ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Метод відображення сутностей бази даних PostgreSQL в прикладний код на платформі NodeJS»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Нікіта ВАСИЛІНЕНКО
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-61
Нікіта ВАСИЛІНЕНКО

Керівник: Владислав ЯСКЕВИЧ
к.т.н.,

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
«_____» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Василіненку Нікіті Максимовичу

1. Тема кваліфікаційної роботи: «Метод відображення сутностей бази даних PostgreSQL в прикладний код на платформі NodeJS»

керівник кваліфікаційної роботи Владислав ЯСКЕВИЧ, к.т.н., доцент, доцент кафедри ІПЗ,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» жовтня 2024 р. № 320.

2. Строк подання кваліфікаційної роботи «17» грудня 2024 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, вимоги до модульності та масштабованості систем, методи відображення сутностей бази даних в прикладному коді.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд існуючих методів проекції даних: Аналіз традиційних підходів та їх обмежень.
2. Розробка методу проекції на основі метаданих: Ідея.

3. Розробка методу проекції на основі метаданих: Імплементація та виміри результатів.

5. Перелік ілюстративного матеріалу: *презентація*
   1. Діаграми архітектури системи: Відображення структури проектів на основі DTO та метаданих.
   2. Схеми взаємодії компонентів: Як процеси відображення інтегруються в загальну систему.
   3. Приклади коду: Демонстрація різниці між DTO та метаданими у реальних сценаріях.
   4. Результати вимірів: Порівняння об’єму коду та гнучкості обох підходів.
   5. Демонстрація роботи методу на реальних прикладах.
   6. Візуалізації даних: Графічне представлення аналізу методів.

6. Дата видачі завдання «16» жовтня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	16.10-23.10.2024	
2	Вивчення матеріалів для аналізу розвитку технологій багатокамерного трекінгу об’єктів	24.10-31.10.2024	
3	Дослідження методів відображення даних в прикладний код	01.11-08.11.2024	
5	Застосування набутих знань в розробці методу відображення сутностей бази даних в прикладний код	09.11-19.11.2024	
6	Написання двох програм для демонстрації переваг методу в об’єму коду і гнучкості	20.11-23.11.2024	
7	Оформлення роботи: вступ, висновки, реферат	24.11-30.11.2024	
8	Розробка демонстраційних матеріалів	01.12-05.12.2024	
9	Попередній захист роботи	25.11-06.12.2024	

Здобувач вищої освіти _____ Нікіта Василіненко
(підпис)

Керівник
кваліфікаційної роботи _____ Владислав ЯСКЕВИЧ
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 75 стор., 5 табл., 37 рис., 6 джерел.

Мета роботи – покращення технології для відображення сутностей бази даних у прикладний код.

Об'єкт дослідження – Процес відображення сутностей бази даних у прикладний код.

Предмет дослідження – Технології і засоби відображення сутностей бази даних у прикладний код.

У роботі використано різноманітні методи, такі як метапрограмування, RBAC (Role-Based Access Model), ООП, тришарова архітектура.

Проведено аналіз сучасних методів для відображення сутностей бази даних у прикладний код

Розроблено та оптимізовано метод відображення сутностей бази даних в прикладний код на основі інтеграції метаданих в RBAC.

Проведено експерименти для традиційного і запропонованого методу відображення сутностей бази даних в прикладний код на основі двох ідентичних по функціональності API (Application Programming Interface) розроблених в межах кваліфікаційної роботи.

КЛЮЧОВІ СЛОВА: ПРОЕКЦІЯ ДАНИХ, МЕТАДАНІ, МЕТАПРОГРАМУВАННЯ, RBAC, ТРИШАРОВА АРХІТЕКТУРА.

ABSTRACT

Text part of the master's qualification work: 75 pages, 37 pictures, 5 tables, 6 sources.

The purpose of the work is to improve the technology for mapping database entities into application code.

Object of research – is the process of mapping database entities into application code.

Subject of research – is the technologies and tools for mapping database entities into application code.

Various methods such as metaprogramming, RBAC (Role-Based Access Model), OOP (Object-Oriented Programming), and three-tier architecture were used in the work.

An analysis of modern methods for mapping database entities into application code was conducted.

A method for mapping database entities into application code based on the integration of metadata into RBAC was developed and optimized.

Experiments were conducted for the traditional and proposed methods of mapping database entities into application code based on two functionally identical APIs (Application Programming Interfaces) developed within the thesis.

KEYWORDS: DATA PROJECTION, METADATA, METAPROGRAMMING, RBAC, THREE-TIER ARCHITECTURE.

ЗМІСТ

ВСТУП

Інформаційні технології сьогодні є рушієм прогресу у всіх сферах життя – від бізнесу та фінансів до охорони здоров’я та державного управління. Збільшення обсягів даних, які обробляються в цифрових системах, і постійне зростання кількості користувачів вимагають більш гнучких та ефективних підходів до управління інформацією. Одним із ключових аспектів є контроль доступу до даних, що дозволяє забезпечити їхню безпеку, цілісність та конфіденційність. У багатокористувацьких системах важливо не лише захистити інформацію, але й організувати її таким чином, щоб різні користувачі мали доступ до необхідних даних відповідно до своїх повноважень.

Ця проблема є особливо актуальною для сучасних програмних систем, де рольова модель управління доступом (RBAC) виступає важливим інструментом для забезпечення безпеки. Однак традиційні методи контролю, такі як використання DTO (Data Transfer Object), часто призводять до значних витрат часу та ресурсів, ускладнюють масштабування системи і знижують її гнучкість. Саме тому виникає потреба у нових підходах, що дозволяють більш ефективно управляти доступом на рівні окремих властивостей об’єктів, зберігаючи при цьому простоту підтримки та високу продуктивність системи.

Актуальність дослідження. У сучасному світі інформаційні технології стали невід'ємною частиною будь-якої сфери діяльності, починаючи від бізнесу і закінчуючи державною службою. З кожним роком зростає обсяг даних, якими оперують організації, і паралельно з цим ускладнюються процеси управління ними. Важливим аспектом стає захист інформації та правильна організація доступу до неї. У багатокористувацьких системах управління доступом є критично важливим компонентом для забезпечення безпеки та ефективного функціонування. З огляду на збільшення кількості користувачів і ролей у таких системах, зростає потреба у розробці гнучких та масштабованих методів контролю доступу.

Традиційні підходи, зокрема через використання DTO (Data Transfer Object), мають низку недоліків. Вони передбачають створення окремих структур для кожної ролі, що призводить до зростання обсягу коду, ускладнення підтримки та зменшення гнучкості системи. В умовах сучасного розробницького середовища така практика стає менш ефективною. Тому актуальним є пошук нових рішень, які дозволять динамічно адаптувати систему до змін, зменшити дублювання коду та забезпечити високий рівень безпеки.

Мета - покращення підходу до відображення даних.

Об'єкт дослідження. Об’єктом дослідження є процес управління доступом до даних у програмних системах, що функціонують у багатокористувацькому середовищі. У центрі уваги перебуває механізм контролю доступу до окремих властивостей об'єктів за допомогою рольової моделі RBAC (Role-Based Access Control).

Завдання дослідження. Основними завданнями дослідження є:

- Аналіз існуючих підходів до управління доступом у багатокористувацьких системах.
- Виявлення недоліків та обмежень традиційних методів на основі DTO.
- Розробка механізму відображення даних через метадані на рівні окремих властивостей об’єктів.
- Реалізація та тестування запропонованого підходу на прикладі API, створеного з використанням Nest.js та PostgreSQL.
- Порівняння результатів із традиційними підходами та оцінка ефективності запропонованої моделі.

Методи дослідження. У ході роботи були застосовані методи аналізу та синтезу для вивчення існуючих підходів до управління доступом у програмних системах. Для моделювання системи використовувалися сучасні технології, зокрема Nest.js та PostgreSQL. Практична частина дослідження включає в себе створення двох ідентичних CRUD API, які працюють на основі різних підходів до управління доступом, із подальшим порівнянням результатів.

Наукова новизна отриманих результатів. Новизна дослідження полягає в тому, що запропоновано новий підхід до управління доступом у багатокористувацьких системах, який базується на використанні метаданих для контролю доступу на рівні окремих властивостей об’єктів. На відміну від традиційних методів DTO, цей підхід дозволяє динамічно змінювати права доступу без внесення змін до бізнес-логіки чи структури бази даних.

Новизна дослідження. Запропонований підхід дозволяє значно знизити кількість коду, необхідного для реалізації механізму контролю доступу. Це досягається шляхом централізованого зберігання правил доступу у вигляді метаданих та їх динамічної обробки сервером. Практична реалізація продемонструвала, що кількість ендпоінтів зменшується вчетверо, а кількість коду – майже втричі порівняно з підходом DTO.

Результати дослідження можуть бути використані для розробки великих багатокористувацьких систем, де є потреба у масштабуванні та забезпеченні високого рівня безпеки. Запропонований підхід забезпечує більш ефективне використання ресурсів і спрощує підтримку програмного забезпечення, знижуючи витрати на його розробку та оновлення.

1 ОГЛЯД МЕТОДІВ ВІДОБРАЖЕННЯ ДАНИХ І ПІДХОДІВ ДО УПРАВЛІННЯ ДОСТУПОМ У БАГАТОКОРИСТУВАЦЬКИХ СИСТЕМАХ

1.1 Поняття та принципи відображення даних

У сучасних програмних системах відображення даних є одним із найважливіших процесів, що забезпечує ефективну взаємодію між різними рівнями архітектури додатку. Цей процес дозволяє трансформувати дані, що зберігаються у базах даних, у форму, зручну для використання бізнес-логікою або відображення на клієнтській стороні. Відображення даних є фундаментальною частиною будь-якої інформаційної системи, адже саме через цей механізм користувачі отримують доступ до інформації та здійснюють управління нею.

Процес відображення даних охоплює кілька рівнів трансформації: від фізичного рівня збереження інформації у базі даних до рівня бізнес-логіки та представлення результатів користувачу. Ефективне відображення даних дозволяє підвищити продуктивність додатку, мінімізувати помилки та знизити час відгуку системи.

1.1.1 Відображення даних: історія та розвиток

Поняття відображення даних бере свій початок з ранніх етапів розвитку інформаційних технологій. Перші програмні системи створювалися з фіксованими структурами, що ускладнювало масштабування та зміну бізнес-логіки. Проте із розвитком реляційних баз даних та об’єктно-орієнтованого програмування почали з’являтися більш гнучкі методики трансформації та передачі даних.

Зокрема, об’єктно-реляційне відображення (ORM)[1-4] стало відповіддю на проблему трансляції даних між реляційними базами даних і об'єктами у програмному коді. У той же час з'явився підхід DTO (Data Transfer Object)[5], який спрощував передачу даних між рівнями додатку та клієнтом. Сучасні фреймворки,

такі як Nest.js[6], Django[7], Spring[8], активно використовують ці підходи для оптимізації роботи з базами даних та управління доступом до них.

1.1.2 Класифікація відображення даних

Відображення даних можна умовно розділити на кілька рівнів:

- Фізичне відображення – визначає, як дані зберігаються у базі даних або файловій системі. Це найнижчий рівень, де важливу роль відіграють оптимізація запитів та структура таблиць або документів.
- Логічне відображення – забезпечує трансформацію даних між базою даних та бізнес-логікою програми. Логічне відображення дозволяє отримувати дані у вигляді об'єктів або масивів, використовуючи механізми ORM або SQL-запити.
- Відображення на рівні бізнес-логіки – на цьому рівні дані обробляються, перетворюються та готуються для подальшої передачі на клієнтський рівень або для внутрішнього використання. DTO є типовим прикладом відображення на цьому рівні.
- Відображення на рівні представлення (Frontend) – дані передаються у вигляді JSON[9], XML[10] або інших форматів, які обробляються та відображаються на клієнтській стороні.

1.1.3 Роль відображення даних у сучасних системах

У сучасних системах відображення даних виконує кілька важливих функцій:

- Оптимізація продуктивності: Правильна трансформація даних дозволяє знизити навантаження на сервер та зменшити час обробки запитів.
- Інкапсуляція бізнес-логіки: Дані не передаються користувачу в сирому вигляді, що знижує ризик витоку конфіденційної інформації та підвищує безпеку додатку.
- Гнучкість: Відображення даних дозволяє легко адаптувати систему до змін у базі даних без необхідності переписування всієї бізнес-логіки або API.

1.1.4 Зв’язок відображення даних із рівнями архітектури

Відображення даних є невід'ємною частиною кожного рівня програмної архітектури. У більшості сучасних додатків архітектура побудована за принципом трьох рівнів:

- Frontend (клієнтська частина) – відповідає за візуалізацію даних та взаємодію користувача із системою.
- Backend (серверна частина) – обробляє запити, виконує бізнес-логіку та формує дані для клієнта.
- Database (база даних) – зберігає інформацію та забезпечує швидкий доступ до неї.

Відображення даних відбувається на кожному з цих рівнів. Наприклад, серверна частина отримує дані з бази даних, перетворює їх у DTO або інший об’єкт, а потім передає на клієнтський рівень у вигляді JSON. На клієнті ці дані можуть бути додатково оброблені та виведені у вигляді графіків, таблиць або списків.

Розглянемо приклад відображення даних у системі керування товарами в інтернет-магазині.

1. Користувач запитує список товарів через API.
2. Сервер виконує SQL-запит[11-13] до бази даних та отримує результат у вигляді таблиці.
3. Результати перетворюються у DTO-об'єкти, які містять лише необхідні поля (наприклад, назва товару, ціна та зображення).
4. DTO об'єкти передаються клієнту у вигляді JSON-масиву.
5. Клієнтська частина візуалізує дані у вигляді карток товарів.

