та технологічними комплексами» (Київ, 25 листопада 2022 р.), за темою «Комп'ютерна система об'єктно-реляційного відображення».
8. Дата видачі завдання: 6 жовтня 2021 року.

#### Календарний план

| № з/п | Назва етапів виконання магістерської дисертації         | Термін виконання етапів магістерської дисертації | Примітка |
|-------|---------------------------------------------------------|--------------------------------------------------|----------|
| 1.    | Вивчення літературних матеріалів за тематикою МД        | 1.10.2021                                        |          |
| 2.    | Створення та наступне узгодження технічного завдання МД | 4.10.2021                                        |          |
| 3.    | Аналіз існуючих рішень                                  | 10.10.2021                                       |          |
| 4.    | Підготовка матеріалів першого розділу МД                | 18.05.2022                                       |          |
| 5.    | Підготовка матеріалів другого розділу МД                | 11.06.2022                                       |          |
| 6.    | Розробка ПЗ                                             | 3.07.2022                                        |          |
| 7.    | Тестування ПЗ                                           | 10.10.2022                                       |          |
| 8.    | Документація розробленого ПЗ                            | 19.10.2022                                       |          |
| 9.    | Оформлення МД                                           | 20.11.2022                                       |          |
| 10.   | Попередній розгляд на кафедрі                           | 1.12.2022                                        |          |

Студент

\_\_\_\_\_

(підпис)

Дмитро ПУШНЯК

(ім'я, прізвище)

Науковий

керівник дисертації

\_\_\_\_\_

Юлія БОЯРІНОВА

## РЕФЕРАТ

**Актуальність теми** обумовлена тим, що велика кількість нових середніх та малих проєктів, в особливості стартапів, яким необхідні інтенсивні темпи розробки, під час роботи із реляційними базами даних зупиняють свій вибір на використанні систем об'єктно-реляційного відображення, які вирішують велику кількість проблем перетворення сутностей бази даних в об'єкти моделей, відповідно до об'єктно-орієнтованої парадигми, на якій побудований та якою маніпулює сервіс. Крім того, такі системи допомагають полегшити написання запитів до бази даних, даючи можливість писати динамічні запити, транзакції та міграції в об'єктно-орієнтованому стилі із використанням синтаксису тієї самої мови програмування, замість використання "сирого" SQL. Основна ідея полягає в тому, що такі системи потрібні для полегшення роботи розробникам та економії їх часу, адже, як відомо, час коштує грошей. Отже, подальше розширення можливостей, що допомагатимуть бізнесу економити ресурси, позбавляючи розробників потреби в низькорівневій архітектурі та даючи можливість замість цього зосередитись саме на бізнес-логіці, є актуальним.

**Об'єктом дослідження** є взаємодія між об'єктно-орієнтованим середовищем (сервісом) та реляційною базою даних.

**Предметом дослідження** є комп'ютерні системи об'єктно-реляційного відображення.

**Мета роботи:** створення прототипу комп'ютерної системи об'єктно-реляційного відображення із розширеним функціоналом для роботи із міграціями для мови програмування TypeScript.

**Наукова новизна** полягає в тому, що вперше було запропоновано новий функціонал системи для автоматичної обробки внесення змін в моделі, які співвідносяться із таблицею в базі даних та генерації міграцій.

**Практична цінність** отриманих в роботі результатів полягає в тому, що розроблена комп'ютерна система окрім своїх типових загальних цілей, призначена для полегшення життя розробника під час роботи із міграціями, надаючи можливість більш швидко вносити зміни в існуючі в коді моделі даних

та позбавити потреби витратити зайвий час на створення міграцій після змін вручну.

**Апробація роботи.** Основні ідеї та результати роботи обговорювались на XV науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2022 (Київ, 16-18 листопада 2022 р.). Опублікована стаття на конференції «IX Міжнародна науково-технічна Internet-конференція «Сучасні методи, інформаційне, програмне та технічне забезпечення систем керування організаційно-технічними та технологічними комплексами» (Київ, 25 листопада 2022 р.).

**Структура та обсяг роботи.** Магістерська дисертація складається з вступу, чотирьох розділів та висновків.

*У вступі* подано загальну характеристику роботи, зроблено оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі досліджень, показано наукову новизну отриманих результатів і практичну цінність роботи.

*У першому розділі* описано контекст, сутність та принцип роботи систем об'єктно-реляційного відображення даних в об'єктно-орієнтованому середовищі виконання.

*У другому розділі* проаналізовано доступний інструментарій і на основі цього обрано конкретні інструменти для подальшої імплементації програмного продукту.

*У третьому розділі* описано реалізований програмний продукт в деталях, описано його архітектуру та принципи роботи, також описано процес тестування розробленої комп'ютерної системи та порівняння вихідних результатів з очікуваними.

*У висновках* підсумовані результати проведеної роботи.

Робота представлена на 74 аркушах, містить посилання на список використаних літературних джерел.

**Ключові слова:** об'єктно-реляційне відображення даних, ORM, PostgreSQL, бази даних.

## ABSTRACT

**The relevance of the topic** is due to the fact that a large number of new medium and small projects, especially startups that need an intensive pace of development, when working with relational databases, opt for the use of object-relational mapping systems that solve a large number of problems of converting database entities into model objects, in accordance with the object-oriented paradigm on which the service is built and manipulated. In addition, such systems help to facilitate the writing of database queries by making it possible to write dynamic queries, transactions, and migrations in an object-oriented style using the syntax of the same programming language, instead of using “raw” SQL. The basic idea is that such systems are needed to facilitate the work of developers and save their time, because, as we know, time costs money. Therefore, further expansion of capabilities that will help businesses save resources, depriving developers of the need for low-level architecture and allowing them to focus on business logic instead, is relevant.

**The object of research** is the interaction between object-oriented environment (service) and relational database.

**The subject of research** is computer systems of object-relational mapping.

**The goal of the work:** creation of a prototype of a computer system of object-relational mapping with advanced functionality for working with migrations for the TypeScript programming language.

**The scientific novelty** lies in the fact that for the first time a new functionality of the system was proposed to automatically process changes to models that correlate with a table in the database and generate migrations.

**The practical value** of the results obtained in the work is that the developed computer system, in addition to its typical general purposes, is designed to facilitate the life of the developer when working with migrations, providing the ability to more quickly make changes to existing data models in the code and eliminate the need to spend extra time creating migrations after changes manually.

**Approbation of the work.** The main ideas and results of the work were discussed at the XV scientific conference of undergraduate and graduate students “Applied

Mathematics and Computing” PMC-2022 (Kyiv, November 16-18, 2022). The article was published at the conference “IX International Scientific and Technical Internet-Conference “Modern methods, information, software and technical support of control systems for organizational, technical and technological complexes” (Kyiv, November 25, 2022).

**Structure and scope of work.** The master's thesis consists of an introduction, four chapters and conclusions.

In the introduction, the general characteristics of the work are given, the current state of the problem is assessed, the relevance of the research direction is substantiated, the aim and objectives of the research are formulated, the scientific novelty of the results and the practical value of the work are shown.

*The first section* describes the context, essence, and principle of operation of object-relational data display systems in an object-oriented runtime environment.

In *the second section*, the available tools are analyzed and based on this, specific tools are selected for further implementation of the software product.

*The third section* describes the implemented software product in detail, describes its architecture and operating principles and the testing of the developed computer system and comparison of the output results with the expected.

*The conclusion* summarizes the results of the work.

The work is presented on 74 pages, contains references to the list of sources that were used.

**Keywords:** object-relational data mapping, ORM, PostgreSQL, databases.

## ЗМІСТ

|                                                                                                              |    |
|--------------------------------------------------------------------------------------------------------------|----|
| ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ                                                               | 4  |
| ВСТУП                                                                                                        | 5  |
| 1. АНАЛІЗ ПРИНЦИПУ РОБОТИ СИСТЕМ ОБ'ЄКТНО-РЕЛЯЦІЙНОГО<br>ВІДОБРАЖЕННЯ В ОБ'ЄКТНО-ОРІЄНТОВАНОМУ<br>СЕРЕДОВИЩІ | 9  |
| 1.1.Об'єктно-орієнтоване програмування                                                                       | 9  |
| 1.2.Фундаментальні принципи об'єктно-орієнтованого програмування                                             | 10 |
| 1.2.1.Наслідування                                                                                           | 10 |
| 1.2.2.Інкапсуляція                                                                                           | 11 |
| 1.2.3.Поліморфізм                                                                                            | 12 |
| 1.2.3.1.Ad-hoc поліморфізм                                                                                   | 12 |
| 1.2.3.2.Параметричний поліморфізм                                                                            | 13 |
| 1.2.3.3.Поліморфізм підтипів                                                                                 | 14 |
| 1.3.SQL                                                                                                      | 14 |
| 1.4.Реляційні бази даних                                                                                     | 17 |
| 1.4.1.Можливості сучасних СУБД                                                                               | 17 |
| 1.4.2.Використання реляційних баз даних в об'єктно-орієнтованих<br>програмах                                 | 19 |
| 1.4.3.Драйвери баз даних та API                                                                              | 20 |
| 1.4.4.Невідповідність імпедансу                                                                              | 21 |
| 1.4.5.Зв'язки між таблицями                                                                                  | 22 |
| 1.4.6.Типи даних                                                                                             | 23 |

|                                                                                              |    |
|----------------------------------------------------------------------------------------------|----|
| 1.5.Об'єктно-реляційне відображення (проекція)                                               | 25 |
| 1.6.Принцип роботи систем об'єктно-реляційного відображення                                  | 28 |
| 1.7.Існуючі системи об'єктно-реляційного відображення для популярних мов програмування       | 29 |
| 1.8.Порівняння існуючих рішень для мови програмування TypeScript                             | 30 |
| ВИСНОВКИ ДО ПЕРШОГО РОЗДІЛУ                                                                  | 35 |
| 2. ОПИС ОБРАНОГО ІНСТРУМЕНТАРІЮ ІМПЛЕМЕНТАЦІЇ СИСТЕМИ ОБ'ЄКТНО-РЕЛЯЦІЙНОГО ВІДОБРАЖЕННЯ      | 37 |
| 2.1.Мова програмування для імплементції обраного програмного забезпечення                    | 37 |
| 2.2.Опис інструментів для обраної мови програмування                                         | 44 |
| 2.2.1.Lodash                                                                                 | 44 |
| 2.2.2.node-postgres                                                                          | 44 |
| 2.2.3.Середовище розробки                                                                    | 47 |
| ВИСНОВКИ ДО ДРУГОГО РОЗДІЛУ                                                                  | 51 |
| 3. СТРУКТУРА, ОПИС ТА ТЕСТУВАННЯ РЕАЛІЗОВАНОЇ КОМП'ЮТЕРНОЇ СИСТЕМИ                           | 52 |
| 3.1.Архітектура розробленого програмного забезпечення                                        | 52 |
| 3.2.Принцип роботи комп'ютерної системи                                                      | 55 |
| 3.3.Можливості розробленої системи                                                           | 56 |
| 3.4.Технічні характеристики пристрою тестування                                              | 57 |
| 3.5.Тестування роботи створеної комп'ютерної системи та аналіз надійності отриманих міграцій | 57 |

|                                                                            |    |
|----------------------------------------------------------------------------|----|
| 3.6.Порівняння результатів комп'ютерної системи з очікуваними результатами | 6  |
| 8                                                                          |    |
| ВИСНОВОК ДО ТРЕТЬОГО РОЗДІЛУ                                               | 73 |
| ВИСНОВКИ                                                                   | 74 |
| СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ                                             | 76 |

## ДОДАТКИ

**Додаток А. Лістинг програми**

**Додаток Б. Презентація магістерської дисертації**

**Додаток В. Наукові публікації**

## ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

Міграція — Міграцією називається інструмент керування інкрементними, зворотними змінами схеми реляційної бази даних. Міграція схеми виконується над базою даних коли необхідно оновити стан схеми бази даних до якоїсь новішої, або повернути до старішої версії;

ООП — об'єктно-орієнтоване програмування;

СУБД — Система управління базами даних;

OODBMS (англ. Object-Oriented Database Management System) — об'єктно-орієнтована система управління базами даних;

ORM (англ. Object-Relational Mapping) — процес об'єктно-реляційного відображення даних;

TypeScript — мова програмування, яка являє собою надбудову над JavaScript та надає їй статичну типізацію на етапі компіляції.

## ВСТУП

На сьогоднішній день об'єктно-орієнтоване програмування вже є загальноприйнятим підходом для розробки бізнес-додатків. Об'єктно-орієнтована розробка надає більш придатні методи та засоби для моделювання об'єктів реального світу, ніж, наприклад, парадигма структурного програмування. Мова йде про такі об'єкти, як клас, який є одиницею визначення даних і поведінки, або об'єкт, який є екземпляром класу, що виконується під час виконання. Якщо додаток розробляється на об'єктно-орієнтованій мові, ймовірно, буде використано інкапсуляцію як засіб для приховування даних і поведінки (методів). Ще однією дуже корисною концепцією об'єктно-орієнтованого програмування є концепція наслідування. За допомогою наслідування можна розширити існуючі класи новими даними та/або поведінкою. У спеціалізованому підкласі можна використовувати всі наслідувані дані та поведінку, а також можемо перевизначати наслідувану поведінку. Скажімо, у нас є клас А, і класи В і С, наслідувані від класу А. Концепція абстракції дозволяє нам працювати з екземплярами В і С так, як якщо б вони були екземплярами А. Дуже потужною особливістю є поліморфізм, коли поведінка об'єкта залежить від його фактичного типу. Широко розповсюдженою мовою для написання програм в об'єктно-орієнтованому стилі сьогодні є Java з її фреймворками, технологіями, API та специфікаціями. Java стала чимось на зразок мейнстріму за час своєї більш ніж десятирічної доступності для спільноти. Іншою платформою для розробки об'єктно-орієнтованих додатків є платформа .NET від Microsoft. Ця платформа молодша за Java і не настільки поширена, як Java. Тим не менш, вона користується популярністю серед багатьох розробників. Якщо немає особливих вимог до продукту, то вибір буде зроблено на користь однієї з вищезгаданих технологій, якщо мова йде про великі корпоративні додатки, або невеликі бізнес-додатки, але в яких надважливою є постійна робота із великими числами. Раніше рішення на основі Smalltalk або C++ були традиційними, якщо говорити про об'єктно-орієнтовану розробку. Також дуже популярним на сьогодні є TypeScript, який є розширює можливості JavaScript, додаючи можливість

типізації, яка дійсна на момент компіляції й роблячи його більш привабливим в якості основної мови для написання серверу із бізнес-імплементацією у певних ситуаціях.

Дані, з якими працює бізнес-додаток, як правило, повинні бути персистентними. Персистентність даних/об'єктів зазвичай забезпечується системами баз даних. Оскільки ми працюємо з об'єктами, простим рішенням було б зберігати об'єкти безпосередньо в об'єктно-орієнтованій базі даних (OODBMS). Багато людей повірили в цю ідею в 90-х роках. Постачальники OODBMS обіцяли, що це стане мейнстрімом для зберігання об'єктів. Серед основних причин чому об'єктно-орієнтовані бази не стали мейнстрімом знаходяться гетерогенні середовища. Об'єктно-орієнтовані середовища розробки та розгортання були абсолютно різними технологічно та концептуально. Ці середовища базувалися, в першу чергу, на таких мовах, як C++, Smalltalk або Java. Деякі фахівці навіть не вважають C++ об'єктно-орієнтованою мовою. На противагу цьому, Smalltalk вважається повністю об'єктно-орієнтованим середовищем з повною підтримкою рефлексії. Java базується на тих же принципах, що і Smalltalk, але з традиційним синтаксисом, схожим на C++. У 90-х роках були великі очікування щодо переходу на об'єктно-орієнтовані бази даних. Мова йде про Enterprise-додатки великих банків, страхових компаній тощо. В той час існувало багато застарілих систем баз даних, заснованих, наприклад, на мережевій моделі, визначеній CODASYL з COBOL у якості мови хоста, або вони зберігали свої дані в реляційних базах даних, які також були вже широко розповсюджені в той час. Для великих компаній було дуже складно перейти на цю нову платформу, оскільки всі їхні важливі бізнес-дані зберігалися в цих застарілих системах. Замість того, щоб об'єктно-орієнтовані бази даних стали мейнстрімом, реляційні бази даних залишилися в цій ролі. Наприкінці 60-х років Едгар Кодд опублікував свою першу статтю про реляційні бази даних [1]. З цього часу реляційні бази даних зазнали значного розвитку. Реляційні бази даних строго орієнтовані на дані. Навіть відношення між даними є даними. Стандарт SQL як мова запитів до реляційних баз даних також розвивався, і багато постачальників розширюють його новими можливостями. Діаграми "сутність-

зв'язок" є корисним інструментом для високорівневого моделювання реального світу для потреб проєктування баз даних. Діаграма "сутність-зв'язок" складається з сутностей, їх атрибутів та зв'язків. Сутність — це об'єкт реального світу, наприклад, автомобіль. Атрибутом автомобіля є, наприклад, `engineType`. Ми також можемо визначити зв'язки між сутностями. Ці зв'язки можуть бути кратності 1:1, 1:n, m:n. Також можлива наявність n-арних зв'язків, де  $n > 2$ . При проєктуванні логічної моделі даних (таблиць) ми моделюємо ці зв'язки як атрибути відповідних таблиць.

Для повноти картини згадаємо про об'єктно-реляційні бази даних. Це реляційні бази даних, які дозволяють застосовувати деякі принципи ООП. Наприклад, в Oracle, починаючи з 8-ї версії, з'явилася можливість визначати абстрактні типи даних та вказівники. Більшість сучасних серверів баз даних вважаються об'єктно-реляційними — наприклад, DB2, PostgreSQL та вже згаданий Oracle [2]. Використання реляційних баз даних для зберігання даних є типовим для сьогоденної ситуації в програмній інженерії. З іншого боку, більша частина бізнес-додатків розробляються в об'єктно-орієнтованих середовищах. Нівелювання цієї невідповідності між реляційним сховищем даних та об'єктно-орієнтованим програмним забезпеченням необхідне для забезпечення персистентності об'єктів. Зауважимо, що ми хочемо визначити цю персистентність для повнофункціональних об'єктів. Ми не хочемо визначати модель, в якій ми можемо працювати з об'єктами в якомусь обмеженому вигляді. Коли потрібно зробити об'єкти персистентними, ми повинні визначити процедури, як відобразити об'єкти в строго орієнтовані на дані структури реляційних баз даних. Мова йде про шар в бізнес-додатку, який відображає об'єкти на базову реляційну базу даних і дозволяє ефективно зберігати та отримувати їх. У дипломній роботі буде більш детально розглянуто проблеми та методи, пов'язані з об'єктно-реляційним відображенням. Також буде описано деякі загальноприйняті інструменти/фреймворки для об'єктно-реляційного відображення та проведемо простий бенчмарк цих інструментів. На основі даних фреймворків буде імплементовано нову систему, з функціоналом, який допоможе більш швидко писати базові міграції в процесі розробки.

# 1. АНАЛІЗ ПРИНЦИПУ РОБОТИ СИСТЕМ ОБ'ЄКТНО-РЕЛЯЦІЙНОГО ВІДОБРАЖЕННЯ В ОБ'ЄКТНО-ОРІЄНТОВАНОМУ СЕРЕДОВИЩІ

## 1.1. Об'єктно-орієнтоване програмування

Об'єктно-орієнтоване програмування (ООП) сьогодні є домінуючою парадигмою в розробці основного програмного забезпечення, хоча воно мало вельми невпевнений початок. Simula представила багато з основ ООП в 1960-х роках [3], але тільки протягом останніх кількох десятиліть ООП стало популярним в мейнстрімі. Однак воно дійсно стало популярним; важко переоцінити поширеність ООП в промислових, наукових та дидактичних додатках. ООП виражає себе за допомогою аналогій з реальним світом. Ідея полягає в побудові програмних абстракцій, які моделюють об'єкти реального світу. Як програмна абстракція, об'єкт — це дискретна сутність зі станом, поведінкою та ідентичністю. В ідеалі він відповідає чітко визначеному, значущому "бізнес-об'єкту" в проблемній області, наприклад, конкретному клієнту, якого компанія бажає відстежувати. Об'єкти можуть взаємодіяти один з одним, обмінюватися повідомленнями та обробляти дані. В об'єктно-орієнтованій мові об'єкти групуються в класи, які інкапсулюють функціональність, спільну для багатьох об'єктів. Наприклад, багато об'єктів клієнтів можуть бути членами одного класу "клієнт", який представляє сукупність всіх клієнтів. Клас клієнта визначає структуру і поведінку, загальну для всіх клієнтів. Наприклад, він записує, що всі клієнти мають властивість "ім'я", і всі клієнти мають поведінку "купівля" товару. Більш явно, клас визначає стан (тобто, абстрактну структуру даних, яку поділяють всі екземпляри класу) і поведінку. Абстрактні члени даних, визначені класом, називаються його атрибутами, властивостями або просто членами даних. Поведінка, яку він визначає, називається його методами [4].

