МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ПРИРОДОКОРИСТУВАННЯ
ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

другого (магістерського) рівня вищої освіти

на тему: «Розробка програми автоматизованого оцінювання складності міграції реляційних баз даних»

Виконав: здобувач 6 курсу гр. Іт-62
Спеціальність 126 «Інформаційні системи та технології»

Бурко Денис Олегович

Керівник: Шувар Б.І.

Рецензент: ____________

ДУБЛЯНИ-2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ПРИРОДОКОРИСТУВАННЯ
ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Освітній ступінь «Магістр»
Спеціальність 126 «Інформаційні системи та технології»

ЗАТВЕРДЖУЮ
Завідувач кафедри
____
(підпис)
д.т.н., професор, Тригуба А. М.
(вч. звання, прізвище, ініціали)
"____" ___________2024 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Бурка Дениса Олеговича
(прізвище, ім'я, побатькові)

1. Тема роботи *«Розробка програми автоматизованого оцінювання складності міграції реляційних баз даних»*

керівник роботи *к.е.н., доцент Шувар Б. І.*
(наук.ступінь, вч. звання, прізвище, ініціали)

затверджені наказом Львівського НУП______________________

2. Строк подання студентом роботи 15.12.2024 р.

3. Вихідні дані: *Міжнародні стандарти щодо управління даними, міграції реляційних баз даних, забезпечення сумісності та безпеки даних, а також вимоги щодо оптимізації процесів переносу даних.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які необхідно розробити) *1. Стан проблемної активності. 1.1 Огляд проблемної області. 1.2 Виклики автоматизації оцінювання міграції баз даних. 1.3 Аналіз наявних рішень. 2. Інформаційне та математичне забезпечення. 2.1 Реляційні бази даних. 2.2 Docker. 2.3 Java. Spring Boot. PostgreSQL. Програмне та технічне забезпечення. Встановлення Docker. 3.2 Ініціалізація проєкту3.4 Підготовка до розробки. 3.5 Розробка проєкту. 3.6 Тестування проєкту. 4. Охорона праці та безпека у надзвичайних ситуаціях. 4.1. Охорона праці. 4.2. Безпека у надзвичайних ситуаціях. 5. Визначення ефективності. 5.1 Аналіз ефективності програми. 5.2 Оцінка ефективності програми. Висновки і пропозиції. Список використаних джерел. Додатки.*

5. Перелік ілюстраційного матеріалу (з точним зазначенням обов'язкових схем та моделей): рисунки, таблиці у вигляді презентації

6. Консультанти з розділів

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата		Відмітка про виконання
		завдання видав	завдання прийняв	
1, 2, 3	Шувар Б.І., доцент кафедри ІТ	16.09.2024р.	16.09.2024р.	
4	Городецький І.М., доцент кафедри управління проектами та безпеки виробництва	17.09.2024р.	17.09.2024р.	

7. Дата видачі завдання 16.09.2024 р

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Приміт-ка
1	*Отримання завдання. Вивчення рекомендованої літератури по темі роботи.*	03.09 - 10.09.24	
2	*Аналіз існуючих підходів до міграції реляційних баз даних.*	10.09 - 16.09.24	
4.	*Розробка моделі бази даних для програми автоматизованого оцінювання міграції.*	16.09 - 31.09.24	
5	*Написання програми для автоматизованого оцінювання міграції баз даних*	01.10 - 31.10.24	
6	*Тестування програми та завершення роботи*	01.11 - 30.11.24	
7	*Виправлення зауважень та перевірка на плагіат*	01.12 – 10.12.24	

Студент __________ Бурко Д.О.
(підпис) (прізвище та ініціали)

Керівник роботи __________ Шувар Б.І
(підпис) (прізвище та ініціали)

УДК ___004.42 : 004.65____

Розробка програми автоматизованого оцінювання складності міграції реляційних баз даних.

Бурко Д.О. Кафедра ІТ – Дубляни, Львівський НУП, 2024.

Кваліфікаційна робота: _78_ с. текст. част., _57_ рис., _1_ табл., _35_ джерел.

Об’єктом дослідження є процес автоматизованої оцінки складності міграції реляційних баз даних, що включає аналіз об’єктів бази, їх взаємозалежностей, а також специфічних механізмів збереження та обробки даних.

Метою роботи є розробка програмного забезпечення, яке спрощує та прискорює аналіз і оцінку складності міграції реляційних баз даних. Таке рішення має забезпечувати автоматичне виявлення залежностей між об’єктами, оцінку специфічних характеристик бази та аналіз складних SQL-запитів.

Предметом дослідження є реляційні бази даних, які є основою сучасних інформаційних систем, а також використання сучасних технологій, таких як Docker для контейнеризації, Java як основної мови розробки, Spring Boot для використання як фреймворку, та PostgreSQL як однієї з провідних та бесплатних реляційних систем управління базами даних.

Практичне значення роботи полягає у створенні інструменту, що дозволить автоматизувати процес оцінки складності міграції баз даних. Це значно полегшить планування міграційних проектів, зменшить витрати часу і ресурсів, а також підвищить якість аналізу завдяки детальному виявленню потенційних складнощів.

Ключові слова: База даних, міграція, Docker, Java, Spring Boot, PostgreSQL, SQL, автоматизація.

ЗМІСТ

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ 7

1.1 Огляд проблемної області 7

1.2 Виклики автоматизації оцінювання міграції баз даних 8

1.3 Аналіз наявних рішень 8

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ 14

2.1 Реляційні бази даних 14

2.2 Docker 16

2.3 Java 17

2.4 Spring Boot 19

2.5 PostgreSQL 20

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ 23

3.1 Встановлення Docker 23

3.2 Ініціалізація проєкту 23

3.4 Підготовка до розробки 25

3.5 Розробка проєкту 27

3.5.1 Створення Docker зображення 27

3.5.2 Створення структури бази даних 29

3.5.3 Розробка серверної частини 38

3.6 Тестування проєкту 57

РОЗДІЛ 4. ОХОРОНА ПРАЦІ 60

4.1 Охорона праці 60

4.2 Безпека у надзвичайних ситуаціях 61

РОЗДІЛ 5. ВИЗНАЧЕНЯ ЕФЕКТИВНОСТІ 63

5.1 Аналіз ефективності програми 63

5.2 Оцінка ефективності програми 66

ВИСНОВКИ І ПРОПОЗИЦІЇ 66

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 70

ДОДАТКИ .... **Помилка! Закладку не визначено.**

ВСТУП

Реляційні бази даних лежать в основі багатьох сучасних систем, від банківських платформ до сервісів електронної комерції. Однак із часом вони застарівають, і їх потрібно переносити на новіші платформи, які краще справляються з великими обсягами даних, складними запитами та інтеграцією з іншими системами. Міграція таких баз - це завжди складний процес, що включає аналіз структури бази, виявлення зв’язків між таблицями, оцінку складності SQL-запитів і розуміння того, як саме працює база.

Часто все це робиться вручну, що займає багато часу і не завжди гарантує точність. Наприклад, якщо база складається з тисяч таблиць і запитів, пропустити щось або неправильно оцінити залежності між об’єктами дуже легко. У результаті з'являються проблеми під час самої міграції: якісь дані можуть втратитися, запити перестають працювати, або платформа починає гальмувати через помилки в налаштуваннях.

Щоб уникнути цих проблем, потрібен інструмент, який самостійно аналізує базу даних. Він має визначати, як таблиці пов’язані одна з одною, наскільки складними є запити, та чи використовуються унікальні механізми, які потрібно врахувати при перенесенні. Такий інструмент може значно спростити підготовку до міграції та зробити її більш передбачуваною.

Для його створення використовуються технології, які роблять розробку швидкою та зручною. Наприклад, Docker дозволяє запускати бази у тестовому середовищі без зайвих налаштувань, Java підходить для реалізації складних обчислень, а Spring Boot спрощує розробку веб-додатків. Як базу даних обрали PostgreSQL - вона безкоштовна і має всі необхідні можливості для інтеграції з таким інструментом.

У результаті це рішення дозволить швидко й точно оцінювати бази даних перед міграцією, зменшуючи ризик помилок і прискорюючи підготовку до перенесення.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1 Огляд проблемної області

Міграція реляційних баз даних - це складний процес, який вимагає глибоко аналізу існуючої бази, а також оцінки її складності та розробки чіткого плану переходу на нову платформу [7]. Цей процес також ускладнюється розміром бази даних, кількості її об'єктів, взаємозв'язками між ними та використанням спеціфічних функцій певної бази даних.

Сьогодні міграція баз даних може стати викликом для багатьох компаній, які прагнуть перейти на сучасніші платформи або адаптувати свої дані для роботи з хмарними технологіями. Ручна оцінка складності такого переходу може бути тривалою, трудомісткою та помилковою, що вимагає автоматизації цього процесу.

До того ж оцінка складності міграції баз даних часто здійснюється вручну або за допомогою частково автоматизованих інструментів, які, однак, не завжди забезпечують повноту аналізу. Основними викликами є об’єм метаданих, що необхідно обробити, складність взаємозв'язків між об'єктами бази, а також специфічні залежності від функціональних можливостей обраної Бази даних.

Автоматизація цього процесу дозволить значно полегшити та прискорити оцінку складності міграції реляційних баз даних, зменшити кількість людських помилок і забезпечити більш точний і детальний аналіз [8]. Це особливо важливо в умовах постійного розвитку технологій баз даних, а також з огляду на зростання обсягів даних і необхідність їхньої інтеграції з новими системами. Тому автоматизація оцінки складності міграції реляційних баз даних стає все більш актуальною і необхідною для забезпечення безпечного та ефективного переходу на нові платформи.

1.2 Виклики автоматизації оцінювання міграції баз даних

Різноманіття типів об'єктів у базах даних, таких як таблиці, представлення, процедури, функції, тригери та індекси [9], вимагає врахування їхніх специфічних механізмів зберігання, виклику та обробки. Взаємозалежність між об'єктами додає складності, адже зміна одного з них може спричинити каскадні зміни в інших. Крім того, кожна база даних має свої унікальні особливості, що стосуються збереження даних, управління транзакціями та забезпечення продуктивності. Складні SQL-запити, зокрема в процедурах і функціях, часто оптимізовані для конкретної бази даних, що потребує адаптації або переписування під час міграції.

Автоматизація цього процесу дає значні переваги. Вона дозволяє суттєво заощадити час на оцінку складності міграції, мінімізувати ймовірність людських помилок, пов’язаних із ручним збором та аналізом метаданих, а також забезпечує зручність обміну даними, наприклад, завдяки експорту у форматі HTML для інтеграції з іншими системами чи подальшого аналізу.

Масштабованість автоматизованого інструменту дозволяє працювати як із невеликими базами даних, так і з великими корпоративними системами, тоді як можливість інтеграції з іншими аналітичними та міграційними інструментами розширює його функціональність.

Гнучкість програмного забезпечення забезпечується динамічною зміною конфігурацій, що дозволяє адаптувати його під різні потреби користувачів.

1.3 Аналіз наявних рішень

На даний час не існує програм, які би могли автоматично оцінити складність баз даних і оцінити скільки ця міграція займе часу, існують тільки програми, які можуть мігрувати ці дані автоматично, або напівавтоматично, тому давайте проаналізуємо їх:

AWS Database Migration

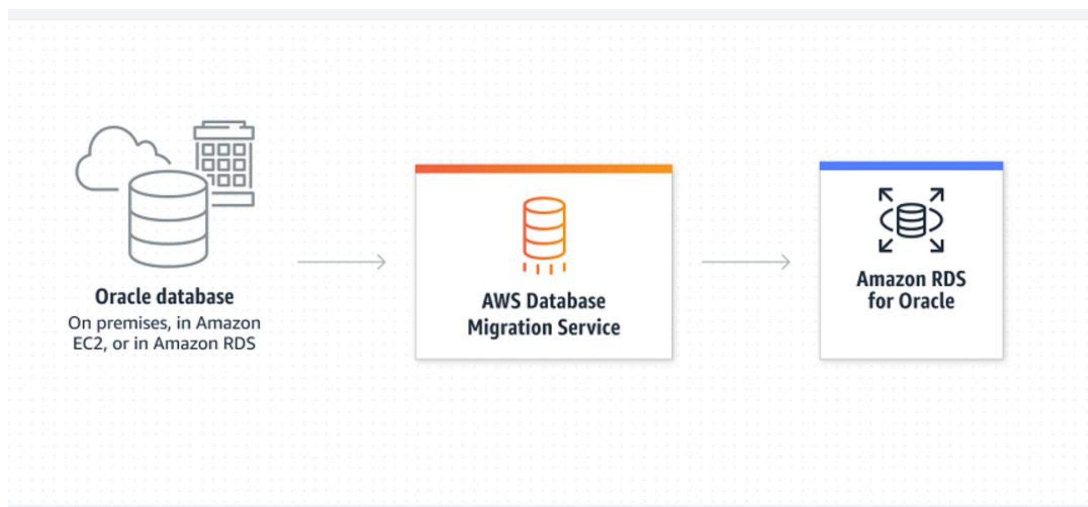


Рисунок 1.1. AWS Database Migration Service

AWS DMS (Database Migration Service) - це повністю керована служба для міграції баз даних, що забезпечує швидке переміщення баз даних та аналітичних навантажень до хмарних рішень Amazon. Вона пропонує інструменти реплікації та міграції, підтримуючи як гомогенні (міграція між однаковими системами, наприклад, з MySQL до MySQL), так і гетерогенні (міграція між різними системами, наприклад, з Oracle до PostgreSQL) сценарії. Ця гнучкість робить AWS DMS корисним інструментом для широкого спектра завдань.

Однак у AWS DMS є певні недоліки. Зокрема, вона орієнтована переважно на міграцію до продуктів Amazon, таких як Amazon RDS чи Amazon Aurora. Це обмежує можливості використання сервісу в тих випадках, коли цільовою платформою є не Amazon.

Крім того, AWS DMS не надає функцій для попередньої оцінки складності міграції. Сервіс зосереджується на автоматизованій передачі даних, що може бути ризикованим. Унікальні функції деяких баз даних, наприклад, збережені процедури, тригери чи специфічні типи даних, не завжди можуть бути коректно перенесені. Це може призвести до втрати функціональності або вимагати додаткового ручного втручання для адаптації, що збільшує вартість і час міграції.

Таким чином, AWS DMS підходить для задач, де міграція є прямолінійною, але для складних чи нестандартних сценаріїв можуть бути потрібні додаткові інструменти [1].

Azure Migrate

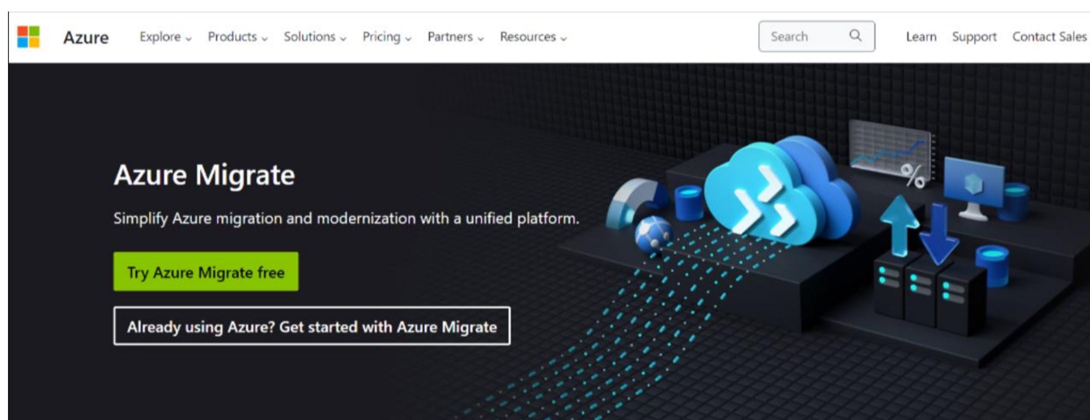


Рисунок 1.2. Azure Migrate

Azure Migrate - це комплексна служба від Microsoft, яка допомагає організаціям оцінювати та мігрувати свої локальні ресурси, додатки, дані та робочі навантаження у хмару Azure. Завдяки інтегрованим інструментам оцінки, Azure Migrate дозволяє аналізувати готовність до міграції, виявляти можливі проблеми, пропонувати варіанти оптимізації та планувати переміщення даних і робочих процесів. Це робить процес переходу більш структурованим та передбачуваним.

Azure Migrate підтримує широкий спектр сценаріїв, включаючи міграцію віртуальних машин (VMware, Hyper-V та фізичних серверів), баз даних, контейнерів, програм і навіть комплексних багаторівневих додатків. Завдяки можливості інтеграції з іншими службами Azure, такими як Azure Site Recovery та Azure Database Migration Service, процес перенесення стає ще більш зручним та безпечним.

Попри свої переваги, Azure Migrate має і певні обмеження. Наприклад, ця служба орієнтована на автоматизовану міграцію, але не пропонує засобів для детальної оцінки складності міграції специфічних баз даних. Це може бути важливо для баз даних із високим рівнем кастомізації чи залежностями, які потребують глибшого аналізу.

Крім того, Azure Migrate підтримує перенесення даних лише у середовище Azure, наприклад до Azure SQL Database чи Azure Virtual Machines. Це створює обмеження для організацій, які планують використовувати інші хмарні платформи або потребують мультихмарних рішень [2].