1.2 Традиційні підходи до відображення даних

Відображення даних у програмних системах є критичним етапом, який визначає, як інформація зберігається, обробляється та передається на різні рівні додатку. У багатокористувацьких системах, де безпека та ефективність мають першочергове значення, вибір методу відображення даних суттєво впливає на

продуктивність і масштабованість. У цьому розділі розглянуто традиційні підходи до відображення даних, такі як DTO (Data Transfer Object) та ORM (Object-Relational Mapping), їхні переваги, недоліки та приклади використання.

1.2.1 DTO (Data Transfer Object)

DTO (Об'єкт для передачі даних) – це клас або структура, яка використовується для перенесення даних між частинами додатку, зазвичай між серверною і клієнтською частинами або між компонентами всередині сервісів. Основна мета DTO – інкапсуляція даних та мінімізація обсягу інформації, яка передається через мережу.

DTO зазвичай формується на основі результатів SQL-запиту до бази даних. Сервер заповнює DTO необхідними даними та передає їх клієнтській частині у вигляді JSON або XML. Це дозволяє уникнути передачі зайвих даних та приховати внутрішню структуру бази даних від клієнта.

Приклад реалізації DTO в Nest.js[14, 15]:

```
export class ProductDTO {
  id: number;
  name: string;
  price: number;
  imageUrl: string;

  constructor(product: ProductEntity) {
    this.id = product.id;
    this.name = product.name;
    this.price = product.price;
    this.imageUrl = product.imageUrl;
  }
}
```

Рис. 1.1 Реалізація DTO в Nest.js

Переваги DTO:

1. Продуктивність. Передача лише необхідних даних знижує навантаження на сервер і мережу.

2. Безпека. DTO приховують від клієнта зайві або конфіденційні поля бази даних.

3. Гнучкість. Легко адаптуються до змін у структурі даних, що спрощує масштабування.

Недоліки DTO:

1. Дублювання коду. Збільшується кількість класів із кожним новим сценарієм або роллю користувача.

2. Складність у підтримці. У системах із великою кількістю DTO підтримка та оновлення стають трудомісткими.

3. Ручне створення. DTO часто потребують ручного написання коду для кожного нового запиту або ролі користувача.

1.2.2 ORM (Object-Relational Mapping)

ORM (Об'єктно-реляційне відображення) – це технологія, яка забезпечує автоматичне перетворення даних між об'єктами програми та реляційними базами даних. ORM дозволяє працювати з базою даних через об'єктно-орієнтовані класи, а SQL-запити генеруються автоматично під час виконання програми.

ORM надає можливість створювати та оновлювати записи в базі даних, використовуючи об'єктну модель. Програміст працює з сутностями як з об'єктами, уникаючи написання складних SQL-запитів.

Приклад реалізації ORM (TypeORM у Nest.js):

```
@Entity()
export class ProductEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  name: string;

  @Column()
  price: number;

  @Column()
  imageUrl: string;
}
```

Рис. 1.2 Реалізація ORM в Nest.js

У цьому прикладі “ProductEntity” є сутністю, яка автоматично відображається на таблицю product у базі даних. ORM створює таблицю та керує всіма SQL-операціями на основі визначеної моделі.

Переваги ORM:

1. Автоматизація: Зменшується обсяг ручного написання SQL-запитів.
2. Чистота коду: Бізнес-логіка та логіка роботи з базою даних чітко розділені.
3. Швидкий розвиток: Додаючи нові поля до моделі, автоматично оновлюється схема бази даних.

Недоліки ORM:

1. Продуктивність: Автоматично генеровані SQL-запити можуть бути менш оптимізованими, ніж написані вручну.
2. Обмеження складних запитів: Для дуже складних запитів ORM може бути недостатньо гнучким, що потребує ручного написання SQL.
3. Навчання: Робота з ORM вимагає знання специфічного синтаксису та концепцій.

1.2.3 Порівняння DTO та ORM

Таблиця 1.1

Порівняння характеристик підходів DTO та ORM

Критерій	DTO	ORM
Продуктивність	Висока	Залежить від складності запитів
Складність реалізації	Висока (ручне написання класів)	Помірна
Захист даних	Високий	Середній
Дублювання коду	Часто зустрічається	Мінімальне
Простота оновлення структури	Низька (потрібно оновлювати всі DTO)	Висока (ORM автоматично оновлює схему)
Контроль доступу на рівні API	Високий	Мінімальний

DTO дозволяє більш точно контролювати, які дані передаються клієнту, але має високу вартість підтримки. ORM, у свою чергу, автоматизує багато процесів, проте може бути менш ефективним у питаннях безпеки та продуктивності.

1.2.4. Active Record vs Data Mapper

У межах ORM можна виділити дві основні моделі – Active Record та Data Mapper:

- Active Record – модель, де кожен клас сутності безпосередньо відповідає за збереження, оновлення та видалення своїх даних у базі.
- Data Mapper – модель, яка розділяє бізнес-логіку та логіку збереження даних, що робить систему більш масштабованою.

Висновок: традиційні підходи, такі як DTO та ORM, мають як переваги, так і обмеження. DTO забезпечує кращий контроль доступу, тоді як ORM прискорює розробку та спрощує роботу з базою даних. У наступному розділі буде розглянуто сучасні підходи до управління доступом на основі метаданих, що поєднують гнучкість ORM з безпекою DTO.

1.3 Підходи до управління доступом у багатокористувацьких системах

У багатокористувацьких інформаційних системах критично важливо забезпечити правильне управління доступом до даних. Контроль доступу гарантує, що користувачі мають змогу отримувати тільки ту інформацію, яка відповідає їхнім правам та ролям у системі. У цьому розділі розглянуто основні моделі контролю доступу, їхні переваги та недоліки, а також особливості їх застосування в сучасних програмних системах.

1.3.1 Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC) – це один із найпоширеніших підходів до управління доступом, який базується на ролях користувача. У цій моделі кожному користувачу надається одна або кілька ролей, які визначають, до яких ресурсів або даних він має доступ.

Основні принципи RBAC:

1. Призначення прав через ролі: Користувачі отримують доступ до даних або функцій не напряму, а через ролі, що спрощує адміністрування.
2. Принцип найменших привілеїв: Користувач отримує лише той рівень доступу, який необхідний для виконання його завдань.
3. Ієрархія ролей: Ролі можуть бути організовані в ієрархічній структурі, де вищі ролі успадковують права нижчих.

Приклад реалізації RBAC у Nest.js:

```
export enum Role {
  Admin = 'admin',
  Manager = 'manager',
  User = 'user',
}

@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const requiredRoles = this.reflector.getAllAndOverride<Role[]>('roles', [
      context.getHandler(),
      context.getClass(),
    ]);
    if (!requiredRoles) return true;

    const { user } = context.switchToHttp().getRequest();
    return requiredRoles.some((role) => user.roles?.includes(role));
  }
}
```

Рис. 1.3 Реалізація RBAC у Nest.js

У цьому прикладі “RolesGuard” перевіряє, чи містить роль користувача необхідний рівень доступу для виконання певної дії.

Переваги RBAC:

1. Простота адміністрування. Централізоване управління правами користувачів через ролі.
2. Масштабованість. Легко адаптується для великих організацій з численними рівнями доступу.
3. Прозорість: Адміністратор чітко бачить, які права мають користувачі.

Недоліки RBAC:

1. Обмежена гнучкість: RBAC не дозволяє враховувати контекстні фактори, такі як час, місцезнаходження чи атрибути даних.
2. Перевантаження ролями: У системах із великою кількістю функцій може виникнути надлишок ролей, що ускладнює управління.

1.3.2 Attribute-Based Access Control (ABAC)

Attribute-Based Access Control (ABAC)[18] – це модель управління доступом, яка ґрунтується на атрибутах користувача, об'єкта або середовища. У цій моделі рішення щодо доступу приймається на основі політик, які враховують різні атрибути, такі як:

- Роль користувача.
- Час запиту.
- Тип даних або чутливість інформації.

Принципи роботи ABAC:

1. Гнучкість. Враховується широкий спектр атрибутів і контекстів.
2. Модульність. Політики доступу можуть бути налаштовані окремо для різних типів об'єктів.
3. Динамічність. Доступ може змінюватися залежно від змін атрибутів у реальному часі.

Приклад реалізації політик ABAC у Node.js:

```
const policy = {
  resource: 'product',
  action: 'read',
  conditions: {
    role: 'manager',
    department: 'sales',
  },
};
```

Рис. 1.4 Реалізація політик ABAC у Nest.js

Політика дозволяє менеджерам відділу продажів отримувати доступ до певних продуктів.

Переваги ABAC:

1. Гнучкість налаштувань. Підтримує складні сценарії доступу.

2. Деталізованість. Можна налаштовувати доступ до окремих атрибутів даних.

3. Контекстний підхід. Враховуються різні умови виконання запиту.

Недоліки ABAC:

1. Складність налаштування. Вимагає значних зусиль для створення політик та атрибутів.

2. Обчислювальна складність. Високе навантаження на систему при великій кількості правил.

1.3.3 Discretionary Access Control (DAC)

Discretionary Access Control (DAC)[19] – це модель, у якій власник ресурсу визначає, хто має до нього доступ. Власник може передати свої права іншому користувачу або змінити рівень доступу до об'єкта.

DAC часто використовується в файлових системах або базах даних, де користувач може керувати доступом до своїх файлів або таблиць.

1.3.4 Mandatory Access Control (MAC)

Mandatory Access Control (MAC) є суворою моделлю контролю доступу, де рішення щодо доступу приймаються централізовано і засновані на загальних правилах і політиках, які визначені на вищому адміністративному рівні. У цій системі користувачі не мають змоги самостійно змінювати права доступу до ресурсів; замість цього, доступ строго контролюється на основі політик безпеки, які аплікovані до всіх об'єктів і суб'єктів в системі.

MAC часто використовується у сферах, де необхідний високий рівень захисту інформації, таких як військові або урядові установи. Ця модель особливо ефективна для запобігання випадковому або навмисному розголошенню конфіденційних даних, оскільки всі дозволи строго регулюються заздалегідь заданими правилами.

1.3.5 Порівняння RBAC, ABAC, DAC і MC

Таблиця 1.2

Порівняльна характеристика моделей управління доступом RBAC, ABAC, DAC і MAC

Модель управління доступом	Гнучкість	Складність реалізації	Масштабованість	Безпека
RBAC	Середня	Низька	Висока	Висока
ABAC	Висока	Висока	Середня	Висока
DAC	Низька	Низька	Низька	Середня
MAC	Низька	Висока	Висока	Висока

Висновок: різні моделі управління доступом мають свої переваги та недоліки. У сучасних системах часто застосовують гібридний підхід, поєднуючи RBAC та ABAC для забезпечення гнучкості та високого рівня безпеки. У наступному розділі буде розглянуто метадані як інструмент для динамічного управління доступом на рівні об'єктів.

1.4. Тришарова архітектура програмних систем

Тришарова архітектура є одним із найбільш поширених підходів до побудови програмних систем. Вона забезпечує чіткий поділ обов’язків між різними рівнями додатку, що сприяє модульності, гнучкості та легкості підтримки системи. У цьому підрозділі розглянемо основні компоненти тришарової архітектури, їх взаємодію та роль у контексті управління доступом до даних.

1.4.1 Структура тришарової архітектури

Тришарова архітектура[21] передбачає поділ системи на три основні шари:

1. Шар клієнта (Presentation Layer):

- Відповідає за взаємодію з користувачем.
- Реалізується у вигляді веб-інтерфейсів, мобільних додатків або десктопних програм.
- Основне завдання – отримання введених даних від користувача та відображення результатів.

2. Шар бізнес-логіки (Business Logic Layer):

- Виконує всі основні операції над даними: перевірку, обробку, виконання бізнес-правил.
- Відповідає за прийняття рішень і передачу даних між іншими шарами.
- Реалізується у вигляді серверної логіки, яка може включати використання API, сервісів і мікросервісів.

3. Шар доступу до даних (Data Access Layer):

- Відповідає за роботу з базою даних або іншими сховищами даних.
- Реалізує збереження, оновлення, видалення та отримання даних.
- Використовує ORM, SQL-запити або інші інструменти для взаємодії з базою даних.