## 1.2. Фундаментальні принципи об'єктно-орієнтованого програмування

### 1.2.1. Наслідування

Одним з найважливіших прийомів ООП є наслідування, при якому програміст створює нові класи шляхом розширення існуючих. Похідні класи успадковують поведінку і структуру базового класу і додають до цього функціональність [5]. Менш поширеною назвою наслідування є узагальнення, оскільки базові класи можна розглядати як узагальнення їх похідних класів. На практиці архітектори програмного забезпечення намагаються обмежити глибину дерев успадкування на користь зрозумілості коду. Деякі мови, такі як C++, Python та Common Lisp, дозволяють множинне наслідування, в якому класи можуть успадковувати функції від більш ніж одного базового класу. У такій мові можна було б реалізувати більш складну ієрархію успадкування. Деякі інші мови, такі як Smalltalk та Modula-3, своєю чергою, виключають множинне наслідування. Багато мов використовують якийсь проміжний підхід. Наприклад, Java, .NET Framework та ObjectiveC мають як класи, так і інтерфейси (в Objective-C вони називаються протоколами). Інтерфейс визначає набір методів, які повинні реалізувати спадкоємці, але повністю опускає їх реалізацію, залишаючи специфікацію реалізації повністю похідним класам. Ці три середовища підтримують множинне наслідування для інтерфейсів, але тільки одиночне наслідування для класів [6]. Приклад простого наслідування зображено на рис. 1.1.

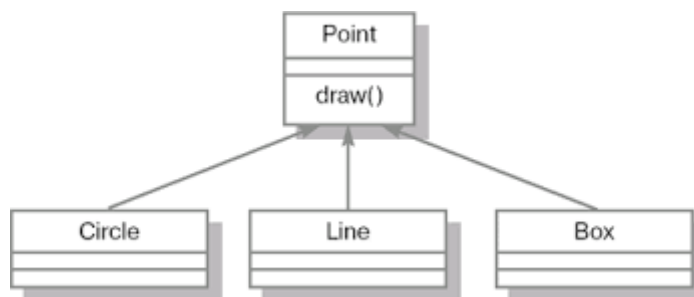


Рисунок 1.1 – Приклад принципу наслідування в ООП.

### 1.2.2. Інкапсуляція

Ще однією часто згадуваною властивістю об'єктно-орієнтованих систем є інкапсуляція. Цей термін може мати будь-яке з декількох різних значень. Є три основні концепції інкапсуляції, які існують в літературі. Тут ми розглянемо кожен концепцію по черзі та ступінь її застосовності до існуючих об'єктно-орієнтованих мов.

Інкапсуляція — це процес, який використовується для пакування даних разом з функціями, що діють над ними. В контексті ООП під інкапсуляцією розуміється властивість, що дані та методи упаковуються разом в класи [7]. Таким чином, всі мови ООП підтримують інкапсуляцію, оскільки всі мови ООП мають класи, що містять як дані, так і методи. Більш строго, однак, можна сказати, що для того, щоб об'єкт можна було назвати інкапсульованим, всі операції над його станом повинні бути частиною його інтерфейсу. У цьому строгому розумінні лише деякі мови забезпечують або навіть допускають інкапсуляцію. Наприклад, у мові JavaScript для приведення плаваючого числа  $x$  до найближчого нижнього цілого числа використовується `Math.floor(x)`. Суворі інкапсуляція вимагала б, щоб це було у форматі `x.floor()`.

- Інкапсуляція означає приховування деталей реалізації об'єктів так, щоб клієнти отримували доступ до об'єктів тільки через їх загальнодоступні інтерфейси;
- Інкапсуляція вигідна тим, що звільняє клієнта від необхідності розбиратись з реалізацією класів. Якщо реалізація класу змінюється, клієнти можуть без ризику проігнорувати цю зміну, оскільки загальнодоступний інтерфейс і логічна поведінка класу залишаються незмінними. Багато об'єктно-орієнтованих мов здійснюють інкапсуляцію двома способами. По-перше, клієнти не мають можливості вивчити код реалізації методів класів, тільки їх сигнатури, тому реалізація методів прихована від клієнтів. По-друге, у багатьох

об'єктно-орієнтованих мовах є функція доступу, яка дозволяє оголошувати члени класу (включаючи як атрибути, так і методи) не загальнодоступними, а приватними, роблячи їх доступними тільки для самого класу, а не для клієнтів. Часто існує також специфікатор доступу `protected`, який робить члени доступними тільки для похідних класів, а іноді існують інші специфікатори доступу, такі як `friend access` в C++, який надає доступ тільки оголошеним "друзям" класу, і `internal access` в C#, який надає доступ тільки клієнтам в тій же збірці. Через переваги, які надає інкапсуляція, архітектори програмного забезпечення, як правило, роблять більшість членів даних закритими та відкривають тільки ті методи, які необхідні клієнтам для використання класу.

Інкапсуляція є поєднанням цих двох ідей. Кажучи точніше, інкапсуляція передбачає упакування даних з функціями, що діють на них, в дискретну одиницю, деталі реалізації якої приховані від зовнішнього світу.

### 1.2.3. Поліморфізм

Ще одним поняттям, яке часто згадується у визначеннях ООП, є поліморфізм, який дозволяє використовувати один інтерфейс з різними типами даних. Існує декілька типів поліморфізму, які зустрічаються в об'єктно-орієнтованих (та інших) мовах.

#### 1.2.3.1. Ad-hoc поліморфізм

Це найбільш обмежена форма поліморфізму. Вона дозволяє параметризувати функцію (або клас тощо) на невеликому (явно перерахованому) наборі типів. Найяскравішим прикладом є перевантаження методів [8]. Клас `Math`

в Java, наприклад, має декілька методів `abs`. Усі методи `abs` отримують один аргумент, але параметри мають різні типи. Є один метод `abs`, який отримує тип `int` і повертає його абсолютне значення, також як `int`; інший отримує тип `double` і повертає його абсолютне значення як `double`; і так далі. Коли клієнт викликає `Math.abs`, Java викликає відповідне перевантаження на основі типу параметра. Якби Java не підтримувала спеціальний поліморфізм у вигляді перевантажень, ця функціональність вимагала б різнойменних методів, по одному для кожного підтримуваного типу. Перевантаження можуть залежати не тільки від типу параметра, але й від кількості параметрів. У середовищі .NET Framework клас `Int32` має метод `ToString`, який перетворює ціле число в його рядкове представлення. Цей метод має чотири перевантаження з різною кількістю та типами параметрів. Іншою формою спеціального поліморфізму іноді вважається примус до типу. Більшість мов підтримують деяке неявне приведення типів, а в деяких мовах, таких як, наприклад, C++, розробники класів можуть визначати нові примуси. Тоді функція, яка отримує аргумент типу `T`, може приймати аргументи будь-якого типу, які можуть бути неявно приведені до `T`. Це найслабший вид поліморфізму, оскільки аргументи фактично перетворюються до цільового типу до того, як над ними виконується функція.

#### 1.2.3.2. Параметричний поліморфізм

Це набагато потужніша форма поліморфізму, в якій загальний код пишеться для роботи з різними типами. Шаблони C++ є гарним прикладом. Програмісти на C++ можуть писати код на зразок того що зображено на рис. 1.2 Ця функція приймає два аргументи будь-якого типу, для яких визначений оператор `>`, включаючи примітивні типи та класи, що мають перевантажений оператор, і повертає той, який є більшим. Загальні типи, подібні до шаблонів C++, були додані до мови Java у 2004 році [9] та до системи загальних типів .NET у 2005 році [10]. Параметричний поліморфізм жодним чином не є специфічним для об'єктно-орієнтованої парадигми (як і поліморфізм *ad hoc*). Такі мови

програмування як-от Haskell або Ada та багато інших не об'єктно-орієнтованих мов підтримують параметричний поліморфізм, і фактично ML, функціональна мова, була мовою, яка запровадила цю можливість.

```
template <class anyType>
anyType GetMax (anyType a, anyType b) {
    return (a>b ? a : b);
}
```

Рисунок 1.2 – Приклад шаблону в мові C++

### 1.2.3.3. Поліморфізм підтипів

Цей вид поліморфізму характерний для мов, які підтримують підтипи, особливо об'єктно-орієнтованих мов, які можуть підтримувати поліморфізм підтипів через успадкування. Він пов'язаний з примусом типу і полягає в тому, що в багатьох мовах функції, які приймають параметр типу T, також приймають екземпляр будь-якого підтипу T. Це означає, що функції, які можуть працювати з типом T, гарантовано підтримують і підтипи T. Однак, поліморфізм підтипів не обмежується цим. Перевизначення методів робить поліморфізм підтипів потужним інструментом. У мовах, що підтримують перевизначення методів, підтипи можуть надавати власні реалізації інтерфейсів у своїх базових типах.

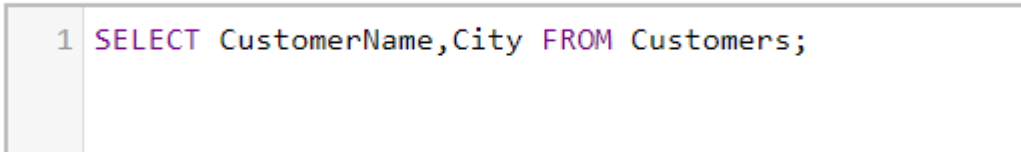
## 1.3. SQL

В теорії можна було б розробити багато мов, які могли б виступати в якості інструкцій для запитів до реляційних баз даних. На практиці, на сьогодні SQL є єдиною мовою, яка заслуговує на увагу. SQL використовується як мова запитів практично у всіх реляційних СУБД на сьогоднішній день. Мова SQL, яка на

світанні свого існування називалася SEQUEL - "Structured English Query Language", зародилася в 1970-х роках для підтримки СУБД System R, розробленої дослідницькою групою IBM. Після появи інших реалізацій SQL, включаючи СУБД Oracle, мова SQL була стандартизована Американським національним інститутом стандартів у 1986 році та Міжнародною організацією зі стандартизації у 1987 році. Ревізії з'явилися в 1989, 1992, 1999 і 2003 роках, і стандарти SQL часто називають такими іменами, як SQL-86, SQL-89 і так далі [11]. Всупереч стандартизації, значні відмінності між реалізаціями SQL залишаються. Як наслідок, сценарії SQL, як правило, не переносяться між різними типами реляційних СУБД. SQL (як це було задумано спочатку) є декларативною мовою програмування, а не імперативною мовою, як, наприклад, мова програмування загального призначення C. SQL включає в себе команди для пошуку даних, маніпулювання даними та визначення даних, а також адміністративні команди, що підтримують такі функції, як транзакції бази даних, дозволи та тригери бази даних.

Розглянемо деякі приклади роботи із базою даних за допомогою SQL запитів [12].

Запит формату зображеному на рис 1.3. вибере у кожного запису в таблиці Customers колонки CustomerName та City.



```
1 SELECT CustomerName, City FROM Customers;
```

Рисунок 1.3 – Select запит SQL

Запит формату зображеному на рис 1.4 додасть до таблиці Customers новий запис, де кожній колонці зі списку після Customers буде присвоєне значення зі списку VALUES відповідно.

```

1 INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)
2 VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');|

```

Рисунок 1.4 – Insert запит SQL

Запит формату зображеному на рис 1.5 оновить значення колонок ContactName та City запису в таблиці Customers, де CustomerID дорівнює 1.

```

1 UPDATE Customers
2 SET ContactName='Alfred Schmidt', City='Frankfurt'
3 WHERE CustomerID=1;

```

Рисунок 1.5 – Update запит SQL

Запит формату зображеному на рис 1.6. видалить усі записи із таблиці Customers, де CustomerName дорівнює Alfreds Futterkiste.

```

1 DELETE FROM Customers WHERE CustomerName=' Alfreds Futterkiste';

```

Рисунок 1.6 – Delete запит SQL

Це — приклади ключових операторів, які використовуються для побудування більшості запитів. Крім них є ще велика кількість допоміжних операторів для фільтрування, маніпуляцій, підзапитів, сортування вибірки тощо. Насправді один і той самий складний запит можна написати різними шляхами, але не кожна його варіація буде працювати однаково з точки зору ефективності, адже хоч і база даних під капотом виконує певні оптимізації, в цілому підхід не змінюється. Саме тому, не маючи багато років практичного досвіду, іноді буває дуже складно з першого разу написати оптимальний варіант запиту під конкретні потреби.

Розробники можуть використовувати “сирий” код SQL для написання прямого інтерфейсу між додатком і базою даних. Більшість реляційних баз даних підтримують SQL для побудови інтерфейсів даних і додатків. Він стабільний, і оскільки SQL використовується з 1970-х років, він добре задокументований і підтримується. Програмісти зберігають великий контроль над своїм інтерфейсом даних за допомогою SQL. Він вимагає багато роботи, але він зазвичай більш гнучкий і детальний, ніж абстракція системи ORM.

Використання “сирого” SQL також має свої недоліки. Наприклад, розробник несе відповідальність за безпеку і збереження коду бази даних. SQL-ін'єкція - це проблема, коли введення даних користувачем може вплинути на стан даних, викликаючи проблеми з додатком і цілісністю даних. Системи ORM, своєю чергою, “дезінфікують” код, що дозволяє уникнути цих проблем.

Конструктори запитів додають рівень абстракції над необробленим SQL, не маскуючи всі основні деталі. Конструктори формалізують шаблони запитів і додають методи або функції, які додають елементи екранування для полегшення інтеграції додатків. Вони додають шар шаблонів, щоб допомогти розробникам зрозуміти структуру бази даних в рамках однієї мови програмування. Конструктори шаблонів все ще вимагають від розробників розуміння структури бази даних, що вимагає від них знання мови SQL на базовому рівні.

#### 1.4. Реляційні бази даних

##### 1.4.1. Можливості сучасних СУБД

Маніпулювання даними, визначення даних та управління даними залишаються центральними аспектами в SQL, але SQL пройшов довгий шлях з моменту його стандартизації в 1986 році. Сьогоднішні реалізації SQL включають функції, які роблять його потужним і універсальним інструментом для вирішення різноманітних завдань обробки даних. Зокрема, мова SQL набула об'єктно-

подібних властивостей, які наближають її до об'єктно-орієнтованої парадигми програмування. Ці особливості підвели основні СУБД до межі об'єктно-реляційних СУБД, що поєднують в собі кращі риси реляційної моделі та об'єктно-орієнтованої парадигми. SQL:1999, зокрема, запровадив вражаючий набір об'єктно-орієнтованих можливостей [13]. Конкретні СУБД також додали багато нових можливостей до своїх реалізацій [14], [15]. Однією з важливих змін стало додавання традиційних функцій мови програмування, таких як присвоєння змінних та структур управління потоком, таких як умови та цикли. Transact-SQL (власне розширення SQL, що використовується в продуктах Microsoft та Sybase) включає умовні оператори IF-ELSE; цикли WHILE; а також оператори GOTO, CONTINUE тощо. PL/SQL (розширення SQL від Oracle) аналогічно включає умовні оператори IF-THEN-ELSE, цикли WHILE і FOR, оператори CASE, а також оператор GOTO. Ці розширення, поряд з багатьма іншими реалізаціями, включають також підтримку оголошень і присвоювання змінних та інші можливості мов програмування. Ще однією особливістю сучасних реалізацій SQL є курсори, керуючі структури, які підтримують обхід та построчно (замість множинної) обробку записів. Це дозволяє здійснювати більш складну обробку, ніж це можна зробити за допомогою традиційних декларативних команд маніпулювання даними, надаючи можливість користувачам баз даних застосовувати такі керуючі структури, як умови та цикли, для досягнення більш тонкого контролю над завданнями обробки даних. SQL:1999 передбачає збережені процедури, самодостатні модулі SQL-коду, які можуть приймати параметри та виконувати завдання, керовані даними, а також функції, що визначаються користувачем, які приймають параметри і повертають єдине значення. Ці функції, подібні до підпрограм або функцій у процедурних мовах, добре підтримуються в сучасних реалізаціях SQL і широко використовуються. SQL:1999 також включає тригери, збережені процедури, які запускаються автоматично, коли відбувається певна модифікація даних. Ці конструкції, як і весь вміст реляційної бази даних, можна створювати, скидати та модифікувати за

допомогою самої мови SQL, за допомогою таких команд, як CREATE PROCEDURE і DROP TRIGGER.

#### 1.4.2. Використання реляційних баз даних в об'єктно-орієнтованих програмах

Об'єктно-орієнтоване програмне забезпечення та СУБД не тільки надзвичайно поширені, але й особливо часто використовуються в поєднанні один з одним. У багатьох додатках необхідно зберігати вміст структур даних протягом тривалих періодів часу — і, зокрема, між виконаннями програми. Здатність робити це називається персистентністю. Без персистентності дані існують тільки в пам'яті та втрачаються при вимиканні комп'ютера. Для досягнення персистентності програма повинна зберігати дані на енергонезалежному носії, наприклад, на жорсткому диску. Наприклад, програмне забезпечення, яке працює на споживчих комп'ютерах (наприклад, гра), може зберігати дані у файловій системі, зберігаючи та завантажуючи файли даних на жорсткий диск з метою підтримки стану між виконаннями програми. Однак багато додатків, особливо ті, що працюють на серверах або інших потужних машинах (наприклад, веб-додатки), використовують реляційні бази даних для збереження даних. Як правило, розробники відображають класи в таблиці бази даних досить простим способом, створюючи одну таблицю для кожного класу, який повинен бути персистентним [16]. Наприклад, для класу User, як показано на рис. 1.7 на JavaScript (TypeScript) розробник може створити таблицю в базі даних додатку, використовуючи код, подібний до наступного на рис 1.8. (Цей приклад наведений в PostgreSQL).

```
export class User {
  public readonly id: number;
  public readonly firstName: string;
  public readonly lastName: string;
  public readonly isActive: boolean;
}
```

Рисунок 1.7. Приклад класу User в TypeScript

```
CREATE TABLE User (
  id          integer PRIMARY KEY,
  firstName   varchar NOT NULL,
  lastName    varchar NOT NULL,
  isActive    boolean DEFAULT true
);
```

Рисунок 1.8. Приклад класу User в PostgreSQL

#### 1.4.3. Драйвери баз даних та API

Але яким чином додаток комунікує із базою даних? Кажучи у найпростіших термінах, сучасні об'єктно-орієнтовані програмні додатки підключаються до баз даних через інтерфейси прикладного програмування (API), які забезпечують програмний доступ до драйверів баз даних, програмних компонентів, які вже фактично підключаються до бази даних та взаємодіють з нею. Насправді все може бути складніше, з декількома рівнями API, драйверів і мостів, але з точки зору розробника програми, весь доступ до бази даних здійснюється через API, який часто вбудований в мову або легко доступний через стандартні бібліотеки класів. Стандартні бібліотеки сучасних об'єктно-орієнтованих мов, такі як Java Platform, Standard Edition або ж бібліотека базових класів .NET, забезпечують готову підтримку різноманітних платформ баз даних. Зручність драйверів баз

даних в тому, що вони надають готовий та простий інтерфейс, а їх детальна реалізація прихована від користувача.

#### 1.4.4. Невідповідність імпедансу

Так звана об'єктно-реляційна невідповідність імпедансу є одночасно і стимулом, і перешкодою для об'єктно-реляційного відображення. Термін "імпедансна невідповідність" запозичений з електротехніки, де імпеданс вимірює опір протіканню змінного струму в ланцюзі. Найбільш ефективний обмін між електричними системами відбувається тоді, коли їхні імпеданси добре узгоджуються. Невідповідність імпедансу — це проблема, яка виникає при з'єднанні двох ланцюгів з різними імпедансами, що в кінцевому підсумку може призвести до загасання і перешкод. Проблема можна виправити шляхом узгодження імпедансу. Аналогія полягає в тому, що об'єктно-орієнтовані системи часто здаються невідповідними системам реляційних баз даних. Щоб пояснити метафору, невідповідність імпедансу як в електротехніці, так і в програмній інженерії виникає тоді, коли одна система не може ефективно взаємодіяти з іншою системою. Об'єктно-реляційна невідповідність імпедансу виникає тому, що об'єктно-орієнтована та реляційна парадигми баз даних мають різні концепції даних. Об'єктно-орієнтована парадигма розглядає дані в першу чергу в контексті дій, що виконуються над ними. Тобто об'єкти важливі не просто через дані, які вони містять, а через їх здатність виконувати завдання над даними та обмінюватися інформацією з іншими об'єктами. Реляційна парадигма орієнтована на дані. Вона надає значення самим даним та їх структурним, замість поведінкових, зв'язкам з іншими даними.

### 1.4.5. Зв'язки між таблицями

У реляційній базі даних зв'язок утворюється шляхом співвіднесення рядків, що належать до різних таблиць. Зв'язок між таблицями встановлюється, коли дочірня таблиця визначає стовпець зовнішнього ключа, який посилається на стовпець первинного ключа своєї батьківської таблиці [17].