Fivetran

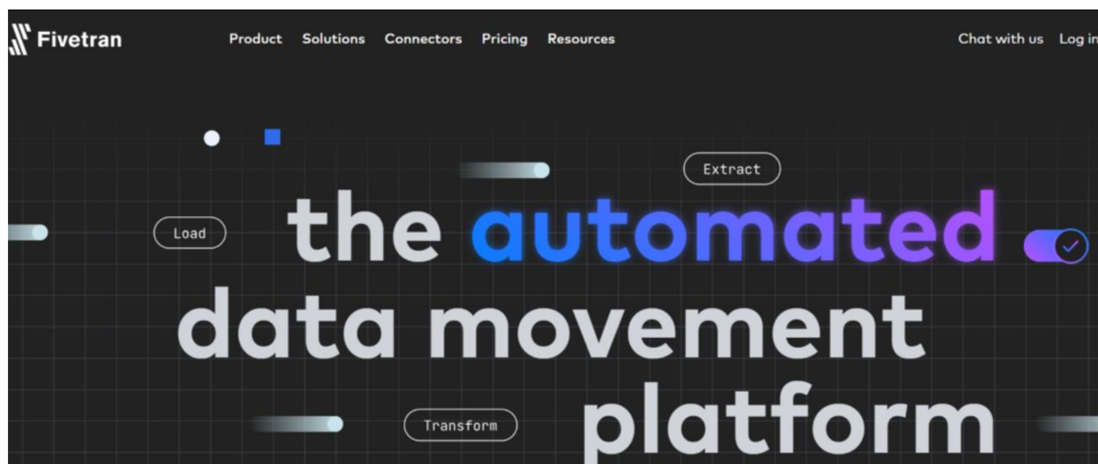


Рисунок 1.3. Fivetran

Fivetran - це автоматизоване рішення ETL (Extract, Transform, Load) [5], яке широко використовується для міграції та інтеграції даних. Завдяки великій кількості конекторів, Fivetran дозволяє безперешкодно переносити дані з різноманітних джерел, таких як бази даних, CRM-системи, аналітичні інструменти, до централізованого сховища даних, наприклад, хмарного data warehouse. Цей процес відбувається у напівавтоматичному режимі, мінімізуючи

необхідність ручного втручання та знижуючи ризики помилок під час передачі даних.

Однією з ключових переваг Fivetran є його фокус на забезпеченні конфіденційності даних. Інструмент використовує найсучасніші засоби безпеки, включаючи шифрування, контроль доступу та моніторинг, щоб гарантувати захист чутливої інформації під час її передачі та зберігання.

Попри ці переваги, Fivetran має і певні обмеження. По-перше, це інструмент, орієнтований на стандартні ETL-пайплайни. Він не має функцій для оцінки складності або адаптації бази даних перед міграцією, що є важливим для проєктів із високим рівнем кастомізації. По-друге, Fivetran вимагає певного рівня технічної експертизи для налаштування та інтеграції, особливо якщо йдеться про нестандартні сценарії використання.

Таким чином, Fivetran є потужним інструментом для спрощення ETL-процесів, але для виконання складних міграційних завдань може знадобитися використання додаткових інструментів [3].

Таблиця 1.1. Порівняня наявних рішень

Інструмент	Аналізує	Оцінює	Мігрує	Широке використання
AWS Migration Service	+	-	+	-
Azure Migrate	+	-	+	-
Fivetran	-	-	-	+

У цій таблиці (табл. 1.1) перераховані переваги та недостатки кожного з інструментів наведених вище [10].

1.4 Актуальність розробки

Розробка програмного забезпечення для автоматизації оцінки складності міграції реляційних баз даних, є технічно складним, але необхідним, тому що зараз таких інструментів практично не існує на широкий загал, а такий інструмент дозволяє значно оптимізувати процеси переходу на нові платформи, що стають критично важливими у зв'язку з постійним розвитком інформаційних технологій та швидкими темпами трансформації бізнесу та відсутністю альтернатив на ринку [11].

Основною метою створення такого програмного забезпечення є зниження витрат на міграцію за рахунок автоматизації ручних процесів, що раніше вимагали значних часових і людських ресурсів. Окрім цього, автоматизація сприяє підвищенню ефективності, адже інструменти аналізу здатні швидко й точно обробляти великі обсяги даних, оцінювати залежності між об'єктами бази, виявляти можливі проблеми та пропонувати шляхи їх розв'язання.

Сучасні компанії активно накопичують величезні масиви даних - від кількох терабайтів до цілих петабайтів. Такі обсяги даних потребують не лише швидкого зберігання, але й швидкого доступу, аналізу та можливості міграції у випадку переходу на інші платформи чи хмарні середовища.

Дані сьогодні справедливо називають «новим золотом», адже вони є основою прийняття бізнес-рішень, розробки інноваційних продуктів і послуг, а також конкурентною перевагою компаній. Тому будь-які інструменти, які допомагають керувати цим «золотом» більш ефективно, отримують високу затребуваність [12].

Розробка програмного забезпечення для автоматизації оцінки складності міграції баз даних не лише відповідає потребам часу, але й відкриває нові можливості для бізнесу, сприяючи його гнучкості, конкурентоспроможності та здатності швидко адаптуватися до змін у світі.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Реляційні бази даних

Реляційна база даних - це тип бази даних, яка зберігає та забезпечує доступ до взаємопов'язаних даних. Реляційні бази даних базуються на реляційній моделі інтуїтивному, простому способі представлення даних у вигляді таблиць. У реляційній базі даних кожний рядок таблиці є записом із унікальним ідентифікатором, який називається ключем. Стовпці таблиці зберігають атрибути даних, і кожен запис зазвичай має значення для кожного атрибута, що спрощує встановлення взаємозв'язків між даними [4].

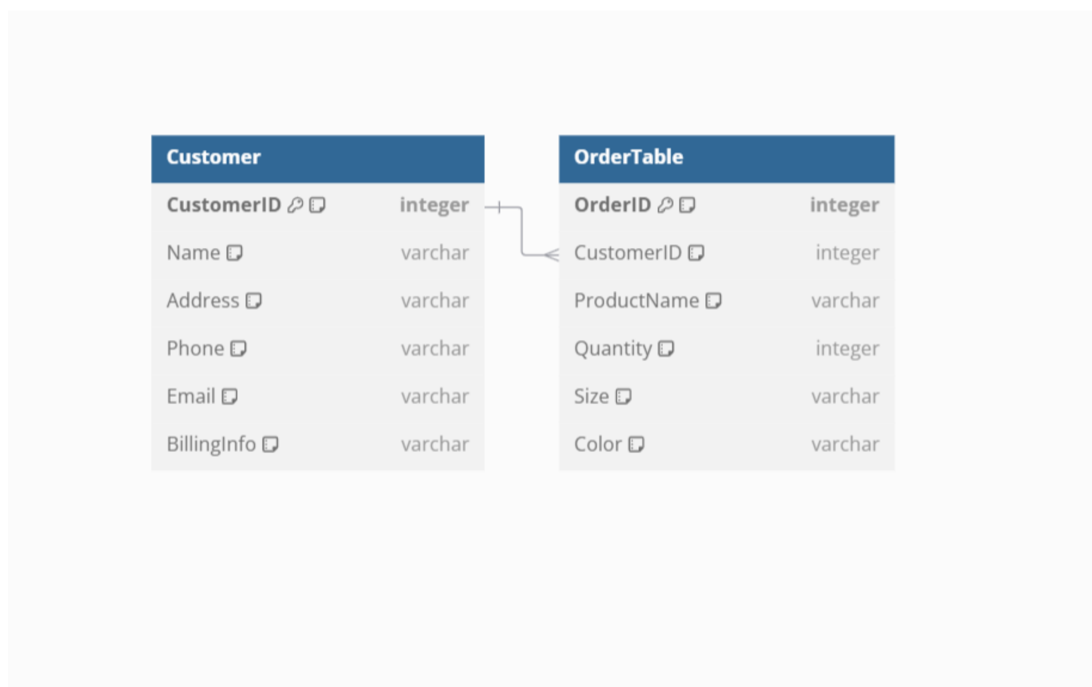


Рисунок 2.1. Реляційна база даних

Ось простий приклад кількох таблиць (рис. 2.1), які може використовувати малий бізнес для обробки замовлень на свої продукти. Перша таблиця - це таблиця інформації про клієнтів, де кожен запис містить ім'я клієнта, адресу,

інформацію для доставки та виставлення рахунків, номер телефону та інші контактні дані. Кожен атрибут (кожен стовпець) має окрему колонку, і база даних призначає унікальний ідентифікатор (ключ) кожному рядку. Друга таблиця - це таблиця замовлень клієнтів, де кожен запис містить ID клієнта, який зробив замовлення, назву продукту, кількість, вибраний розмір і колір тощо - але не ім’я чи контактну інформацію клієнта.

Ці дві таблиці мають лише одну спільну характеристику: стовпець ID (ключ). Завдяки цьому спільному стовпцю реляційна база даних може створити зв’язок між таблицями. Коли програма для обробки замовлень подає замовлення в базу даних, вона може отримати потрібну інформацію про продукт із таблиці замовлень клієнтів і за допомогою ID знайти дані про доставку та виставлення рахунків у таблиці клієнтів. Таким чином, склад отримує правильний продукт, клієнт своєчасно отримує замовлення, а компанія отримує оплату.

Реляційна модель передбачає, що логічні структури даних - таблиці, подання та індекси - відокремлені від фізичних структур зберігання [13]. Це дозволяє адміністраторам баз даних управляти фізичним зберіганням даних, не впливаючи на доступ до них у логічній структурі. Наприклад, перейменування файлу бази даних не змінює назви таблиць, які в ньому зберігаються.

Це розділення також застосовується до операцій бази даних, які визначають дії для маніпулювання даними. Логічні операції дозволяють програмам вказувати потрібний зміст, а фізичні операції визначають, як отримати ці дані.

Для забезпечення точності даних реляційні бази дотримуються певних правил цілісності. Наприклад, правило цілісності може забороняти дублікати рядків у таблиці, щоб уникнути помилкової інформації.

У ранні роки існування баз даних кожна програма зберігала дані у своїй унікальній структурі. Реляційна модель вирішила проблему численних довільних структур даних, забезпечивши стандартний спосіб представлення і запиту даних.

Таблиці стали основною перевагою моделі, оскільки вони забезпечують інтуїтивне, ефективне та гнучке зберігання й доступ до структурованої

інформації. Згодом ще однією перевагою моделі стало використання мови SQL для написання запитів. SQL базується на реляційній алгебрі та забезпечує послідовну математичну основу для оптимізації запитів.

Проста, але потужна реляційна модель підходить для організацій будь-якого масштабу, які потребують надійного управління даними. Реляційні бази використовуються для відстеження запасів, обробки транзакцій, управління інформацією про клієнтів тощо. Вони залишаються найпоширенішою моделлю баз даних з 1970-х років.

Реляційні бази даних найкраще забезпечують послідовність даних між копіями бази (екземплярами). Наприклад, коли клієнт вносить гроші через банкомат і перевіряє баланс через мобільний застосунок, реляційна база гарантує, що обидві дії відображаються миттєво.

Реляційні бази даних дозволяють використовувати збережені процедури - блоки коду, які можна викликати із застосунків. Це спрощує повторювані операції й забезпечує послідовність [14].

Конкурентний доступ до даних вирішується за допомогою блокувань і політик управління. Наприклад, реляційні бази можуть блокувати окремі записи, а не всю таблицю, що зменшує вплив на продуктивність.

2.2 Docker

Docker **-** це відкрита платформа для розробки, доставки та запуску програм. Docker дозволяє відокремити програму від інфраструктури, що дає змогу швидко доставляти її. Використовуючи Docker для доставки, тестування та розгортання коду, можна значно зменшити затримку між написанням коду та його встановленням у клієнта.

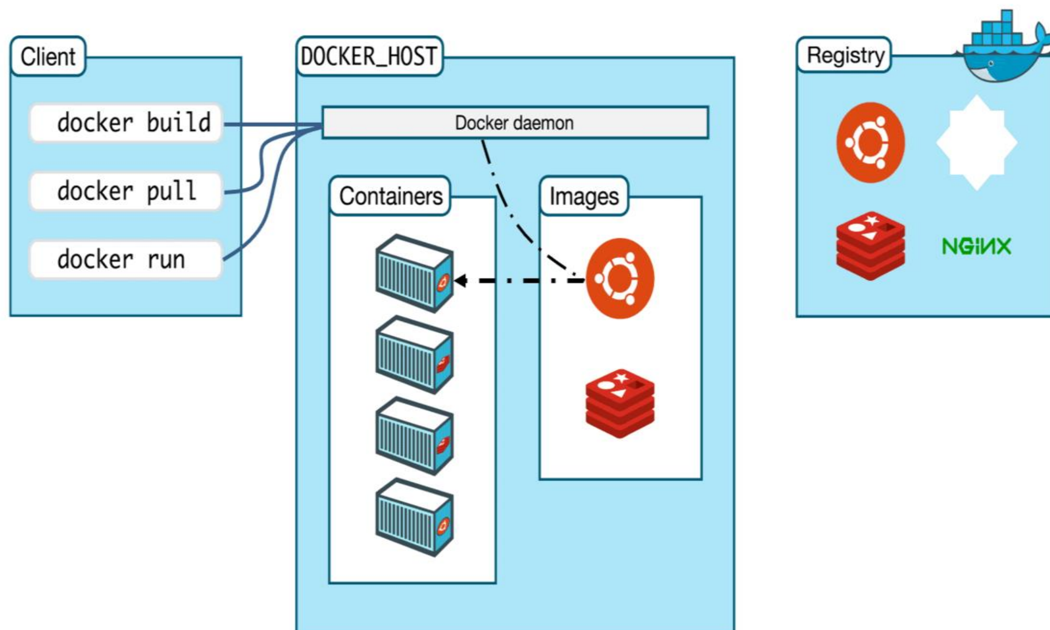


Рисунок 2.2. Архітектура Docker

Docker дозволяє ізолювати програми одна від одної, оскільки кожна програма працює у своєму контейнері, що зменшує ризики конфліктів. Контейнери є портативними, тобто їх можна запускати на будь-якій операційній системі. Docker також дає змогу зайти всередину контейнера і працювати з ним через термінал, як з Linux Bash [15].

Крім того, Docker підтримує масштабування, дозволяючи запускати кілька контейнерів одночасно, що корисно для паралельної оцінки різних баз даних. Усі залежності програми встановлюються під час створення контейнера, що дозволяє налаштувати середовище лише один раз, спрощуючи подальшу роботу.

Docker - це ідеальне рішення для стабільності, масштабованості та простоти роботи розробленої програми. Тому що він пропонує повну ізоляцію, швидко працює, а також його можна встановити на любу операційну систему.

2.3 Java

Java - це одна з найвідоміших мов програмування, відома своєю універсальністю, продуктивністю та надійністю. Для розробки програми вона надає всі необхідні інструмети для підтримки різних баз даних [16].

В першу чергу, Java Database Connectivity (JDBC) є стандартом для роботи з реляційними базами даних. Це ключова перевага, оскільки JDBC дозволяє програмі виконувати наступні завдання [17]:

1. Підключатися до будь-якої реляційної бази даних за допомогою драйверів.
2. Виконувати SQL запити для оцінки структур бази.
3. Отримувати і обробляти метадані бази даних, такі як списки таблиць, зв'язки між об'єктами, індекси, ключі тощо.
4. Адаптувати алгоритми аналізу під специфіку кожної бази, враховуючи її функціональні можливості.

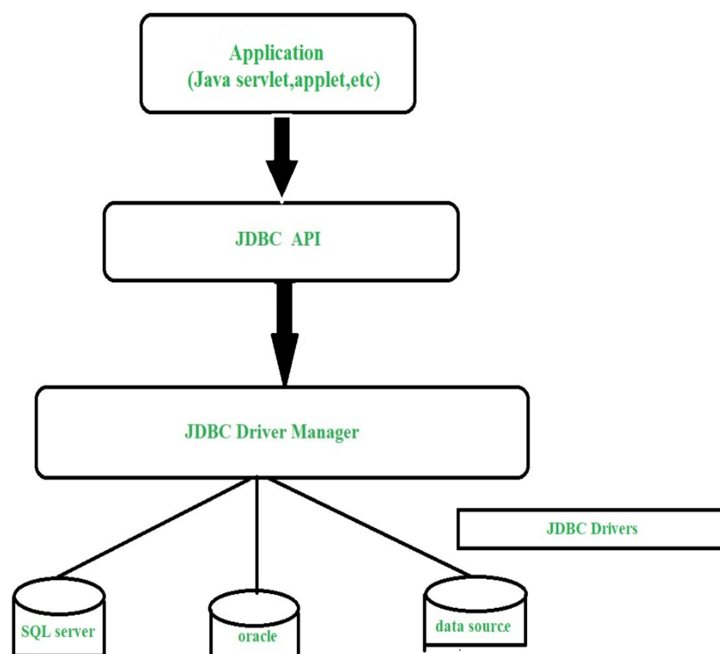


Рисунок 2.3. Архітектура JDBC

По-друге, Java підходить для створення систем, які можуть обробляти великі обсяги даних і виконувати швидкі обчислення. Завдяки багатопоточності та інструментам оптимізації Java дозволяє реалізувати: паралельну обробку даних для прискорення аналізу великих баз та гнучке масштабування програми для підтримки інших баз даних [18].

Також, Java пропонує велику кількість бібліотек для роботи з базами даних. Наприклад, Hibernate ORM [19] якщо потрібно працювати з моделлю об'єктів, або Apache POI і OpenCSV [20] для інтеграції з іншими форматами даних, такими як Excel або CSV, які можуть використовуватися для експорту результатів оцінки. Java має вбудовані механізми безпеки, які дозволяють захищати дані та запити. Наприклад, як підтримка SSL/TLS для безпечного підключення до баз даних [21].