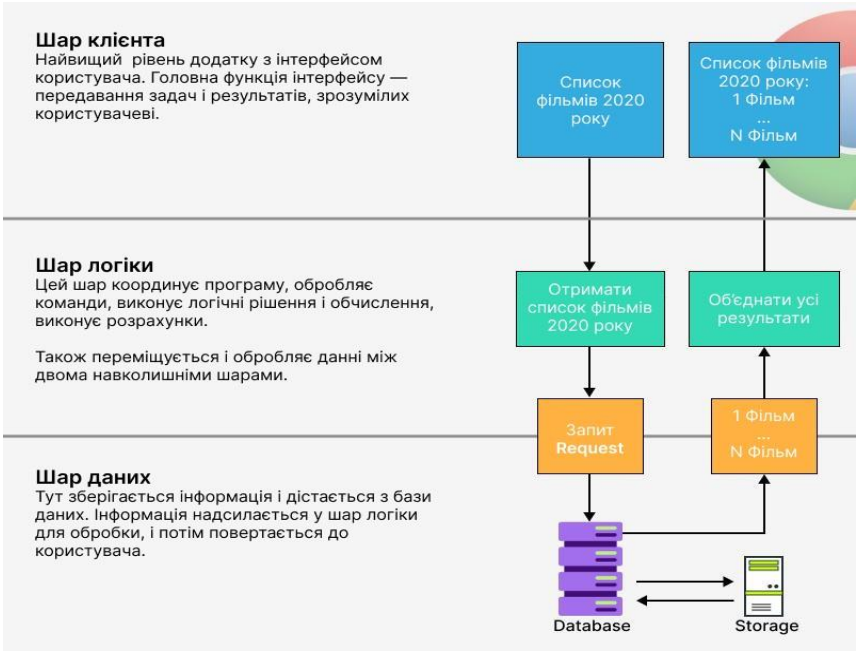


Рис. 1.5. Схема тришарової (трирівневої) архітектури

1.4.2 Взаємодія між шарами

У тришаровій архітектурі кожен шар виконує чітко визначені функції, а взаємодія між ними забезпечується через визначені інтерфейси. Наприклад:

1. Шар клієнта передає запити користувачів до шару бізнес-логіки.
2. Шар бізнес-логіки отримує дані з шару доступу до даних, виконує необхідні обчислення та повертає результат.
3. Шар доступу до даних зберігає результати або передає потрібну інформацію шару бізнес-логіки.

Такий підхід забезпечує модульність системи, дозволяє розробникам зосередитися на конкретних аспектах без необхідності враховувати всі аспекти додатку.

Тришарова архітектура є основою для побудови багатьох сучасних систем, оскільки забезпечує чітке розділення логіки та сприяє зручній розробці та масштабуванню. Шар доступу до даних відіграє ключову роль у впровадженні механізмів управління доступом, що робить його особливо важливим у контексті багатокористувацьких систем. У наступних розділах буде розглянуто, як різні підходи до управління доступом інтегруються в тришарову архітектуру.

2. ОГЛЯД ІСНУЮЧИХ ІНСТРУМЕНТІВ ДЛЯ РЕАЛІЗАЦІЇ ПРОЕКЦІЇ ДАНИХ В ПРИКЛАДНИЙ КОД JAVASCRIPT

2.1 Інструменти для ORM

Проекція даних із реляційних баз даних у прикладний код є критичним етапом створення сучасних додатків. Цей процес забезпечує взаємодію між рівнем зберігання даних (наприклад, PostgreSQL) та бізнес-логікою додатку, дозволяючи працювати з базою даних через об'єкти та класи.

Node.js пропонує широкий набір інструментів для роботи з базами даних, серед яких найважливішу роль відіграють ORM (Object-Relational Mapping) та

SQL-конструктори. Ці інструменти автоматизують проекцію даних, зменшують кількість коду та дозволяють уникати ручного написання SQL-запитів.

2.1.1 TypeORM

TypeORM є найбільш поширеним ORM-інструментом для Node.js. Він підтримує TypeScript і дозволяє визначати структуру бази даних через класи та декоратори.

Архітектура та принцип роботи

TypeORM дотримується класичної архітектури ORM, де кожен клас (entity) відповідає таблиці бази даних. Сутності (Entities) описують таблиці, а декоратори використовуються для визначення типів колонок та зв'язків між ними.

Компоненти архітектури:

- Entities (Сутності): Класи, що представляють таблиці бази даних.
- Repositories (Репозиторії): Виконують CRUD-операції над сутностями.
- Migrations (Міграції): Відстежують зміни в схемі баз даних.
- Relations (Зв'язки): Забезпечують взаємозв'язки між таблицями (one-to-one, one-to-many, many-to-many).

Приклад сутності:

```
@Entity()
export class Product {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  name: string;

  @Column('decimal')
  price: number;

  @ManyToOne(() => Category, (category) => category.products)
  category: Category;
}
```

Рис. 2.1 Сутність TypeORM в Node.js

Переваги:

- Інтеграція з TypeScript. Забезпечує типізацію та автодоповнення.
- Підтримка складних зв’язків. Легка робота з реляційними зв’язками.
- Автоматичні міграції. Дозволяє уникнути помилок при зміні схеми бази даних.

Недоліки:

- Високий рівень складності налаштування. Потребує глибоких знань TypeScript та принципів ООР.
- Зниження продуктивності на великих обсягах даних.

Кейс використання:

TypeORM активно застосовується у великих корпоративних додатках із розгалуженою структурою баз даних. Наприклад, ERP-системи або CRM-додатки часто потребують складних реляційних зв’язків, що робить TypeORM ідеальним вибором.

2.1.2 Sequelize

Sequelize[22] є одним із найдавніших ORM для Node.js, з широкою підтримкою баз даних, включаючи PostgreSQL, MySQL, MariaDB та SQLite.

Архітектура та принцип роботи:

Sequelize використовує моделі для відображення структури таблиць і асоціації для створення зв’язків між ними. Це дозволяє створювати складні реляційні структури за допомогою простого API.

Компоненти архітектури:

- Models (Моделі): Визначають структуру таблиць.
- Associations (Асоціації): Визначають зв’язки між моделями.
- Migrations (Міграції): Дозволяють оновлювати структуру баз даних.

Приклад моделі:

```
const User = sequelize.define('User', {
  username: {
    type: Sequelize.STRING,
    allowNull: false,
  },
  email: {
    type: Sequelize.STRING,
    unique: true,
  },
});
```

Рис. 2.2. Модель в Sequelize

Переваги:

- Простота у використанні. Легкий для вивчення новачками.
- Швидке створення API. Швидке налаштування зв’язків і запуск CRUD-операцій.

Недоліки:

- Обмежена підтримка TypeScript. Нативна інтеграція з TypeScript потребує додаткових налаштувань.
- Низька продуктивність при складних запитах.

Кейс використання:

Sequelize чудово підходить для стартапів і середніх додатків, де швидкість розробки є пріоритетом.

2.1.3 Objection.js

Objection.js – це ORM, яка забезпечує роботу із SQL-запитами через Knex.js[23], зберігаючи об'єктно-орієнтовану модель роботи з таблицями.

Архітектура та принцип роботи

Objection.js поєднує об’єктно-орієнтовані моделі з прямою роботою із SQL-запитами через Knex.js.

Приклад моделі:

```
class Person extends Model {
  static get tableName() {
    return 'persons';
  }
}
```

Рис. 2.3 Приклад моделі Objection.js

Переваги:

- Повний контроль над SQL. Легко реалізовує складні запити.
- Гнучкість у роботі з існуючими базами даних.

Недоліки:

- Відсутність автоматизації міграцій.

Кейс використання:

Objection.js підходить для великих систем аналітики та фінансових додатків, де SQL-запити мають бути точно налаштовані.

2.1.4 Prisma

Prisma[24] – сучасний ORM нового покоління, що пропонує декларативний підхід до взаємодії з базами даних. Його ключова особливість – це генерація типізованого клієнта, який забезпечує автодоповнення та статичну перевірку типів. Prisma надає зручний спосіб роботи з PostgreSQL, MySQL, MariaDB та SQLite.

Архітектура та принцип роботи

Prisma складається з кількох основних компонентів:

- Schema (Схема): Декларативний файл, що описує структуру бази даних.
- Migrations (Міграції): Інструмент для автоматичного створення та оновлення таблиць.
- Client (Клієнт): Генерується автоматично та використовується для виконання запитів до бази даних.

Приклад схеми:

```
model User {
  id    Int    @id @default(autoincrement())
  name  String
  email String @unique
}
```

Рис. 2.4 Приклад схеми в Prisma

Приклад використання клієнта:

```
const user = await prisma.user.create({
  data: {
    name: 'John Doe',
    email: 'john@example.com',
  },
});
```

Рис. 2.5. Приклад використання схеми клієнтом в Prisma

Переваги:

- Декларативний підхід. Схеми легко читаються та підтримуються.
- Автоматична генерація клієнта. Швидка взаємодія з базою даних без написання SQL-запитів вручну.
- Підтримка TypeScript. Prisma автоматично генерує типізовані запити.

Недоліки:

- Обмежена гнучкість. Менше можливостей для кастомізації запитів порівняно з Objection.js чи Knex.js.
- Додатковий рівень абстракції. Це може впливати на продуктивність у деяких сценаріях.

Кейс використання:

Prisma активно використовується для створення GraphQL-серверів, REST API та адміністративних панелей, де важлива швидка взаємодія з базою даних та простота підтримки.

2.1.5 Drizzle ORM

Drizzle ORM[25] – легковаговий ORM, орієнтований на продуктивність і простоту. Відзначається мінімалістичним API та високою швидкістю роботи з базами даних. В основі Drizzle лежить принцип "зберігати SQL-запити прозорими та контрольованими".

Архітектура та принцип роботи

Drizzle ORM фокусується на забезпеченні максимального контролю над SQL-запитами, зберігаючи при цьому зручність ORM.

Компоненти архітектури:

- Models (Моделі): Описуються через TypeScript.
- Query Builder: Вбудований механізм для побудови SQL-запитів.
- Migrations (Міграції): Автоматизоване оновлення структури баз даних.

Приклад моделі:

```
import { pgTable, integer, varchar } from 'drizzle-orm/pg-core';

export const users = pgTable('users', {
  id: integer('id').primaryKey(),
  name: varchar('name', { length: 255 }),
  email: varchar('email', { length: 255 }).unique(),
});
```

Рис. 2.6 Приклад моделі в Drizzle ORM

Приклад запиту:

```
const result = await db.select().from(users).where(eq(users.email, 'test@example.com'));
```

Рис. 2.7. Приклад запиту в Drizzle ORM

Переваги:

- Простота. Мінімалістичний API з високою продуктивністю.
- Повний контроль над SQL. Розробники можуть легко налаштовувати запити під свої потреби.
- TypeScript-first. Опис моделей через TypeScript гарантує точність і безпеку.

Недоліки:

- Відсутність автоматичних зв’язків. Розробник повинен вручну налаштовувати зв’язки між таблицями.
- Молода бібліотека. Можлива нестача функціоналу порівняно з TypeORM або Prisma.

Кейс використання:

Drizzle ORM ідеально підходить для розробки серверних частин, де важлива продуктивність, прозорість SQL-запитів та проста підтримка моделей.

2.1.6 Knex.js

Knex.js – SQL-конструктор для Node.js, який дозволяє створювати SQL-запити за допомогою JavaScript. На відміну від ORM, Knex.js забезпечує повний контроль над створенням запитів, що робить його ідеальним для складних та оптимізованих SQL-операцій.

Архітектура та принцип роботи:

Knex.js працює як SQL-конструктор і не прив’язується до об’єктно-орієнтованої моделі. Це дозволяє використовувати Knex.js як незалежний рівень для взаємодії з PostgreSQL, MySQL або іншими базами даних.

Компоненти архітектури:

- Query Builder: Інструмент для створення SQL-запитів.
- Migrations (Міграції): Інструменти для оновлення схеми баз даних.

Приклад запиту:

```
const users = await knex('users').where({ email: 'test@example.com' }).select('*');
```

Рис. 2.8 Приклад запиту в Knex.js

Приклад створення таблиці:

```
await knex.schema.createTable('users', (table) => {
  table.increments('id').primary();
  table.string('name');
  table.string('email').unique();
});
```

Рис. 2.9. Приклад створення таблиці в Knex.js

Переваги:

- Гнучкість. Дає повний контроль над SQL-запитами.
- Висока продуктивність. Відсутність зайвої абстракції забезпечує швидку роботу.

Недоліки:

- Відсутність об’єктно-орієнтованої моделі. Потрібно створювати класи та зв’язки вручну.

Кейс використання:

Knex.js активно використовується у фінансових системах, де важлива оптимізація SQL-запитів та максимальна продуктивність.

2.2 Інструменти для маніпулювання Access Layer

2.2.1 Casl (Code Access Security Layer)

Casl[26] – це потужна бібліотека для управління доступом, яка дозволяє налаштовувати політики на основі ролей користувача та атрибутів об'єкта. Casl підтримує гнучку проекцію даних, забезпечуючи відображення лише тих об'єктів, до яких користувач має доступ.

Архітектура та принцип роботи

Casl базується на створенні "здібностей" (abilities), які описують, що користувач може робити з певними ресурсами.

Компоненти архітектури:

- Abilities (Здібності): Описують правила доступу для конкретних дій (create, read, update, delete).
- Conditions (Умови): Обмежують доступ до ресурсів на основі атрибутів.
- Subjects (Об'єкти): Визначають тип ресурсу, до якого застосовується політика.

Приклад налаштування політики:

```
const ability = defineAbility((can, cannot) => {
  can('read', 'Article', { published: true });
  cannot('delete', 'Article', { isProtected: true });
});
```

Рис. 2.10 Налаштування політики в Casl

Приклад фільтрації даних:

```
const articles = await Article.find(ability.rulesFor('read', 'Article'));
```

Рис. 2.11. Фільтрація даних в Casl

Переваги:

- Гнучкість. Дозволяє налаштовувати складні політики доступу.
- Динамічність. Можливість змінювати правила доступу під час виконання програми.
- Інтеграція з ORM. Casl легко інтегрується з TypeORM, Prisma та іншими ORM.

Недоліки:

- Висока складність правил. У складних системах конфігурація може стати громіздкою.

Кейс використання:

Casl активно використовується в SaaS-платформах та адміністративних панелях, де права користувача можуть змінюватись динамічно залежно від його ролі та статусу.

2.2.2 NodeAcl

NodeAcl[27] – це бібліотека для реалізації ACL (Access Control List) в Node.js-додатках. Вона забезпечує налаштування доступу на рівні ресурсів, користувачів та ролей.