Таким чином, кожен зв'язок між таблицями бази даних будується на основі стовпців зовнішнього ключа, всього може бути три типи зв'язків між таблицями:

- один-до-багатьох — найпоширеніший тип зв'язку, який пов'язує рядок з батьківської таблиці з декількома рядками в дочірній таблиці. (рис. 1.9) Прикладом такого зв'язку в реальному світі може бути блог і коментарі до нього. У одного блогу може бути безліч коментарів, але один коментар може відноситись тільки до одного певного блогу;
- один-до-одного вимагає, щоб первинний ключ дочірньої таблиці був зв'язаний через зовнішній ключ зі стовпцем первинного ключа батьківської таблиці. (рис. 1.10) Прикладом такого зв'язку в реальному світі може бути блог та його опис. У одного блогу бути лише один опис, і цей опис може належати тільки одному конкретному блогу;
- багато-до-багатьох вимагає, щоб таблиця зв'язку містила два стовпці зовнішнього ключа, які посилаються на дві різні батьківські таблиці. (рис. 1.11) Прикладом такого зв'язку в реальному світі може бути блог та теги. Один блог може мати багато різних тегів одночасно. В той самий час, в кожного тегу є велика кількість пов'язаних блогів.

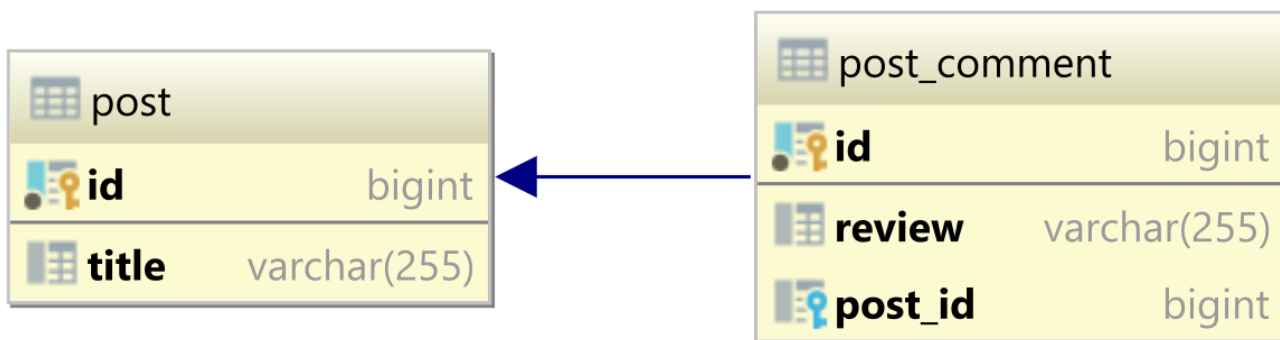


Рисунок 1.9 – Зв’язок між таблицями “один до багатьох”

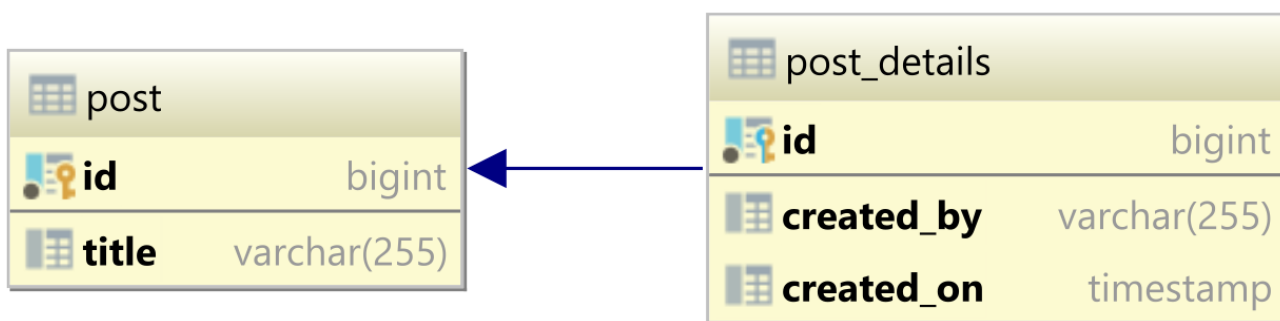


Рисунок 1.10 – Зв’язок між таблицями “один до одного”



Рисунок 1.11 – Зв’язок між таблицями “багато до багатьох”

#### 1.4.6. Типи даних

Типи даних SQL визначають тип значення, яке може зберігатися в стовпці таблиці. Наприклад, якщо ми хочемо, щоб стовпець зберігав тільки цілі значення, то ми можемо визначити його тип даних як `int`.

Типи даних SQL можна умовно розділити на наступні категорії (рис. 1.12):

- Числові типи даних, такі як `int`, `tinyint`, `bigint`, `float`, `real` тощо;
- Типи даних дати та часу, такі як `Date`, `Time`, `Datetime` і т.д.;

- Символьні та рядкові типи даних, такі як char, varchar, text і т.д.;
- Символьно-рядкові типи даних Unicode, наприклад, nchar, nvarchar, ntext і т.д.;
- Двійкові типи даних, такі як binary, varbinary і т.д.;
- Додаткові типи даних - clob, blob, xml, cursor, table тощо.

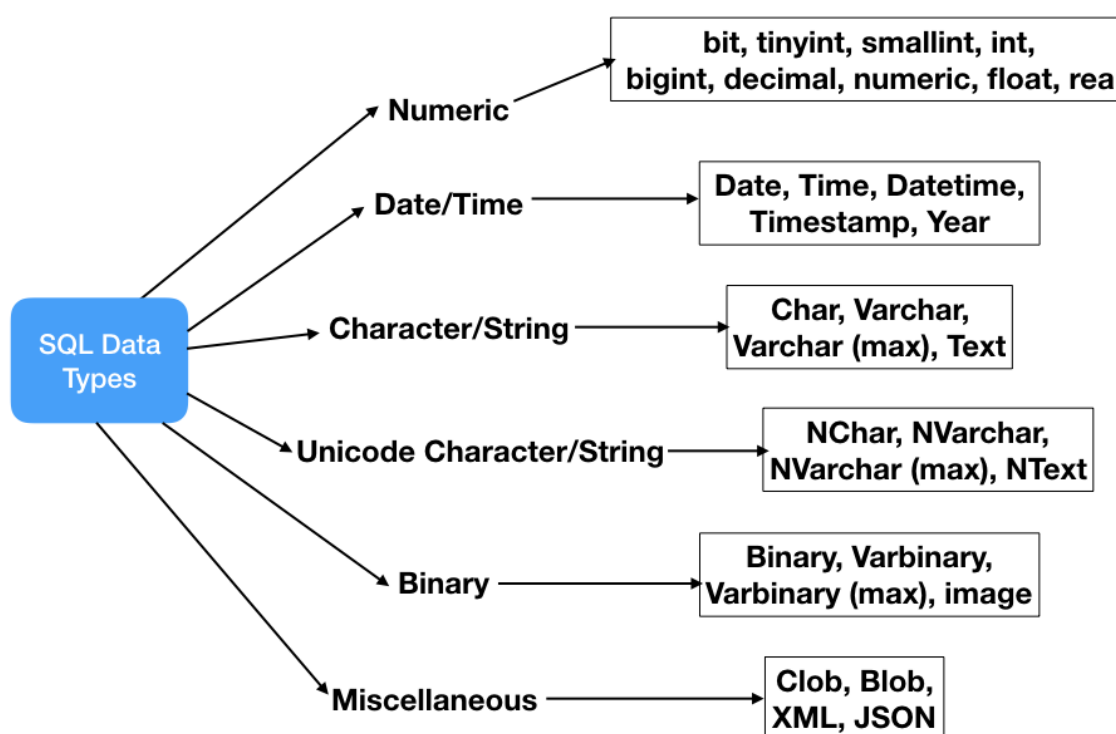


Рисунок 1.12 – Типи даних в SQL

Важливо розуміти, що не всі типи даних підтримуються кожним постачальником реляційних баз даних. Наприклад, база даних Oracle не підтримує тип даних DATETIME, а MySQL не підтримує тип даних CLOB. Тому при розробці схеми бази даних та написанні SQL-запитів слід обов'язково перевіряти, чи підтримуються потрібні типи даних.

Типи даних, які перераховані вище, не включають в себе всі типи даних, це найбільш часто використовувані типи даних. Деякі постачальники реляційних

баз даних мають власні типи даних, які можуть бути відсутніми в цьому списку. Наприклад, Microsoft SQL Server має типи даних money і smallmoney, але оскільки вони не підтримуються іншими популярними постачальниками баз даних, вони не вказані тут [18].

### 1.5. Об'єктно-реляційне відображення (проєкція)

Система ORM передбачає підключення об'єктного коду до реляційної бази даних за допомогою дескрипторів метаданих. Для написання об'єктного коду використовуються об'єктно-орієнтовані мови програмування, такі як Java, C#, TypeScript тощо. Дані, представлені яких у реляційних базах даних і мові ООП не можуть співіснувати, можна трансформувати за допомогою системи ORM [19]. Приклад невідповідності зберігання даних зображено на рис 1.13 та рис 1.14. На рисунках видно ту саму невідповідність імпедансу, яку було згадано в цьому розділі раніше.

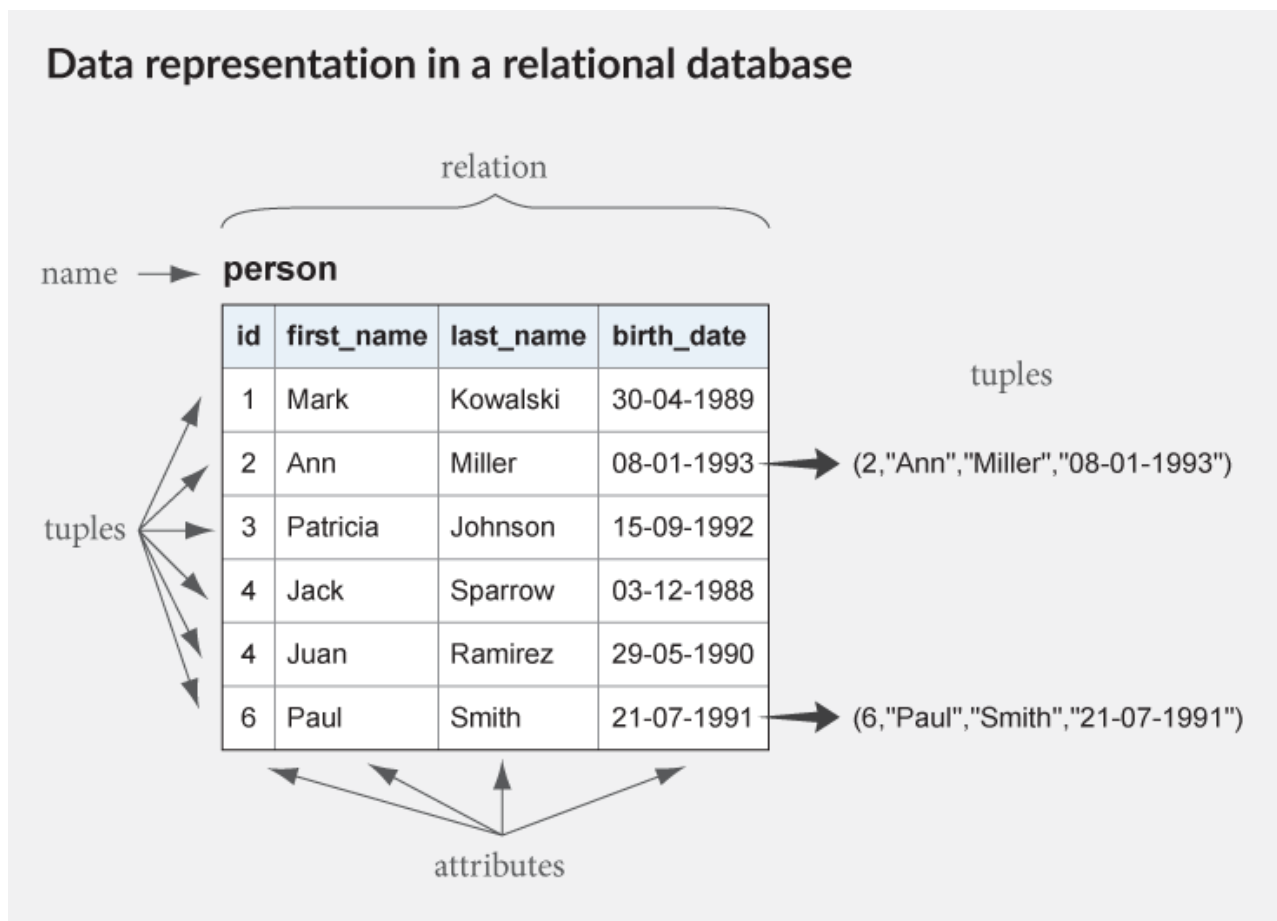


Рисунок 1.13 – Приклад зберігання даних в базі даних

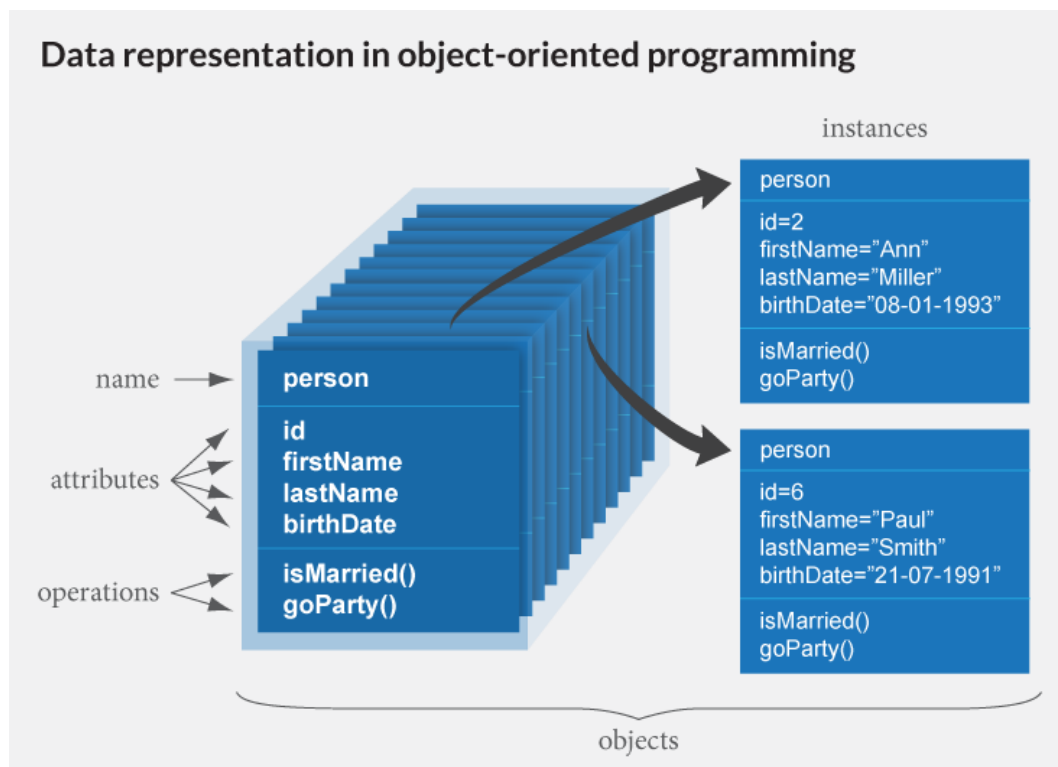


Рисунок 1.14 – Приклад зберігання даних в програмному коді (екземпляри класу)

У порівнянні з вбудованим SQL і рукописними збереженими процедурами, система ORM спрощує розробку, автоматизуючи перетворення об'єкта в таблицю і таблиці в об'єкт. Прозоре кешування об'єктів на рівні програми покращує продуктивність системи. Оптимізовані рішення роблять додатки швидшими та легшими для обслуговування. Використання ORM надає розробнику можливість писати тією мовою, якою він вже користується. Не кожен програміст знає особливості мови SQL на потрібному рівні, адже SQL є потужною мовою, але більшість не дуже часто пишуть запити нею. В той самий час, як правило, ми набагато краще володіємо тією чи іншою мовою програмування. Залежно від ORM користувач отримує багато розширених функцій із коробки, наприклад підтримку транзакцій, пул з'єднань, міграції, сеанси, потоки та інші неймовірно корисні функції. ORM створює новий рівень абстракції над базою даних, тим самим роблячи програму незалежною від системи керування базою даних, що використовується серверною частиною, тому, користуючись інтерфейсом, можна писати загальні запити. Завдяки цьому, у разі переходу на іншу базу даних наявність ORM в проєкті стає дуже вигідною пропозицією. Турботи програміста зводяться тільки до вивчення синтаксису SQL окремо для кожної бази даних, яка використовується для підтримки програми. Замість вирішення проблем із підключенням розробники можуть звернути увагу на оптимізацію коду та підвищення продуктивності. Усі малі чи великі зміни можна вносити через ORM, тому немає сильних обмежень під час роботи з даними. Наприклад, JDBC має багато обмежень щодо отримання набору результатів, його обробки та відправки назад до бази даних. Використовуючи ORM таких обмежень немає. Навіть окремі комірочки в базі даних можна отримати, змінити та зберегти. З'єднання стає надійним, безпечним, оскільки зменшується втручання в код. ORM буде обробляти всі потрібні конфігурації, які необхідні для зіставлення мови програмування додатка із мовою запитів бази даних, оскільки буде менше втручання для сприяння безпечній програмі в цілому.

На ринку присутня досить велика кількість відкритоджерельних ORM відповідно до використовуваної мови програмування. Можна легко вибрати відповідно до бізнес-вимог.

Поява ORM у багатьох розробках додатків викликала розбіжності серед експертів. Основними недоліками у використанні ORM вважаються ситуації коли база даних знаходиться в застарілих файлових системах і є невідсортованою. Виникає завдання впорядкувати цілий масив даних, а потім зіставити його з ORM. Таким чином, пропонується використовувати ORM, коли внутрішня частина досить добре керована. Крім цього, ORM не у всіх ситуаціях працює належним чином, і збережена процедура іноді може бути кращим рішенням. Також, у деяких випадках довіра до ORM може призвести до поганого дизайну бази даних, проте, це все ж більше людський фактор, аніж провина систем ORM.

#### 1.6. Принцип роботи систем об'єктно-реляційного відображення

ORM створюють об'єктно-орієнтовану модель програми з абстракціями високого рівня. Іншими словами, він виконує рівень логіки без глибоких деталей коду. Відображення описує зв'язок між об'єктами та даними, не знаючи структури даних. Потім цю модель можна використовувати для підключення програми до коду SQL, необхідного для керування операціями з даними. Цей «сантехнічний» тип коду не потребує переписування, що значно економить час розробників.

ORM використовує дві різні стратегії: модель активного запису та модель відображення даних:

- Шаблон активного запису. Ця стратегія відображає дані в структурі об'єкта в коді. Ви можете використовувати класи та структури в коді для керування даними. Цей підхід є проблематичним, оскільки структура бази даних тісно

пов'язана з кодом, що ускладнює видалення та переміщення бази даних в іншу програму;

- Шаблон відображення даних. Шаблон відображення даних намагається відокремити бізнес-логіку в об'єкті від бази даних. Це розділення може спростити зміну бази даних і використовувати ту саму логіку програмування.

### 1.7. Існуючі системи об'єктно-реляційного відображення для популярних мов програмування:

Java:

- Hibernate;
- Java Data Objects (JDO);
- Ebean;
- Apache Cayenne.

.NET:

Dapper

- nHydrate;
- nHibernate;
- Entity Framework;
- Quick Objects.

PHP:

- Doctrine ORM;
- Laravel (Eloquent);
- Yii;
- Qcodo;
- Propel.

Python:

- Django ORM;

- SQLAlchemy;
- SQLAlchemy;
- Storm;
- Tryton ORM.

Typescript:

- Prisma;
- TypeORM;
- Sequelize;
- Objection.js;
- Knex.js;
- Bookshelf;
- Waterline.

## 1.8. Порівняння існуючих рішень для мови програмування TypeScript

Оскільки, об'єктивно кажучи, немає великого сенсу порівнювати між собою технічні можливості систем ORM різних мов програмування, в цьому параграфі буде поверхневе порівняння існуючих рішень саме для мови програмування TypeScript, якою, як буде зазначено в наступному розділі, буде розроблюватись програмне рішення, втілене в рамках даної роботи.

Для порівняння візьмемо найпопулярніші системи ORM екосистеми TypeScript: Sequelize, TypeORM, Knex та Objection.

Sequelize має надійну підтримку транзакцій, зв'язків, швидкого та лінивого завантаження, реплікації читання та багато іншого. Це один з найповніших систем OPB для Node.js. Має підтримку Postgres, MySQL, MariaDB, SQLite та Microsoft SQL Server.

Knex — дуже потужний конструктор запитів з підтримкою транзакцій. Він не має всіх функцій, які може мати система OPB, але його продуктивність набагато

краща. Має підтримку Postgres, MSSQL, MySQL, MariaDB, SQLite3, Oracle та Amazon Redshift.

TypeORM має мету завжди підтримувати найновіші функції JavaScript і надавати додаткові можливості, які допоможуть розробляти будь-які додатки, що використовують бази даних. Він підтримує MySQL, MariaDB, Postgres, CockroachDB, SQLite, Microsoft SQL Server, Oracle і MongoDB NoSQL.

Objection побудований на основі Knex, тому підтримує ті ж самі бази даних. Має всі переваги конструктора SQL запитів, а також потужний набір інструментів для роботи з відношеннями, за це його можна вважати системою ORM.