Розробка програми для оцінювання складності міграції реляційних баз даних на Java дає значні переваги, зокрема швидку інтеграцію з різними базами даних завдяки використанню JDBC [6], що дозволяє підключатися до баз незалежно від їх типу. Програма також забезпечує ефективну обробку великих баз, оскільки підтримує паралельні обчислення, що дозволяє одночасно аналізувати таблиці, запити та індекси за допомогою багатопотокової обробки. Крім того, автоматизація запитів дає змогу зчитувати, обробляти та оцінювати базу даних без втручання людини.

Java є ідеальним вибором для цієї програми завдяки своїм можливостям роботи з базами даних через JDBC [6], а також високій продуктивності, гнучкості та безпеці, які необхідні для реалізації ефективного рішення.

2.4 Spring Boot

Spring Boot виділяється серед інших фреймворків, оскільки він надає розробникам програмного забезпечення гнучке налаштування, надійну пакетну обробку, ефективний робочий процес та велику кількість інструментів, допомагаючи швидко розробляти надійні та масштабовані програми на базі Spring [22].

Підримка JDBC: Spring Boot JDBC використовується для підключення Spring Boot програми до бази даних і має контроль над SQL запитами, що виконуються. Spring Boot JDBC спрощує роботу з Spring JDBC, автоматизуючи процеси та кроки за допомогою концепції автоматичної конфігурації. Такі

об'єкти, як JdbcTemplate, DataSource, NamedParameterJdbcTemplate та інші необхідні об'єкти, автоматично конфігуруються і створюються.

Spring Profiles: Spring профайли надають інструменти для розподілу частин конфігурації програми та дозволять зробити компоненти доступними лише в певних профайлах. Будь-який @Component або @Configuration можна позначити за допомогою анотації @Profile, щоб обмежити середовище, де цей компонент буде завантажений. Також профайли дозволять загружати конфігурацію в залежності від профайлу, що дає змогу розділити конфіграції для різних баз даних, що скануються. Наприклад, ми можемо створити одну конфігурацію з набором запитів для бази даних SQL Server, а іншу для Oracle, і міняючи лише профайл, ми поміняємо поведінку програми [23].

Spring Boot Executable JAR: також відомий як Uber JAR або JAR з залежностями - це JAR файл, який містить не тільки Java програму, але й вбудовує її залежності, що дозволяє виконати програму «standalone» без скачування її залежностей [24].

Spring Boot є хорошим вибором для розробки цієї програми, тому-що він дозволяє пришвидшити розробку програми а також має динамічні конфігурації, має підтримку JDBC для зручної роботи з базами даних та можливості створення Executable JAR, що включає всі залежності. Завдяки профілям можна легко налаштовувати різні конфігурації для різних середовищ, що оцінюються.

2.5 PostgreSQL

PostgreSQL - це потужна, відкрита об'єктно-реляційна система управління базами даних, яка використовує та розширює мову SQL, має багато функцій, що безпечно зберігають і масштабують найскладніші запити до даних [25].

PostgreSQL є відкритим програмним забезпеченням з ліцензією free to use, що дозволяє розгортати базу даних у контейнері на стороні клієнта. Вона також підтримує роботу з JSON [26], що дає змогу денормалізувати дані та зберігати їх у цьому форматі. Ця база даних забезпечує високу швидкодію, що дозволяє

створювати швидкі аналітичні представлення, використовуючи їх для загальної оцінки бази даних. До того ж, PostgreSQL має власний JDBC драйвер і нативну підтримку в Spring Boot, що дозволяє централізовано зберігати метадані з просканованих баз для подальшого аналізу та оцінки.

PostgreSQL буде найкращим сховищем метаданих для програми, тому-що він швидко працює з JSON, має JDBC драйвер, а також дозволяє створювати аналітичні представлення, що є основним результатом цієї програми, і головне що він повністю бесплатний, що дозволяє безперешкодно використовувати цю базу даних на середовищах клієнтів [27].

Структура внутрішньої бази даних повинна відповідати потребам програми і зберігати всі необхідні для подальшого аналізу проскановані дані. Це повині бути: бази даних, схеми, таблиці, колонки, індекси, тригери, рутини та представлення [9]. (рис. 2.4).

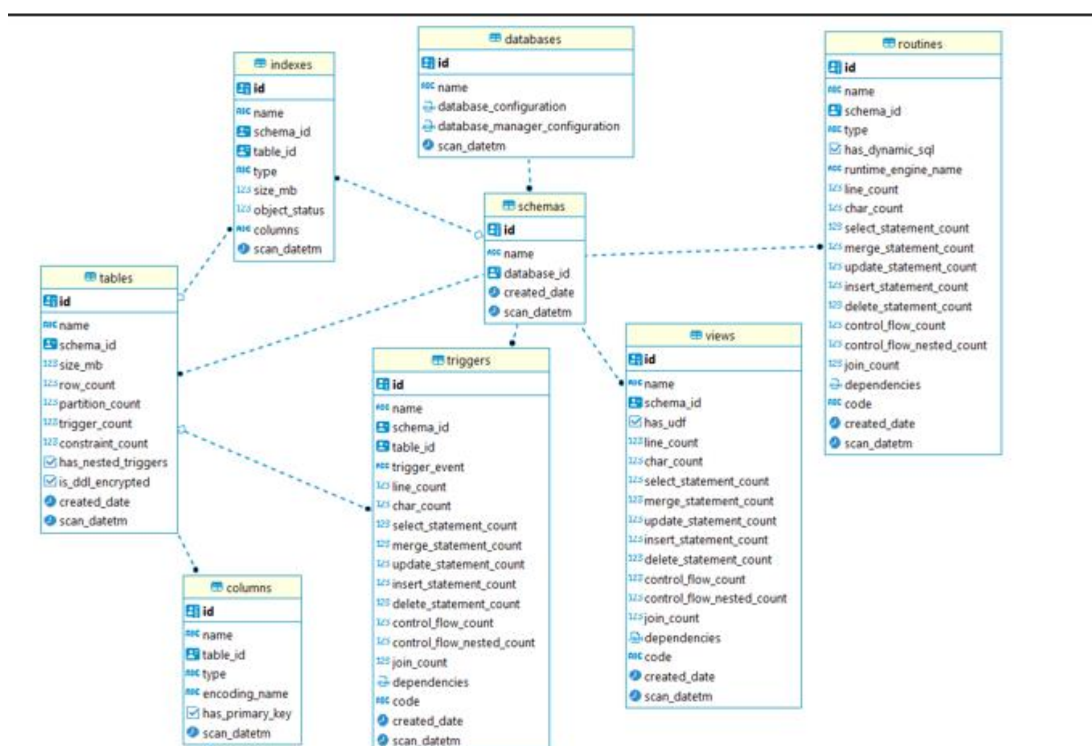


Рисунок 2.4. UML діаграма

На цій даіграмі видно, що головним компонентом у структурі є база даних (таблиця databases). Що є батьківською таблицею для таблиці схем (таблиця

schemas). Від таблиця schemas в свою чергу залежать наступні об’єкти в базі даних:

1. Проскановані таблиці (таблиця tables) яка містить у собі колонки (таблиця columns).
2. Проскановані індекси (таблиця indexes), яка крім зв’язку з таблицею у якій зберігаються схеми, також має зв’язок з таблицею у якій зберігаються таблиці. Це тому що, індекси можуть бути створені як на рівні схеми, так і на рівні таблиці.
3. Проскановані тригери (таблиця triggers), яка також крім зв’язку з таблицею схем, має зв’язок з таблицею де зберігаються таблиці. І це також зв’язано з тим, що тригери можуть бути створені на рівні схеми, без прив’язки до таблиці.
4. Проскановані представлення (таблиця views), яка залежить від таблиці де зберігаються схеми, тобто від схем. І крім основних полів, також містить в собі корисні атрибути типу кількість джоінів (join_count), для подальшого визначення складності цього преставлення і бази даних в цілому.
5. Проскановані рутини (таблиця routines), яка також залежить від таблиці де зберігаються схеми. Рутини у SQL – це процедури та функції, або іншимим словами, код який можна виконувати на базі даних. Також крім основних полів і корисних атрибутів, вона містить у собі об’єкти бази даних від яких вона залежить, що збільшує складність її міграції, тому-що крім неї також потрібно мігрувати і її залежності [9].

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Встановлення Docker

Для того щоб встановити Docker на особистий комп'ютер для розробки, в залежності від вашої операційної системи Windows чи Linux вам знадобиться виконати наступні кроки:

Крок 1: Встановлення на Linux. Для того щоб встановити Docker на операційну систему Linux виконайте наступну команду:

```
sudo apt-get install docker-ce docker-ce-cli containerd.io
docker-buildx-plugin docker-compose-plugin
```

Крок 2: Для того щоб встановити Docker на Windows ви можете використовувати Docker Desktop (для персонального використання), або встановити його на WSL (Windows Subsystem for Linux). Виконавши кроки з цієї статті: https://docs.docker.com/desktop/setup/install/windows-install.

Крок 3: Для того щоб перевірити чи Docker встановився успішно можна виконати наступну команду у терміналі, вона запустить тестовий контейнер:

```
docker run hello-world
```

3.2 Ініціалізація проєкту

Для локальної розробки і тестування, вам буде необхідно встановити Java на комп'ютер. Для розробки цієї програми я буду використовувати безплатну версію OpenJDK 17 від Amazon, яку можна встановити з офіційного сайту https://docs.aws.amazon.com/corretto/latest/corretto-17-ug/downloads-list.html:

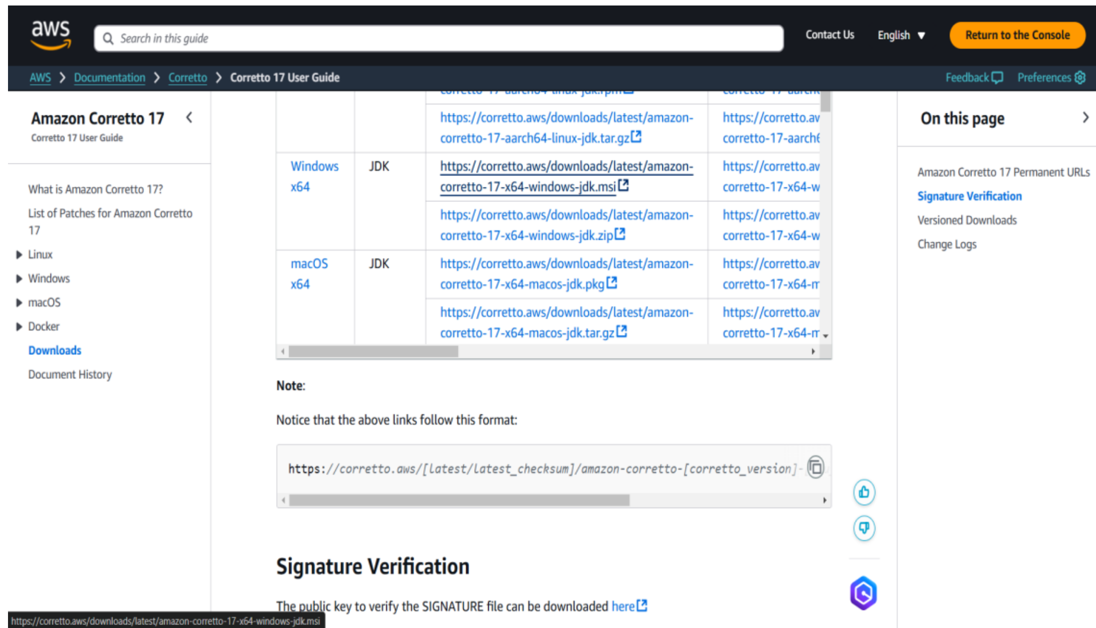


Рисунок 3.1. Завантаження Amazon JDK

Щоб ініціалізувати проєкт, перейдемо на офіційний сайт ініціалізації проєктів Spring https://start.spring.io, який створить проєкт з потрібною структурою та залежностями автоматично:

1. У опції **Language** потрібно вибрати **Java**.
2. У опції **Project** вибираємо **Maven**.

Maven - це інструмент автоматизації збірки, який використовується переважно для проєктів на Java. Maven також можна використовувати для створення та управління проєктами, написаними на C#, Ruby, Scala та інших мовах. Проєкт Maven підтримується Apache Software Foundation, раніше він був частиною проєкту Jakarta.

3. Версію **Spring Boot** вибираємо **3.4.0**, яка є стабільною версією на даний час.
4. У поле **Group** вписуємо: **com.database**.
5. У поле **Artifact** і **Name** вписуємо: **migration**.
6. Опцію **Packaging** вибираємо **Jar**.
7. Опцію **Java** вибираємо **17**, тому-що ми встановили OpenJDK 17 версії.
8. Тепер переходимо до таблиці справа і додаємо залежності:

a. JDBC API – для роботи з базами даних через JDBC.
b. PostgreSQL Driver – для підключення до внутрішньої бази даних, де зберігаються метадані.
c. Spring Data JPA – це Hibernate ORM, що дозволить відтворити структуру бази даних, за допомогою Java об’єктів.
d. MS SQL Server Driver – для підключення до MS SQL баз даних, та їх сканування через JDBC.
e. Oracle Driver – для підключення до Oracle баз даних, та їх сканування через JDBC.
f. Flyway Migration – для пітримки і версіонування внутрішньої PostgreSQL бази даних, та її автоматичного створення чи модифікації.

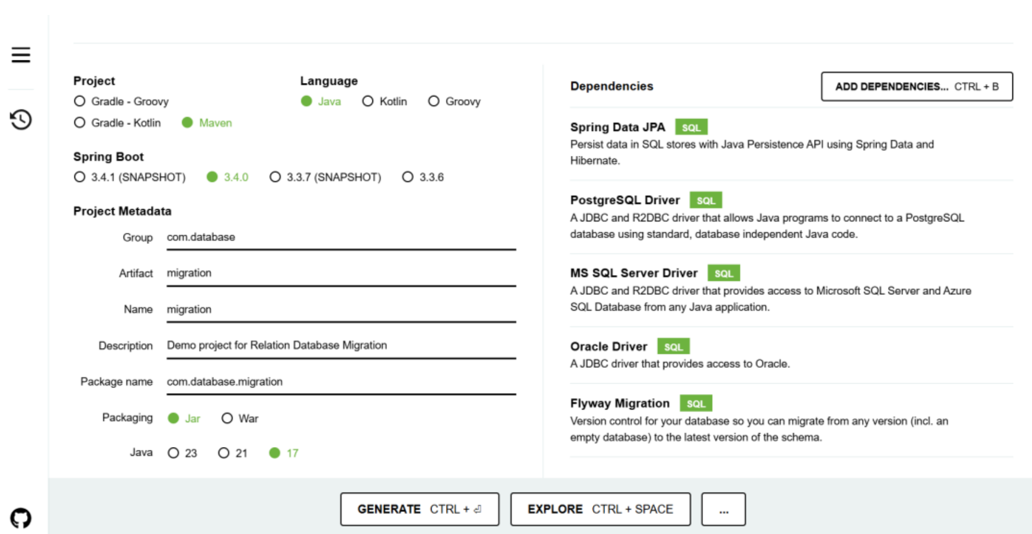


Рисунок 3.2. Ініціалізація проєкту

Тепер потрібно нажати кнопку **Generate**, і скачати архів, після чого розпакувати його. Після розпаковки, у цій папці буде знаходитись стандартний Spring Boot проєкту із всіма залежностями що ми додали.

3.4 Підготовка до розробки

Для розробки програми знадобиться середовище розробки, або **IDE**. Для розробки цього проєкту використовується IDE **IntelliJ IDEA** від компанії JetBrains.

IDE - це комплексне програмне рішення для розробки програмного забезпечення. Зазвичай, складається з редактора початкового коду, інструментів для автоматизації складання та відлагодження програм. Більшість сучасних середовищ розробки мають можливість автодоповнення коду [28].

IntelliJ IDEA має безліч переваг порівняно з іншими середовища розробки, такими як Eclipse чи NetBeans. Ось деякі з них:

1. Розумна допомога з кодом: IntelliJ IDEA пропонує неперевершену допомогу в написанні коду, включаючи інтелектуальне автодоповнення, аналіз і рекомендації. Вона розуміє контекст вашого коду та пропонує рекомендації, що відповідають ситуації, допомагаючи писати код швидше та з меншою кількістю помилок.
2. Розширене рефакторингування: IDE надає широкий спектр автоматизованих інструментів для рефакторингу, що полегшують покращення якості, читабельності та підтримуваності коду. Ви можете безпечно перейменовувати змінні, виділяти методи та багато іншого.
3. Перевірка коду: Завдяки вбудованому аналізу коду IntelliJ IDEA допомагає виявляти потенційні проблеми та помилки в реальному часі. Вона надає швидкі виправлення та рекомендації для підтримки якості коду, зменшуючи час на налагодження.
4. Інтеграція з системами контролю версій: Безшовна інтеграція з популярними системами контролю версій, такими як Git, SVN і Mercurial, спрощує співпрацю та відстеження версій, роблячи командну розробку ефективнішою.

Після відкриття проєкту у терміналі IDE потрібно виконати наступну Maven команду для того щоб встановити залежності:

```
./mvnw clean package
```

3.5 Розробка проєкту

3.5.1 Створення Docker зображення

Для локальної розробки та інтеграційного тестування в першу чергу знадобиться Докер контейнер. Тому розпочнемо з нього:

Крок 1: У корені проєкту створимо папку **docker**.

Крок 2: У папці docker створимо файл **Dockerfile** (рис. 3.3) і наповнимо його кроками необхідними для ініціалізації середовища. Встановимо **Alpine Linux** як операційну систему, також встановимо **Java** і **PostgreSQL** в середині контейнера.