Архітектура та принцип роботи:

NodeAcl дозволяє зберігати списки контролю доступу в пам’яті або базі даних і визначає права для кожного користувача або групи користувачів.

Компоненти архітектури:

- Roles (Ролі): Набори дозволів для користувачів або груп.
- Resources (Ресурси): Об'єкти, до яких застосовуються політики доступу.
- Permissions (Дозволи): Конкретні дії, дозволені для певних ролей (read, write, delete).

Приклад налаштування ролей:

```
acl.allow('admin', 'articles', ['create', 'read', 'update', 'delete']);
acl.allow('user', 'articles', ['read']);
```

Рис. 2.12 Приклад налаштування ролей в NodeAcl

Перевірка доступу:

```
acl.isAllowed('user123', 'articles', 'delete', (err, res) => {
  if (res) {
    console.log('Access granted');
  } else {
    console.log('Access denied');
  }
});
```

Рис. 2.13. Перевірка доступу d NodeAcl

Переваги:

- Простота у використанні. Легко налаштовується та масштабується.
- Гнучкість. Дозволяє створювати ACL на рівні об'єктів або API-ендпоінтів.

Недоліки:

- Обмеженість. Менш динамічний у порівнянні з Casl.
- Витрати на продуктивність. Велика кількість ACL може вплинути на швидкість роботи системи.

Кейс використання:

NodeAcl активно використовується в системах управління файлами та медіа, де важливо контролювати доступ на рівні окремих документів або колекцій.

2.2.3 AccessControl

AccessControl є потужною бібліотекою для Node.js, яка дозволяє розробникам ефективно імплементувати систему управління доступом на основі

ролей (RBAC). Ця бібліотека надає гнучкі засоби для визначення ролей, ресурсів і дозволів, що дозволяє адміністраторам системи або розробникам легко створювати складні політики доступу.

AccessControl використовує декларативний синтаксис для налаштування правил доступу, що робить код більш чистим і зрозумілим. Розробники можуть легко визначати правила у вигляді об'єктів JSON, що дозволяє централізовано управляти політиками безпеки та з легкістю вносити зміни.

Архітектура та принцип роботи:

AccessControl працює на рівні ресурсів та ролей, забезпечуючи простий декларативний синтаксис для налаштування доступу.

Компоненти архітектури:

- Grants (Гранти): Визначають дозволи для кожної ролі.
- Extend (Розширення): Дозволяє успадковувати права від інших ролей.

Приклад створення політик:

```
const ac = new AccessControl();

ac.grant('user').readOwn('profile');
ac.grant('admin').extend('user').updateAny('profile');
```

Рис. 2.14 Приклад створення політик в AccessControl

Перевірка доступу:

```
const permission = ac.can('user').readOwn('profile');
if (permission.granted) {
  console.log('Access allowed');
}
```

Рис. 2.14 Приклад перевірки доступу в AccessControl

Переваги:

- Простота конфігурації: AccessControl використовує декларативний JSON-подібний синтаксис, який є легким для розуміння та налаштування, забезпечуючи швидке впровадження системи контролю доступу.
- Наслідування ролей: Можливість наслідування дозволяє створювати складні ієрархії ролей, спрощуючи управління дозволами та зменшуючи кількість повторюваного коду.
- Висока швидкість виконання: AccessControl оптимізований для швидкої перевірки прав доступу, що є критично важливим для високонавантажених додатків.

Недоліки:

- Менша гнучкість. Порівняно з Casl, обмежена підтримка умов доступу.
- Обмеження у підтримці контекстуальних правил: Хоча бібліотека дозволяє додавати динамічні правила, її можливості для обробки контекстуальних даних (наприклад, час або місцезнаходження) можуть бути обмежені.
- Залежність від структури проекту: AccessControl вимагає чіткої організації коду та дотримання певної структури проекту, що може створити складнощі при інтеграції з існуючими проектами.

Кейс використання:

AccessControl (контроль доступу) застосовується для обмеження доступу до певних частин системи на основі ролей користувачів. У цьому випадку ми розглядаємо багаторівневу систему ролей, де кожен користувач може мати одну або кілька ролей, і доступ до ресурсів чи дій обмежується на основі їхніх прав. Це особливо актуально для API та веб-додатків, де необхідно чітко визначити, хто має доступ до яких функцій чи даних.

3. РОЗРОБКА МЕТОДУ ВІДОБРАЖЕННЯ СУТНОСТЕЙ БАЗИ ДАНИХ В ПРИКЛАДНИЙ КОД НА ОСНОВІ МЕТАДАНИХ ТА РОЛЕЙ КОРИСТУВАЧА

3.1 Визначення та значення проекції даних

Сучасні інформаційні системи мають справу з великим обсягом даних, структурованих у вигляді сутностей та зв’язків між ними. Важливою складовою таких систем є ефективна проекція даних у прикладний код, яка дозволяє оптимізувати доступ до бази даних, спрощує управління структурами даних та мінімізує об’єм коду.

Відображення сутностей бази даних у прикладний код часто виконується за допомогою ORM (Object-Relational Mapping) або SQL-конструкторів, однак традиційні підходи не завжди дозволяють ефективно керувати доступом до даних на рівні окремих полів та зв’язків. Це призводить до створення дублюючих моделей для різних користувачів або ролей, що збільшує технічний борг системи.

Проекція даних є однією з базових концепцій управління інформацією у сучасних інформаційних системах. Вона дозволяє обмежити набір полів або атрибутів сутності, що передаються або використовуються під час запиту. У широкому сенсі проекція визначає, які саме частини даних є доступними для відображення чи обробки у конкретному контексті, що забезпечує гнучкість, оптимізацію ресурсів та зменшення надлишковості даних.

Проекція даних відіграє критичну роль у системах, де інформація повинна бути надана різним групам користувачів з різним рівнем доступу або контексту використання. У таких системах сутності можуть мати десятки атрибутів, проте не всі вони є необхідними для кожного запиту чи ролі. Проекція дозволяє створювати динамічні представлення об'єктів, які містять лише актуальні атрибути, відповідно до заданих умов або ролей користувача.

3.1.1 Значення проекції даних у сучасних інформаційних системах

Проекція є важливим елементом архітектури систем управління базами даних, мікросервісів[28] та веб-додатків[29]. Завдяки вибірковій обробці даних можна значно оптимізувати продуктивність системи, зменшити навантаження на сервер та забезпечити кращий користувацький досвід.

У багатокористувацьких системах із різними рівнями доступу (наприклад, ERP-системи, CRM, банківські додатки) проекція даних дозволяє керувати видимістю атрибутів залежно від ролі користувача. Це підвищує рівень безпеки, запобігає витоку конфіденційної інформації та забезпечує дотримання принципу найменших привілеїв (Principle of Least Privilege).

Ключові аспекти значення проекції даних:

1. Оптимізація запитів та підвищення продуктивності.

Зменшення кількості полів, що обробляються під час виконання запиту, скорочує обсяг переданих даних, знижує час виконання запитів та зменшує навантаження на сервери. Це особливо важливо для систем, які обробляють великі масиви даних у режимі реального часу.

2. Гнучкість та масштабованість.

Проекція дозволяє створювати універсальні моделі сутностей, які адаптуються до різних користувацьких сценаріїв, мінімізуючи потребу в створенні дублюючих класів або структур. Це полегшує масштабування системи та інтеграцію нових функцій.

3. Забезпечення безпеки та конфіденційності.

Проекція сприяє обмеженню видимості чутливих полів для користувачів, які не мають відповідних прав доступу. Це знижує ризик витоку конфіденційної інформації та забезпечує відповідність стандартам захисту даних, таким як GDPR чи HIPAA.

4. Зменшення дублювання коду.

Використання проекції дозволяє уникнути дублювання коду при створенні окремих моделей або запитів для різних ролей користувачів. Єдина модель сутності

може бути використана у кількох контекстах із застосуванням проекції для обмеження видимих атрибутів.

3.1.2 Проекція даних у контексті архітектури інформаційних систем

Проекція є важливою частиною архітектурних шаблонів та концепцій, які лежать в основі проектування інформаційних систем. Вона забезпечує логічне розділення між внутрішньою моделлю сутності та її представленням на рівні клієнта або користувацького інтерфейсу.

- Проекція виконує кілька важливих функцій:
- Контроль за передачею даних між клієнтом та сервером.
- Обмеження доступу до атрибутів об'єктів на рівні API.
- Побудова легковагових об'єктів для передачі даних (DTO – Data Transfer Object).

У мікросервісній архітектурі проекція дозволяє створювати специфічні моделі для різних сервісів, які працюють із єдиною базою даних, але мають різні бізнес-логіки та рівні доступу.

3.1.3 Співвідношення проекції та процесів нормалізації даних

Проекція даних у системах управління базами даних (СУБД) тісно пов’язана з процесами нормалізації та денормалізації даних. Нормалізація спрямована на мінімізацію надлишковості даних та оптимізацію структури баз даних шляхом створення зв’язків між таблицями.

Відмінність між нормалізацією та проекцією:

- Нормалізація розділяє дані між різними таблицями, щоб усунути дублювання.
- Проекція вибірково витягує дані з цих таблиць або сутностей, формуючи підмножину атрибутів для конкретного користувача або запиту.

3.1.4 Роль проекції у моделюванні рівнів доступу

У складних інформаційних системах проекція даних використовується для створення рівнів доступу до атрибутів сутностей. Це дозволяє моделювати багаторівневу систему прав доступу, в якій кожен рівень надає різний набір даних відповідно до ролі користувача.

Основні рівні доступу:

- Повний доступ: Усі поля сутності доступні для адміністратора або супервізора.
- Обмежений доступ: Менеджери або оператори бачать лише ключові поля, необхідні для виконання їхніх завдань.
- Мінімальний доступ: Користувачі отримують доступ лише до базових атрибутів, які не містять чутливої інформації.

3.1.5 Вплив проекції на ефективність управління даними

Проекція даних дозволяє забезпечити:

- Логічне розділення рівнів роботи з даними. Проекція дозволяє створювати незалежні представлення для кожного рівня доступу.
- Скорочення часу виконання запитів. Вибіркове відображення даних зменшує кількість обчислювальних ресурсів, що використовуються для обробки запитів.
- Ефективне управління кешуванням. Проекція дозволяє кешувати лише вибрані атрибути сутностей, що значно знижує обсяг займаної пам’яті.

Проекція даних є важливою складовою сучасних інформаційних систем, яка дозволяє оптимізувати роботу з даними, підвищити рівень безпеки та забезпечити ефективне управління інформацією. Вона є невід'ємною частиною масштабованих і безпечних додатків, дозволяючи створювати динамічні моделі сутностей, що адаптуються до різних контекстів та ролей користувачів.

3.2 Теоретичні моделі проекції даних

У процесі розробки інформаційних систем важливу роль відіграє вибір моделі проекції даних. Теоретичні моделі проекції визначають, як саме дані відображаються на рівні прикладного коду та бази даних, які атрибути сутностей доступні для конкретних користувачів або ролей, та які частини інформації є актуальними для кожної бізнес-операції.

Існує кілька основних підходів до реалізації проекції даних, які формуються на основі архітектурних принципів та вимог до системи. Традиційні методи передбачають створення статичних моделей для кожної сутності та ролі користувача, тоді як сучасні підходи базуються на автоматизації та динамічній генерації моделей через метадані та рольові політики.

3.2.1 Традиційна модель проекції даних

Традиційна проекція базується на створенні статичних класів та моделей сутностей для кожного рівня доступу або ролі користувача. У цьому випадку для кожного типу користувача (адміністратора, менеджера, клієнта тощо) створюється окрема модель, що містить підмножину атрибутів повної сутності.

Основні принципи традиційної моделі:

1. Жорстка прив’язка до рівнів доступу. Для кожної ролі або сценарію використання створюється окрема версія моделі.
2. Статична структура. Моделі не змінюються під час виконання програми та є заздалегідь визначеними.
3. Дублювання коду. Створення кількох варіантів однієї сутності призводить до дублювання логіки, що ускладнює підтримку та масштабування системи.

Переваги традиційної моделі:

- Простота впровадження у невеликих проектах.
- Відсутність складних механізмів динамічної генерації сутностей.
- Чітко визначена структура та контроль над кожною моделлю.

Недоліки традиційної моделі:

- Велика кількість коду. Необхідність створення окремих класів для кожної ролі користувача.
- Складність підтримки. Зміна структури сутності вимагає оновлення всіх версій моделі, що підвищує ризик помилок.
- Обмежена масштабованість. Додавання нових ролей або рівнів доступу потребує створення нових моделей, що ускладнює розвиток системи.

3.2.2 Динамічна (запропонована) модель проекції даних

Динамічна проекція є більш сучасним підходом, який базується на автоматизації створення моделей та їх адаптації до контексту виконання програми. У цьому випадку сутність описується за допомогою метаданих, які визначають її структуру, атрибути та правила доступу. Модель формується в реальному часі відповідно до ролі користувача та бізнес-логіки системи.

Основні принципи динамічної моделі:

- Використання метаданих. Метадані містять опис полів сутності та рівні доступу до них.
- Гнучкість та адаптивність. Моделі можуть змінюватися під час виконання програми, адаптуючись до поточного користувача чи операції.
- Зменшення дублювання коду. Всі моделі генеруються на основі єдиної структури, що дозволяє уникнути дублювання логіки.