Таблиця 1.1 – Порівняння за популярністю серед спільноти

| Система   | Дата створення | Завантажень за тиждень | Залежності |
|-----------|----------------|------------------------|------------|
| Sequelize | 22 липня 2010  | 1,424,313              | 3139       |
| TypeORM   | 29 грудня 2012 | 1,078,473              | 1953       |
| Knex      | 29 лютого 2016 | 1,288,938              | 917        |
| Objection | 14 квітня 2015 | 134,251                | 220        |

Downloads in past 1 Year ▾

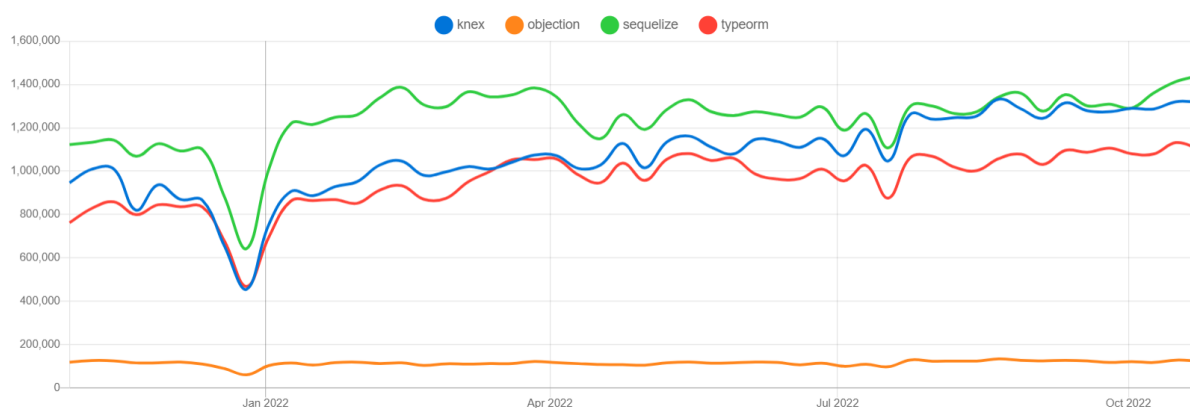


Рисунок 1.15 – Графік завантажень порівнюваних систем за останній рік

Маючи дані з табл. 1.1 та рис. 1.15 (дані взяті з офіційного сайту npm), можемо зробити їх аналіз.

По-перше, одразу видно, що Sequelize, являючись найстарішим, є найпопулярнішою системою ORM. Трохи відстають від нього Knex та TypeORM. Це дуже цікавий момент що Knex використовується майже так само часто, як і Sequelize, незважаючи на те, що він є менш дружнім рішенням, бо являється, по своїй суті, інструментом створеним, в основному, тільки для спрощення процесу побудування запитів до бази даних.

По-друге, бачимо, що за останній рік всі три системи показали стале зростання популярності, на відміну від Objection, який своєю чергою демонструє практично нульовий приріст щотижневих завантажень, утримуючи його на рівні близько 130 тис. завантажень на тиждень.

Таблиця 1.2 – Порівняння результатів швидкодії читання простих даних з бази

| Система   | Середня затримка, мс | Середня к-ть запитів в секунду |
|-----------|----------------------|--------------------------------|
| Sequelize | 2202                 | 42.9                           |
| TypeORM   | 1277                 | 75.91                          |
| Knex      | 1424                 | 67.8                           |
| Objection | 1467                 | 65.66                          |

Система оцінювання для табл. 1.9. наступна:

- 0 - відсутня;
- 1 - є, слабка/погано описана;
- 2 - є, з певними недоліками;
- 3 - є, відмінна.

Таблиця 1.3 – Порівняння офіційної документації

| Система   | Підтримка TypeScript | Якість документації |
|-----------|----------------------|---------------------|
| Sequelize | 1                    | 2                   |
| TypeORM   | 3                    | 3                   |
| Knex      | 2                    | 2                   |
| Objection | 3                    | 3                   |

Аналізуючи дані з таблиць 1.2 та 1.3, можна відмітити що TypeORM є, одним із найкращих рішень, доступних на сьогодні. Також хорошими рішеннями є Knex та Objection. Sequelize є надійним інструментом випробуваним часом, проте він все ж є застарілим. З не самою прямолінійною документацією, в якій немає вичерпних прикладів використання, крім того, на момент написання цієї роботи, деякі методи системи взагалі не мають описаних TypeScript інтерфейсів, що звісно не є зручним під час роботи з ним. Навіть в самій документації прямим текстом вказано, що оскільки Sequelize значною мірою покладається на присвоєння властивостей під час виконання, TypeScript не буде дуже корисним з коробки [20]. Для того, щоб зробити моделі працездатними, потрібна доволі велика кількість ручних оголошень типів.

## ВИСНОВКИ ДО ПЕРШОГО РОЗДІЛУ

Написання коду SQL для з'єднання реляційних баз даних з об'єктно-орієнтованими середовищами виконання може бути трудомісткою діяльністю, яка не принесе користі для бізнес-програм. Розробники можуть писати необроблений код SQL або використовувати конструктор запитів SQL, щоб покращити процес, але обидва підходи все одно вимагають глибоких знань бази даних і навичок програмування SQL. ORM підвищують продуктивність, створюючи високоабстрактні моделі даних і автоматично генеруючи код SQL. Ці інструменти також спрощують відокремлення бази даних від логіки програмування, надаючи розробникам більше гнучкості. Але системи ORM також мають своїх критиків. Загальні скарги включають низьку продуктивність та необхідність зайвий код, але це дуже залежить від якості самої використаної ORM.

Системи ORM рекомендуються для використання у малих та середніх за розмірами проєктах. Розробники повинні виконати певне вивчення ринку перед тим як прийняти рішення про остаточне використання тієї чи іншої ORM, щоб прискорити процес розробки. Доступна велика кількість комерційних інструментів із відкритим кодом для різних мов програмування. Оскільки ці інструменти існують уже багато років, більшість з них добре задокументована та підтримуються великою спільнотою розробників.

В той самий час, додатки, засновані на простих структурах даних, що не мають багато логіки для управління ними, швидше за все, простіше і зрозуміліше писати на звичайному sql.

Команди з високим рівнем знань баз даних і невеликим досвідом в ОО технологіях, швидше за все, будуть більш ефективні при використанні чистого SQL. Звичайно, в залежності від додатків, які вони пишуть, команді може бути рекомендовано переключити фокус.

Своєю чергою, команди з високим рівнем знань в області ООП і лише базовим досвідом роботи з базами даних, швидше за все, будуть більш ефективні при використанні ORM.

## 2. ОПИС ОБРАНОГО ІНСТРУМЕНТАРІЮ ІМПЛЕМЕНТАЦІЇ СИСТЕМИ ОБ'ЄКТНО-РЕЛЯЦІЙНОГО ВІДОБРАЖЕННЯ

### 2.1. Мова програмування для імплементції обраного програмного забезпечення

Для розробки обраного програмного забезпечення було обрано мову програмування TypeScript. Мова TypeScript пропонує всі можливості JavaScript, а також додатковий рівень над ними: систему типів TypeScript.

Наприклад, JavaScript надає примітиви мови, такі як `string` і `number`, але він не перевіряє, чи правильно їх було присвоєно, бо JavaScript являє собою мову з динамічною типізацією. А TypeScript дозволяє робити це.

Це означає, що дійсний код JavaScript також є кодом TypeScript. Основна перевага TypeScript полягає в тому, що він виявляє неочікувану поведінку у вашому коді, зменшуючи ймовірність помилок на етапі компіляції процесу розробки [21].



```
let helloWorld = "Hello World";  
let helloWorld: string
```

Рисунок 2.1 – Базовий приклад коду мовою TypeScript

Знаючи та розуміючи, як працює JavaScript, TypeScript може допомогти створити систему типів, яка приймає код JavaScript, але має статичну типізацію. Це забезпечує систему типів без додавання додаткових символів, щоб зробити типи явними в коді. Саме таким чином у прикладі на рис. 2.1 TypeScript знає, що `helloWorld` є рядком (типом `string`).

Так, наприклад, редактор Visual Studio Code під час роботи з JavaScript має можливість автозавершення. Для цього функціоналу, Visual Studio Code використовує TypeScript під капотом, щоб полегшити роботу з JavaScript.

В JavaScript можна використовувати широкий спектр шаблонів проєктування. Однак деякі шаблони ускладнюють автоматичне визначення типів. Для покриття цих випадків TypeScript підтримує розширення мови JavaScript, яке пропонує місця, де можна вказати TypeScript, якими повинні бути типи.

Наприклад, щоб створити об'єкт з виведеним типом, який включає ім'я з типом рядок та ідентифікатор з типом число, можна написати як зображено на рис 2.2. Явно описати форму цього об'єкту можна з допомогою оголошення інтерфейсу як зображено на 2.3. Потім можна оголосити, що об'єкт JavaScript відповідає формі створеного нового інтерфейсу, використовуючи синтаксис типу : TypeName після оголошення змінної як зображено на рис. 2.4. Таким чином, якщо буде надано об'єкт, який не відповідає визначеному інтерфейсу, TypeScript попередить про це таким способом, як це зображено на рис 2.5.

```
const user = {  
  name: "Hayes",  
  id: 0,  
};
```

Рисунок 2.2 – Приклад базового об'єкта

```
interface User {  
  name: string;  
  id: number;  
}
```

Рисунок 2.3 – Приклад інтерфейсу

```
const user: User = {  
  name: "Hayes",  
  id: 0,  
};
```

Рисунок 2.4 – Приклад типізації об'єкта за допомогою інтерфейсу

```
interface User {
  name: string;
  id: number;
}

const user: User = {
  username: "Hayes",
  id: 0,
};
```

Type '{ username: string; id: number; }' is not assignable to type 'User'.  
Object literal may only specify known properties, and 'username' does not exist in type 'User'.

Try

Рисунок 2.5 – Приклад помилки типізації

Оскільки JavaScript підтримує класи та об'єктно-орієнтоване програмування, то і TypeScript також. З класами також можна використовувати оголошення інтерфейсу, як зображено на рис 2.6.

```
interface User {
  name: string;
  id: number;
}

class UserAccount {
  name: string;
  id: number;

  constructor(name: string, id: number) {
    this.name = name;
    this.id = id;
  }
}

const user: User = new UserAccount("Murphy", 1);
```

Try

Рисунок 2.6 – Типізація за допомогою інтерфейсу класу

За допомогою TypeScript можна створювати складні типи шляхом об'єднання між собою більше простих типів. Існує два популярних способи зробити це: за допомогою об'єднань і за допомогою узагальнень.

За допомогою об'єднання можна оголосити, що тип може бути одним з багатьох типів. Наприклад, таким чином, як зображено на рис. 2.7, можна описати булевий тип як істинний або хибний.

```
type MyBool = true | false;
```

Рисунок 2.7 – Приклад описаного складного типу

Популярним випадком використання об'єднаних типів є опис набору рядкових або числових літералів, якими може бути значення, як-от на рис. 2.8.

```
type WindowStates = "open" | "closed" | "minimized";
type LockStates = "locked" | "unlocked";
type PositiveOddNumbersUnderTen = 1 | 3 | 5 | 7 | 9;
```

Рисунок 2.8 – Приклад типів з описом наборів рядкових та числових літералів

Узагальнення надають змінним типи. Типовим прикладом є масив. Масив без узагальнень може містити будь-що. Масив з узагальненнями може описувати значення, які містять в собі масив.

```
type StringArray = Array<string>;
type NumberArray = Array<number>;
type ObjectWithNameArray = Array<{ name: string }>;
```

Рисунок 2.9 – Узагальнення в TypeScript

Один з основних принципів TypeScript полягає в тому, що перевірка типу фокусується на формі, яку мають значення. Це іноді називають "качиною типізацією" або "структурною типізацією".

Таким чином, у системі структурних типів, якщо два об'єкти мають однакову форму, вони вважаються одного типу, як-от на рис. 2.10 Тут змінна `point` ніколи не оголошується як тип `Point`. Однак, при перевірці типу TypeScript порівнює форму точки з формою типу `Point`. Вони мають однакову форму, тому код проходить.

```

interface Point {
  x: number;
  y: number;
}

function logPoint(p: Point) {
  console.log(`${p.x}, ${p.y}`);
}

// logs "12, 26"
const point = { x: 12, y: 26 };
logPoint(point);

```

Рисунок 2.10 – Приклад “качиної” типізації TypeScript

В TypeScript модифікатори доступу керують доступністю членів класу. У TypeScript є два модифікатори доступу - `public` та `private`. За замовчуванням члени класу є загальнодоступними, але до них можна явно додати модифікатор `public` або `private`. Розглянемо наступний фрагмент коду на рис. 2.11, в якому оголошується дві загальнодоступні (`public`) та одна приватна (`private`) змінні. У наведеному коді оголошується клас з іменем `Customer`, який має три члени: `customerId`, `companyName` та `country`. Оскільки за замовчуванням члени класу є загальнодоступними, то до `customerId` можна отримати доступ із зовнішнього світу, як показано у вікні `IntelliSense`. У тих же рядках компанія `CompanyName` явно позначена як загальнодоступна і, отже, вона доступна із зовнішнього світу. Країна, однак, позначена як приватна і, отже, не буде доступна за межами класу, як видно з вікна `IntelliSense`.

```

class Customer {
    customerId: number;
    public companyName: string;
    private country: string;
}

var obj = new Customer();
obj.

```

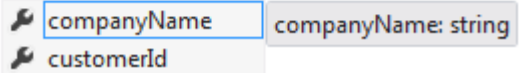


Рисунок 2.11 – Дві загальнодоступні та одна приватна змінні класу

Також TypeScript дозволяє писати перевантажені функції. Таким чином, можна викликати різні реалізації функції в залежності від параметра. Однак слід пам'ятати, що перевантаження функцій TypeScript є трохи дивним і вимагає перевірки типу під час реалізації. Це обмеження пов'язане з тим, що код на TypeScript в кінцевому підсумку компілюється у звичайний JavaScript, а JavaScript не підтримує концепцію перевантаження функцій в її істинному розумінні. Наступний простий приклад на рис. 2.12 показує, як можна створити дві версії функції мовою TypeScript. Як бачимо, перші дві сигнатури є просто сигнатурами функцій без будь-якої реалізації. Перша сигнатура має один параметр типу `number`, тоді як друга сигнатура має параметр типу `string`. Третя функція містить власне реалізацію і має параметр типу `any`. Тип даних `any` вказує на те, що параметр може бути переданий будь-якого типу. Далі реалізація має ряд блоків `if`, кожен з яких перевіряє тип переданого параметру. Відповідно виконується інший фрагмент коду.

```

addCustomer(custId: number);
addCustomer(company:string);
addCustomer(value: any) {
    if (value && typeof value == "number") {
        alert("First overload - " + value);
    }
    if (value && typeof value == "string") {
        alert("Second overload - " + value);
    }
}
}

```

Рисунок 2.12 – Перевантаження в TypeScript

Слід ще раз зазначити, що користуючись TypeScript в кінцевому результаті врантаймі все одно буде використовуватися JavaScript (\*.js), а не на файл коду TypeScript (\*.ts). Для компіляції коду TypeScript в JavaScript є два варіанти — можна скористатися компілятором TypeScript з командного рядка (tsc.exe) або плагіном редактора TypeScript для Visual Studio.

Отже, ось в чому виграє розробник використовуючи TypeScript:

- Дослідження показали, що TypeScript може виявити 15% поширених помилок;
- Читабельність — легше зрозуміти, що повинен робити код. А при роботі в команді легше зрозуміти, що мали на увазі інші розробники;
- Це популярна та потрібна технологія — знання TypeScript дозволить тому хто його вивчає претендувати на більше цікавих робочих місць;
- Вивчення TypeScript надасть краще розуміння і новий погляд на JavaScript.

Недоліки TypeScript:

- TypeScript займає більше часу для написання, ніж JavaScript, оскільки вам потрібно вказувати типи, тому для невеликих короткотривалих проєктів, можливо, не варто його використовувати;
- TypeScript потрібно компілювати, що може зайняти певний час, особливо при використанні у великих проєктах.

## 2.2. Опис інструментів для обраної мови програмування

### 2.2.1. Lodash

Під час імплементації системи було використано бібліотеку Lodash.

Lodash являє собою сучасну бібліотеку утиліт JavaScript, що забезпечує модульність, продуктивність та додаткові можливості. Він робить JavaScript простішим, позбавляючи розробника клопоту при роботі з масивами, числами, об'єктами, рядками тощо.

Модульні методи Lodash чудово підходять для:

- Ітерації масивів, об'єктів та рядків;
- Маніпулювання та тестування значень;
- Створення складних функцій.

### 2.3. node-postgres

У якості драйверу, для роботи із базою даних було обрано бібліотеку pg (node-postgres).

node-postgres - це збірка модулів node.js для взаємодії з базою даних PostgreSQL. Він має підтримку зворотних викликів, промісів, async/await, пулів з'єднань, підготовлених операторів, курсорів, потокового передавання результатів, C/C++ зв'язування, розширеного розбору типів та багато іншого.

Наприклад на рис. 2.13 зображено найпростіший спосіб з'єднання, запиту та роз'єднання з асинхронністю.

```
const { Client } = require('pg')
const client = new Client()
await client.connect()

const res = await client.query('SELECT $1::text as message', ['Hello world!'])
console.log(res.rows[0].message) // Hello world!
await client.end()
```

Рисунок 2.13 – Приклад використання драйверу pg

API для виконання запитів підтримує як колбеки, так і проміси. Приклад для обох стилів наведені на рис. 2.14 та рис. 2.15. По суті, `pool.query` внутрішньо делегує виконання безпосередньо клієнту бази даних з рис. 2.13.

```
const { Pool } = require('pg')
const pool = new Pool()

;(async () => {
  const { rows } = await pool.query('SELECT * FROM users WHERE id = $1', [1])
  console.log('user:', rows[0])
})().catch(err =>
  setImmediate(() => {
    throw err
  })
})
```

Рисунок 2.14 – Приклад запиту з очікуванням промісу

```
const { Pool } = require('pg')
const pool = new Pool()

pool
  .query('SELECT * FROM users WHERE id = $1', [1])
  .then(res => console.log('user:', res.rows[0]))
  .catch(err =>
    setImmediate(() => {
      throw err
    })
  )
```

Рисунок 2.15 – Приклад запиту з колбеком

Також драйвер `node-postgres` надає інтерфейс для створення транзакцій. На рис. 2.16 зображено приклад створення транзакції через проміси використовуючи синтаксис `async/await`.

```

const { Pool } = require('pg')
const pool = new Pool()
;(async () => {
  // note: we don't try/catch this because if connecting throws an exception
  // we don't need to dispose of the client (it will be undefined)
  const client = await pool.connect()

  try {
    await client.query('BEGIN')
    const queryText = 'INSERT INTO users(name) VALUES($1) RETURNING id'
    const res = await client.query(queryText, ['brianc'])

    const insertPhotoText = 'INSERT INTO photos(user_id, photo_url) VALUES ($1, $2)'
    const insertPhotoValues = [res.rows[0].id, 's3.bucket.foo']
    await client.query(insertPhotoText, insertPhotoValues)
    await client.query('COMMIT')
  } catch (e) {
    await client.query('ROLLBACK')
    throw e
  } finally {
    client.release()
  }
})();
}).catch(e => console.error(e.stack))

```

Рисунок 2.16 – Приклад створення транзакції

Крім цього слід зазначити, що PostgreSQL має багату систему підтримуваних типів даних. І node-postgres робить все можливе, щоб підтримувати найпоширеніші типи даних "з коробки" і постачає розширюваний аналізатор типів, що дозволяє користувачеві серіалізувати та аналізувати типи.

Драйвер node-postgres перетворить тип бази даних в рядок JavaScript, якщо він не має зареєстрованого аналізатора типів для цього типу бази даних. Більш того, можна відправити будь-який тип на сервер PostgreSQL у вигляді рядка і node-postgres пропустить його через себе, ніяк не модифікуючи.

Для дати, драйвер node-postgres перетворить екземпляри об'єктів дати JavaScript в очікуване вхідне значення для сервера PostgreSQL. Аналогічно, при зчитуванні дати, мітки часу або значення стовпця timestamp назад в JavaScript, node-postgres розбере значення на екземпляри об'єктів JavaScript Date, приклад зображено на рис 2.17.

```

const createTableText = `
CREATE TEMP TABLE dates(
  date_col DATE,
  timestamp_col TIMESTAMP,
  timestamptz_col TIMESTAMPTZ
);
`;

// create our temp table
await client.query(createTableText)

// insert the current time into it
const now = new Date()
const insertText = 'INSERT INTO dates(date_col, timestamp_col, timestamptz_col) VALUES ($1, $2, $3)'
await client.query(insertText, [now, now, now])

// read the row back out
const result = await client.query('SELECT * FROM dates')

console.log(result.rows)
// {
//   date_col: 2017-05-29T05:00:00.000Z,
//   timestamp_col: 2017-05-29T23:18:13.263Z,
//   timestamptz_col: 2017-05-29T23:18:13.263Z
// }

```

Рисунок 2.17 – Приклад перетворення типів даних

## 2.4. Середовище розробки

Не менш важливим фактором під час розробки також є вибір середовища розробки. Хоч воно і не впливає на результуючий програмний продукт напряму, все ж від середовища залежить те наскільки швидко та зручно буде проходити процес розробки.