Alpine Linux - це дистрибутив Linux, розроблений для того, щоб бути компактним, простим і безпечним. Він використовує musl, BusyBox та OpenRC замість більш поширених glibc, GNU Core Utilities та system [29].

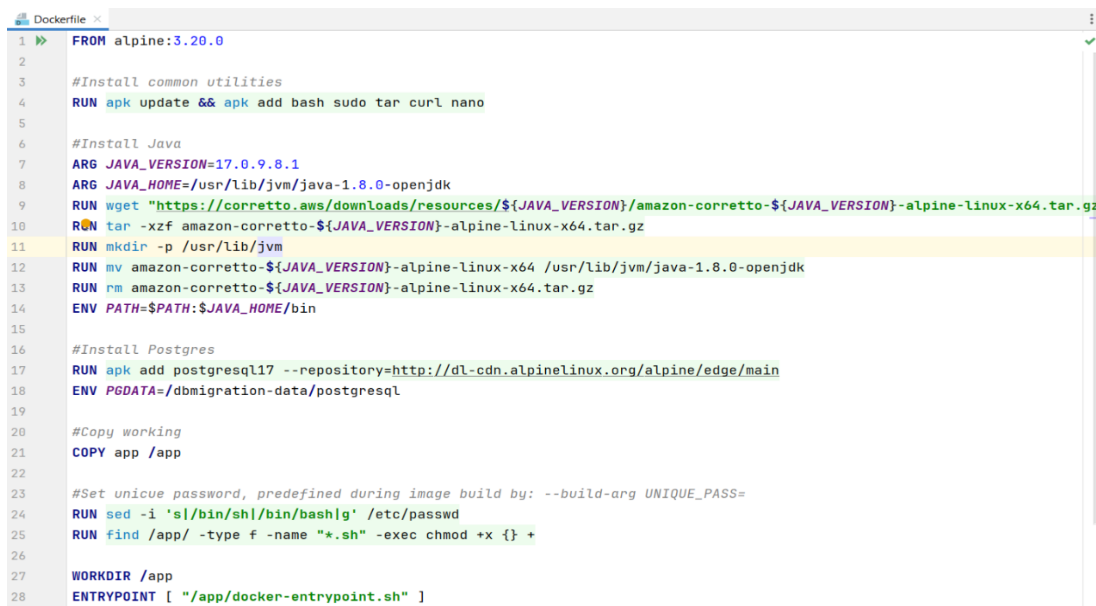


Рисунок 3.3. Файл Dockerfile

У цьому файлі **Dockerfile** (рис. 3.3) або (дод. А) виконуються наступні команди:

1. `FROM alpine` (лінія 1) – встановленя Alpine Linux як ядро контейнера.
2. `RUN apk update && apk add` (лінія 4) – завантаження всіх необхідних для роботи в контейнери утиліт.
3. `ARG JAVA_VERSION` (лінія 7) – додавання змінної середовища для версії Джави.
4. `ARG JAVA_HOME` (лінія 8) – додавання змінної середовища для домашньої папки Джави.
5. `RUN wget, RUN tar, RUN mkdir, RUN mv, RUN rm` (лінії 7 – 13) – набір команд для того щоб скачати Java Amazon Corretto з офіційного сайту, розпакувати її, перемістити та видалити.
6. `RUN apk add, ENV PGDATA` (лінії 17 – 18) – завантаження PostgreSQL і установлення змінної середовища для робочого середовища PostgreSQL.
7. `COPY` (лінії 21) – копіювання папки app з локального середовища всередину контейнера.
8. `RUN sed, RUN find` (лінії 24 - 25) – можливість використувувати bash команди без паролю, а також всі файли з розширенням .sh всередині папки app стають виконуваними.
9. `WORKDIR` (ліня 27) – встановлення папки app як домашньої папки.
10. `ENTRYPOINT` (ліня 28) – виконати цей скрипт при запуску контейнера, у ньому стартує внутрішня база даних.

Крок 3: На рівні із файлом **Dockerfile** створимо папку **app** і у ній файл **docker-entrypoint.sh** (рис. 3.4) для того щоб автоматично створити базу даних і користувача PostgreSQL запуску контейнера.

Папка **app** – це інтерфейс програми. Вся взаємодія з програмою буде відбуватись саме через цю папку за допомогою **Bash (.sh)** команд, які будуть запускати Java програми.

Bash (Bourne Again Shell) - це безкоштовна та вдосконалена версія оболонки Bourne, що постачається з операційними системами Linux та GNU. Bash подібний

до оригіналу, але має додаткові функції, такі як редагування командного рядка [29]

```
#!/bin/bash
set -e

mkdir -p "$PGDATA" /run/postgresql
chown -R postgres:postgres "$PGDATA" /run/postgresql
chmod 0750                 "$PGDATA" /run/postgresql

if [ ! -d "$PGDATA" ]; then
    echo -e "\n${GREEN}Initializing PostgreSQL${NC}\n"
    su - postgres -c "initdb -D $PGDATA"
fi

echo -e "\n${GREEN}Starting PostgreSQL server${NC}\n"
exec su - postgres -c "postgres -D $PGDATA"
```

Рисунок 3.4. Файл docker-entrypoint.sh

У файлі **docker-entrypoint.sh** (рис. 3.4) створюються необхідні папки для старту PostgreSQL, після чого, користувачу postgres надаються права на цю папку. Потім перевіряється чи база даних уже ініціалізована, якщо ні – то база ініціалізується за допомогою команди initdb -D, після чого база даних запускається за допомогою команди postgres -D. Обидві команди приймають змінну середовища PGDATA, яка була встановлена у файлі **Dockerfile** (рис. 3.4).

3.5.2 Створення структури бази даних

За допомогою **Flyway**, ми можемо створити версіонування внутрішньої бази даних для програми. Для швидкості і зручності розробки IntelliJ IDEA пропронує бесплатний плагін який ми можемо встановити, який буде створювати Flyway файли у правильному форматі.

Flyway - це інструмент для міграції баз даних з відкритим кодом, який спрощує процес керування та версіювання змін у схемі бази даних.

Крок 1. Для початку створимо схему assessment у базі даних за допомогою Flyway. Для цього у папці src/main/resources/db.migration за допомогою Flyway плагіну створюємо файл з назвою create_schema.sql (рис. 3.5) та наповнюємо цей файл SQL запитом CREATE SCHEMA, який створить необхідну схему.

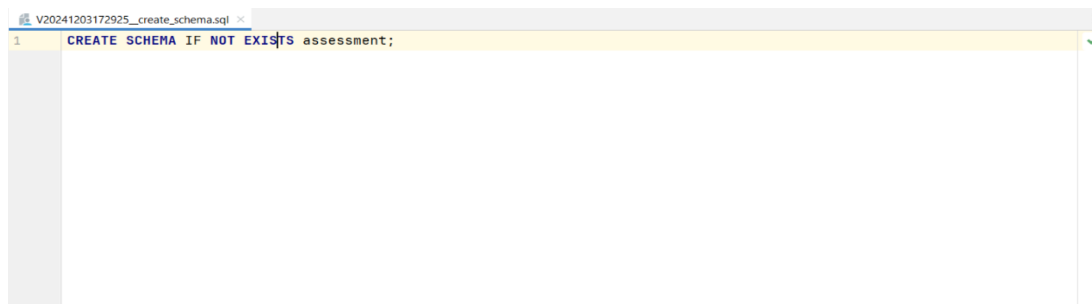


Рисунок 3.5. Файл create_schema.sql

Мова структурованих запитів (SQL) - це мова програмування для зберігання та обробки інформації в реляційній базі даних. Реляційна база даних зберігає інформацію у табличній формі, де рядки й стовпці представляють різні атрибути даних та взаємозв'язки між значеннями даних [31].

Крок 2. Створення таблиці, де будуть зберігатись проскановані бази даних під назвою databases. У цій ж папці де є файл зі створенням схеми, створюємо файл з назвою database.sql (рис. 3.6) та наповнюємо цей файл SQL запитом CREATE TABLE, який створить необхідну таблицю.

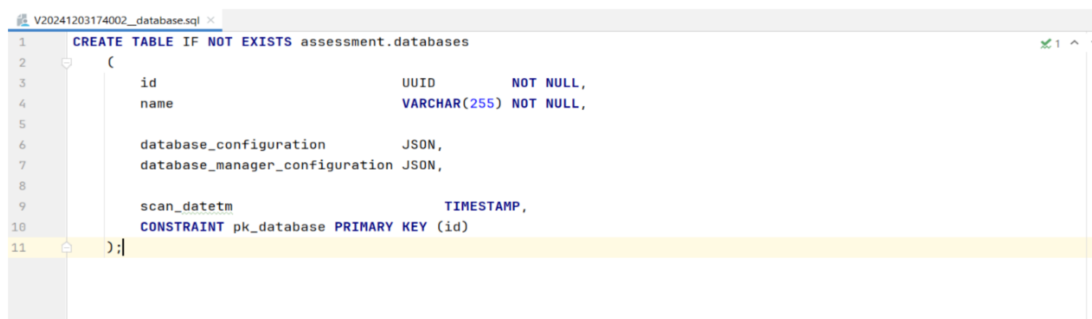


Рисунок 3.6. Файл database.sql

Також, тут доданий PRIMARY KEY (первинний ключ) на поле id, для того щоб була змога зв'язати всі об'єкти які належать до цієї бази даних по цьому ключу.

SQL бази даних, також відомі як реляційні бази даних - це системи, які зберігають колекції таблиць та організовують структуровані набори даних у табличному форматі з колонками та рядками, подібному до формату електронної таблиці.

Первинний ключ у SQL - це одне або група полів чи стовпців, які можуть однозначно ідентифікувати рядок у таблиці. Простими словами, це стовпець, який приймає унікальні значення для кожного рядка [37].

Крок 3. Тепер створимо таблицю з назвою schemas, де будуть зберігатись проскановані схеми. Для цього створимо файл schema.sql (рис. 3.7) та наповнюємо цей файл.

```
V20241203175436__schema.sql
1   CREATE TABLE IF NOT EXISTS assessment.schemas
2       (
3           id                  UUID         NOT NULL,
4           name                VARCHAR(255) NOT NULL,
5
6           database_id                 UUID         NOT NULL,
7
8           created_date                TIMESTAMP,
9           scan_datetm                 TIMESTAMP,
10          CONSTRAINT pk_schema PRIMARY KEY (id),
11          CONSTRAINT fk_schema_database FOREIGN KEY (database_id) REFERENCES assessment.databases (id)
12      );
```

Рисунок 3.7. Файл schema.sql

У цій таблиці, крім первинного ключа на поле id, є ще FOREIGN KEY (зовнішній ключ), який вказує що поле database_id в таблиці schemas, посилається на поле id в таблиці databases. Таким чином, **Hibernate ORM** матиме змогу відтворити ці звязки, при побудові моделі в пам'яті Java.

SQL схема - це корисна концепція бази даних. Вона допомагає створити логічну групу об'єктів, таких як таблиці, збережені процедури та функції.

FOREIGN KEY (зовнішній ключ) - це поле (або набір полів) в одній таблиці, яке посилається на PRIMARY KEY (первинний ключ) в іншій таблиці. Таблиця із зовнішнім ключем називається дочірньою таблицею, а таблиця з первинним ключем називається пов'язаною або батьківською таблицею [31].

Hibernate - це інструмент з відкритим кодом для об'єктно-реляційного відображення (ORM), який надає фреймворк для відображення об'єктно-орієнтованих доменних моделей у реляційні бази даних для веб-застосунків [19].

Крок 4. Створення таблиці tables у якій будуть зберігатись проскановані таблиці. Для цього потрібно створити файл table.sql (рис. 3.8) та наповнити його.

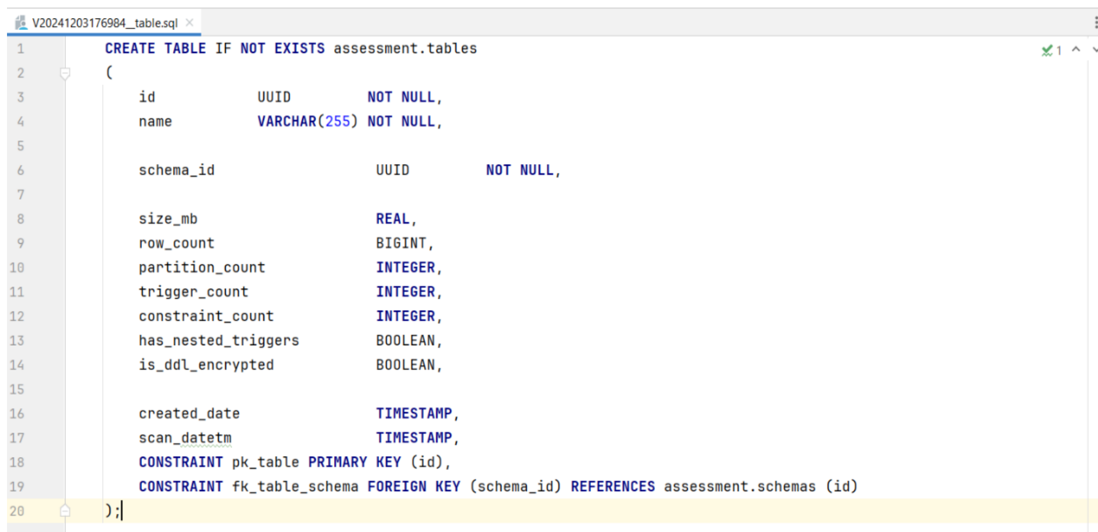


Рисунок 3.8. Файл table.sql

Тут також використовується поле id як первинний ключ (PRIMARY KEY) та поле schema_id як зовнішній ключ (FOREIGN KEY), що посиляється на поле id таблиці schemas.

Таблиці - це об'єкти бази даних, які містять всі дані в базі даних. У таблицях дані логічно організовані в форматі рядків і стовпців, подібному до формату електронної таблиці [9].

Крок 5. Створення таблиці columns у якій будуть зберігатись проскановані колонки таблиць. Для цього потрібно створити файл column.sql (рис. 3.9) та наповнити його.

V20241203190951__column.sql

```sql
CREATE TABLE IF NOT EXISTS assessment.columns
(
    id          UUID         NOT NULL,
    name        VARCHAR(255) NOT NULL,

    table_id            UUID         NOT NULL,

    type                VARCHAR(255),
    encoding_name       VARCHAR(50),
    has_primary_key     BOOLEAN,

    scan_datetm         TIMESTAMP,
    CONSTRAINT pk_column PRIMARY KEY (id),
    CONSTRAINT fk_column_table FOREIGN KEY (table_id) REFERENCES assessment.tables (id)
);
```

Рисунок *3.9*. Файл column.sql

У цьому файлі, видно що колонка за допогою зовнішнього ключа (FOREIGN KEY) на поле table_id посилається на поле id у таблиці tables, це означає що колонка належить до таблиці.

Записи та поля в таблицях SQL містять рядки та стовпці, де рядки називаються записами, а стовпці - полями. Стовпець - це набір значень даних певного типу (наприклад, чисел або літер).

Крок 6. Створення таблиці indexes у якій будуть зберігатись проскановані індекси. Для цього потрібно створити файл index.sql (рис. 3.10) та наповнити його.

V20241203184941__index.sql

```sql
CREATE TABLE IF NOT EXISTS assessment.indexes
(
    id          UUID         NOT NULL,
    name        VARCHAR(255) NOT NULL,

    schema_id           UUID,
    table_id            UUID,

    type                VARCHAR(255),
    size_mb             REAL,
    columns             TEXT,

    scan_datetm         TIMESTAMP,
    CONSTRAINT pk_index PRIMARY KEY (id),
    CONSTRAINT fk_index_schema FOREIGN KEY (schema_id) REFERENCES assessment.schemas (id),
    CONSTRAINT fk_index_table FOREIGN KEY (table_id) REFERENCES assessment.tables (id)
);
```

Рисунок 3.10. Файл index.sql

Тут варто відмітити, що індекс може належати як до таблиці, так і бути створеним на рівні схеми. Таким, чином у нас є два зовнішніх ключі (FOREIGN KEY), поле schema_id, яке посилається на поле id в таблиці schemas, та поле table_id яке посилається на поле id в таблиці tables. Або, бути взагалі без таблиці і схеми. Тому поля schema_id і table_id не мають правила **NOT NULL** і можуть містити **NULL**.

Індекси - це спеціальні таблиці пошуку, які повинні використовуватися пошуковим механізмом бази даних для пришвидшення отримання даних [9].

Поле зі значенням NULL - це поле без значення. Якщо поле в таблиці є необов'язковим, можна вставити новий запис або оновити існуючий запис без додавання значення до цього поля. У такому разі поле буде збережене зі значенням NULL [31].

Крок 5. Створення таблиці triggers у якій будуть зберігатись проскановані тригери. Для цього потрібно створити файл trigger.sql (рис. 3.11) та наповнити його.