Переваги динамічної моделі:

- Гнучкість. Легко додавати нові ролі та налаштовувати доступ до атрибутів сутності.
- Оптимізація коду. Зменшення кількості вручну створених моделей та сутностей.
- Зниження витрат на підтримку. Зміни в структурі даних вносяться в одному місці – у метаданих.

Недоліки динамічної моделі:

- Ускладненість реалізації. Для впровадження динамічної проекції необхідно створити механізм генерації моделей на основі метаданих.
- Збільшення навантаження на процесор. Генерація моделей у реальному часі може призвести до збільшення часу обробки запитів.
- Необхідність ретельного тестування. Динамічні системи вимагають детального тестування, щоб уникнути неочікуваних помилок у процесі генерації моделей.

3.2.3. Порівняння традиційної та динамічної моделей проекції

Таблиця 3.1

Метрики традиційної і динамічної моделей

Критерій	Традиційна модель	Динамічна модель
Продуктивність	Висока (моделі створюються наперед)	Середня (генерація під час виконання)
Масштабованість	Обмежена	Висока
Обсяг коду	Великий (багато дублювання)	Мінімальний
Зручність підтримки	Низька	Висока
Гнучкість	Низька	Висока
Простота реалізації	Висока (легка імплементація)	Середня (потрібна складна логіка)
Адаптивність	Низька	Висока

3.3. Метадані як основа проекції

Метадані відіграють ключову роль у сучасних інформаційних системах, забезпечуючи можливість гнучкої та масштабованої проекції даних. Вони дозволяють описувати структуру сутностей, атрибути, зв’язки та правила доступу, що робить процес управління даними більш централізованим і автоматизованим.

Метадані є своєрідним «описом даних», який дозволяє системі адаптувати вивід інформації в залежності від ролі користувача, контексту запиту або умов виконання. Завдяки цьому інформаційні системи можуть динамічно генерувати моделі, контролювати видимість атрибутів сутностей та забезпечувати відповідність принципам RBAC (Role-Based Access Control) та ABAC (Attribute-Based Access Control).

3.3.1. Визначення та класифікація метаданих

Метадані – це структуровані дані, які описують інші дані. У контексті баз даних і прикладного коду метадані є проміжним шаром, який містить інформацію про:

- Структуру сутностей: які поля містить об’єкт, їхні типи, обов'язковість та обмеження.
- Зв’язки між сутностями: типи реляційних зв’язків (one-to-one, one-to-many, many-to-many).
- Правила доступу: визначення того, які поля є видимими для конкретних ролей або користувачів.
- Контекстні умови: додаткові умови, за яких доступ до даних може бути обмеженим або розширеним.

Класифікація метаданих для проекції даних:

1. Структурні метадані:

- Визначають структуру сутності (назви полів, їхні типи, обмеження).
- Містять інформацію про ключові атрибути, індекси, унікальні поля та зовнішні ключі.

2. Рольові метадані:

- Описують правила доступу до полів сутності залежно від ролі користувача.
- Визначають набір атрибутів, які є доступними для кожної ролі або групи користувачів.

3. Зв’язкові метадані[30]:

- Містять інформацію про відношення сутності до інших таблиць або моделей (наприклад, зв’язок продуктів із категоріями).

4. Контекстні метадані:

- Враховують умови, які визначають доступ до сутності або її частини залежно від поточного стану об’єкта (наприклад, статус замовлення або рівень доступу).

3.3.2 Роль метаданих у побудові інформаційних систем

Метадані є фундаментальним компонентом при створенні масштабованих інформаційних систем, що дозволяє:

- Автоматизувати процес створення моделей. Створення моделей та їхніх атрибутів відбувається на основі конфігурацій, описаних у метаданих, що зменшує необхідність ручної роботи.
- Підвищити гнучкість системи. Додавання нових полів або атрибутів не потребує змін у коді програми, достатньо оновити метадані.
- Забезпечити централізоване управління доступом. Правила доступу можуть змінюватися без оновлення основної логіки програми, що спрощує підтримку та масштабування.
- Мінімізувати дублювання коду. Завдяки єдиній структурі сутностей, метадані дозволяють створювати динамічні моделі без дублювання логіки.
- Оптимізувати взаємодію між компонентами системи. Метадані можуть служити мостом між різними модулями і службами, гарантуючи, що вся система працює узгоджено і ефективно.

Приклад метаданих для сутності "Product":

```
Product:
  fields:
    id: integer
    name: string
    price: decimal
    categoryId: integer
    stock: integer
  relations:
    category:
      type: many-to-one
      target: Category
  roles:
    admin: ["id", "name", "price", "categoryId", "stock"]
    manager: ["id", "name", "price", "categoryId"]
    user: ["id", "name", "price"]
```

Рис. 3.1 Метадані для сутності “Product”

3.3.3 Структура метаданих у багаторівневих системах

У складних системах метадані можуть бути організовані у вигляді багаторівневої структури:

1. Глобальні метадані – містять базові описи сутностей, які є доступними для всіх рівнів системи.
2. Модульні метадані – налаштовуються для конкретних частин програми (наприклад, модуль управління клієнтами чи замовленнями).
3. Користувацькі метадані – налаштовуються для індивідуальних користувачів або ролей з урахуванням специфічних потреб бізнесу.

3.3.4 Автоматизація створення сутностей на основі метаданих

Однією з ключових функцій метаданих є можливість автоматичного створення сутностей та їхніх атрибутів у прикладному коді. Це дозволяє розробникам уникати дублювання коду та підтримувати актуальність моделей навіть при внесенні змін до структури бази даних.

Процес автоматизації створення сутностей включає такі етапи:

1. Зчитування метаданих із конфігураційного файлу (JSON, YAML, XML).
2. Генерація класів або об’єктів у прикладному коді.
3. Застосування обмежень та зв’язків між сутностями відповідно до опису метаданих.
4. Формування проекції відповідно до ролі користувача.

Приклад генерації класу на основі метаданих:

```
function generateModel(metadata) {
  return class {
    constructor(data) {
      Object.keys(metadata.fields).forEach((field) => {
        this[field] = data[field] || null;
      });
    }
  };
}
```

Рис 3.2. Функція-фабрика класів для генерації Моделей на основі метаданих

3.3.5. Використання метаданих для динамічної проекції даних

Метадані дозволяють створювати різні представлення однієї сутності для різних користувачів. Це означає, що користувач із роллю admin бачитиме повний набір атрибутів сутності, тоді як звичайний користувач – лише базові поля.

Проекція на основі метаданих працює за принципом фільтрації полів сутності залежно від ролі користувача. У процесі виконання запиту система зчитує метадані, ідентифікує роль користувача та формує відповідний об’єкт із доступними полями.

3.4. Взаємозв’язок метаданих та рольової моделі доступу

Рольова модель доступу (RBAC – Role-Based Access Control) є одним із найбільш ефективних способів керування правами доступу до даних у багатокористувацьких системах. Вона дозволяє централізовано контролювати, які сутності або атрибути є доступними для користувачів залежно від їхньої ролі в організації чи системі.

Метадані відіграють важливу роль у впровадженні та підтримці рольової моделі, забезпечуючи:

- Гнучкість у визначенні політик доступу;
- Динамічність у формуванні проекцій даних;
- Централізацію управління атрибутами сутностей.

Метадані, які описують сутності та їхні поля, поєднуються з правилами RBAC/ABAC, формуючи основу для автоматичної фільтрації та проекції даних залежно від ролі користувача. Завдяки цьому, доступ до певних атрибутів або методів може обмежуватися або дозволятися без необхідності змінювати основну бізнес-логіку додатку.

3.4.1. Використання метаданих для реалізації RBAC

Метадані забезпечують опис сутностей і дозволяють формувати правила доступу у вигляді конфігурацій або окремих таблиць у базі даних. Вони є проміжним рівнем між користувацькими запитами та бізнес-логікою додатку.

Приклад метаданих для RBAC:

```yaml
Order:
  fields:
    id: integer
    customer_name: string
    total_price: decimal
    status: string
    internal_notes: string
  roles:
    admin:
      access: ["id", "customer_name", "total_price", "status", "internal_notes"]
    manager:
      access: ["id", "customer_name", "status"]
    user:
      access: ["id", "status"]
```

Рис. 3.3 Метадані для RBAC

3.4.2 ABAC та розширення можливостей проекції

ABAC (Attribute-Based Access Control) розширює можливості RBAC, додаючи до системи атрибутивний підхід, який враховує не тільки ролі користувача, але й додаткові умови (контекст, атрибути сутності, час виконання запиту тощо).

Приклад метаданих для ABAC:

```yaml
Order:
  fields:
    id: integer
    customer_name: string
    status: string
    delivery_date: string
  rules:
    - condition: "user.role == 'manager' && order.status == 'pending'"
      access: ["id", "customer_name", "status"]
    - condition: "user.role == 'admin'"
      access: ["id", "customer_name", "status", "delivery_date"]
```

Рис. 3.4 Метадані для ABAC

Пояснення:

- Менеджери бачать замовлення лише у статусі pending.
- Адміністратори мають доступ до всіх замовлень та додаткових атрибутів (delivery_date).

3.4.3 Взаємозв’язок метаданих та рольової моделі

Метадані є невід'ємною частиною впровадження RBAC та ABAC, оскільки дозволяють зберігати правила доступу у форматі, що легко редагується. Це забезпечує централізоване управління політиками доступу та спрощує їхню підтримку.

Етапи взаємодії метаданих та ролей:

- Ідентифікація ролі користувача. Під час запиту система зчитує роль користувача із JWT-токена або бази даних.
- Читання метаданих. Система звертається до метаданих для визначення доступних полів сутності.
- Проекція даних. Формується об’єкт із доступними полями відповідно до ролі користувача.
- Відповідь користувачу. Відповідь містить лише ті атрибути, які дозволені для поточної ролі або контексту.

3.4.4 Архітектурна схема інтеграції метаданих та RBAC

Компоненти системи:

- Метадані виступають як центральний репозиторій, де зберігається вся необхідна інформація про структуру даних і політики доступу. Ці дані можуть бути збережені у форматі конфігураційних файлів, таких як XML або YAML, або у вигляді таблиць в базі даних. Конфігураційні файли часто використовуються для їхньої зручності та легкості читання людиною, в той час як таблиці в базі даних дозволяють динамічно змінювати метадані без перезапуску системи. Метадані

містять деталізований опис кожної сутності, такі як поля даних, типи даних та обмеження, а також політики доступу, що визначають, хто може бачити або змінювати дані.

- Рольова модель (RBAC) є критично важливим компонентом, що контролює доступ до ресурсів на основі ролей користувачів. Вона взаємодіє з метаданими для забезпечення дотримання політик доступу. У цій моделі ролі присвоюються користувачам на основі їхніх обов'язків та відповідальностей у системі. Кожна роль має набір дозволів, які визначають доступ до певних дій або ресурсів. Це забезпечує гнучкість та зменшує складність управління індивідуальними правами користувачів, оскільки зміни в доступі можуть бути реалізовані шляхом зміни ролі без необхідності переконфігурування прав кожного користувача.

Блок-схема інтеграції:

Рис. 3.5 Блок-схема інтеграції метаданих та RBAC

Взаємозв’язок між метаданими та рольовою моделлю доступу є ключовим фактором у побудові масштабованих та адаптивних інформаційних систем. Поєднання цих підходів дозволяє значно зменшити обсяг коду за рахунок автоматизації створення сутностей, динамічної проекції даних та централізованого управління політиками доступу. Це знижує необхідність дублювання моделей для кожної ролі користувача, спрощуючи підтримку та розвиток системи.

Застосування метаданих забезпечує:

- Зменшення обсягу коду за рахунок автоматичної генерації сутностей і контролю за доступом на рівні атрибутів.
- Гнучкість у зміні правил доступу без необхідності переписувати бізнес-логіку чи змінювати структуру моделей.
- Централізоване управління доступом, що спрощує підтримку та масштабування великих інформаційних систем.
- Підвищення якості даних через стандартизацію форматів та правил валідації, які можуть бути централізовано визначені та управляються через метадані, що знижує ризик помилок при введенні даних і сприяє забезпеченню консистентності даних у всій системі.

Один із важливих аспектів використання метаданих – це зменшення технічного боргу та підвищення адаптивності системи до нових бізнес-вимог. У результаті впровадження цього підходу знижується час, необхідний для додавання нових ролей або розширення функціоналу, що особливо важливо для динамічно зростаючих проектів.

4 ПРАКТИЧНЕ ПОРІВНЯННЯ DTO-ПІДХОДУ ТА ПІДХОДУ З МЕТАДАНИМИ ДЛЯ ВІДОБРАЖЕННЯ СУТНОСТЕЙ

4.1 Постановка задачі до проектів

У цьому розділі представлено практичне порівняння двох підходів до відображення сутностей баз даних у прикладний код:

1. DTO-підхід (Data Transfer Object).
2. Метадані та динамічна проекція.

Для створення API-проектів використано фреймворк Nest.js, який забезпечує модульну архітектуру та спрощує організацію коду. В якості бази даних використовується PostgreSQL, що дозволяє ефективно працювати з реляційними даними та реалізовувати складні запити.

Проекти базуються на тренувальній базі даних AdventureWorks, яка містить великий набір таблиць, пов’язаних із продукцією, замовленнями та клієнтами. Це дозволяє змоделювати реальні бізнес-процеси та перевірити ефективність підходів у великій системі.

Мета розділу:

- Створити два API-проекти, які реалізують однакову функціональність за допомогою різних підходів.
- Довести, що підхід із використанням метаданих є більш гнучким та потребує меншої кількості коду для підтримки та розширення.