Під час імплементації комп'ютерної системи, було використано редактор коду Visual Studio Code.

Visual Studio Code - це легкий, але потужний редактор джерельного коду, який є десктопним додатком і доступний для Windows, macOS і Linux. Він є найпопулярнішим вибором серед розробників, згідно зі статистикою на рис. 2.18. Visual Studio Code має вбудовану підтримку JavaScript, TypeScript та Node.js, а

також багату екосистему розширень для інших мов (таких як C++, C#, Java, Python, PHP та Go) та середовищ виконання (таких як .NET та Unity).

### Most Popular Development Environments

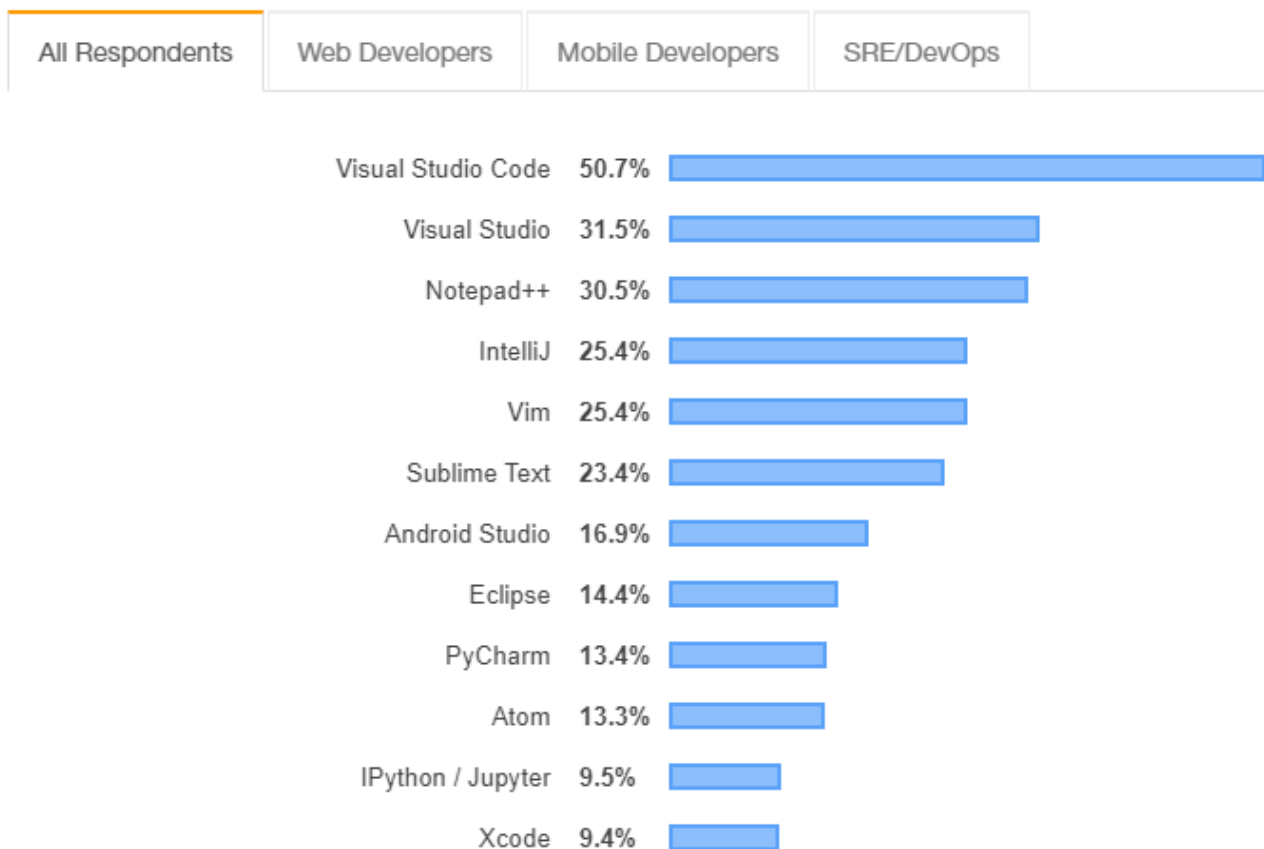


Рисунок 2.18 – Статистика найпопулярніших редакторів коду

Окрім загальної ідеї бути легким та швидко запускатися, VS Code має:

- Завершення коду IntelliSense для змінних, методів та імпортованих модулів;
- Графічне налагодження лінтування, багатокурсорне редагування, підказки параметрів та інші потужні функції редагування;
- Витончену навігацію та рефакторинг коду;
- Вбудований контроль вихідного коду, включаючи підтримку Git.

Багато з цього було адаптовано з технології Visual Studio.

Сам VS Code побудований з використанням оболонки Electron, Node.js, TypeScript та протоколу Language Server і оновлюється щомісяця. Розширення оновлюються так часто, як це необхідно. Багатство підтримки залежить від різних

мов програмування та їх розширень, починаючи від простого підсвічування синтаксису та узгодження дужок і закінчуючи налагодженням та рефакторингом.

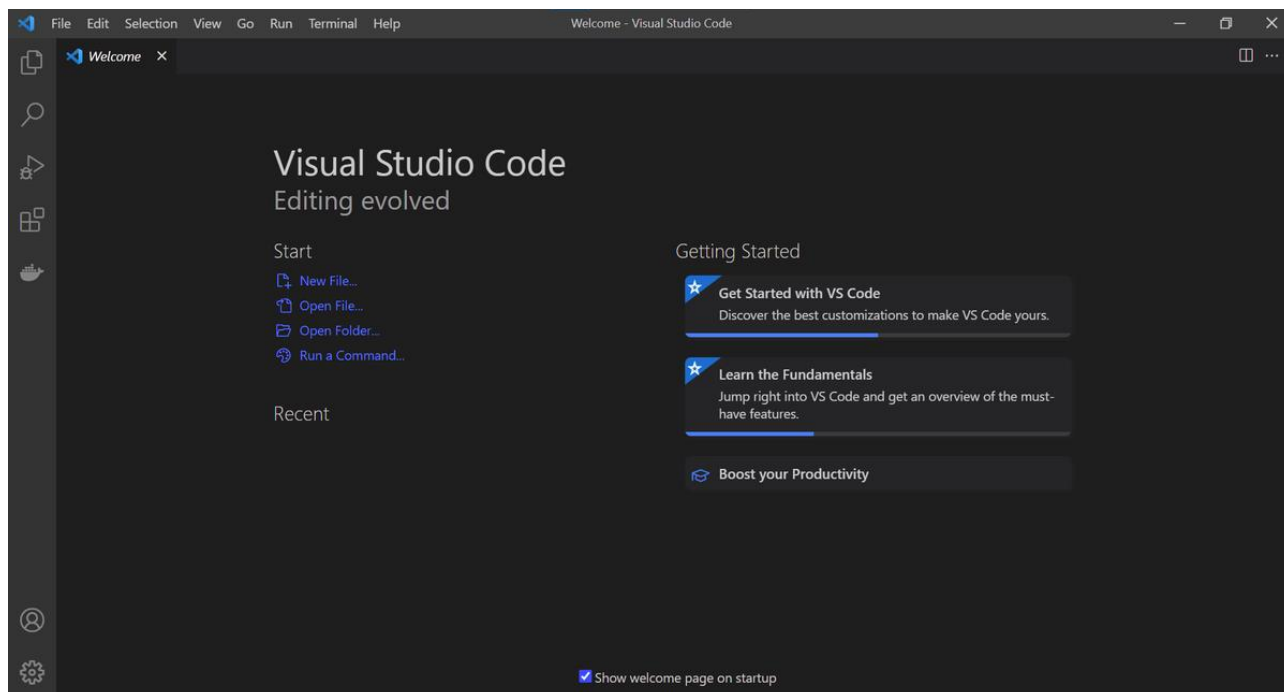


Рисунок 2.19 – Інтерфейс редактора коду Visual Studio Code

Особливістю VS Code є можливість встановлення різноманітних додаткових користувацьких розширень, про які вже було згадано вище. На ринку VS Code є тисячі розширень, і нові з'являються кожен день. Розширення можуть служити багатьом цілям. Від тем інтерфейсу до підтримки мов програмування, налагодження, інтеграції з Git'ом і навіть плеєрів Spotify. Без них VS Code був би не набагато більше, ніж прославленим текстовим редактором з гарним дизайном і базовим автозаповненням де-не-де.

Наприклад, в процесі розробки було використано декілька розширень, та певно найкориснішим було розширення “PostgreSQL” (рис. 2.20), яке надає можливість під'єднання до бази даних PostgreSQL прямо в редакторі VS Code. Це дуже зручно в процесі тестування під час розробки, адже відпадає потреба постійно перемикатись між різними робочими вікнами — вікном з кодом та вікном із клієнтом для бази даних.

Visual Studio Code > Other > PostgreSQL
New to Visual Studio Code



## PostgreSQL

Chris Kolkman | 431,691 installs | ★★★★★ (45) | Free

PostgreSQL Management Tool

Install
[Trouble Installing?](#)

[Overview](#)
[Version History](#)
[Q & A](#)
[Rating & Review](#)

### PostgreSQL

This is a query tool for PostgreSQL databases. While there is a database explorer it is *NOT* meant for creating/dropping databases or tables. The explorer is a visual aid for helping to craft your queries.

There is a language service that should keep at most a single connection open to the database (assuming one was selected). This facilitates query diagnostics, code completion, and function signatures. All code completion keywords, functions, tables, and field names are pulled from the current connection to try to keep everything relevant.

It is possible there are some queries that won't run and report problems - even if they are completely valid (DROP DATABASE comes to mind).

*Note: PostgreSQL versions before 9.4 are not supported.*

*Note: AWS RedShift is also not supported - there are information schema queries needed that just won't work there*

#### Categories

Other

#### Tags

keybindings postgres

#### Works with

Universal

#### Resources

[Issues](#)  
[Repository](#)  
[Homepage](#)  
[License](#)  
[Changelog](#)  
[Download Extension](#)

Рисунок 2.20 – Розширення “PostgreSQL” для VS Code

## ВИСНОВКИ ДО ДРУГОГО РОЗДІЛУ МД

В цьому розділі було описано та розглянуто базовий функціонал обраного для реалізації програмного продукту інструментарію. У якості мови програмування використано TypeScript — мову, джерельний код якої компілюється в JavaScript. У якості реляційної бази даних — PostgreSQL, а у якості драйверу для роботи з обраною базою даних — модуль node-postgres.

Обраний інструментарій існує протягом доволі тривалого часу, тож він випробуваний часом, має певну репутацію та велику спільноту поціновувачів, а отже має вичерпну документацію та велику кількість статей та обговорень актуальних питань на різноманітних форумах, що послугувало додатковим фактором полегшення процесу розробки із використанням цих інструментів.

### 3. СТРУКТУРА, ОПИС ТА ТЕСТУВАННЯ РЕАЛІЗОВАНОЇ КОМП'ЮТЕРНОЇ СИСТЕМИ

#### 3.1. Архітектура розробленого програмного забезпечення

На даному етапі, архітектура проєкту складається з шести основних модулів: Driver, Connection, Metadata, Migration, Executor та QueryBuilder.

Модуль Driver являє собою загальний інтерфейс над специфічними драйверами під певну базу даних. В нашому випадку, використовується тільки драйвер для PostgreSQL, проте, в майбутньому, маючи окремий модуль з уніфікованим інтерфейсом для специфічних драйверів, буде легше додати підтримку нових драйверів під інші реляційні бази даних, що додає гнучкості.

Модуль Connection відповідає за встановлення і підтримку одного з'єднання із базою даних, з цим модулем напряду працює більшість інших модулів. Це важливо, для правильного розподілу відповідальності між модулями.

Модуль Metadata працює з метаданими описаних класів та їх атрибутів, метадані необхідні, для того, щоб мати інформацію про те який клас відповідає якій сутності в таблиці бази даних, які трансформації та маніпуляції з атрибутами повинні бути проведені, для коректного відображення цієї сутності між середовищем та базою даних. Самі ж метадані описуються в коді за допомогою декораторів. Знову ж таки, важливо виділити це в окремий модуль, адже це розподілення відповідальності між модулями.

Модуль Migration відповідає за все що в системі пов'язано із міграціями, їх генерацією та запуском для їх використання та зміни поточного стану бази даних і перехід до нового стану.

Модуль Executor необхідний для прямого виконання запитів до бази даних, інші модулі, які напряду працюють з базою даних, делегують виконання SQL запитів саме на цей модуль.

І нарешті модуль QueryBuilder - модуль, який надає інтерфейс для побудування запитів до бази даних в об'єктно-орієнтованому стилі. Він займається

розпізнаванням ланцюга запиту та транслює його в коректний SQL-запит, намагаючись максимально оптимізувати його, який потім своєю чергою виконується модулем Executor.

```

src > core > executor > TS index.ts > Executor
1 import { Connection } from '../connection';
2
3 export class Executor {
4     private readonly connection;
5
6     constructor(connection: Connection) {
7         this.connection = connection.activeConnection;
8     }
9
10    async query(query: string, params?: any[]): Promise<any[]> {
11        const safeQuery = Helper.escape(query);
12        let parameters = [];
13
14        if (params && params.length) {
15            parameters = Helper.transformParameters(params);
16        }
17
18        return this.connection.query(safeQuery, parameters);
19    }
20 }
21
22 abstract class Helper {
23     static escape(str: string) {
24         if (typeof str !== 'string') return str;

```

PROBLEMS OUTPUT TERMINAL GITLENS COMMENTS

> TERMINAL

PS F:\code\k35orm>

Рисунок 3.1 – Метод query модуля Executor, який виконує запит до бази даних

Загалом, структура проекту зображена на рис. 3.2.

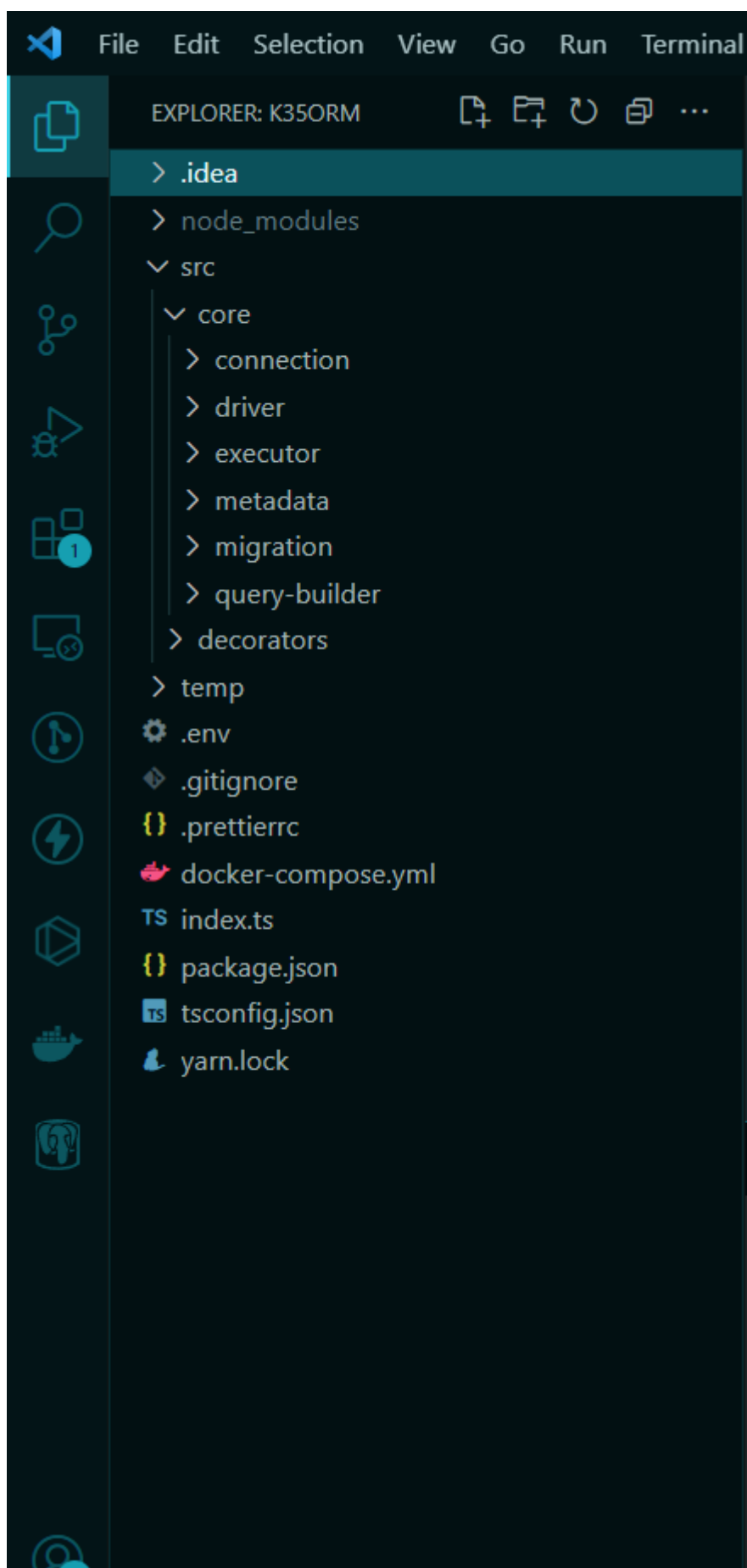


Рисунок 3.2 – Структура модулів

Директорія `decorators` з рис. 3.2 містить в собі декоратори — функції, які слугують обгорткою над атрибутами класів та самого класу. Декоратори, використовуючи `Reflect-metadata`, встановлюють класам-сутностям та їх метадані, які необхідні для правильного відображення даних між базою та сервісом. Декораторами, наприклад, є `Model` - декоратор, для опису моделі даних, яка співвідноситься із якоюсь таблицею в базі даних. Або `Column` - декоратор для опису атрибутів моделі, які потім будуть відображені у якості атрибута відповідної таблиці та у зворотньому напрямку.

### 3.2. Принцип роботи комп'ютерної системи

Загальний принцип взаємодії модулів був описаний в розділі 3.1, тут же буде розписано детальніше про алгоритм, за яким відбувається генерація міграцій на основі змін внесених в моделі даних, які відповідають сутностям бази даних. Розглянемо алгоритм, схематично зображений на рис. 3.3.

Під час запуску сервісу, після встановлення з'єднання з базою відбувається завантаження в оперативну пам'ять останньої версії стану моделей даних. Наступним кроком відбувається перетворення до необхідного формату завантажених з бази моделей та порівняння відповідних збережених та поточних моделей між собою, якщо між ними знаходяться якісь невідповідності між, тоді на основі цих невідповідностей генеруються відповідні міграції — по одній на кожну модель даних, де були знайдені невідповідності. Останнім етапом відбувається збереження в базу даних, в спеціальну таблицю для моделей, актуальних моделей, які будуть використані для порівняння під час наступного запуску додатка.

Тобто, наприклад, якщо буде помічена відмінність в типі в якійсь моделі, що означає що в моделі були зміни — буде згенерована міграція із SQL запитом, яка вносить такі ж зміни в тип даних таблиці бази даних, якій відповідає вищезазначена

модель, та SQL запитом, який анулює ці зміни, на випадок, якщо буде необхідно повернути стан бази даних до минулого.