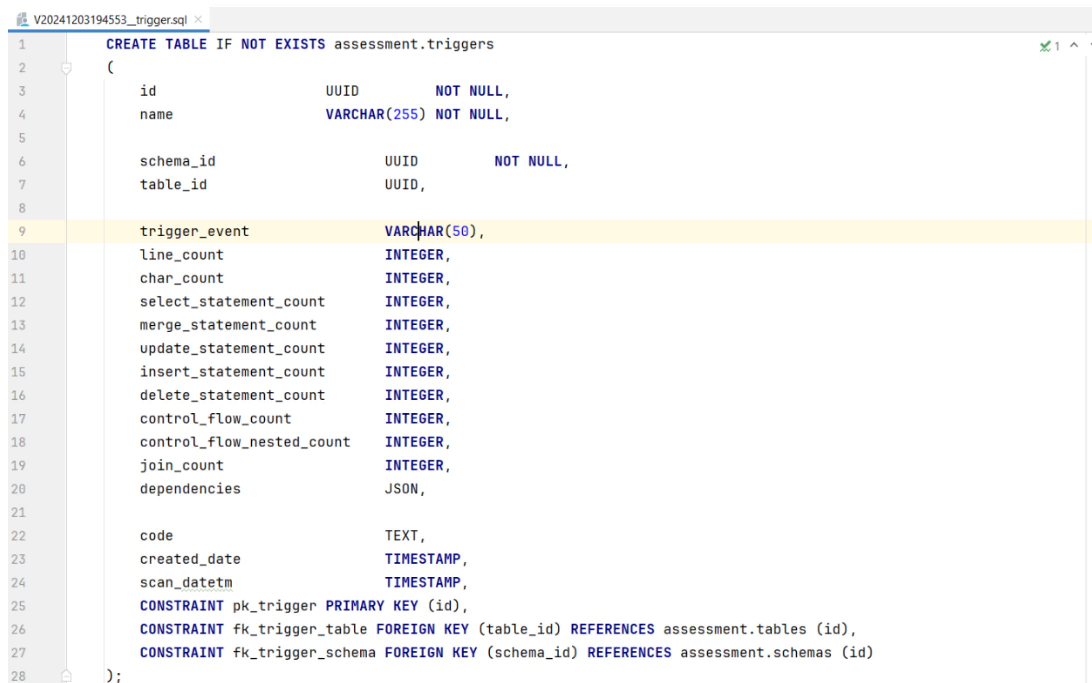


Рисунок 3.11. Файл triggers.sql

Тригери завжди створюються на рівні схеми, тому тут є зовнішній ключ (FOREIGN KEY), поле schema_id яке посилається на поле id в таблиці schemas. Однак тригери, інколи, також можуть належати до таблиць тому також доданий зовнішній ключ на поле table_id, яке може містити **NULL**, який посилається на поле id у таблиці tables.

Крім того, у ця таблиця тримає інформацію про корисні атрибути які ми можемо дістати із SQL тексту, типу кількість **JOIN** у запиті (поле join_count), чи кількість циклів (поле control_flow_count). Саме на цих даних ми будемо оцінювати базу даних. В додаток зберігається і сам SQL код тригера яким він, для потенційної його подальшої обровки.

Також, зберігаються об’єкти від яких залежить цей тригер (поле dependencies), тому-що це також впливає на його складність. Оскільки, ці об’єкти також потрібно буде змігрувати.

Тригери в SQL - це збережені процедури, які автоматично виконуються у відповідь на певні події в конкретній таблиці або поданні бази даних. Вони використовуються для підтримання цілісності даних, впровадження бізнес-правил та автоматизації завдань. Тригери можна налаштувати на виконання до або після операцій INSERT, UPDATE чи DELETE.

JOIN - використовується для об'єднання рядків з двох або більше таблиць, на основі зв'язаного стовпця між ними [31].

Деякі об'єкти бази даних мають залежності від інших об'єктів бази даних. Наприклад, тригери та збережені процедури залежать від наявності таблиць, які містять дані, що повертаються цими уявленнями або процедурами.

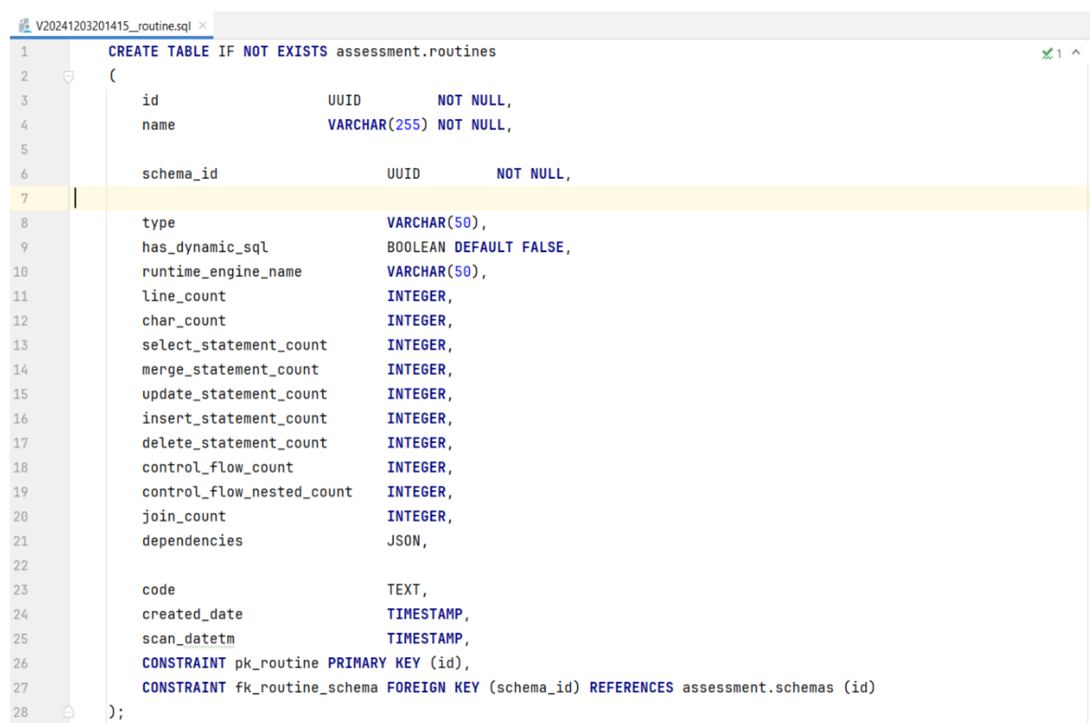
Крок 6. Створення таблиці routines у якій будуть зберігатись проскановanрутини (функції та процедури). Для цього потрібно створити файл routine.sql (рис. 3.12) та наповнити його.

Рутини мають зовнішній ключ (FOREIGN KEY) на поле schema_id, що посилається на поле id таблиці schemas, що означає що вони залежні від схем. Крім того, рутини також зберігають корисні атрибути які можна дістати з їх SQL

коду та оцінити їх складність, як і сам SQL код, яким вони були створені, а також об'єкти від яких ці рутини залежать.

Крім того, вони зберігють мову на якій вони були написані (поле runtime_engine_name) , у більшості випадків це **SQL**, але інколи вони можуть бути написані на інших мовах, наприклад **JavaScript**, що робить їх міграцію надзвичайно складною.

Також вони зберігають інофрмацію про те чи використовується **динамічний SQL** – це також вагомий фактор при міграції, тому-що його практично неможливо відтворити, так як він може генеруватись в залежності від умов.



```
CREATE TABLE IF NOT EXISTS assessment.routines
(
    id                   UUID         NOT NULL,
    name                 VARCHAR(255) NOT NULL,

    schema_id                   UUID         NOT NULL,

    type                        VARCHAR(50),
    has_dynamic_sql             BOOLEAN DEFAULT FALSE,
    runtime_engine_name         VARCHAR(50),
    line_count                  INTEGER,
    char_count                  INTEGER,
    select_statement_count      INTEGER,
    merge_statement_count       INTEGER,
    update_statement_count      INTEGER,
    insert_statement_count      INTEGER,
    delete_statement_count      INTEGER,
    control_flow_count          INTEGER,
    control_flow_nested_count   INTEGER,
    join_count                  INTEGER,
    dependencies                JSON,

    code                        TEXT,
    created_date                TIMESTAMP,
    scan_datetm                 TIMESTAMP,
    CONSTRAINT pk_routine PRIMARY KEY (id),
    CONSTRAINT fk_routine_schema FOREIGN KEY (schema_id) REFERENCES assessment.schemas (id)
);
```

Рисунок 3.12. Файл routine.sql

SQL рутини - є важливими компонентами, які допомагають ефективно обробляти та керувати даними. Ці рутини сприяють модульності та повторному використанню коду, забезпечуючи виконання повторюваних завдань без

дублювання коду. Існують два типи рутин Stored Procedures і Functions разом вони допомагають оптимізувати складні операції, підвищити безпеку, покращити продуктивність та спростити завдання з управління базою даних.

Динамічний SQL - це техніка програмування, яка дозволяє формувати SQL-запити під час виконання програми [32].

Двигун виконання (runtime engine) - це інтерпретатор, який виконує рутинний код у певній мові програмування, такій як JavaScript, або SQL [33].

Крок 7. Створення таблиці views у якій будуть зберігатись проскановані представлення. Для цього потрібно створити файл view.sql (рис. 3.13) або (дод. Б) та наповнити його.

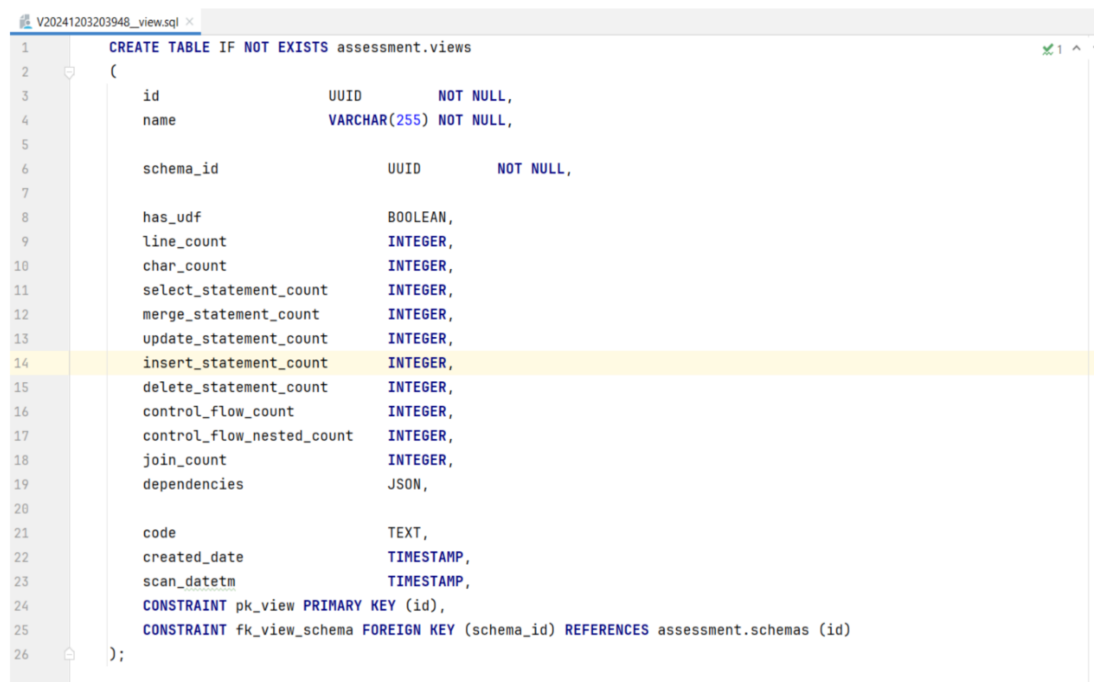


Рисунок 3.13. Файл view.sql

Представлення, як і рутини, мають зовнішній ключ (FOREIGN KEY) на поле schema_id, що посилається на поле id таблиці schemas, тобто це означає що вони залежні від схем. Представлення також зберігають корисні атрибути які можна дістати з їх SQL коду та оцінити їх складність, як і сам SQL код, яким вони були створені, а також об’єкти від яких представлення залежать.

Крім цього, вони зберігають поле has_udf, що дає змогу побачити чи використовують ці представлення **User Defined Function** (функції створені користувачем, у цій програмі попадають у таблицю routines), що кардинально також впливає на складність їх міграції, тому-що або цій функції потрібно шукати заміну, або мігрувати її також.

Представлення (view) - це віртуальна таблиця, вміст якої визначається запитом. Як і таблиця, представлення складається з набору іменованих стовпців та рядків даних. Рядки та стовпці даних отримуються з таблиць, згаданих у запиті, який визначає представлення, і генеруються динамічно, коли представлення викликається [9].

Користувацькі функції (UDF) - це потужні інструменти в SQL, які дозволяють розробникам інкапсулювати часто використовувану логіку в багаторазові модулі. Ці функції можуть спростити складні запити, покращити читабельність коду та сприяти повторному використанню коду [34].

3.5.3 Розробка серверної частини

Крок 1. Для того щоб програма могла взаємодія з внутрішньою базою даних, яка уже є розгорнута всередині Докера, необхідно налаштувати з’єднання. Для цього у папці src/main/resources потрібно створити файл application.yml (рис. 3.14) та наповнити його необхідними змінними для підключення до бази даних.

```yaml
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/postgres?currentSchema=assessment
    username: postgres
    password: postgres
  jpa.properties.hibernate.dialect: org.hibernate.dialect.PostgreSQLDialect
```

Рисунок 3.14. Файл application.yml

1. Змінна **url** – це текст у форматі **JDBC** драйвера, який вказує що використовується базу даних від провайдера postgresql і підключаємося до бази даних з доменом localhost (по локальній мережі, тому-що база даних і

програма існують в одному і тому ж контейнері), і використовуємо базу даних під назвою postgres.

2. Змінна **username** – це назва користувача під яким програма підключається до бази даних. У цьому випадку це postgres.
3. Змінна **password** – це пароль користувача під яким програма підключається до бази даних. У цьому випадку це також postgres.
4. Остання змінна **jpa.properties.hibernate.dialect** – це змінна яка вказує який SQL діалект (мову) використовувати при формуванні SQL запитів при створенні, зберіганні чи оновленні моделей. Тут вказано щоб він використовував PostgreSQLDialect.

Крок 2. Наступним кроком потрібно розділити програму на два режими роботи – сканування бази даних та експорт просканованих даних. Цей функціонал ми реалізуємо за допомогою Spring Profiles (профайлів), де вкажемо який функціонал використовувати залежно від профайлу. Для цього необхідно:

У пакеті com.database.migration створимо пакет executor та інтерфейс Executor (рис. 3.15) з методом execute().

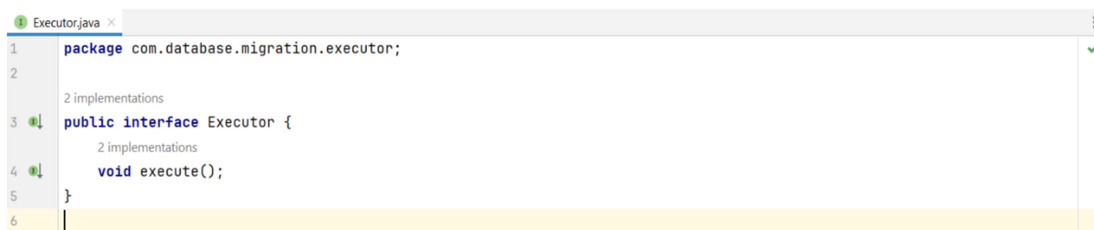


Рисунок 3.15. Інтерфейс Executor

Цей інтерфейс буде необхідний для того, що його наслідники переоприділяли цей метод execute() та імплементували власну логіку поведінки програми.

Наступним кроком створимо клас ScannerExecutor (рис. 3.16), який буде наслідником інтерфейсу Executor і буде викликатись тільки коли встановлений профайл « scan ».

```java
package com.database.migration.executor;

import ...

@Component
@Profile("scan")
public class ScannerExecutor implements Executor {

    @Autowired
    private ScanService scanService;

    @Autowired
    private DatabaseRepository databaseRepository;

    @Override
    public void execute() {
        List<Database> databases = scanService.scan();
        databaseRepository.saveAll(databases);
    }

}
```

Рисунок 3.16. Клас ScannerExecutor

Поки-що не потрібно імплементувати логіку цього метода, залишимо її пустою. Також можна помітити що над класом є дві анотації **@Component** яка дає змогу Spring розпізнавати цей клас при створенні контексту, та **@Profile** зі значенням «scan», що говорить щоб Спринг підтягував цей клас, тільки якщо встановлений цей профайл.

Анотація @Component у Spring використовується для позначення класу як компонента. Це означає, що фреймворк Spring автоматично виявлятиме такі класи для впровадження залежностей при використанні конфігурації на основі анотацій.

Аналогічно зробимо і клас ExportExecutor (рис. 3.17), який також буде наслідником інтерфейсу Executor, але буде викликатись, коли встановлений профайл « export ».

```java
package com.database.migration.executor;

import ...

@Component
@Profile("export")
public class ExportExecutor implements Executor {

    @Autowired
    private ExportService exportService;

    @Override
    public void execute() { exportService.export(); }

}
```

Рисунок 3.17. Клас ExportExecutor

Тепер в класі MigrationApplication (рис. 3.18) що є точкою входу у програму потрібно викликати цей інтерфейс і Спринг автоматично в залежності від встановленого профайлу викличе потрібну логіку.

```
package com.database.migration;

import ...

@SpringBootApplication
public class MigrationApplication {

    public static void main(String[] args) {
        SpringApplication.run(MigrationApplication.class, args)
                .getBean(Executor.class)
                .execute();
    }

}
```

Рисунок 3.18. Клас MigrationApplication

Цей клас був створений автоматично, при створенні проєкту. У ньому можна побачити main() метод, що вказує Java на те, що саме його потрібно викликати при запуску програми, а також анотація @SpringBootApplication, яка також вказує на те, що це є точкою входу для Спринга. Крім того, саме метод getBean(Executor.class) дає Спрингу зрозуміти, що потрібно витягнути екзепляр інтерфейсу Executor, в залежності від встановленого профайлу, це клас ExportExecutor або клас ScannerExecutor.