Очікувані результати:

1. Скорочення обсягу коду у проекті з метаданими в порівнянні з DTO-підходом.
2. Підвищення гнучкості системи за рахунок зменшення кількості змін у коді при додаванні нових ролей.

4.2 Проект 1: API з DTO-підходом

Проект розроблений на основі фреймворку Nest.js та бази даних PostgreSQL, використовує базу даних Adventure Works для демонстрації своїх можливостей. Adventure Works є добре відомою навчальною базою даних, яка часто використовується для тренування і демонстрації можливостей систем управління базами даних і технологій програмування. Вона включає різноманітні дані, пов'язані з виробництвом, продажем та доставкою товарів, що робить її ідеальною для використання в умовах реального світу.

Основні аспекти проекту з базою даних Adventure Works:

- Інтеграція з Nest.js і Sequelize: Використання Sequelize як ORM дозволяє ефективно інтегрувати структури даних з бази Adventure Works з бізнес-логікою, написаною на Nest.js. Моделі Sequelize відображають таблиці бази даних, забезпечуючи легкий доступ і маніпуляцію даними через сучасні методи програмування.
- Використання DTO для різних рівнів доступу: На основі даних з Adventure Works створено окремі DTO-класи для різних ролей користувачів, що відповідають за вибіркове відображення інформації в залежності від рівня доступу.

Рівні доступу:

1. Адміністратори мають повний доступ до всіх даних, включно з фінансовими показниками та управлінською інформацією.
2. Менеджери бачать обмежену інформацію, яка включає дані про продажі та доступ до деяких маркетингових показників, але без доступу до собівартості продукції.
3. Клієнти або звичайні користувачі доступають тільки до основної інформації про продукцію, такої як назва товару та ціна.

4.2.1 Файлова структура проекту

Для наочності файлова структура представлена в скороченому вигляді:

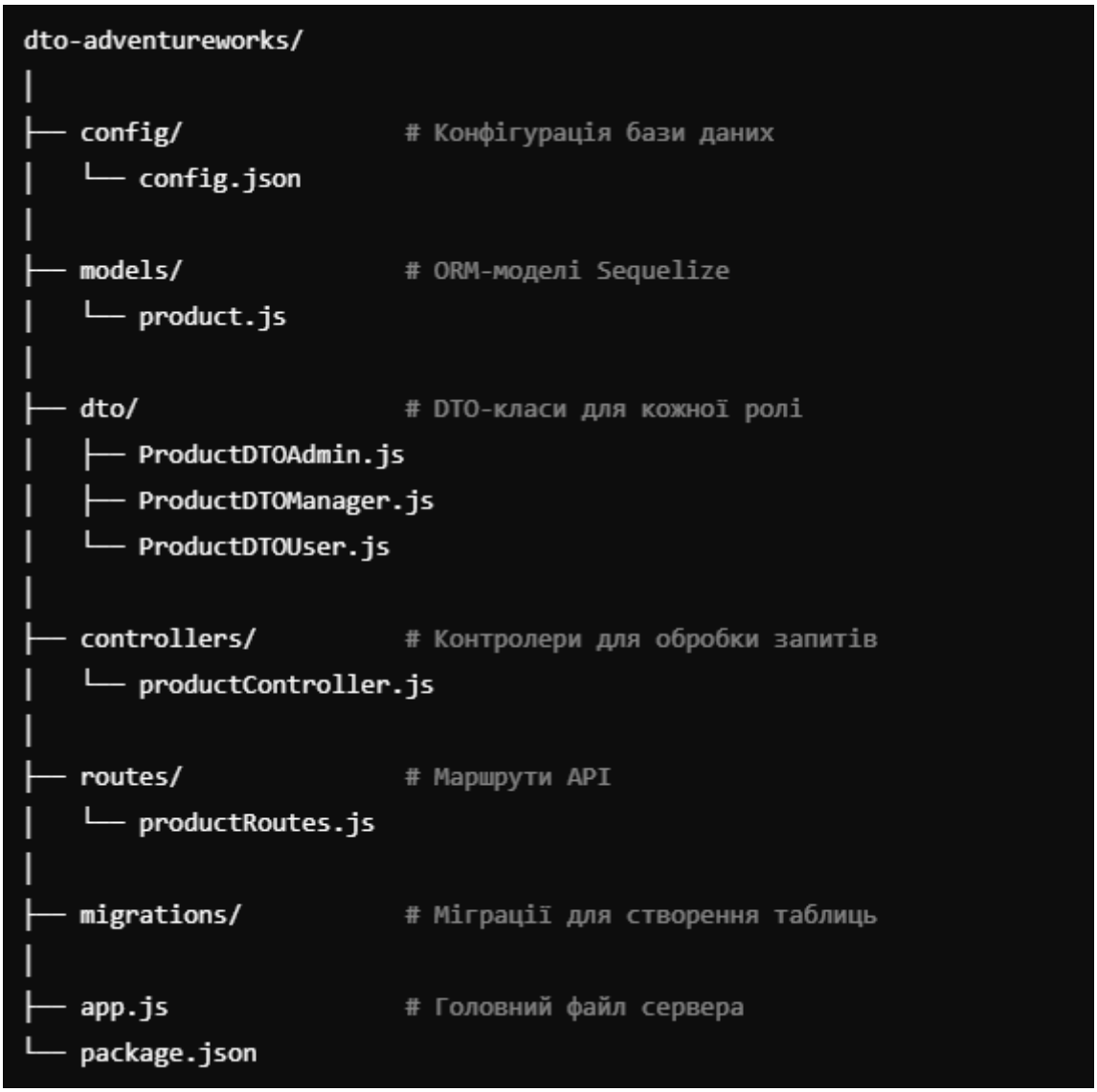


Рис. 4.1. Файлова структура першого проекту

- config/: Директорія містить конфігураційні файли, такі як config.json, які зберігають налаштування бази даних та інші конфігураційні параметри необхідні для проекту.
- models/: Тут розміщені ORM-моделі Sequelize, які відображають таблиці бази даних у коді JavaScript. Наприклад, product.js може містити модель для товарів з визначеннями полів і відносин.
- dto/: Директорія містить DTO-класи, які визначають структури даних, що передаються між сервером і клієнтом. Є окремі файли для різних

ролей, наприклад, ProductDTOAdmin.js для адміністраторів, ProductDTOManager.js для менеджерів і ProductDTOUser.js для звичайних користувачів, що дозволяє налаштовувати вивід відповідно до рівня доступу користувача.

- controllers/: Контролери обробляють запити, прийняті через API, і керують логікою взаємодії з базою даних. productController.js може включати функції для отримання, додавання, оновлення і видалення інформації про продукти.
- routes/: Містить маршрути API, описані в productRoutes.js, які визначають endpoint'и URL для доступу до функцій контролерів. Маршрутизація відіграє ключову роль у визначенні того, які HTTP запити обробляються різними частинами програми.
- migrations/: Директорія для міграцій бази даних, яка містить скрипти для створення, зміни або видалення таблиць у базі даних, забезпечуючи версіонування і управління структурою бази даних.

4.2.2 Ключові моменти коду

1. Створення моделі товарів (Product):

```
module.exports = (sequelize, DataTypes) => {
  const Product = sequelize.define('Product', {
    name: DataTypes.STRING,
    price: DataTypes.DECIMAL,
    cost: DataTypes.DECIMAL,
    category: DataTypes.STRING
  }, {});
  return Product;
};
```

Рис. 4.2 Створення моделі товарів

2. DTO-класи:

```
class ProductDTOAdmin {
  constructor(product) {
    this.id = product.id;
    this.name = product.name;
    this.price = product.price;
    this.cost = product.cost;
    this.category = product.category;
  }
}
```

Рис. 4.3 DTO об’єкт для адміністратора

```
class ProductDTOManager {
  constructor(product) {
    this.id = product.id;
    this.name = product.name;
    this.price = product.price;
    this.category = product.category;
  }
}
```

Рис. 4.4. DTO об’єкт для менеджера (без собівартості)

```
class ProductDTOUser {
  constructor(product) {
    this.id = product.id;
    this.name = product.name;
    this.price = product.price;
  }
}
```

Рис. 4.5. DTO об’єкт для користувача (мінімальна інформація)

3. Контролер для отримання товарів:

```
const { Product } = require('../models');
const { ProductDTOAdmin, ProductDTOManager, ProductDTOUser } = require('../dto');

async function getProducts(req, res) {
  const products = await Product.findAll();
  const role = req.user.role;

  let result;
  if (role === 'admin') {
    result = products.map(p => new ProductDTOAdmin(p));
  } else if (role === 'manager') {
    result = products.map(p => new ProductDTOManager(p));
  } else {
    result = products.map(p => new ProductDTOUser(p));
  }

  res.json(result);
}
```

Рис. 4.6. Контролер для отримання товарів

1. Запит: Користувач відправляє запит на отримання товарів, можливо зазначивши свою роль (наприклад, admin, manager, user). Запит обробляється методом getProducts.

2. Отримання даних: Контролер викликає метод findAll() з ProductService, який запитує всі записи товарів з бази даних.

3. Проекція за допомогою DTO: Залежно від ролі користувача, отримані дані про товари трансформуються до відповідного формату DTO. Це дозволяє відправити клієнту лише ті дані, до яких у нього є доступ.

4. Відправлення відповіді: Отримані DTO передаються клієнту у відповіді з статусом HTTP 200 OK.

4.2 Проект 2: API з метаданими та проекціями

Проект реалізовано на основі Nest.js та PostgreSQL, з використанням підходу метаданих для динамічної проекції даних. На відміну від DTO, цей метод передбачає створення єдиної моделі для сутності Product, яка проектується залежно від ролі користувача під час виконання запиту.

Всі правила доступу та проекції описуються у конфігураційних YAML-файлах, що забезпечує централізоване управління доступом. Система зчитує метадані та формує відповідний об’єкт для клієнта, враховуючи роль користувача.

4.2.1 Файлова структура проект

Для наочності файлова структура представлена в скороченому вигляді.

```
metadata-adventureworks/
│
├── config/
│   └── config.json
│
├── models/
│   └── product.js
│
├── metadata/              # YAML-файли метаданих
│   └── metadata.yaml
│
├── controllers/
│   └── productController.js
│
├── routes/
│   └── productRoutes.js
│
├── migrations/
│
├── app.js
└── package.json
```

Рис. 4.7 Файлова структура проекту 2

4.2.2 Ключові моменти коду

1. Створення метаданих для ролей (metadata/metadata.yaml):

```yaml
Product:
  fields:
    id: integer
    name: string
    price: decimal
    cost: decimal
    category: string
  roles:
    admin: ["id", "name", "price", "cost", "category"]
    manager: ["id", "name", "price", "category"]
    user: ["id", "name", "price"]
```

Рис. 4.8 Метадані в yaml файлі

2. Контролер API з метаданими:

```javascript
const fs = require('fs');
const yaml = require('js-yaml');
const { Product } = require('../models');

const metadata = yaml.load(fs.readFileSync('./metadata/metadata.yaml', 'utf8'));

async function getProducts(req, res) {
  const products = await Product.findAll();
  const role = req.user.role;

  const result = products.map(product => projectData(product, role));
  res.json(result);
}

function projectData(entity, role) {
  const allowedFields = metadata.Product.roles[role];
  const result = {};

  allowedFields.forEach(field => {
    result[field] = entity[field];
  });

  return result;
}
```

Рис. 4.9 Контролер API проекту №2

Контролер API з метаданими в проекті на основі Nest.js виконує динамічну проекцію даних під час кожного запиту. Замість того, щоб повертати повну модель сутності, він формує відповідь, виходячи з ролі користувача та конфігурації, що зберігається в YAML-файлі з метаданими.

4.3 Порівняння проектів

У цьому підрозділі проводиться порівняння двох API-проектів, створених на основі бази даних AdventureWorks. Мета – оцінити різницю в обсязі коду та гнучкості підходів, а також виявити ключові переваги та недоліки кожного з них.

Критерії порівняння:

1. Кількість коду – загальний обсяг коду в кожному проекті.
2. Гнучкість – кількість змін у коді при додаванні нової ролі.

4.3.1 Порівняння обсягу коду

Для порівняння підраховується кількість рядків коду у ключових компонентах проектів:

1. DTO-підхід: Кожна нова роль вимагає створення окремого DTO-класу.
2. Метадані: Всі зміни обмежуються оновленням конфігураційного YAML-файлу.

Таблиця 4.1

Порівняння обсягу коду

Компонент	DTO-підхід (рядків коду)	Метадані (рядків коду)
Моделі	150	150
Контролери	230	180

Продовження таблиці 4.1

Порівняння обсягу коду

Компонент	DTO-підхід (рядків коду)	Метадані (рядків коду)
DTO-класи	450	0
Метадані (YAML)	0	120
Всього	830	450

Діаграма кількості ендпоінтів показує загальну кількість API ендпоінтів у проекті, що вказує на кількість різних функцій або сервісів, доступних через API.

Цей показник корисний для розуміння масштабу API, керування навантаженням на сервер та планування ресурсів. Більша кількість ендпоінтів може вказувати на більшу складність і більші вимоги до документації та тестування.