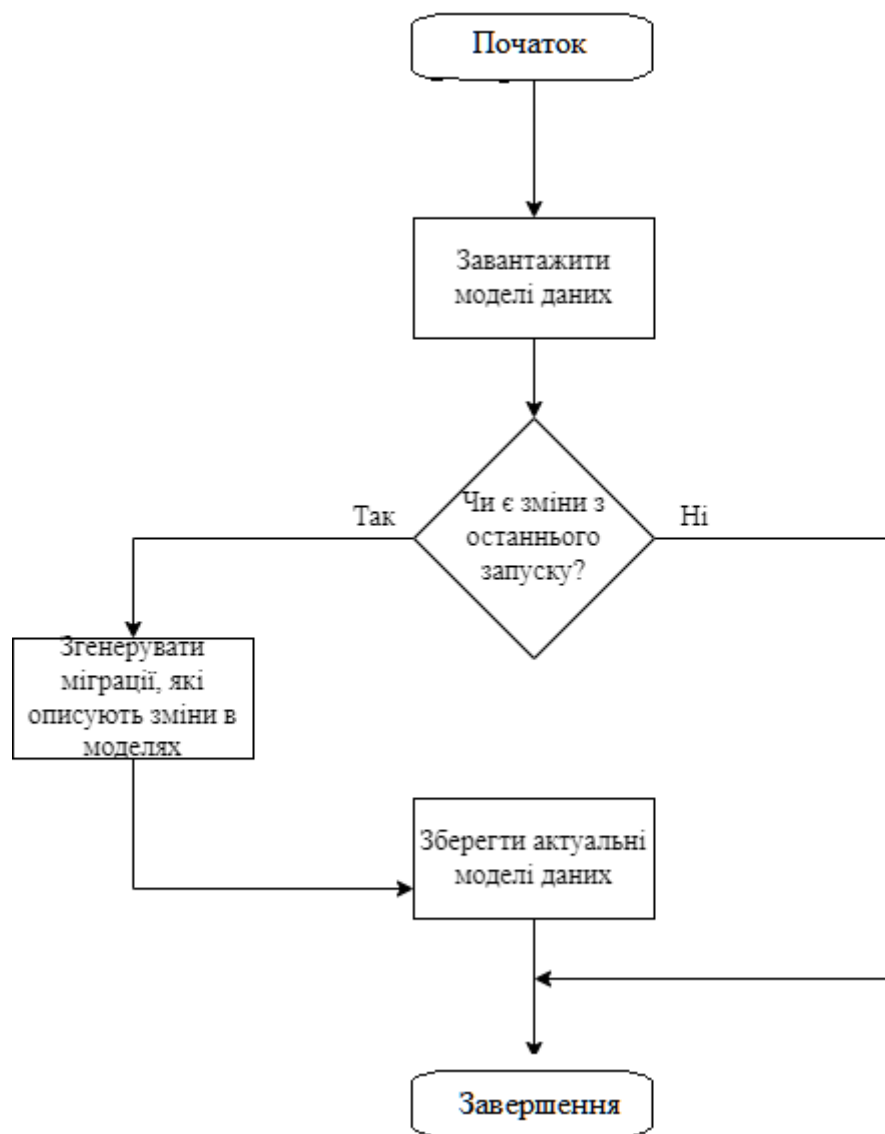


Рисунок 3.3 – Схематична версія алгоритму генерації міграцій

### 3.3. Можливості розробленої системи

Розроблена комп'ютерна система задовольняє наступні потреби розробника при роботі з базою даних:

- Відображення даних між базою та сервісом;
- Генерація міграцій;
- Виконання міграцій;
- Відміна по одній останній міграції;
- Динамічне побудування запитів до бази даних використовуючи класи;
- Декларативність опису сутностей в коді;

### 3.4. Технічні характеристики пристрою тестування

Тестування відбуватиметься на ноутбучі Lenovo Legion5 із наступними технічними характеристиками:

- ОС - Windows 10, версія 21H1;
- Процесор - Intel Core i7 із 6 ядрами та 12 потоками;
- Оперативна пам'ять - 16 ГБ;
- Фізична пам'ять - 1 ТБ HDD, 500 ГБ SSD;
- Графічна карта - NVIDIA GeForce RTX 3060, 6 ГБ відеопам'яті.

Також необхідними моментами є наявність доступу в мережу Інтернет та наявність встановленого node.js середовища. Встановити його можна з офіційного сайту. При тестуванні використовувалась версія node.js 16.3.1.

### 3.5. Тестування роботи створеної комп'ютерної системи та аналіз надійності отриманих міграцій

Процес тестування буде відбуватися наступним чином. Маючи наступний docker-compose файл (рис. 3.4), який описує інфраструктуру потрібних сервісів, запустимо його командою `docker-compose up`, щоб підняти в докері базу даних

PostgreSQL та почати працювати з нею. Консольні рядки з рис. 3.5 сигналізують, що сервер успішно піднявся, працює і готовий встановлювати підключення.

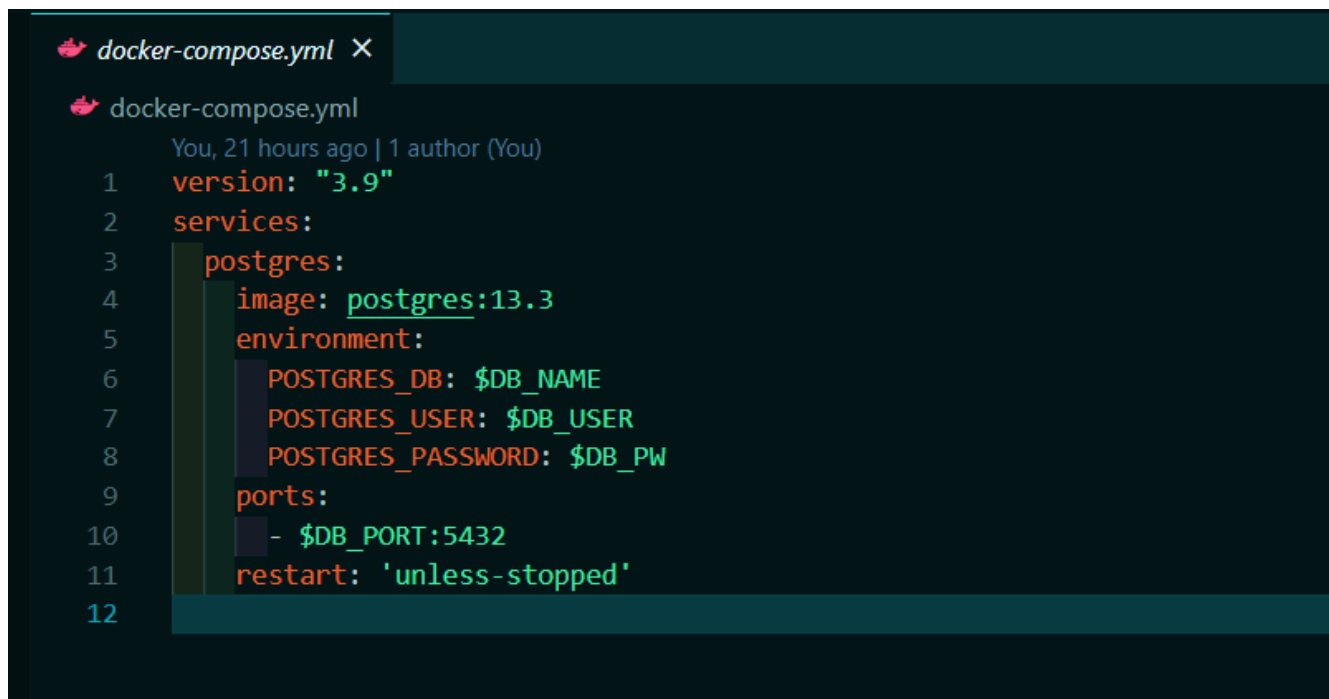


Рисунок 3.4 – Docker-compose опис

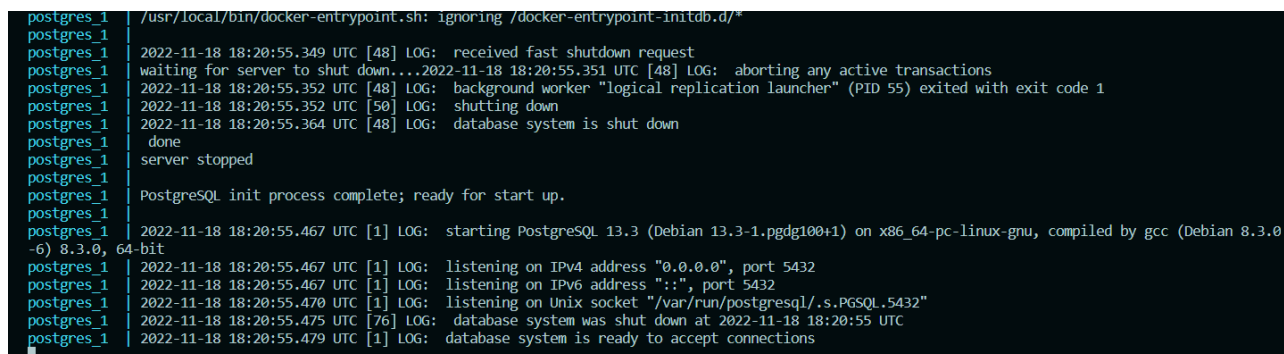


Рисунок 3.5 – Сервер PostgreSQL піднято в Docker

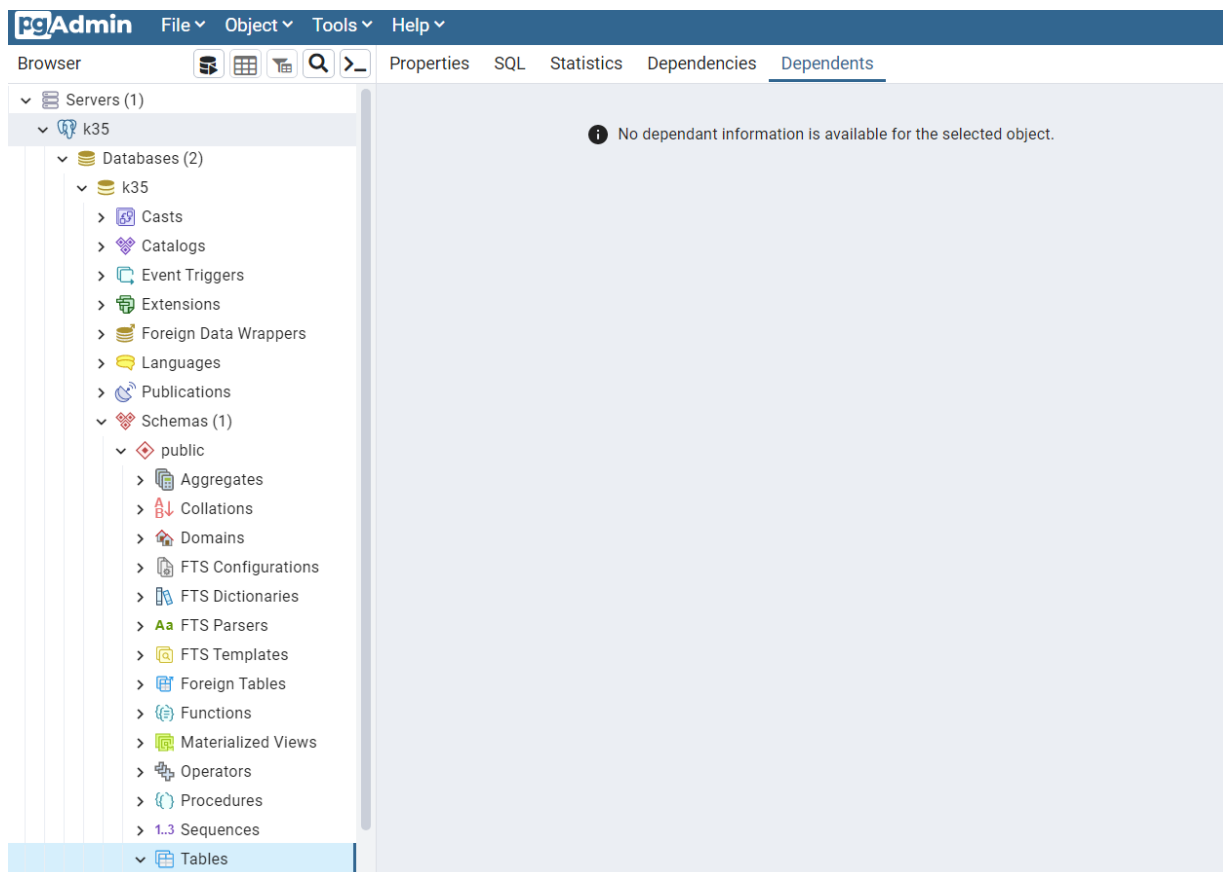


Рисунок 3.6 – Підключений до сервера бази даних клієнт PgAdmin 4

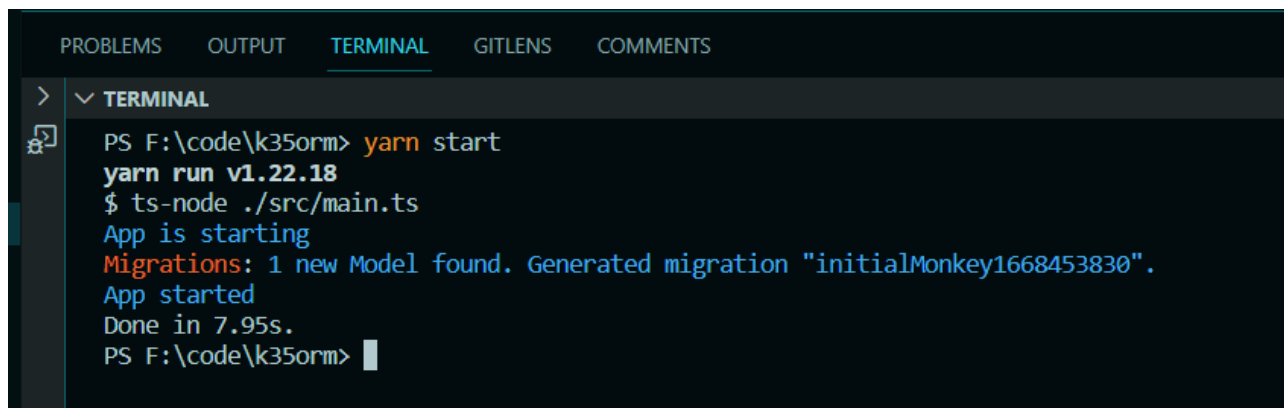
Підключившись до бази даних, можна переходити безпосередньо до тестування розробленого ПЗ.

Першим кроком створимо нову модель даних Monkey (рис. 3.7.).

```
temp > TS example.ts > Monkey
You, 16 seconds ago | 1 author (You)
1 | import { BaseModel } from '../src/core/base-model';
2 | import { Column, Model } from './decorators';
3
You, 16 seconds ago | 1 author (You)
4 | @Model({
5 |   unique_id: 0,
6 |   name: 'monkey',
7 | })
8 | export class Monkey extends BaseModel {
9 |   @Column({
10 |     unique_id: 0,
11 |     primary: true,
12 |   })
13 |   id: number;
14
15 |   @Column({
16 |     unique_id: 1,
17 |     type: 'varchar',
18 |     nullable: false,
19 |   })
20 |   name: string;
21
22 |   @Column({
23 |     unique_id: 2,
24 |     type: 'smallint',
25 |     nullable: false,
26 |   })
27 |   age: number;
28 | }
29
```

Рисунок 3.7 – Приклад. Модель Monkey

Наступним кроком, запусимо додаток командою `yarn start`. Логи описують процес ініціалізації додатка, на рис. 3.8 бачимо, що модулем міграцій було знайдено нову модель даних і на її основі було згенеровано міграцію.

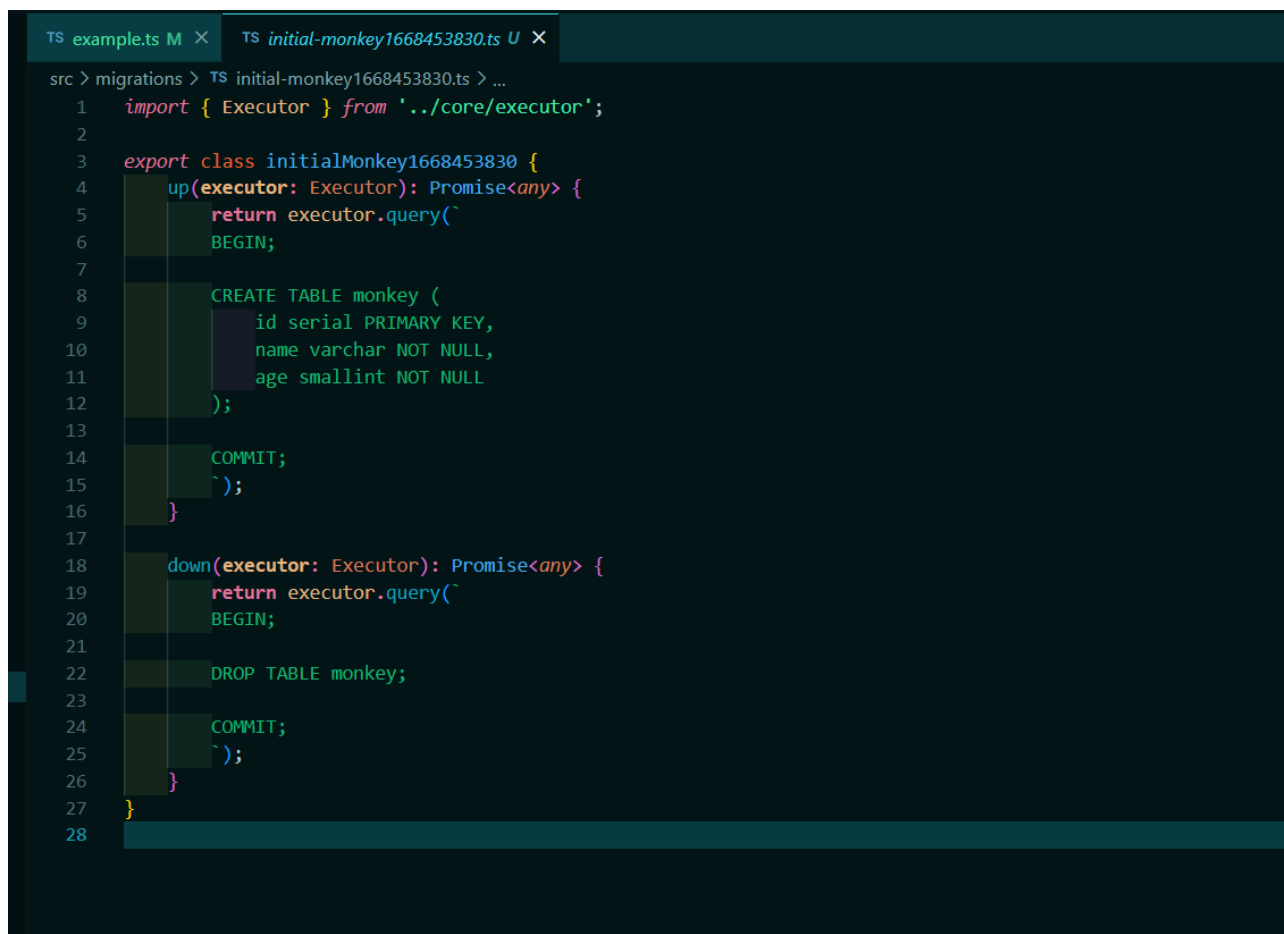
A screenshot of a terminal window with a dark background. The terminal shows the execution of the 'yarn start' command. The output includes the command being run, the version of yarn (v1.22.18), the command to run the application (ts-node ./src/main.ts), a message 'App is starting', a log entry 'Migrations: 1 new Model found. Generated migration "initialMonkey1668453830".', a message 'App started', and the execution time 'Done in 7.95s.'. The prompt 'PS F:\code\k35orm>' is visible at the bottom.

```
PS F:\code\k35orm> yarn start
yarn run v1.22.18
$ ts-node ./src/main.ts
App is starting
Migrations: 1 new Model found. Generated migration "initialMonkey1668453830".
App started
Done in 7.95s.
PS F:\code\k35orm> 
```

Рисунок 3.8 – Запуск додатка

На рис. 3.9 бачимо міграцію, яку згенерувала розроблена система. Міграція містить в собі всі атрибути, які були описані в моделі. Наявні методи `up` для використання міграції та `down` для відміни цієї міграції, якщо це знадобиться в майбутньому.

Тепер запусимо міграцію командою `yarn migrations:up` (рис. 3.10.).

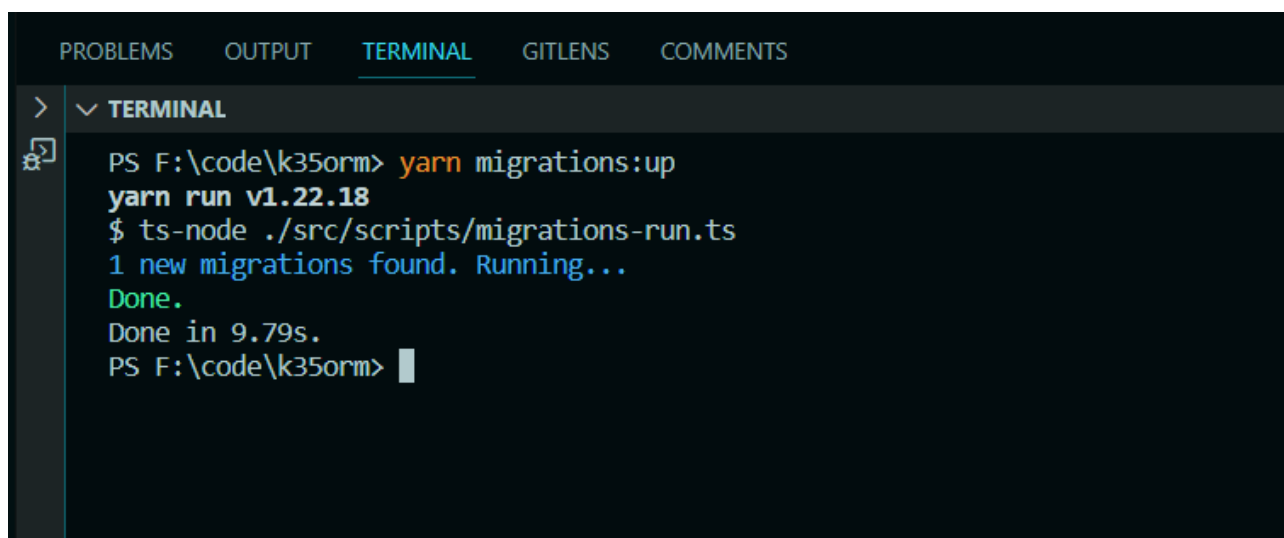


```

src > migrations > TS initial-monkey1668453830.ts > ...
1  import { Executor } from '../core/executor';
2
3  export class initialMonkey1668453830 {
4      up(executor: Executor): Promise<any> {
5          return executor.query(`
6              BEGIN;
7
8              CREATE TABLE monkey (
9                  id serial PRIMARY KEY,
10                 name varchar NOT NULL,
11                 age smallint NOT NULL
12             );
13
14             COMMIT;
15         `);
16     }
17
18     down(executor: Executor): Promise<any> {
19         return executor.query(`
20             BEGIN;
21
22             DROP TABLE monkey;
23
24             COMMIT;
25         `);
26     }
27 }
28

```

Рисунок 3.9 – Згенерована міграція для описаної моделі



```

PROBLEMS  OUTPUT  TERMINAL  GITLENS  COMMENTS
>  ✓ TERMINAL
PS F:\code\k35orm> yarn migrations:up
yarn run v1.22.18
$ ts-node ./src/scripts/migrations-run.ts
1 new migrations found. Running...
Done.
Done in 9.79s.
PS F:\code\k35orm>

```

Рисунок 3.10 – Запуск скрипту, який виконує міграції

Тепер, заглянувши в клієнт бази даних, побачимо, що була створена таблиця, яка повністю відповідає моделі в коді (рис 3.11). Крім цього, бачимо, що на рис 3.11 крім таблиці `monkey` з'явилися ще дві таблиці - `migrations` (рис. 3.12) та `models`.