Анотація @SpringBootApplication у Spring Boot використовується для позначення класу конфігурації, який оголошує один або більше методів з анотацією @Bean, а також запускає автоконфігурацію та сканування компонентів.

Крок 3. Підключення до бази даних яку потрібно просканувати. Для того щоб підключитись до бази даних, яку потрібно просканувати, необхідно створити нові Spring конфігурації і створити методи (bean), над якими поставити анотації **@Bean** і **@Qualifier**. За допомогою першої анотації Спринг помістить ці методи

у свій контекст і зможе автоматично вставляти їх, коли потрібно, а за допомогою другої анотації він матиме зможу розрізняти біни з одинаковим типом.

Однією з найважливіших анотацій у Spring є анотація **@Bean**, яка застосовується до методу для вказівки, що він повертає бін, який буде керуватися контекстом Spring.

У Spring анотація **@Qualifier** використовується для визначення, який саме бін має бути впроваджений, коли доступно кілька бінів одного типу.

Створимо клас StorageDatabaseConfiguration (рис. 3.19) який буде відповідати за підключення до внутрішньої бази даних, де ми зберігаємо проскановані дані.

```
StorageDatabaseConfiguration.java
1   package com.database.migration;
2
3   import ...
11
12  @Configuration
13  public class StorageDatabaseConfiguration {
14
15
16      @Bean
17      @ConfigurationProperties("spring.datasource")
18      public DataSourceProperties storageDataSourceProperties() {
19          return new DataSourceProperties();
20      }
21
22      @Bean
23      @Primary
24      public DataSource storageDataSource() {
25          return storageDataSourceProperties().initializeDataSourceBuilder().build();
26      }
27
28      @Bean
29      public NamedParameterJdbcTemplate storageJdbcTemplate(DataSource dataSource) {
30          return new NamedParameterJdbcTemplate(dataSource);
31      }
32
33  }
34
```

Рисунок 3.19. Клас StorageDatabaseConfiguration

Тут варто відмітити анотацію **@ConfigurationProperties** з значенням spring.datasource, що означає що ці змінні будуть доступні тільки під цим префіксом, ці змінні уже вказані у конфігураційному файлі application.yml. Також варто відмітити змінну **@Primary** яка вказує на те, що при наявності в контексті Спринга двох бігів з одинаковим типом, Спринг завжди буде надавати перевагу цьому біну і піставляти його.

У Spring анотація **@ConfigurationProperties** використовується для прив'язки зовнішніх конфігураційних властивостей до об'єкта Java.

У Spring анотація **@Primary** використовується для надання вищого пріоритету біну, коли існує кілька бінів одного типу.

Створимо клас ScanDatabaseConfiguration (рис. 3.20), у якому створимо потрібні біни.

Рисунок 3.20. Клас ScanDatabaseConfiguration

Підключення до бази даних буде ініцілізуватись тільки коли увімкнений профайл «scan», анотація **@ConfigurationProperties** з значенням scanner.datasource дозволяє вказати що url, username та password для бази даних яка сканується повинні бути вказані з цим префіксом, а анотація **@Autowired** (лінія 21 – 22), вказує Spring те, що він повинен у змінну sessionDatabaseProperties вставити цей клас (бін) зі змінними.

У Spring анотація **@Autowired** використовується для автоматичного впровадження залежностей.

Крок 4. Створення моделі бази даних як Java об’єктів. Для того щоб ми могли використовувати Hibernate ORM який транслюватиме моделі в базі даних в Java об’єкти. Для цього створимо пакет новий пакет model.

Створимо клас Database (рис. 3.21) який буде репрезентацією таблиці databases у Java програмі.

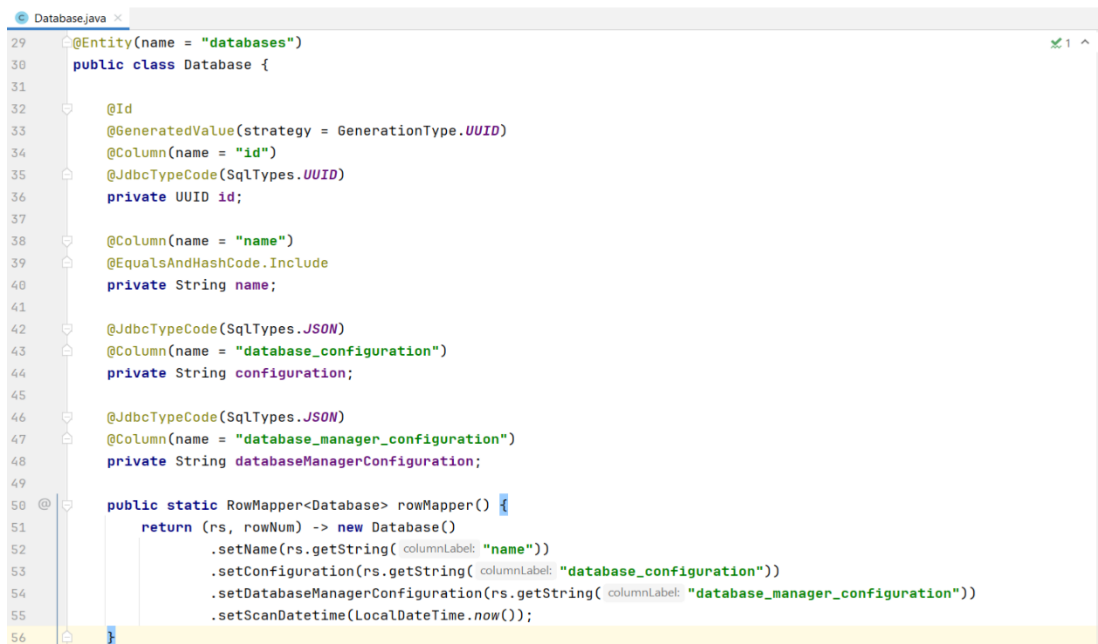


Рисунок 3.21. Клас Database

У цьому класі варто відмітити анотацію **@Entity** якою ми даємо Hibernate зрозуміти, що цей клас є репрезентації таблиці databases, а також анотацію **@Column** щоб позначити що це поле, є колонкою у цій таблиці. Анотація **@Id** над полем, вказує що це поле це PRIMARY KEY, а анотації **@OneToMany** та **@JoinColumn**, вказують що до цієї бази даних, може належати багато схем з’єднані за допомогоб поля database_id.

Також, варто додати, що у цьому класі, як і у всіх наступних буде реалізований метод rowMapper(), який необхідний для того, щоб створити цю модель даних, які будуть полчені із запитів до бази даних яка сканується.

В Java анотація **@Entity** використовується для позначення класу як сутності (entity) у контексті JPA (Java Persistence API). Це означає, що клас

відображатиметься у таблицю бази даних, а його екземпляри представлятимуть рядки цієї таблиці.

Створимо клас Schema (рис. 3.22), який буде репрезентацією таблиці schemas.

Рисунок 3.22. Клас Schema

В Java анотація @Column використовується для відображення атрибута класу сутності (entity) на стовпець у таблиці бази даних. Ця анотація є частиною JPA (Java Persistence API) і дозволяє налаштувати властивості відображення між полем класу та стовпцем бази даних.

Також потрібно створити всі інші моделі: клас Tables (рис. 3.23), для таблиці tables, клас TableColumn для таблиці columns, клас Index для таблиці indexes та інші.

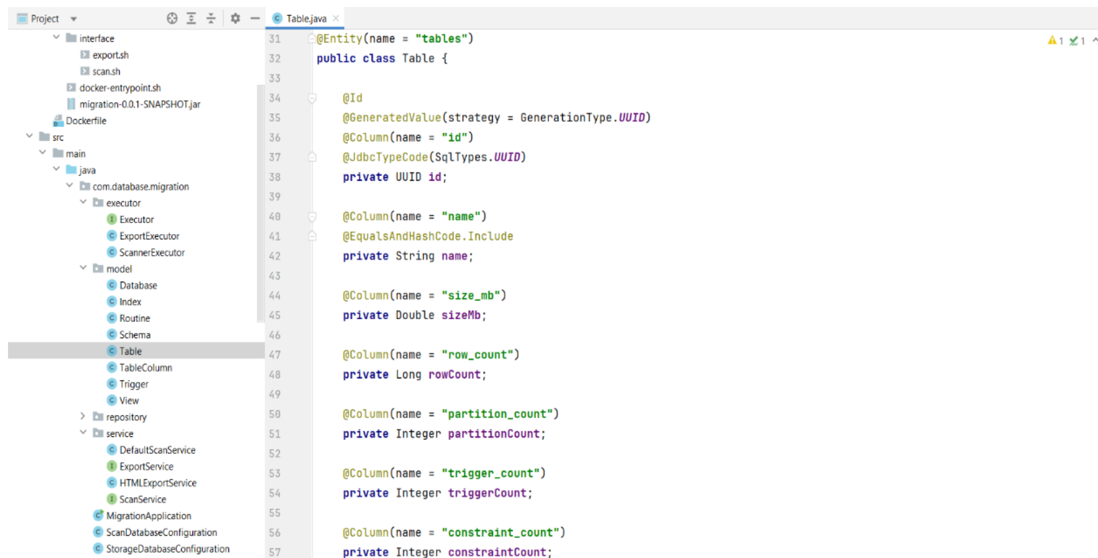


Рисунок 3.23. Клас Table та інші

Можна побчити, що клас Table є репрезентацією таблиці tables у базі даний, а також, що всі інші моделі створені у пакеті model. Також, тут варто додати, що анотація **@GeneratedValue**, при записі в базі даних автоматично створить унікальний **UUID** ідентифікатор об’єкта.

В JPA анотація **@GeneratedValue** використовується для вказівки того, що значення первинного ключа (**@Id**) має генеруватися автоматично.

UUID (Universal Unique Identifier) — це 128-бітне значення, яке використовується для унікальної ідентифікації об'єкта чи сутності в інтернеті.

Крок 5. Створення репозиторію. Для того щоб зберігати проскановані об’єкти в базу даних, необхідно створити репозиторій. Так як у нас всі об’єкти належать до бази даних (рис. 3.21), можна створити репозиторій тільки для цього об’єкту, всі інші об’єкти збережуться транзитивно. Створюємо інтерфейс DatabaseRepository (рис. 3.24).

DatabaseRepository.java

C:\Users\Denys_Burko\IdeaProjects\migration\src\main\java\com\database\migration\repository\DatabaseRepository.java

```java
pac

import ...

@Repository
public interface DatabaseRepository extends JpaRepository<Database, UUID> {
}
```

Рисунок 3.24. Інтерфейс DatabaseRepository

Цей інтерфейс наслідується від інтерфейсу **JpaRepository** - це специфічне розширення інтерфейсу Repository для JPA. Воно включає повний набір API інтерфейсів CrudRepository і PagingAndSortingRepository.

Крок 6. Написання основної логіки сканування бази даних. Для цього створимо пакет service і у цьому пакеті створимо інтерфейс ScanService, у якому буде метод scan(), що буде повертати набір класу Database (рис. 3.21), та його нащадок клас DefaultScanService (рис. 3.25) або (дод. В), який буде імплементувати цей клас та застосовуватись до всіх баз даних, якщо вони не мають своєї імплементації.

DefaultScanService.java

```java
@Service
@Profile("scan")
public class DefaultScanService implements ScanService {

    @Autowired
    @Qualifier("scanner")
    private NamedParameterJdbcTemplate jdbcTemplate;

    @Autowired
    private Environment environment;

    @Override
    public List<Database> scan() {
        List<Database> databases = jdbcTemplate.query(getQuery( key: "query.databases"), Database.rowMapper());
        List<Schema> schemas = jdbcTemplate.query(getQuery( key: "query.schemas"), Schema.rowMapper());
        List<Table> tables = jdbcTemplate.query(getQuery( key: "query.tables"), Table.rowMapper());
        List<TableColumn> columns = jdbcTemplate.query(getQuery( key: "query.columns"), TableColumn.rowMapper());
        List<Index> indexes = jdbcTemplate.query(getQuery( key: "query.indexes"), Index.rowMapper());
        List<Routine> routines = jdbcTemplate.query(getQuery( key: "query.routines"), Routine.rowMapper());
        List<View> views = jdbcTemplate.query(getQuery( key: "query.views"), View.rowMapper());
        List<Trigger> triggers = jdbcTemplate.query(getQuery( key: "query.triggers"), Trigger.rowMapper());

        databases.forEach(database -> database.addSchemas(schemas));
        schemas.forEach(schema -> {
            schema.addTables(tables);
            schema.addIndexes(indexes);
            schema.addRoutines(routines);
            schema.addViews(views);
            schema.addTriggers(triggers);
```

Рисунок 3.25. Клас DefaultScanService

У цьому сервісі, спочатку зчитуються набори даних (**ResultSet**), з бази даних за допомогою запитів із конфіг файлу, які за допомогою методу rowMapper(), який імплементований у кожному класі, перетворює рядки з цього набору на об’єкти, а потім ці об’єкти об’єднуються між собою. Для кожної бази даних по назві шукаються її схеми, для кожної схеми відбувається пошук її таблиць, індексів, рутин, представлень та тригерів, а також для кожної таблиці відбувається пошук її колонок.

ResultSet - це таблиця даних, створена в результаті виконання запитів до бази даних.

Крок 7. Створення запитів. Тепер необхідно створити запити, які будуть виконуватись на базі даних які сканується, для того щоб отримати ці дані. Запити під кожну технологію можуть бути різними і щоб це вирішити, також допоможуть Spring профайли. Достатньо створити конфігураційний файл, наприклад, application-source-mssql.yml (рис. 3.26), і увімкнути профайл source-mssql, таким чином Спринг буде розуміти, що йому потрібно взяти ці змінні із цього конфігураційного файлу.

application-source-mssql.yml

```
1  query:
2    database
3        SELECT
4          DB_NAME() AS name,
5          NULL AS database_configuration,
6          NULL AS database_manager_configuration
7      "
8    schemas: "
9            SELECT
10             DB_NAME() AS database_name,
11             s.name as name,
12             (select min(o.create_date) from sys.objects o WHERE o.schema_id  = s.schema_id) as created_date
13           FROM sys.schemas s
14         "
```

Рисунок 3.26. Файл application-source-mssql.yml)

Тут можна відмітити імплементацію запитів для баз даних (databases) та схем (schemas). Кожен запит дістає саме ті атрибути, які вказані в методі rowMapper() кожного класу та які будуть використовуватись для подальшого аналізу.

Наступним кроком потрібно імплементувати запити для таблиць (рис. 3.27).

```
application-source-mssql.yml
15    tables: "
16      select
17        schema_name(t.schema_id) as schema_name,
18        t.NAME as name,
19        t.create_date as created_date,
20        p.rows as row_count,
21        max(p.partition_number) as partition_count,
22        count(distinct fk.name) as constraint_count,
23        'FALSE' as has_nested_triggers,
24        cast(round((sum(a.used_pages) * 8) / 1024.00,2) as numeric(12,2)) as size_mb,
25        'FALSE' as is_ddl_encrypted
26      from
27        sys.tables t
28          inner join sys.partitions p
29                     on t.object_id = p.OBJECT_ID
30          inner join sys.allocation_units a
31                     on p.partition_id = a.container_id
32          left join sys.external_tables ext_t
33                    on t.object_id = ext_t.object_id
34          left join sys.foreign_keys fk
35                    on t.object_id = fk.parent_object_id
36          left join sys.triggers tr
37                    on t.object_id = tr.parent_id
38      WHERE
39        a.type = 1
40      group by
41        t.object_id,
42        t.Name,
```

Рисунок 3.27. Запит для таблиць

Можна побачити, що цей запит вже значно складніший ніж два попередні, де використовуються агрегації, а також INNER JOIN та LEFT JOIN.

LEFT JOIN, також відомий як LEFT OUTER JOIN, — це тип операції SQL JOIN, яка вибирає всі записи з лівої таблиці (table1) та відповідні записи з правої таблиці (table2). Якщо у правій таблиці немає відповідних записів, то результуючі поля будуть містити значення NULL.

Також потрібно імплементувати всі інші запити, на рисунку буде показано приклад запиту для представлень, всі інші будуть подібні (рис. 3.28).

```
application-source-mssql.yml
  views: "
    select
      schema_name(v.schema_id) as schema_name,
      v.name as name,
      v.create_date as created_date,
      (select
           count(o.type)
       from
           sys.sql_expression_dependencies d
               left join sys.objects o
                         on o.type in('AF','FN','FS','FT','IF')
                             and o.object_id = d.referenced_id
       where d.referencing_id = v.object_id) as has_udf,
      (select
           referencing_id,
           referenced_id,
           REPLACE(referenced_server_name, '\\', '\\\\') as server_name,
           case when referenced_database_name is null
            then  db_name() else referenced_database_name
            end
           as database_name,
           referenced_schema_name as schema_name,
           referenced_entity_name as entity_name,
           o.type as entity_type
       from sys.sql_expression_dependencies e
       left join sys.objects o on e.referenced_id=o.object_id
       where e.referencing_id = v.object_id for json path,root('dependencies')) as dependencies,
      mod.definition as code
```

Рисунок 3.28. Запит для представлень

Тут також варто відмітити комплексніть цього запиту, а також те що він містить підзапити для того щоб зформувати об’єкти.