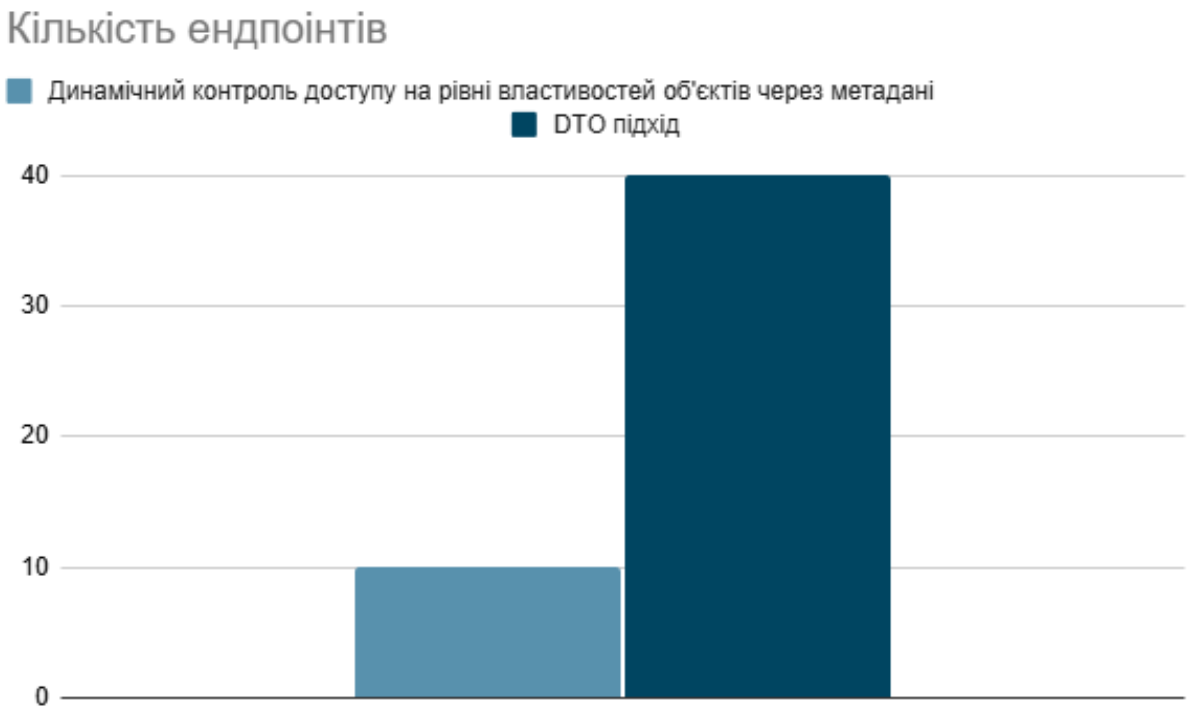


Рис. 4.10 Діаграма кількості ендпоінтів на проект

Діаграма кількості рядків коду на проект призначена для відображення загальної кількості рядків коду у кожному з проектів. Вона дозволяє оцінити обсяг роботи, складність та потенційну витрату часу на розробку та підтримку проекту.

Аналізуючи кількість рядків коду, можна зробити висновки про ефективність написання коду, наявність можливих дублікацій, та оцінити загальну структуру проекту. Це також допомагає в порівнянні з іншими проектами або з індустріальними стандартами.

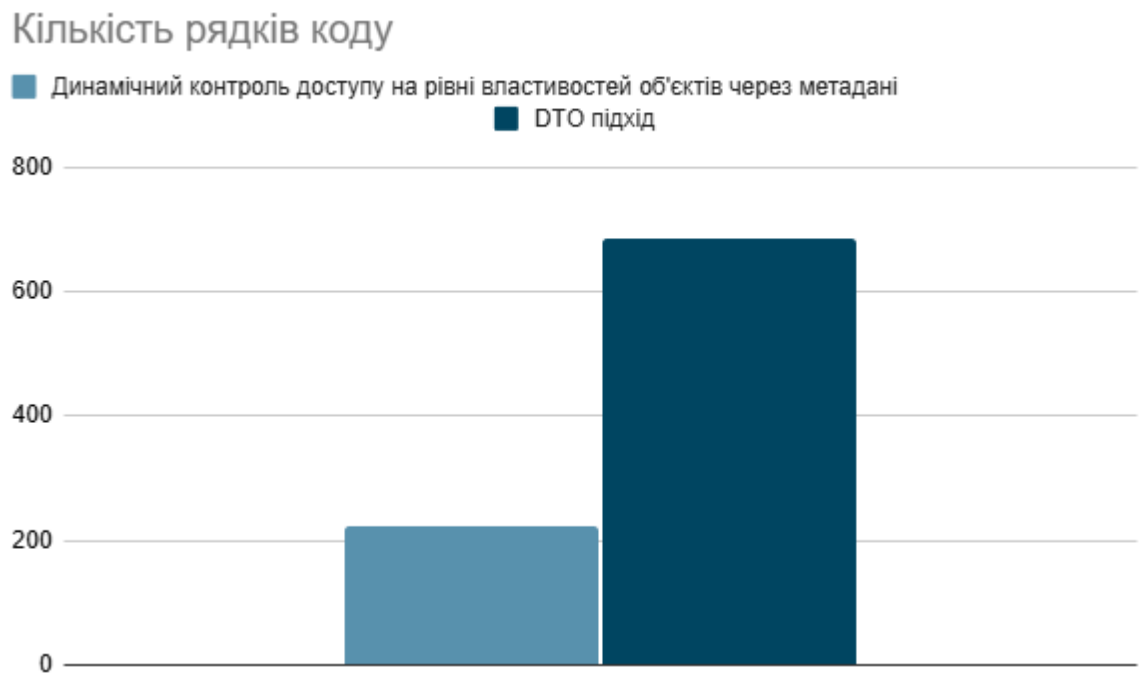


Рис. 4.11 Діаграма кількості рядків коду на проект

Діаграма кількості файлів на проект відображає кількість файлів, що входять до складу проекту, що дає уявлення про його розмір та структуру.

Оцінка кількості файлів допомагає зрозуміти ступінь модульності проекту та його організацію. Модульна структура з багатьма файлами може сприяти легшій підтримці та розширенню, але також вимагає більш ретельного управління залежностями та інтеграції компонентів.

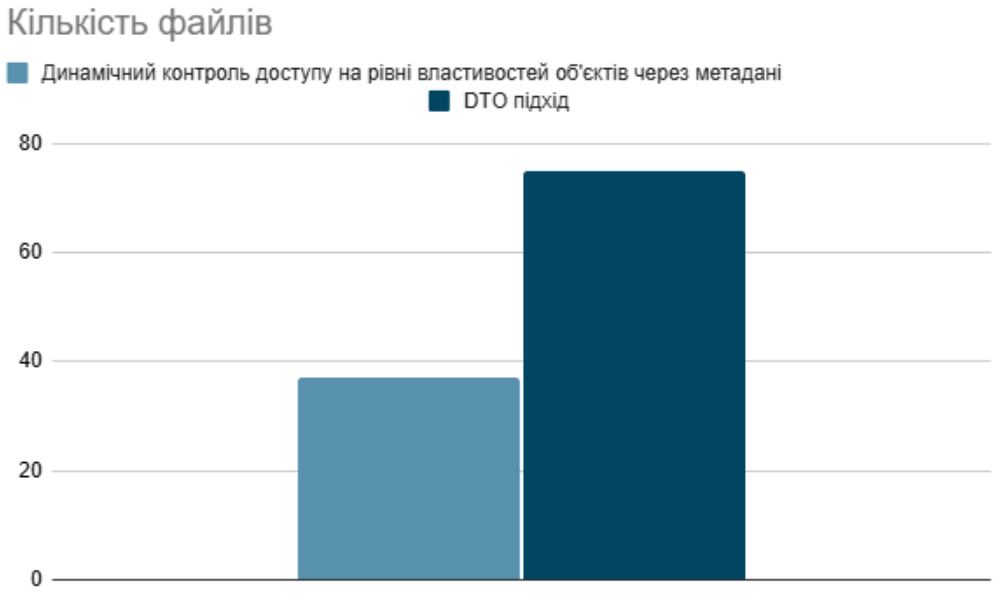


Рис. 4.12 Діаграма кількості файлів на проект

Аналіз:

- Дублювання коду: Оскільки кожен новий клас DTO потребує окремої реалізації, це призводить до значного дублювання логіки доступу до даних. Таке дублювання ускладнює підтримку коду та може вести до більшої кількості помилок, особливо при необхідності внесення змін у бізнес-логіку.
- Централізація логіки проекції: Застосування метаданих дозволяє зосередити всі правила проекції та доступу до даних у єдиному YAML-файлі. Це спрощує управління правилами доступу, адже замість редагування численних класів DTO, зміни можна внести лише у відповідні метадані.
- Зменшення обсягу коду: Підхід з використанням метаданих значно знижує кількість коду у системі, оскільки усуває потребу у створенні численних класів DTO. За оцінками, такий підхід може зменшити обсяг коду приблизно в 2 рази, що сприяє більшій ефективності розробки та легшій підтримці.

4.3.2 Порівняння гнучкості

Для оцінки гнучкості було проведено експеримент:

- Додано нову роль supervisor, яка має обмежений доступ до атрибутів товару (тільки id, name та price).
- Виміряна кількість змін у коді, необхідних для реалізації цієї ролі.

Таблиця 4.2

Порівняння кількості необхідних змін

Компонент	DTO-підхід (зміни)	Метадані (зміни)
Створення нового класу DTO	1	0
Оновлення контролера	1	0
Додавання ролі в метадані	0	1
Загальна кількість змін	2	1

Аналіз:

- DTO-підхід потребує створення нового класу DTO (ProductDTOSupervisor) та внесення змін у контролер. Це збільшує ризик помилок і вимагає додаткових перевірок.
- Підхід з метаданими дозволяє додати роль supervisor шляхом редагування YAML-файлу без змін у контролерах чи моделях. Уся логіка доступу централізована у метаданих.

ВИСНОВОК

У ході виконання дипломної роботи було досліджено та розроблено метод проекції даних із бази даних PostgreSQL у прикладний код на платформі Node.js з використанням Nest.js. Основна увага приділялася порівнянню двох підходів: класичного використання DTO (Data Transfer Object) та динамічної проекції даних на основі метаданих.

Практична частина роботи включала створення двох API-проектів на основі бази даних AdventureWorks, що дозволило змоделювати реальні сценарії управління доступом у багаторівневих системах. Під час реалізації проектів було досягнуто наступних результатів:

- Проект із DTO продемонстрував значне зростання кількості коду при додаванні нових ролей та атрибутів. Відсутність централізованого управління доступом призвела до дублювання коду та ускладнення підтримки.
- Проект із метаданими забезпечив зниження обсягу коду приблизно в 2 рази за рахунок відсутності необхідності створення окремих DTO-класів для кожної ролі. Усі правила доступу були централізовані у YAML-файлах, що дозволило легко додавати нові ролі та адаптувати систему до змін бізнес-логіки.

Проведений аналіз продемонстрував, що підхід із використанням метаданих є більш гнучким та потребує написання меншої кількості коду. Додавання нових ролей вимагало мінімальних змін у коді – лише оновлення метаданих, тоді як у DTO-системі необхідно було створювати нові класи та вносити зміни у контролери.

Показник кількості змін у коді при додаванні нової ролі був обраний як основний критерій оцінки гнучкості системи. У проекті з метаданими ця кількість була мінімальною, що підтверджує ефективність запропонованого підходу.

Запропонований метод дозволяє значно зменшити технічний борг, спростити рефакторинг та забезпечити швидке масштабування системи. Його впровадження є доцільним для великих інформаційних систем із багаторівневим доступом, де кількість ролей користувачів може постійно змінюватися.

Отримані результати можуть бути використані у майбутніх проектах для оптимізації процесу управління доступом до даних та підвищення гнучкості інформаційних систем.

Результати дослідження апробовано та опубліковано у наступних тезах доповіді на конференціях:

1. Василіненко Н.М., Яксевич. В.О. Гнучкий підхід до управління доступом у багатокористувацьких системах: від DTO до метаданих. // VIII. Міжнародна науково-практична конференція «Інтеграція світових наукових процесів як основа суспільного прогресу». – подано до друку.

2. Василіненко Н.М., Яксевич. В.О. Роль метаданих у динамічному управлінні доступом до даних: підхід на основі RBAC. // VIII. Міжнародна науково-практична конференція «Інтеграція світових наукових процесів як основа суспільного прогресу». – подано до друку.

ПЕРЕЛІК ПОСИЛАНЬ

1. Ihechikara Abba. What is an ORM – The Meaning of Object Relational Mapping Database Tools. 15.11.2020 URL: https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/ (дата звернення 15.09.2024).
2. Justin Ellingwood. What is an ORM? 13.11.2022 URL: https://www.prisma.io/dataguide/types/relational/what-is-an-orm (дата звернення 13.09.2024).
3. Rahul Awati. What is an ORM? 15.07.2021 URL: https://www.theserverside.com/definition/object-relational-mapping-ORM (дата звернення 16.09.2024).
4. Mia Liang. Understanding Object-Relational Mapping: Pros, Cons, and Types 11.03.2021 URL: https://www.altexsoft.com/blog/object-relational-mapping/ (дата звернення 23.09.2024).
5. Владислав Огородніков. Про проблему DTO та шляхи її вирішення. 27.06.2023 URL: https://dou.ua/forums/topic/43912/ (дата звернення 24.09.2024).
6. Brad Beighton. Why I choose NestJS for a SaaS API. 29.07.2023 URL: https://bradbeighton.medium.com/nestjs-the-pros-and-cons-aff714607b07 (дата звернення 25.09.2024).
7. Alex Ryabtsev. Top 14 Pros of Using Django for Python Web Development. 28.10.2024 URL: https://djangostars.com/blog/top-14-pros-using-django-web-development/ (дата звернення 30.10.2024).
8. Jakub Orczyk. Spring Framework vs. Spring Boot – pros and cons. 20.08.2023 URL: https://vmsoftwarehouse.com/spring-framework-vs-spring-boot-pros-and-cons (дата звернення 01.11.2024).
9. Jean-Paul Rustom. JWT explained in 4 minutes (With Visuals). 24.01.2024 URL: https://dev.to/jaypmedia/jwt-explained-in-4-minutes-with-visuals-g3n (дата звернення 01.11.2024).