Таблиця migrations зберігає в собі ім'я та час міграцій, які були виконані, таким чином, гарантуючи, що одна й та сама міграція не буде виконана двічі. Таблиця models необхідна для зберігання стану моделей, на основі цього стану в подальшому й відбувається генерація міграцій в разі змін.

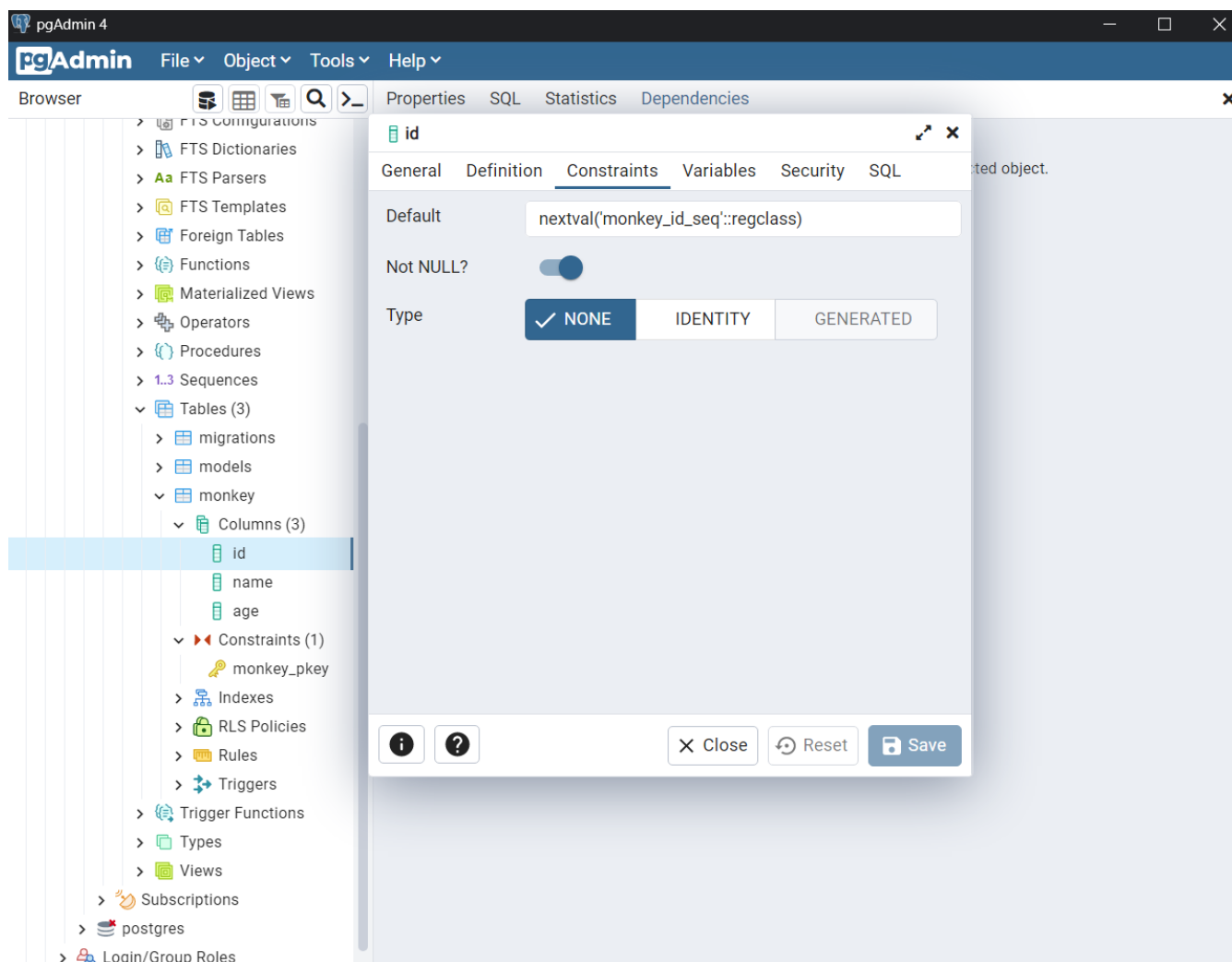


Рисунок 3.11 – Таблиця створена міграцією

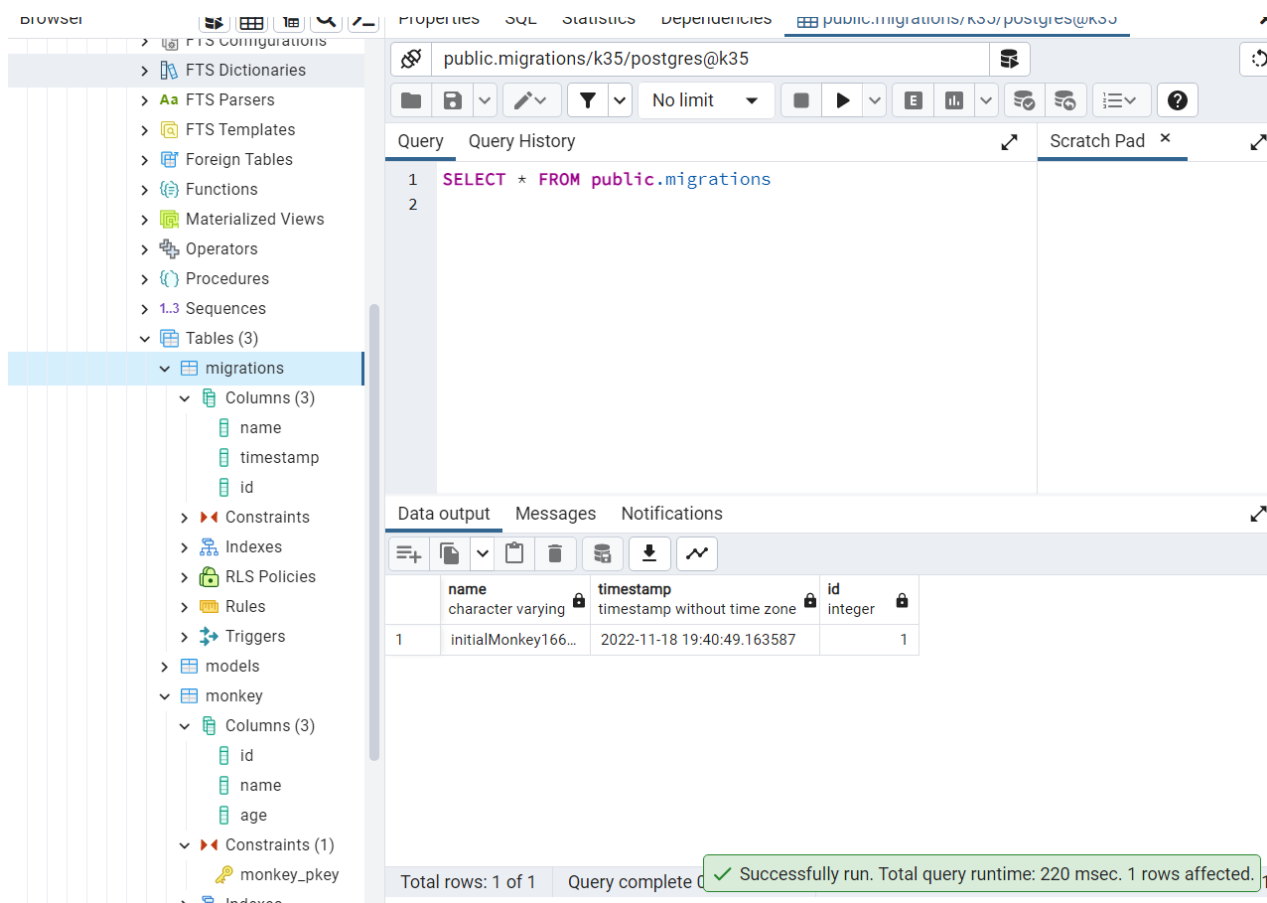


Рисунок 3.12 – Таблиця із виконаними міграціями

Тепер відмінимо міграцію, яку виконали, щоб впевнитись, що ця частина функціонала теж працює коректно (рис. 3.13 та 3.14). Бачимо, що таблиця `monkey` зникла, а з таблиці міграцій зникла міграція із назвою `initialMonkey1668453830`.

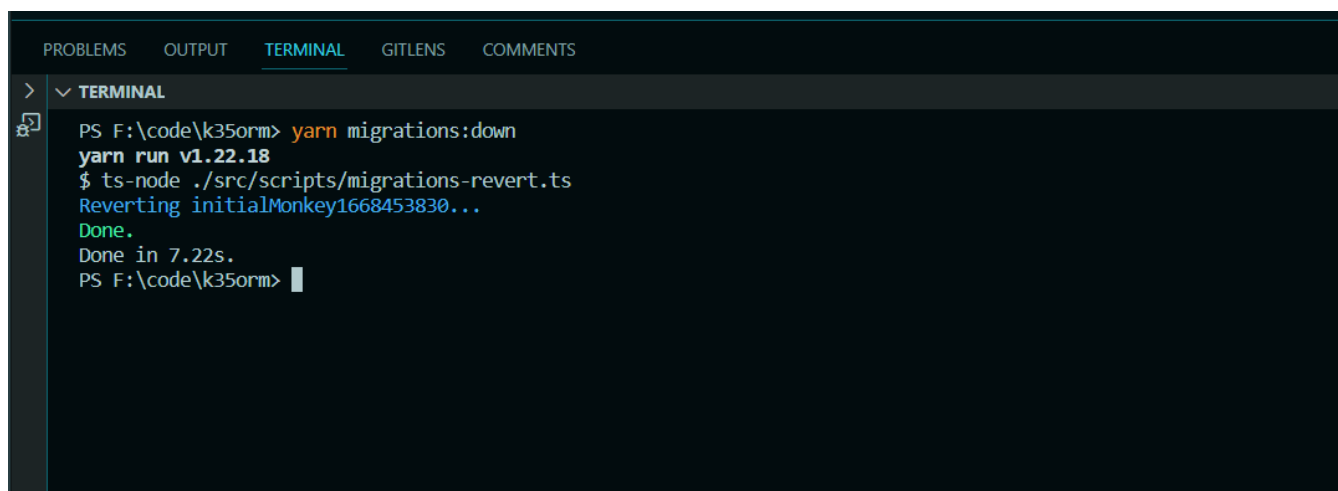


Рисунок 3.13 – Відміна останньої міграції

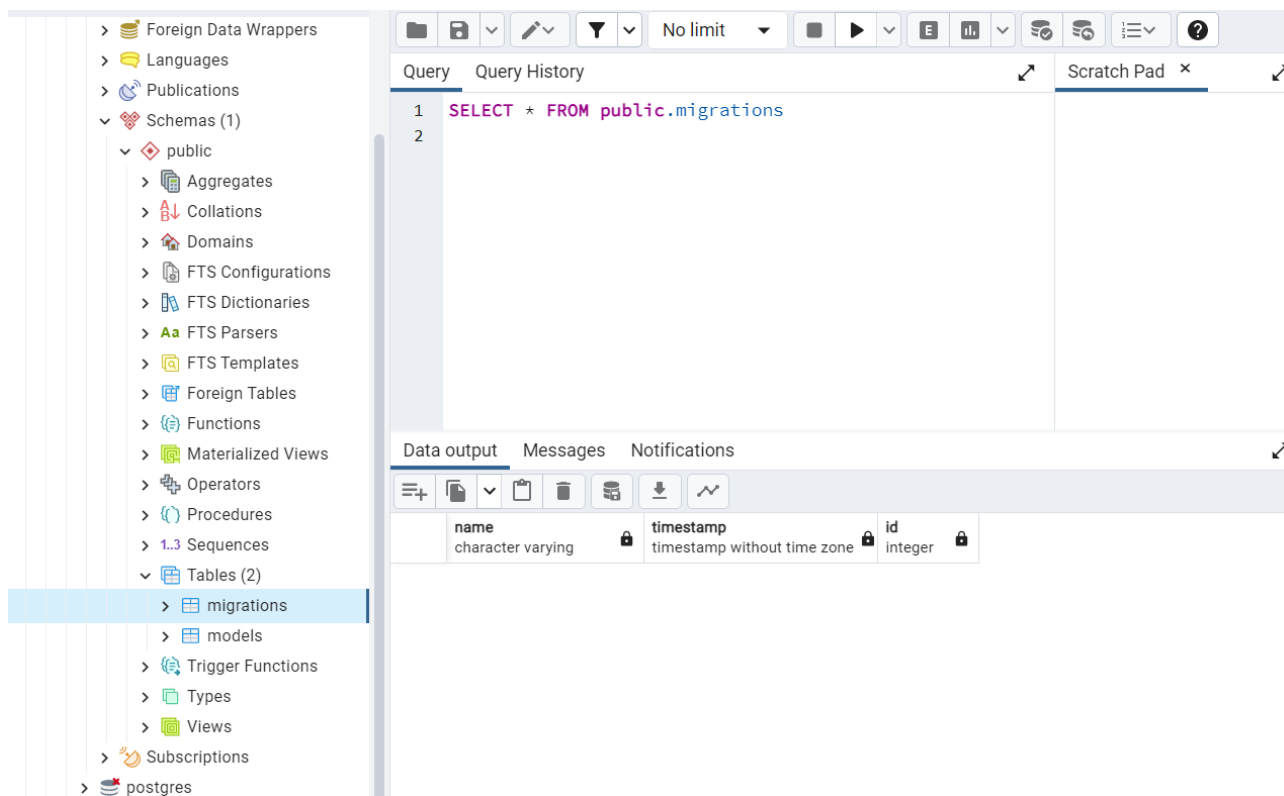


Рисунок 3.14 – База даних повернулась до попереднього стану

Розглянемо генерування міграцій на зміни у вже ініціалізованих та синхронізованих із базою даних моделях. Для цього повернемо міграцію запустив ще раз `yarn migrations:up`.

Тепер внесемо зміни до моделі `Monkey`. Наприклад, виставимо `nullable: true` для атрибутів `name` та `age`. Оновлена модель зображено на рис. 3.15.

Після чого знову запусимо додаток, щоб в процесі його запуску модуль міграцій помітив зміни. В результаті, отримано міграції, що зображено на рис 3.17. Запустимо міграції, щоб виконати їх та оновити стан бази даних.

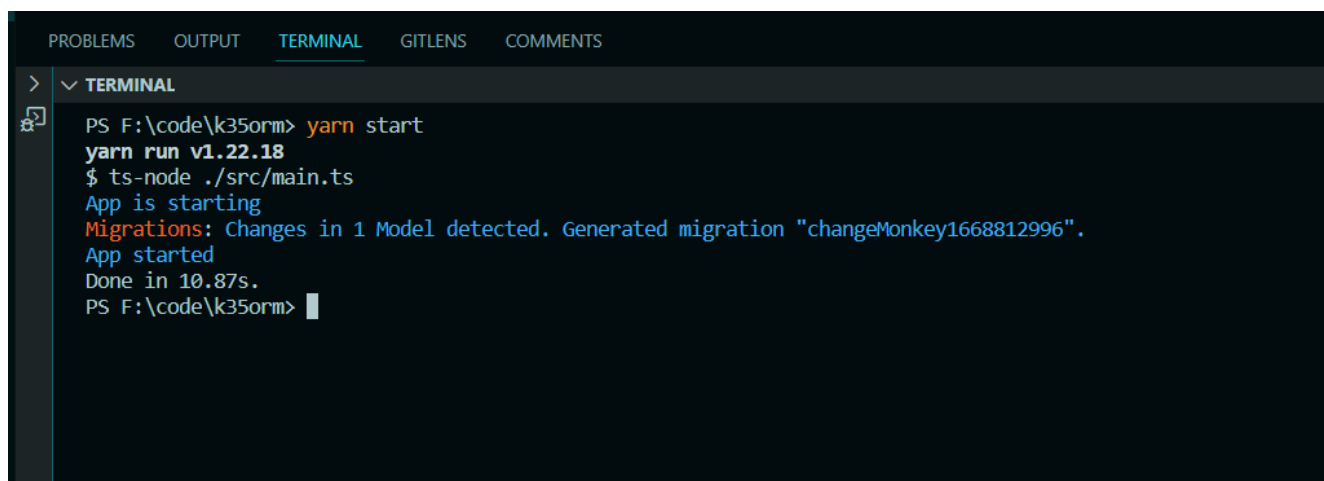
На рис 3.18 та 3.19 видно, що процес пройшов успішно і міграції були виконані.

```
export class Monkey extends BaseModel {
  @Column({
    unique_id: 0,
    primary: true,
  })
  id: number;

  @Column({
    unique_id: 1,
    type: 'varchar',
    nullable: true,
  })
  name: string;

  @Column({
    unique_id: 2,
    type: 'smallint',
    nullable: true,
  })
  age: number;
}
```

Рисунок 3.15 – Оновлена модель даних Monkey



```
PROBLEMS  OUTPUT  TERMINAL  GITLENS  COMMENTS
>  TERMINAL
PS F:\code\k35orm> yarn start
yarn run v1.22.18
$ ts-node ./src/main.ts
App is starting
Migrations: Changes in 1 Model detected. Generated migration "changeMonkey1668812996".
App started
Done in 10.87s.
PS F:\code\k35orm> 
```

Рисунок 3.16 – Генерація міграцій для змін

```

src > migrations > TS change-monkey1668812996.ts > ...
1  import { Executor } from '../core/executor';
2
3  export class changeMonkey1668812996 {
4      up(executor: Executor): Promise<any> {
5          return executor.query(`
6              BEGIN;
7
8              ALTER table monkey ALTER column name DROP NOT NULL,
9              ALTER column age DROP NOT NULL;
10
11             COMMIT;
12             `);
13     }
14
15     down(executor: Executor): Promise<any> {
16         return executor.query(`
17             BEGIN;
18
19             ALTER table monkey ALTER column name SET NOT NULL,
20             ALTER column age SET NOT NULL;
21
22             COMMIT;
23             `);
24     }
25 }
26

```

Рисунок 3.17 – Міграції з оновленням таблиці monkey

The screenshot shows the pgAdmin 4 interface. On the left, the 'Schemas (1)' tree is expanded to 'public', and the 'Tables (3)' list includes 'migrations', 'models', and 'monkey'. The 'monkey' table is selected, showing its 'Columns (3)': 'id' (integer), 'name' (character varying), and 'age' (timestamp without time zone). The 'Data output' pane at the bottom displays the results of a query: 'SELECT \* FROM public.migrations'. The results show two rows of migration data.

|   | name                     | timestamp                  | id |
|---|--------------------------|----------------------------|----|
| 1 | initialMonkey16684538... | 2022-11-18 23:11:53.311751 | 2  |
| 2 | changeMonkey166881...    | 2022-11-18 23:13:49.134307 | 3  |