Крок 8. Збереження об’єктів у базу даних. Для того щоб зберегти проскановані об’єкти у внутрішню базу даних, у класі ScannerExecutor (рис. 3.29).

```java
ScannerExecutor.java
package com.database.migration.executor;

import ...

@Component
@Profile("scan")
public class ScannerExecutor implements Executor {

    @Autowired
    private ScanService scanService;

    @Autowired
    private DatabaseRepository databaseRepository;

    @Override
    public void execute() {
        List<Database> databases = scanService.scan();
        databaseRepository.saveAll(databases);
    }

}
```

Рисунок 3.29. Клас ScannerExecutor

У цьому класі викликається метод scan() із сервісу ScanService, потім за допомогою репозиторію DatabaseRepository (рис. 32) збережемо набір цих об’єктів екзеплярів класу (рис. 3.21).

Крок 9. Створення аналітичних представлень. Усі аналітичні представлення будуть створені на внутрішній базі даних PostgreSQL.

У папці src/main/resources/db.migration створимо файл table_analysis.sql (рис. 3.30), та наповнимо його представленням (**view**) яке буде агрегувати проскановані дані та аналізувати складність таблиць.

View - це віртуальна таблиця, вміст якої визначається запитом. Як і таблиця, вид складається з набору іменованих стовпців і рядків даних.

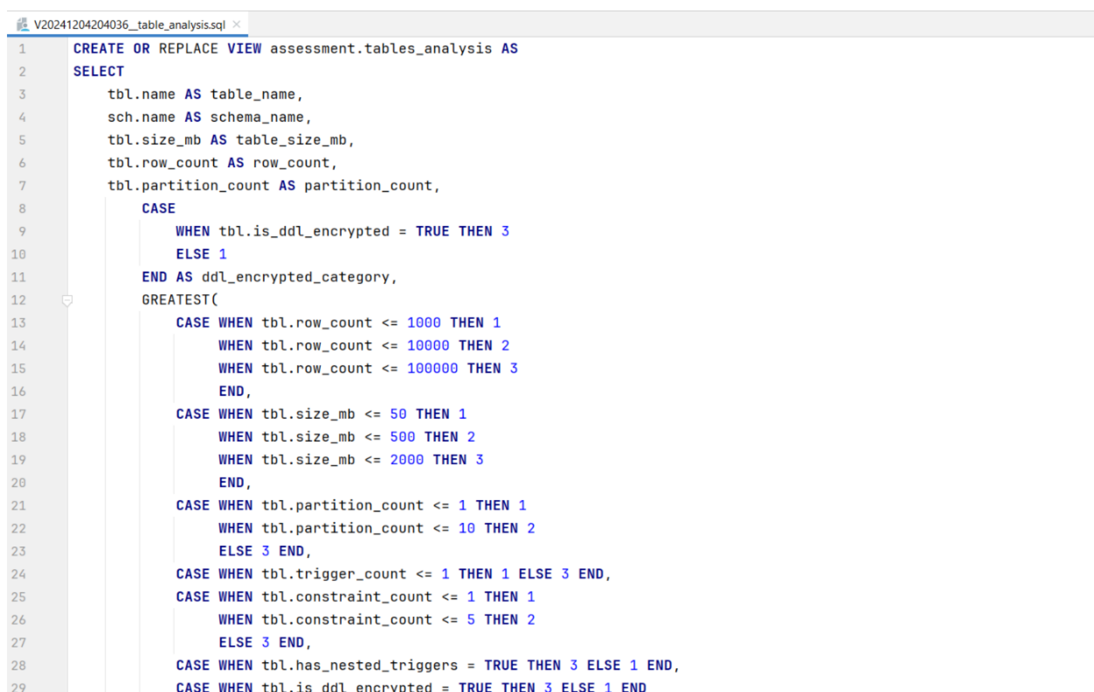


Рисунок 3.30. Файл table_analysis.sql

Подібні преставлення створимо для всіх інших об’єктів, які потрібно оцінити. (індекси, тригери, представлення, рутини та інші) (рис. 3.31).

```
Project
    MigrationApplication
    ScanDatabaseConfiguration
    StorageDatabaseConfiguration
  resources
    db.migration
      V20241203172925__create_schema.sql
      V20241203174002__database.sql
      V20241203175436__schema.sql
      V20241203176984__table.sql
      V20241203184941__index.sql
      V20241203190951__column.sql
      V20241203194553__trigger.sql
      V20241203201415__routine.sql
      V20241203203948__view.sql
      V20241204204036__table_analysis.sql
      V20241204210133__index_analysis.sql
      V20241204210227__column_analysis.sql
      V20241204210256__trigger_analysis.sql
      V20241204210317__routine_analysis.sql
      V20241204210342__view_analysis.sql
      V20241204211101__schema_analysis.sql
      V20241204211128__database_analysis.sc
    application.yml
    application-source-mssql.yml
    template.html
  test
    java
      com.database.migration
  target
  .gitattributes
  .gitignore
  HELP.md
  mvnw
  mvnw.cmd
```

```sql
-- V20241204210133__index_analysis.sql
CREATE OR REPLACE VIEW assessment.indexes_analysis AS
SELECT
    idx.name AS index_name,
    sch.name AS schema_name,
    tbl.name AS table_name,
    idx.size_mb AS index_size_mb,
    idx.columns AS index_columns,
        GREATEST(
            CASE WHEN idx.size_mb <= 10 THEN 1
                 WHEN idx.size_mb <= 100 THEN 2
                 WHEN idx.size_mb <= 500 THEN 3
                 END,
            CASE WHEN LENGTH(idx.columns) <= 50 THEN 1
                 WHEN LENGTH(idx.columns) <= 200 THEN 2
                 ELSE 3 END
        ) AS complexity,
    idx.scan_datetm AS index_scan_datetm,
    CASE
        WHEN idx.size_mb <= 10 THEN 1
        WHEN idx.size_mb <= 100 THEN 2
        WHEN idx.size_mb <= 500 THEN 3
        ELSE 4
    END AS index_size_category,
    CASE
        WHEN LENGTH(idx.columns) <= 50 THEN 1
        WHEN LENGTH(idx.columns) <= 200 THEN 2
        ELSE 3
    END AS columns_count_category
FROM assessment.indexes idx
```

Рисунок 3.31. Файл index_analysis.sql

Зверніть увагу на дерево файлів зліва, всі аналітичні представлення є створені.

Тепер потрібно створити файл database_analysis.sql (рис. 3.32). Це представлення буде використовуватись при формунні основного звіту.

```sql
-- V20241204211128__database_analysis.sql
CREATE OR REPLACE VIEW assessment.databases_analysis AS
SELECT
    db.name AS database_name,

       GREATEST(
           CASE WHEN COUNT(DISTINCT sch.id) <= 10 THEN 1
                WHEN COUNT(DISTINCT sch.id) <= 50 THEN 2
                ELSE 3 END,
           CASE WHEN COUNT(DISTINCT tbl.id) <= 100 THEN 1
                WHEN COUNT(DISTINCT tbl.id) <= 1000 THEN 2
                ELSE 3 END,
           CASE WHEN COUNT(DISTINCT idx.id) <= 50 THEN 1
                WHEN COUNT(DISTINCT idx.id) <= 200 THEN 2
                ELSE 3 END,
           CASE WHEN COUNT(DISTINCT col.id) <= 500 THEN 1
                WHEN COUNT(DISTINCT col.id) <= 2000 THEN 2
                ELSE 3 END,
           CASE WHEN COUNT(DISTINCT trg.id) <= 10 THEN 1
                WHEN COUNT(DISTINCT trg.id) <= 50 THEN 2
                ELSE 3 END,
           CASE WHEN COUNT(DISTINCT r.id) <= 10 THEN 1
                WHEN COUNT(DISTINCT r.id) <= 50 THEN 2
                ELSE 3 END,
           CASE WHEN COUNT(DISTINCT v.id) <= 10 THEN 1
                WHEN COUNT(DISTINCT v.id) <= 50 THEN 2
                ELSE 3 END
        ) AS database_complexity,

    SUM(idx.size mb) AS total index size mb.
```

Рисунок 3.32. Файл database_analysis.sql

У цьому представленні, крім визначення складності, ми ще збираємо кількість всіх об'єктів що є в базі даних, а також сумуємо основні їх атрибути.

Крок 10. Імплементація експорту аналітичних представлень в HTML. Для того щоб експортувати дані в HTML потрібно створити шаблон, який ми будемо заповняти а також написати клас HTMLExportService (рис. 3.34), у якому буде логіка заповнення цього шаблону.

Для того щоб створити шаблон потрібно у папці src/main/resources створити файл template.html (рис. 3.33), у якому потрібно описати основну структуру сторінки, а також залишит плейсхолдери, які будуть заповнятись актуальними даними з аналітичних представлень.

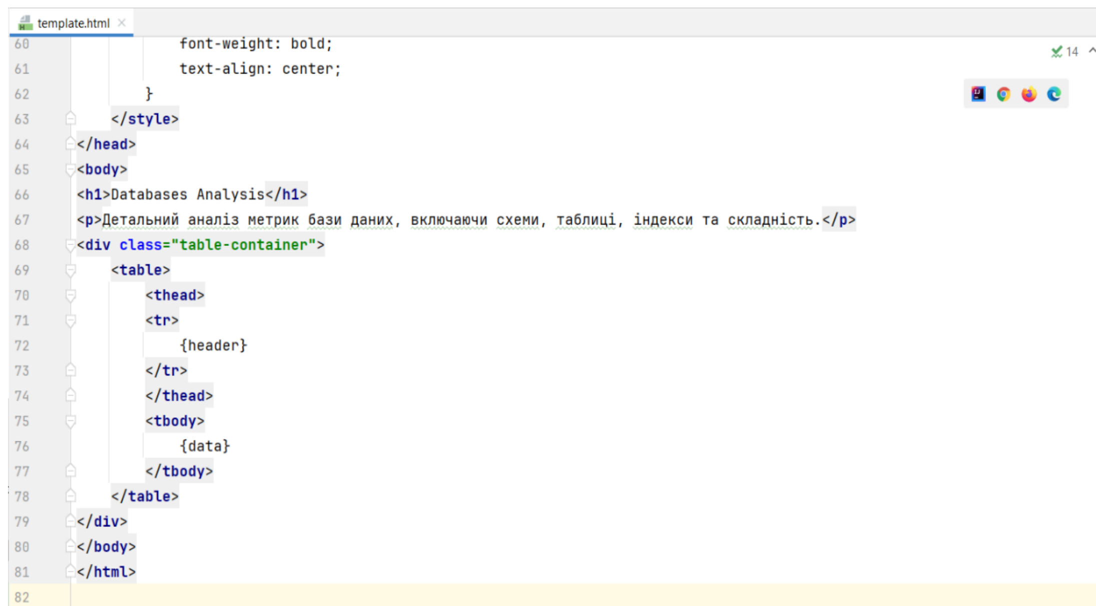


Рисунок 3.33. Файл template.html

Плейсхолдери {header} та {data} – це саме ті плейсхолдери які будуть заповнятись актуальними даними з аналітичних представлень.

Placeholder - це частина тексту, яка тимчасово розміщується в документі чи іншому місці, доки на пізньому етапі там не буде вставлено остаточний текст.

Тепер потрібно створити клас HTMLExportService (рис. 3.34), який буде брати з дані з аналітичних представлень та заповнювати ними плейсхолдери в HTML шаблоні.

```java
HTMLExportService.java

@Override
public void export() { views.forEach(this::exportHTML); }

@SneakyThrows
private void exportHTML(String name) {
    List<Map<String, Object>> result = jdbcTemplate.queryForList( sql: "SELECT * FROM " + name);

    if (!result.isEmpty()) {
        Resource resource = resourceLoader.getResource( location: "classpath:template.html");
        String template = resource.getContentAsString(Charset.defaultCharset());

        String header = result.get(0).keySet().stream()
                .map(column -> "<th>" + column.replace( target: "_",  replacement: " ") + "</th>")
                .collect(Collectors.joining( delimiter: "\n"));

        String data = result.stream() Stream<Map<...>>
                .map(row -> row.values().stream() Stream<Object>
                        .map(value -> "<td>" + value + "</td>") Stream<String>
                        .collect(Collectors.joining( delimiter: "\n"))) Stream<String>
                .map(row -> "<tr>" + row + "</tr>")
                .collect(Collectors.joining());

        String output = template.replace( target: "{header}", header).replace( target: "{data}", data);

        Path folder = Path.of( first: "/app",  ...more: "output");

        if (Files.notExists(folder)) {
            Files.createDirectory(folder);
```

Рисунок 3.34. Клас HTMLExportService

У цьому класі у export() методі перебираються всі назви представлень по черзі і для кожної з цих назвах викликається метод exportHTML(), який саме вичитує шаблон template.html, після чого робить запит в базу даних, щоб прочитати дані з аналітичних представлень, потім з першого рядка вибираються всі назви колонок і з них формується шапка таблиці (**header**), після цього зі всіх рядків представлення формуються рядки бази даних, після чого у шаблоні заміняються плейсхолдери, та вже готовий результат записується в файл з назвою представлення + “html” розширенням.

Тег <th> визначає заголовкову комірку в таблиці HTML. Таблиця HTML має два типи комірок: комірки заголовка - містять інформацію заголовка (створюються за допомогою елемента <th>) і комірки даних - містять дані (створюються за допомогою елемента <td>).

Крок 11. Розроблення CLI інтерфейсу. Інтерфейс програми буде розроблений як **CLI,** тобто користувач буде використувати Bash команди в середині контейнера.

Спочатку у папці docker/app створимо папку interface. Після цього у цій папці створимо файл scan.sh (рис. 3.35) та заповнимо його.

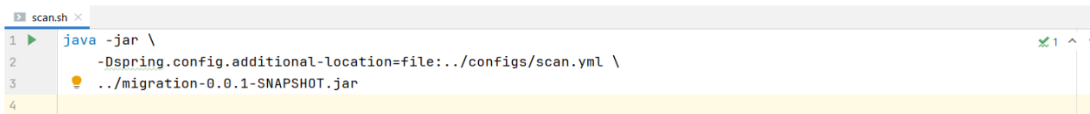


Рисунок 3.35. Файл scan.sh

У цьому файлі використовується команда java -jar для того, щоб запустити виконання програми, після цього передається параметр spring.config.additional-location для того щоб вказати Спрингу додаткову конфігураю, та вказується шлях до **JAR** файлу (виконуваний файл з байт-кодом програми).

Файл **JAR** (Java Archive) - це формат пакувального файлу, який використовується в Java для об'єднання декількох файлів в один архів, зазвичай для спрощення розповсюдження та розгортання. Це, по суті, ZIP-файл із розширенням .jar, який використовується для пакування файлів класів Java, ресурсів і метаданих (наприклад, MANIFEST.MF).

Коли користувацькі місця розташування конфігурацій налаштовуються за допомогою spring.config.additional-location, вони використовуються на додаток до стандартних місць розташування. Додаткові місця розташування шукаються перед стандартними місцями розташування.

Тепер потрібно створити конфігураційний файл. Для цього створимо папку configs та у ній створимо файл scan.yml (рис. 3.36).

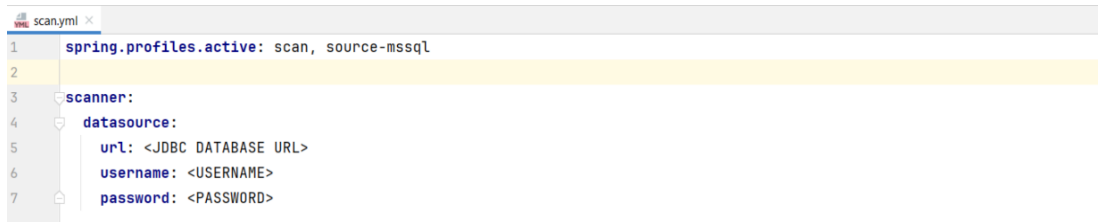


Рисунок 3.36. Файл scan.yml

Ця конфігурація автоматично включає профайли “scan” та “source-mssql” (лінія 1) використовуючи spring.profiles.active, а також вимагає заповнення JDBC посилання до бази даних, а також назву і пароль від користувача бази даних.

У звичному середовищі Spring можна використовувати властивість середовища spring.profiles.active, щоб вказати, які профілі активні.

Аналогічно у папці interface та у папці configs, створимо Bash команду, export.sh (рис. 3.37) та конфіг файл export.yml (рис. 3.38) для експортера.

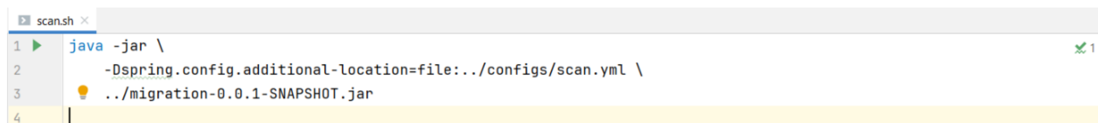


Рисунок 3.37. Файл export.sh

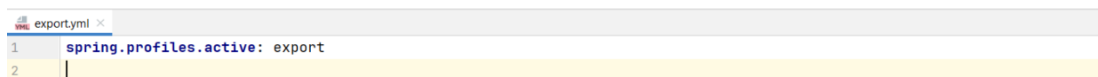


Рисунок 3.38. Файл export.yml

У цьому профайлі вказано лише, що потрібно застосовувати профайл “export”.

Крок 12. Для того щоб зібрати Java проєкт спочатку потрібно виконати команду Maven clean package у терміналі:

```
./mvnw clean package
```

Після цього потрібно перейти в папку docker і виконати комаду Docker build:

```
docker build -t dbmigration docker
```

Ця команда створить імедж з назвою dbmigration і помістить усі файли з папки docker всередину контейнера (в його контекст).