10. Vijay Kanade. What Is XML (Extensible Markup Language)? Meaning, Elements, and Benefits. 08.03.2023. URL: https://www.spiceworks.com/tech/tech-general/articles/what-is-xml/ (дата звернення 01.11.2024).
11. Zubair Idris Aweda. How to Use SQL Triggers. 21.01.2023. URL: https://www.freecodecamp.org/news/sql-triggers/ (дата звернення 01.11.2024).
12. Ravikiran A S. Understanding Stored Procedures in SQL: Benefits & Creation! 07.07.2024 URL: https://www.simplilearn.com/tutorials/sql-tutorial/stored-procedure-in-sql (дата звернення 02.11.2024).
13. Chanuka Vidanapathirana. Stored procedures advantages and disadvantages. 10.01.2024 URL: https://medium.com/@chanukasuboth1/stored-procedures-advantages-and-disadvantages-c0aa2d50004b (дата звернення 02.11.2024).
14. Jay McDoniel. NestJS Metadata Deep Dive. 18.07.2023 URL: https://trilon.io/blog/nestjs-metadata-deep-dive (дата звернення 03.11.2024).
15. Alaran Ayobami. How To Set Up TypeORM DataSource in Your NestJS Project. 25.04.2024 URL: https://www.freecodecamp.org/news/how-to-setup-typeorm-datasource-nestjs-app/ (дата звернення 03.11.2024).
16. Casciaro M., Mammino L. Node.js Design Patterns. 3rd ed. Birmingham: Packt Publishing, 2020. 526 p.
17. Schönig H.-J. Mastering PostgreSQL 12. 3rd ed. Birmingham: Packt Publishing, 2019. 488 p.
18. Ihrig C.J., Bretz A. Pro Node.js for Developers. New York: Apress, 2014. 300 p.
19. Obe R., Hsu L. PostgreSQL: Up and Running. 3rd ed. Sebastopol: O'Reilly Media, 2018. 372 p.
20. Mardan A. Practical Node.js: Building Real-World Scalable Web Apps. 2nd ed. New York: Apress, 2018. 320 p.
21. Juba S., Vannahme A. Learning PostgreSQL 11. Birmingham: Packt Publishing, 2018. 452 p.
22. Syed B.A. Beginning Node.js. New York: Apress, 2014. 308 p.
23. Riggs S., Ciolli G. Expert PostgreSQL. 2nd ed. New York: Apress, 2015. 350 p.
24. Wilson J. Node.js 8 the Right Way. 2nd ed. Pragmatic Bookshelf, 2018. 320 p.

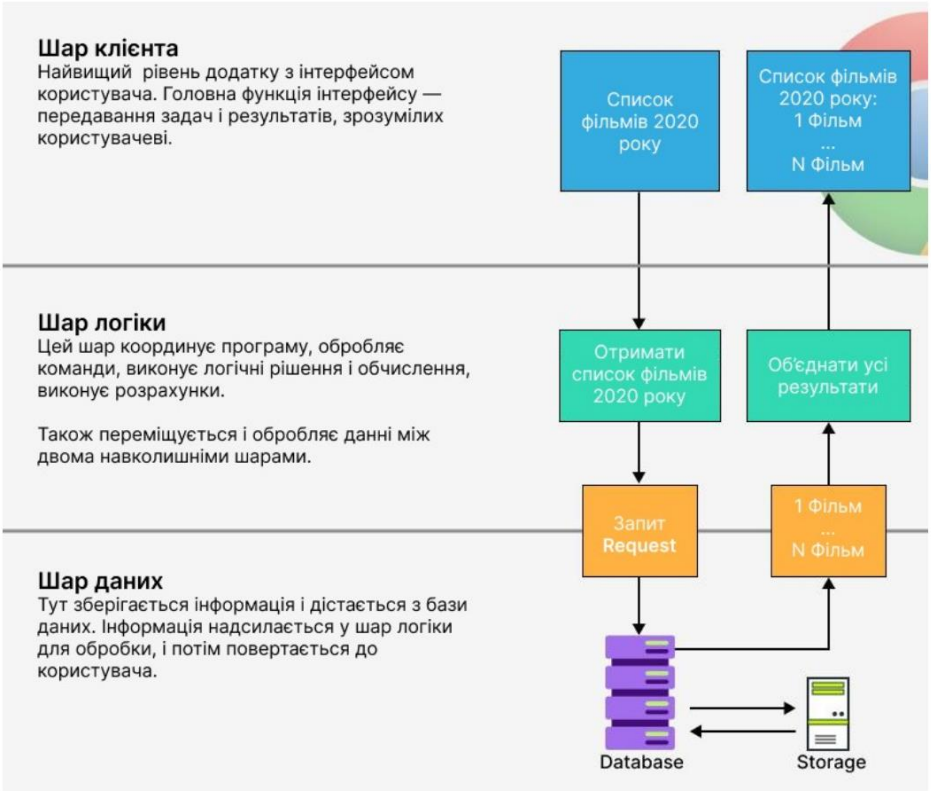
25. Ahmed I., Fayyaz A., Shahzad A. PostgreSQL Developer’s Guide. Birmingham: Packt Publishing, 2015. 504 p.
26. Herron D. Node.js Web Development. 4th ed. Birmingham: Packt Publishing, 2016. 422 p.
27. Dar U., Krosing H. PostgreSQL Server Programming. 2nd ed. Birmingham: Packt Publishing, 2015. 300 p.
28. Cantelon M., Harter M., Holowaychuk T.J., Rajlich N. Node.js in Action. 2nd ed. Manning Publications, 2017. 384 p.
29. Churcher C. Beginning Database Design. 2nd ed. New York: Apress, 2012. 252 p.
30. Norman M. The Definitive Guide to PostgreSQL. 3rd ed. New York: Apress, 2019. 610 p.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: покращення технології для відображення сутностей бази даних у прикладний код.
Об'єкт дослідження: Процес відображення сутностей бази даних у прикладний код.
Предмет дослідження: Технології і засоби відображення сутностей бази даних у прикладний код.

ТРИШАРОВА АРХІТЕКТУРА

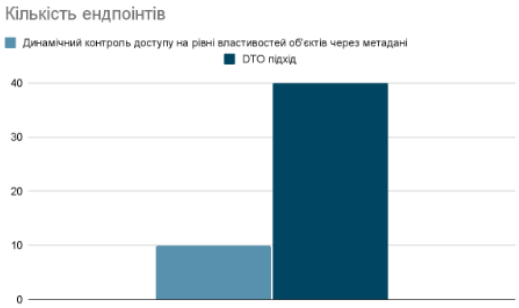


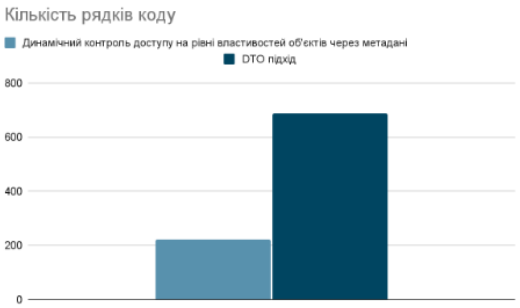
АРХІТЕКТУРНА СХЕМА ІНТЕГРАЦІЇ МЕТАДАНИХ ТА RBAC (ROLE-BASED ACCESS CONTROL)

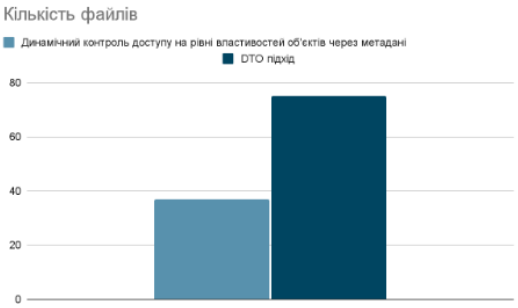
5

ПРАКТИЧНИЙ РЕЗУЛЬТАТ

ПОРІВНЯЛЬНИЙ АНАЛІЗ







8

ПРАКТИЧНИЙ РЕЗУЛЬТАТ

Файлові структури

Динамічний контроль доступу на рівні властивостей об'єктів через метадані

```
src/
├── app.module.ts
├── main.ts
├── employees/
│   ├── employee.controller.ts
│   ├── employee.service.ts
│   ├── employee.entity.ts
├── products/
│   ├── product.controller.ts
│   ├── product.service.ts
│   ├── product.entity.ts
├── access/
│   ├── access.decorator.ts      # Реалізація декораторів для ролей
│   ├── access.service.ts        # Сервіс для фільтрації властивостей на основі ролей
│   ├── access.guard.ts          # Опціонально: Guard для перевірки ролі в токені
├── sales-orders/
│   ├── sales-order.controller.ts
│   ├── sales-order.service.ts
│   ├── sales-order.entity.ts
```

DTO підхід

```
src/
├── app.module.ts
├── main.ts
├── employees/
│   ├── employee.controller.ts
│   ├── employee.service.ts
│   ├── dto/
│   │   ├── employee-admin.dto.ts
│   │   ├── employee-manager.dto.ts
│   │   ├── employee-self.dto.ts
│   │   ├── employee-customer.dto.ts
│   ├── employee.entity.ts
├── products/
│   ├── product.controller.ts
│   ├── product.service.ts
│   ├── dto/
│   │   ├── product-admin.dto.ts
│   │   ├── product-manager.dto.ts
│   │   ├── product-self.dto.ts
│   │   ├── product-customer.dto.ts
│   ├── product.entity.ts
├── sales-orders/
│   ├── sales-order.controller.ts
│   ├── sales-order.service.ts
│   ├── dto/
│   │   ├── sales-order-admin.dto.ts
│   │   ├── sales-order-manager.dto.ts
│   │   ├── sales-order-self.dto.ts
│   │   ├── sales-order-customer.dto.ts
│   ├── sales-order.entity.ts
```

7

ВИСНОВКИ

Проведений аналіз продемонстрував, що підхід із використанням метаданих є більш гнучким та потребує написання меншої кількості коду. Додавання нових ролей вимагало мінімальних змін у коді – лише оновлення метаданих, тоді як у DTO-системі необхідно було створювати нові класи та вносити зміни у контролери.

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Тези доповідей:

1. Василіненко Н.М., Яксевич. В.О. Гнучкий підхід до управління доступом у багатокористувацьких системах: від DTO до метаданих. // VIII. Міжнародна науково-практична конференція «Інтеграція світових наукових процесів як основа суспільного прогресу». – подано до друку
2. Василіненко Н.М., Яксевич. В.О. Роль метаданих у динамічному управлінні доступом до даних: підхід на основі RBAC. // VIII. Міжнародна науково-практична конференція «Інтеграція світових наукових процесів як основа суспільного прогресу». – подано до друку

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

services/product.service.ts

```
import { Injectable } from '@nestjs/common';
import { InjectModel } from '@nestjs/sequelize';
import { Product } from '../models/product.model';

@Injectable()
export class ProductService {
 constructor(
  @InjectModel(Product)
  private productModel: typeof Product
 ) {}

 async findAll() {
  return this.productModel.findAll();
 }
}
```

controllers/product.controller.ts

```
import { Controller, Get, Request } from '@nestjs/common';
import { ProductService } from '../services/product.service';
import * as yaml from 'js-yaml';
import * as fs from 'fs';

@Controller('products')
export class ProductController {
 private metadata;

 constructor(private readonly productService: ProductService) {
  // Завантаження метаданих під час ініціалізації контролера
  this.metadata = yaml.load(
   fs.readFileSync('./metadata/product.yaml', 'utf8')
  );
 }

 @Get()
 async getProducts(@Request() req) {
  const role = req.user.role;  // Отримання ролі користувача
  const products = await this.productService.findAll();

  // Динамічна проекція даних
  const projectedData = products.map((product) =>
   this.projectData(product, role)
  );
  return projectedData;
 }

 // Проекція даних на основі ролі та метаданих
 private projectData(entity, role) {
  const allowedFields = this.metadata.Product.roles[role] || [];
  const projectedEntity = {};

  allowedFields.forEach((field) => {
   projectedEntity[field] = entity[field];
  });

  return projectedEntity;
 }
}
```

metadata/product.yaml

```
Product:
 fields:
  id: integer
  name: string
  price: decimal
  cost: decimal
  category: string
 roles:
  admin: ["id", "name", "price", "cost", "category"]
  manager: ["id", "name", "price", "category"]
  user: ["id", "name", "price"]
class ProductDTOUser {
 constructor(product) {
  this.id = product.id;
  this.name = product.name;
  this.price = product.price;
 }
}
class ProductDTOManager {
 constructor(product) {
  this.id = product.id;
  this.name = product.name;
  this.price = product.price;
  this.category = product.category;
 }
}
class ProductDTOAdmin {
 constructor(product) {
  this.id = product.id;
```

```
    this.name = product.name;
    this.price = product.price;
    this.cost = product.cost;
    this.category = product.category;
  }
}
```