Рисунок 3.18 – Міграції з оновленням таблиці monkey

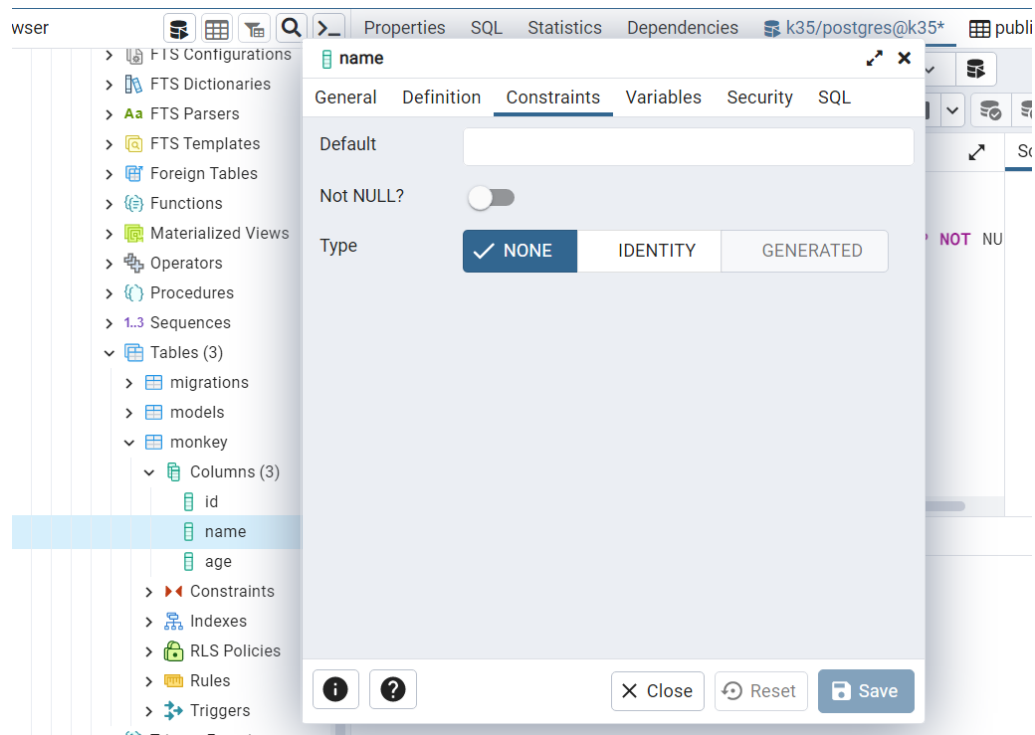


Рисунок 3.19 – Колонка name більше не має обмеження на значення NULL

Тепер перевіримо, як відбувається динамічне відображення даних між базою даних та програмним середовищем. На рис. 3.20 зображено екземпляр класу `Monkey`, в якого встановлені атрибути `age` 9 та `name` `Fluffy`. Після виклику метода `save`, який є в моделі від класу-протитипу `BaseModel`, цей екземпляр класу було збережено до бази даних в таблицю `monkey`, з приведенням типів згідно метаданих моделі, що можна бачити на рис. 3.21.

```
const monkey = new Monkey();

monkey.age = 9;
monkey.name = 'Fluffy';

monkey.save();
```

Рисунок 3.20 – Екземпляр класу `Monkey`

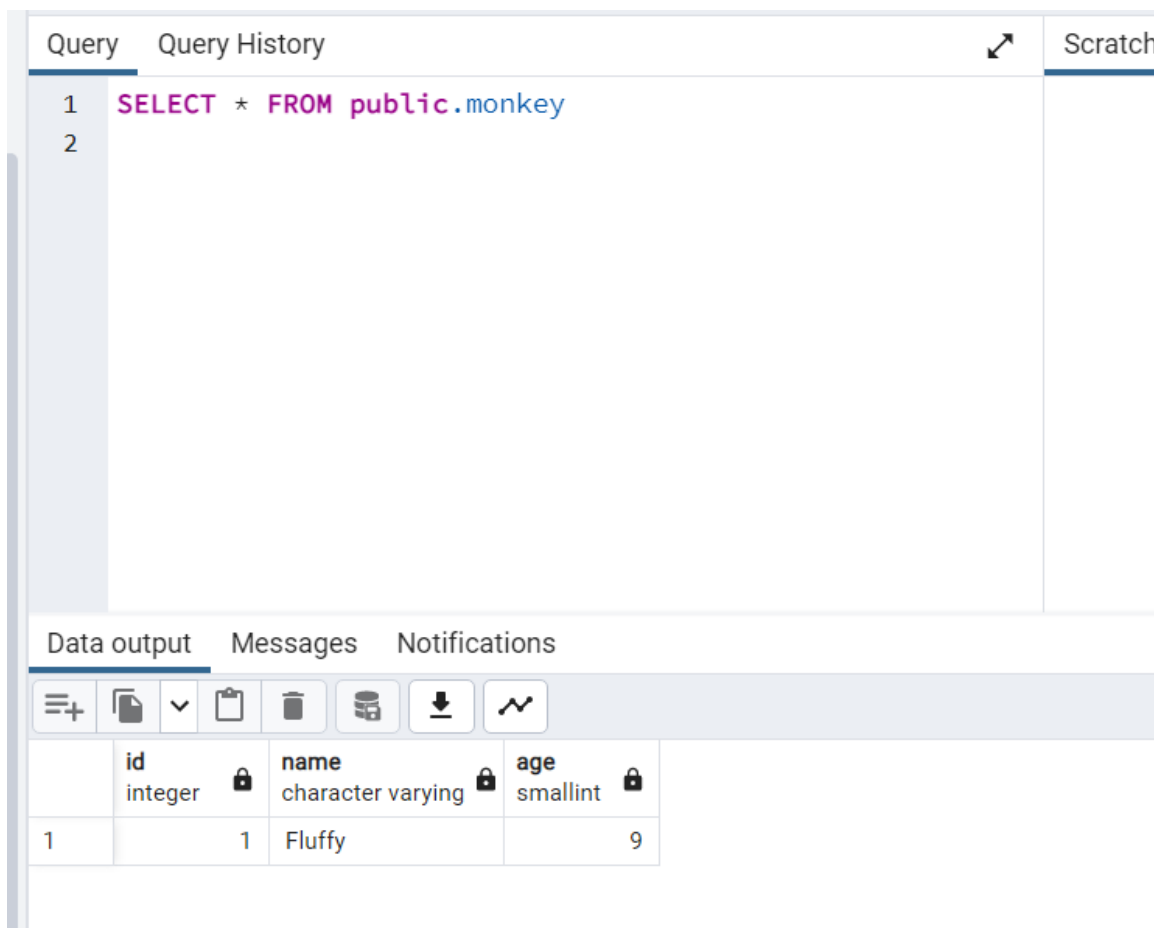


Рисунок 3.21 – Екземпляр класу Monkey збережений в базу даних

### 3.6. Порівняння результатів комп'ютерної системи з очікуваними результатами

Проведено заміри параметрів швидкодії та ефективності запитів, які були згенерованих системою. Було створено простий веб-сервер, де з localhost адреси надсилалися запити до сервера, який в свою чергу використовував розроблену систему.

Для цього тестування було використано наступну схему бази даних (рис. 3.22 та рис. 3.23).



Рисунок 3.22 – Схематичне зображення бази даних

```

1  Table queues {
2    id int [pk]
3    doctor varchar
4    date timestamp
5  }
6
7  Table patients {
8    id int [pk]
9    name varchar
10 }
11
12 Table queue_patient {
13   queue_id int [ref: > queues.id]
14   patient_id int [ref: > patients.id]
15 }
16

```

Рисунок 3.23 – Опис схеми бази даних псевдокодом

Таблиця 3.1 - Середня затримка, мс

|                |         |
|----------------|---------|
| Простий GET    | 1431.41 |
| Вкладений GET  | 4591.19 |
| Вкладений POST | 719.17  |

Таблиця 3.2 - Середня кількість запитів в секунду

|                |      |
|----------------|------|
| Простий GET    | 41.9 |
| Вкладений GET  | 13   |
| Вкладений POST | 83.4 |

Аналізуючи результати з табл. 3.1 та табл. 3.2 можна зробити висновки, що прості GET та вкладені POST запити показують хороші результати, а от вкладені GET запити демонструють результати трохи гірші за бажані, що пов'язано з не найефективнішими за своєю структурою запитами, побудовані QueryBuilder, що своєю чергою призводить до не самих оптимізованих планів виконання запиту, які створює планувальник PostgreSQL. В майбутньому необхідно оптимізувати цей процес, щоб досягти бажаного результату. Графік порівняння з іншими існуючими системами наведені на рис. 3.24, рис. 3.25 та рис. 3.26.

Що стосується самого процесу автоматичної генерації міграцій під час запуску сервісу, які призначені полегшити життя розробника, важко заміряти ефективність та об'єктивно оцінити пришвидшення процесу розробки, адже це не якийсь функціональний алгоритм, що виконується машиною. Міграції зазвичай пишуться людиною, і час, необхідний для цього варіюється від розробника до розробника. На це впливають різні фактори, наприклад, мотивація розробника, загальна швидкість його роботи тощо. Тож не можливо емпірично схарактеризувати покращення таким чином, щоб результат був справедливим для всіх розробників. На особистому прикладі, згідно із замірами та порівняннями, економія часу складає в середньому 12%, проте варто ще раз наголосити, що це

індивідуальний показник і не можна результати якоїсь однієї людини екстраполювати на всіх.

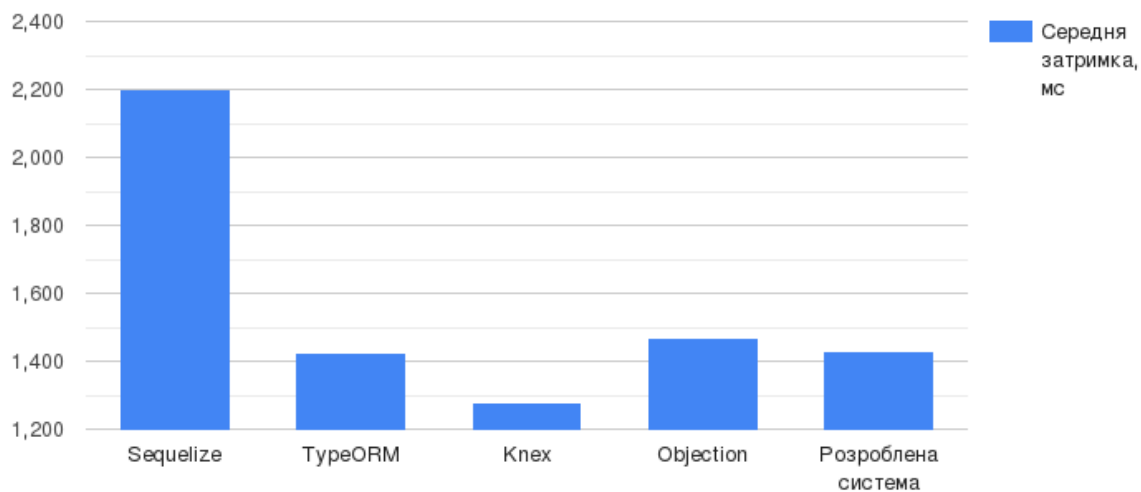


Рисунок 3.24 – Порівняння затримки виконання простого GET

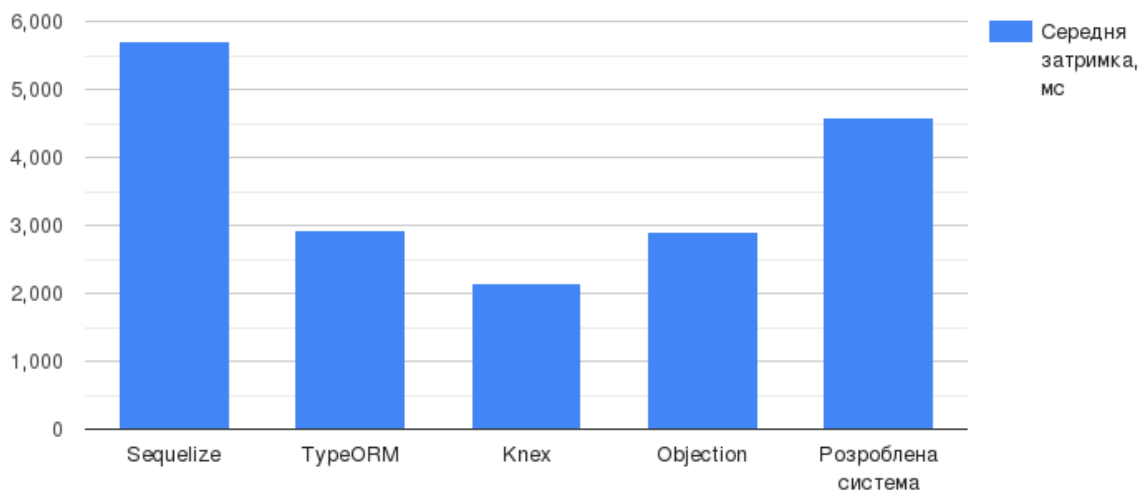


Рисунок 3.25 – Порівняння затримки виконання вкладеного GET

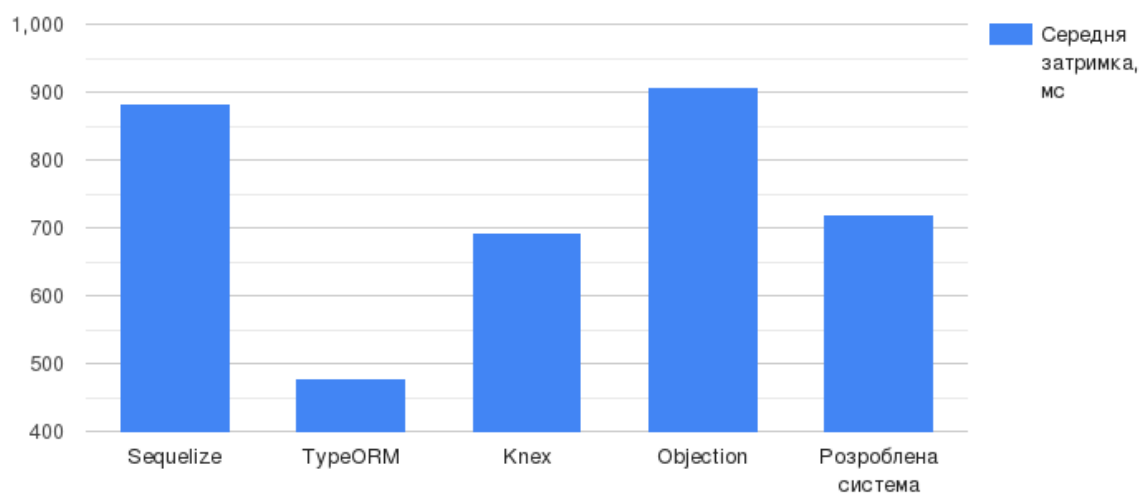


Рисунок 3.26 – Порівняння затримки виконання вкладеного POST

## ВИСНОВОК ДО ТРЕТЬОГО РОЗДІЛУ

Третій розділ присвячено розробці та подальшому тестуванню запропонованого способу. Описано розроблений програмний продукт з архітектурної точки зору та принцип його роботи, зазначено які в проєкті наявні модулі та яке їх призначення. Детально описано алгоритм, за яким відбувається автоматична генерація міграцій для модифікації стану бази даних на основі змін в моделях даних. Також перелічено усі можливості, які на цей момент надає розроблена система. Також було описано технічні характеристики пристрою тестування. Протестовано наступний функціонал системи: автоматична генерація міграцій, запуск міграцій, відміна міграцій та об'єктно-реляційна проєкція даних між базою даних та сервісом. Також задокументовано процес тестування в деталях. Вихідні міграції показали себе надійними, такими, що зберігають консистентність бази даних. Виконано порівняння з існуючими системами часу, необхідного для виконання запитів до бази даних що є одним з основних функціоналів систем такого роду, де розроблений програмний продукт показав себе приблизно на одному рівні з існуючими, проте все ж залишається простір для покращень.

## ВИСНОВКИ

В процесі виконання магістерської дисертації, було розглянуто загальні принципи роботи комп'ютерних систем об'єктно-реляційної проєкції. Порівняно між собою популярні існуючі системи об'єктно-реляційного відображення для мови програмування TypeScript. Розроблено прототип власної системи, в якій вперше запропоновано можливість автоматичної генерації міграцій, на основі змін внесених в моделі даних. Це необхідно для полегшення роботи розробників шляхом уникнення потреби робити це вручну кожен раз самостійно, адже це забирає зайвий час.

В першому розділі розглянуто теоретичні засади, необхідні для розуміння проблеми яку вирішують системи об'єктно-реляційної проєкції, їх переваги та недоліки, загальні принципи їх роботи тощо. Було виконане порівняння швидкодії деяких популярних систем в рамках простого веб-сервісу.

В другому розділі акцентована увага на виборі інструментів для реалізації власного програмного забезпечення, зроблено детальний опис цих інструментів та можливостей, які вони надають.

В третьому розділі приділено увагу розробленому прототипу комп'ютерної системи об'єктно-реляційного відображення, описано архітектуру розробки, алгоритми, які він використовує для досягнення мети магістерської дисертації, а також задокументовано процес тестування розробки, описано кроки які виконувались для цього, технічні характеристики машини, на якій проходило тестування. Кожен крок підкріплений рисунками, які демонструють процес тестування. В кінці розділу проведено тестування швидкодії розробленого прототипу на базі веб-серверу. Результати швидкодії були порівняні зі значеннями існуючих систем та зображено на графіках. В цілому, швидкодія вже імплементованих запитів знаходиться приблизно на тому ж рівні, що й в існуючих системах, що безумовно є хорошим показником, проте все ж є простір для покращень, тому як розробка поки що знаходиться на стадії робочого прототипу.

Вихідна комп'ютерна система об'єктно-реляційного відображення даних дозволяє розробникам прискорити процес розробки додатків, які взаємодіють із реляційними базами даними, завдяки економії часу на написанні міграцій власноруч. В середньому, це допоможе зекономити від 8 до 15 відсотків часу необхідних на задачі, що включають в себе внесення змін в архітектуру бази даних, в залежності від особливостей швидкості роботи розробника, його мотивації тощо.

В найближчому майбутньому, було б корисно продовжити займатись оптимізацією генеруємих запитів під час використання QueryBuilder та додати підтримку таких СУБД як MySQL, SQLite та MariaDB, попрацювати із кешуванням.

## СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. E. Codd. A Relational Model of Data for Large Shared Data Banks. [Електронний ресурс]. — Режим доступу: <https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf>
2. C. J. Date; H. Darwen. The Third Manifesto. 2007.
3. Ole-Johan Dahl; Kristen Nygaard. The Development of the Simula Languages. [Електронний ресурс]. — Режим доступу: [https://hannemyr.com/cache/knojd\\_acm78.pdf](https://hannemyr.com/cache/knojd_acm78.pdf)
4. Classes. The Java Tutorials. [Електронний ресурс]. — Режим доступу: <https://docs.oracle.com/javase/tutorial/java/javaOO/classes.html>
5. Armstrong, Deborah J. The quarks of object-oriented development. [Електронний ресурс]. — Режим доступу: <http://portal.acm.org/citation.cfm?id=1113040>
6. Abstract classes (C++). Microsoft. [Електронний ресурс]. — Режим доступу: <https://learn.microsoft.com/en-us/cpp/cpp/abstract-classes-cpp?view=msvc-170>
7. Connolly, Thomas M.; Begg, Carolyn E. Introduction to Object DMBS § Object-oriented concepts. Database systems: a practical approach to design, implementation, and management. 2005.
8. C. Strachey. Fundamental Concepts in Programming Languages. [Електронний ресурс]. — Режим доступу: [https://www.ics.uci.edu/~jajones/INF102-S18/readings/05\\_strachey\\_1967.pdf](https://www.ics.uci.edu/~jajones/INF102-S18/readings/05_strachey_1967.pdf)
9. Java Specification Requests. Sun Microsystems. [Електронний ресурс]. — Режим доступу: <http://jcp.org/en/jsr/all>
10. What's New in the .NET Framework Version 2.0. Microsoft. [Електронний ресурс]. — Режим доступу: [https://learn.microsoft.com/en-us/previous-versions/t357fb32\(v=vs.100\)](https://learn.microsoft.com/en-us/previous-versions/t357fb32(v=vs.100))
11. C. J. Date; H. Darwen. A Guide to the SQL standard: a users guide to the standard database language SQL. 1997.
12. SQL Examples. W3schools. [Електронний ресурс]. — Режим доступу: [https://www.w3schools.com/sql/sql\\_examples.asp](https://www.w3schools.com/sql/sql_examples.asp)

13. Jim Melton. Advanced SQL:1999: Understanding Object-Relational and Other Advanced Features. 2002.
14. Michael M. Gorman. Is SQL a real standard anymore? [Электронный ресурс]. — Режим доступа: <http://www.tdan.com/i016hy01.htm>
15. Peter Gultzan. Standard SQL. [Электронный ресурс]. — Режим доступа: <http://dbazine.com/db2/db2-disarticles/gultzan3>
16. Gary Orenstein, Conor Doherty, Kevin White, Steven Camina. Building Real-Time Data Pipelines. 2015.
17. Database relationships. IBM Documentation. IBM. [Электронный ресурс]. — Режим доступа: <https://www.ibm.com/docs/en/mam/7.6.0?topic=structure-database-relationships>
18. Pankaj. SQL Data Types. 2022. [Электронный ресурс]. — Режим доступа: <https://www.digitalocean.com/community/tutorials/sql-data-types>
19. Josh Berkus. Wrecking Your Database. Серпень, 2009. [Электронный ресурс]. — Режим доступа: <https://www.toolbox.com/tech/data-management/blogs/wrecking-your-database-080509>
20. Documentation. Sequelize. [Электронный ресурс]. — Режим доступа: <https://sequelize.org/docs/v7/other-topics/typescript/>
21. Documentation. TypeScript. [Электронный ресурс]. — Режим доступа: <https://www.typescriptlang.org/docs/handbook/intro.html>