Команда mvn clean package в Maven виконує дві основні задачі: спочатку очищає директорію target, видаляючи попередні зібрані файли, а потім компілює вихідний код, виконує тести та упаковує код у JAR, WAR або інший артефакт, зазначений у pom.xml. Це забезпечує побудову проєкту з нуля без залишкових файлів від попередніх зібрань.

Docker створює образи використовуючи команду build, читаючи інструкції з Dockerfile. Dockerfile - це текстовий файл, що містить інструкції для побудови вашого вихідного коду.

3.6 Тестування проєкту

Крок 1. Спочатку потрібно запустити контейнер за допомогою команди docker run (рис. 3.39):

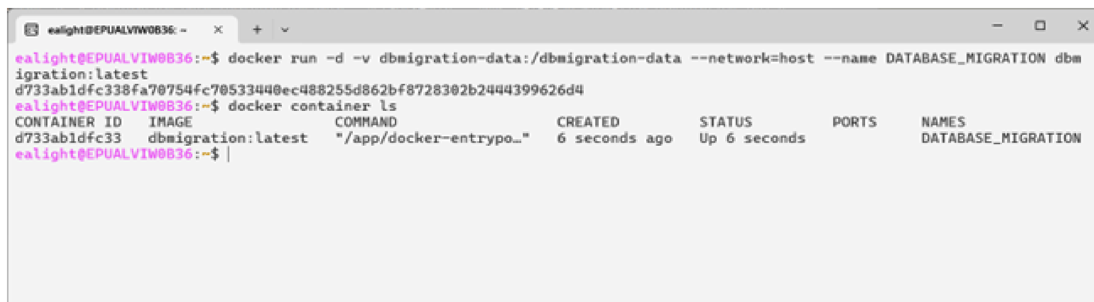


Рисунок 3.39. Запуск контейнера

Крок 2. Наступним кроком потрібно зайти всередину контейнера за допомогою комади docker exec (рис. 3.40):

Рисунок 3.40. Авторизація в контейнері

Крок 3. Заповнення конфігурації підключення до бази даних у файлі scan.yml (рис. 3.41) у папці configs:

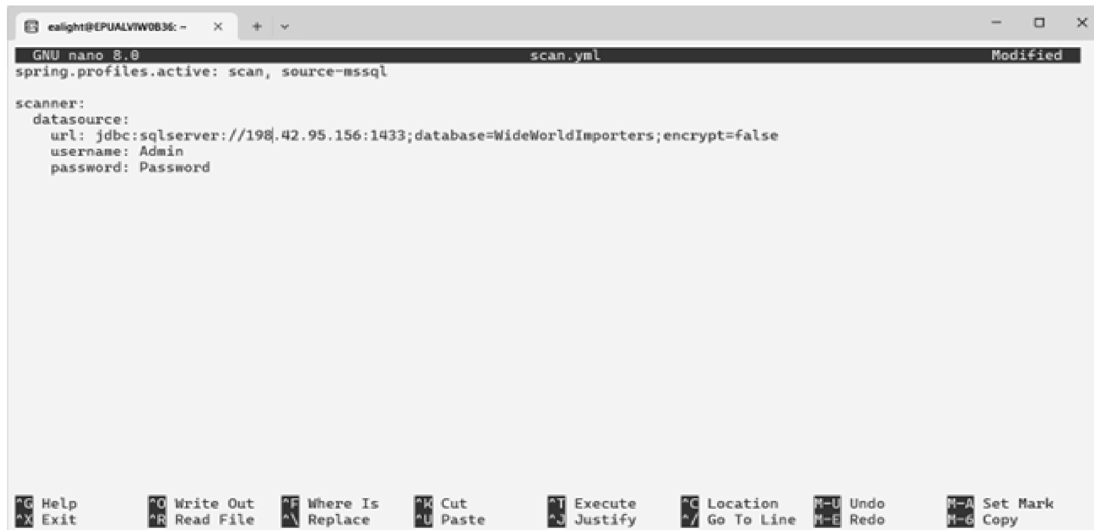


Рисунок 3.41. Файл scan.yml

Крок 4. Для того щоб просканувати базу даних, потрібно перейти у папку interface та викликати bash команду scan.sh (рис. 3.42):

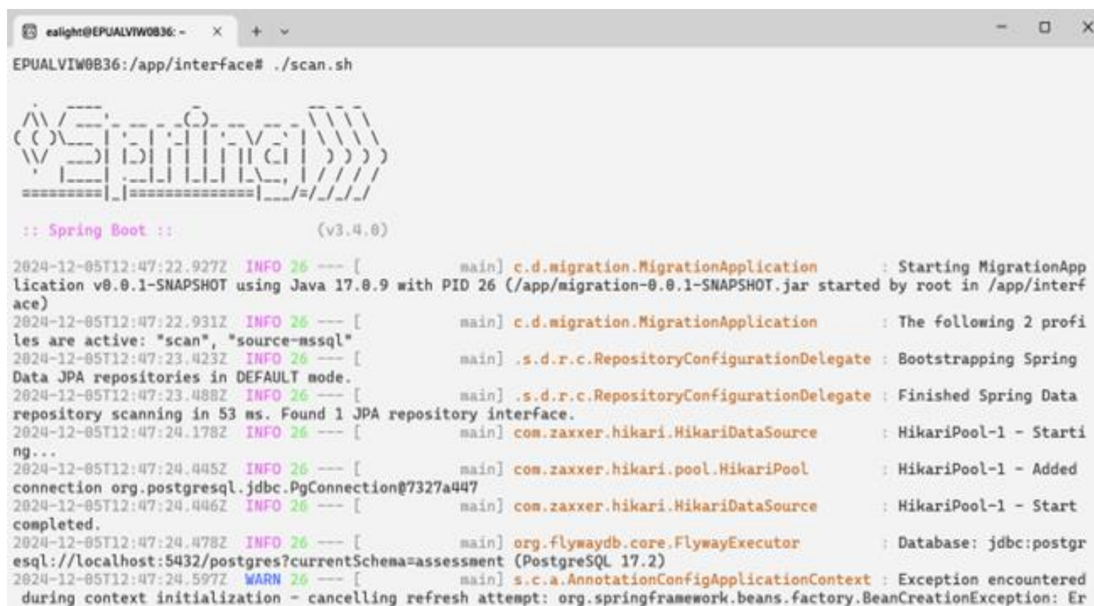


Рисунок 3.42. Сканування бази даних

Крок 5. Для того щоб експортувати результат, потрібно перейти у папку interface та викликати bash команду export.sh (рис. 3.43):

```
ealight@EPUALVIW0B36: ~
EPUALVIW0B36:/app/interface# ls
export.sh  scan.sh
EPUALVIW0B36:/app/interface# ./export.sh

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/

 :: Spring Boot ::                (v3.4.0)

2024-12-05T13:55:55.225Z  INFO 859 --- [           main] c.d.migration.MigrationApplication       : Starting MigrationApplication v0.0.1-SNAPSHOT using Java 17.0.9 with PID 859 (/app/migration-0.0.1-SNAPSHOT.jar started by root in /app/interface)
2024-12-05T13:55:55.227Z  INFO 859 --- [           main] c.d.migration.MigrationApplication       : The following 1 profile is active: "export"
2024-12-05T13:55:55.706Z  INFO 859 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2024-12-05T13:55:55.761Z  INFO 859 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 43 ms. Found 1 JPA repository interface.
2024-12-05T13:55:56.347Z  INFO 859 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2024-12-05T13:55:56.535Z  INFO 859 --- [           main] com.zaxxer.hikari.pool.HikariPool        : HikariPool-1 - Added connection org.postgresql.jdbc.PgConnection@3d20e575
2024-12-05T13:55:56.537Z  INFO 859 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2024-12-05T13:55:56.564Z  INFO 859 --- [           main] org.flywaydb.core.FlywayExecutor         : Database: jdbc:postgresql://localhost:5432/postgres?currentSchema=assessment (PostgreSQL 17.2)
```

Рисунок 3.43. Експорт результатів

Крок 6. Тепер перейдемо у папку /app/output (рис. 3.44): і перевіримо результати:

```
ealight@EPUALVIW0B36: ~
EPUALVIW0B36:/app/output# ls -al
total 292
drwxr-xr-x    2 root     root          4096 Dec  5 13:54 .
drwxr-xr-x    1 root     root          4096 Dec  5 13:54 ..
-rw-r--r--    1 root     root        193672 Dec  5 13:55 columns_analysis.html
-rw-r--r--    1 root     root          3066 Dec  5 13:56 databases_analysis.html
-rw-r--r--    1 root     root         28627 Dec  5 13:56 routines_analysis.html
-rw-r--r--    1 root     root         12594 Dec  5 13:56 schema_analysis.html
-rw-r--r--    1 root     root         29316 Dec  5 13:56 tables_analysis.html
-rw-r--r--    1 root     root          6216 Dec  5 13:56 views_analysis.html
```

Рисунок 3.44. Папка output

РОЗДІЛ 4. ОХОРОНА ПРАЦІ

4.1 Охорона праці

Охорона праці – це комплексна система, що забезпечує безпечні та здорові умови роботи для всіх співробітників. Вона охоплює правові, організаційні, технічні, соціальні та психологічні аспекти, спрямовані на зменшення виробничих ризиків і впливу шкідливих факторів на людину. Основою цієї системи є законодавчі акти, які регулюють охорону праці та визначають механізми реалізації заходів для захисту здоров’я і життя працівників.

Ключовим документом у цій сфері є Закон України "Про охорону праці" [35], який закріплює права й обов’язки роботодавців і працівників, а також регламентує контроль за дотриманням вимог безпеки. Додатково до цього використовуються Державні нормативно-правові акти з охорони праці (ДНАОП), які визначають конкретні вимоги до умов роботи, безпеки обладнання, інструментів та технологічних процесів.

Системний підхід до охорони праці передбачає оцінку ризиків, що можуть спричинити травми або захворювання. Цей процес включає:

1. визначення потенційно небезпечних факторів (фізичних, хімічних, біологічних, психологічних тощо), які можуть впливати на працівників;
2. аналіз ймовірності виникнення небезпеки та можливих наслідків для здоров’я;
3. розробку та впровадження заходів для зменшення ризиків.

Ефективне управління охороною праці вимагає створення організаційної структури, яка включає:

1. розподіл обов’язків і відповідальності серед працівників, керівників і служб безпеки;

2. регулярні інструктажі та навчання персоналу з техніки безпеки;
3. формування служби охорони праці, що контролює дотримання вимог безпеки й об'єднує представників адміністрації та працівників.

4.2 Безпека у надзвичайних ситуаціях

Надзвичайні ситуації (НС) - це несподівані події, які можуть спричинити серйозну шкоду здоров'ю людей, майну або довкіллю. Вони поділяються на техногенні (аварії на виробництві, транспорті, енергетичних об'єктах), природні (землетруси, повені, урагани), соціальні (масові заворушення, терористичні акти) та екологічні (радіаційні аварії, забруднення навколишнього середовища).

До основних видів НС належать:

1. Техногенні: аварії, вибухи, витоки токсичних речовин на промислових чи інфраструктурних об'єктах.
2. Природні: стихійні лиха, такі як бурі, паводки чи землетруси
3. Соціальні та політичні: громадянські заворушення, терористичні дії
4. Екологічні: катастрофи, що призводять до забруднення середовища.

Для ефективного реагування на НС необхідно підготуватися заздалегідь:

1. Розробити плани евакуації та ліквідації наслідків.
2. Організувати навчання працівників, включаючи тренування та інструктажі.
3. Створити систему раннього оповіщення для швидкого реагування (сигналізація, моніторинг умов).
4. Забезпечити персонал засобами індивідуального захисту (протигазами, захисними костюмами, рятувальним обладнанням).

У разі виникнення НС необхідно:

1. Евакуювати людей із небезпечних зон та надати першу медичну допомогу.
2. Ліквідувати наслідки (гасіння пожеж, очищення територій, відновлення інфраструктури).
3. Координувати дії з владою та екстреними службами.

Особлива увага приділяється забезпеченню безпеки під час пожеж або перебоїв з електропостачанням. В офісах встановлюють вогнегасники, проводять тренінги для працівників, організовують резервне живлення (генератори, блоки безперебійного живлення) та альтернативні канали зв’язку.

В умовах війни або конфліктів безпека працівників і захист інфраструктури є критичними. IT-компанії мають забезпечити:

1. Фізичну безпеку працівників (сховища, евакуація).
2. Захист даних та систем (резервування, безпечні комунікаційні канали).
3. Можливість віддаленої роботи для зниження ризиків.

Такі заходи включають тренінги, розробку планів дій під час криз та співпрацю з державними органами. Використання сховищ, резервних систем і віддаленої роботи допомагає мінімізувати ризики для співробітників і зберегти безперервність бізнес-процесів.

Охорона праці та підготовка до надвичайних ситуацій - це багаторівневий процес, що вимагає комплексного підходу. Ефективна стратегія безпеки знижує ризики, мінімізує економічні втрати та забезпечує стабільність діяльності організацій навіть у кризових умовах.

РОЗДІЛ 5. ВИЗНАЧЕНЯ ЕФЕКТИВНОСТІ

5.1 Аналіз ефективності програми

У файлі databases_analysis.html можна побачити загальну оцінку складності баз даних (рис. 5.1). Ця складність оцінюється на основі об'єктв що існують в базі даних, тобто ідексів (колонка Index Complexity), тригерів (колонка Column Complexity), та інших. Також у цій таблиці можна побачити кількість об'єктів кожного типу.

Databases Analysis

Детальний аналіз метрик бази даних, включаючи схеми, таблиці, індекси та складність.

DATABASE NAME	DATABASE COMPLEXITY	INDEX COMPLEXITY	COLUMN COMPLEXITY	TRIGGER COMPLEXITY	ROUTINE COMPLEXITY	VIEW COMPLEXITY	TOTAL SCHEMAS	TOTA
WideWorldImporters	2	3	3	3	3	3	23	57

Рисунок 5.1. Результати аналізу баз даних

У файлі schemas_analysis.html можна побачити оцінку складності кожної схеми (рис. 5.2). Вони крім того що показують оцінку всіх компонентів, що є у схемі мають і свою оцінку (колонка Schema Complexity).

Databases Analysis

Детальний аналіз метрик бази даних, включаючи схеми, таблиці, індекси та складність.

SCHEMA NAME	SCHEMA COMPLEXITY	TRIGGER COMPLEXITY	ROUTINE COMPLEXITY	VIEW COMPLEXITY	INDEX COMPLEXITY	COLUMN COMPLEXITY	TOTAL TABLES	T
db_denydatawriter	1	3	3	3	3	1	0	0
Warehouse	3	3	3	3	3	3	14	32
db_datawriter	1	3	3	3	3	1	0	0
db_denydatareader	1	3	3	3	3	1	0	0
sys	1	3	3	3	3	1	0	0
PowerBI	1	3	3	3	3	1	0	0
Integration	2	3	3	3	3	1	0	0
DataLoadSimulation	1	3	3	3	3	1	0	0
Sales	3	3	3	3	3	3	13	57
db_datareader	1	3	3	3	3	1	0	0
Website	2	3	3	3	3	1	0	0
db_ddladmin	1	3	3	3	3	1	0	0
db_owner	1	3	3	3	3	1	0	0
Reports	1	3	3	3	3	1	0	0

Рисунок 5.2. Результати аналізу схем

У файлі tables_analysis.html можна побачити оцінку складності усіх таблиць що є в базі даних (рис. 5.3). У цій таблиці можна побачити атрибути кожної з таблиць, а також їх оцінку (колонка Complexity).

Databases Analysis

Детальний аналіз метрик бази даних, включаючи схеми, таблиці, індекси та складність.

TABLE NAME	SCHEMA NAME	DDL ENCRYPTED CATEGORY	COMPLEXITY	TABLE SIZE MB	ROW COUNT	PARTITION COUNT	CONSTRAINT COUNT	HAS NESTED T
SupplierTransactions	Purchasing	1	3	0.31	2438	1	5	false
CustomerTransactions	Sales	1	3	11.33	97147	1	5	false
DeliveryMethods_Archive	Application	1	3	0.02	1	1	0	false
StateProvinces_Archive	Application	1	3	0.24	104	1	0	false
PaymentMethods	Application	1	3	0.03	4	1	1	false
StateProvinces	Application	1	3	0.77	53	1	2	false
TransactionTypes_Archive	Application	1	3	0.02	1	1	0	false
Countries_Archive	Application	1	3	0.12	37	1	0	false
People	Application	1	3	0.91	1111	1	1	false
People_Archive	Application	1	3	0.38	961	1	0	false
SystemParameters	Application	1	3	0.14	1	1	3	false
TransactionTypes	Application	1	3	0.03	13	1	1	false
PaymentMethods_Archive	Application	1	3	0.02	1	1	0	false
Cities	Application	1	3	9.48	37940	1	2	false
Cities_Archive	Application	1	3	0.02	28	1	0	false

Рисунок 5.3. Результати аналізу таблиць

У файлі columns_analysis.html є оцінка складності усіх колонок що є в базі даних (рис. 5.4). У таблиці є атрибути кожної з колонок та оцінка (колонка Complexity). Колонки бувають складними тільки якщо мають обмеження.

Databases Analysis

Детальний аналіз метрик бази даних, включаючи схеми, таблиці, індекси та складність.

COLUMN NAME	TABLE NAME	COLUMN TYPE	COLUMN ENCODING	COMPLEXITY	HAS PRIMARY KEY	COLUMN SCAN DATETM	PRIMARY KEY CATEGOR
OrderID	Orders	int	0	3	true	2024-12-10 18:04:17.290416	3
ValidTo	DeliveryMethods_Archive	datetime2	0	1	false	2024-12-10 18:04:17.405441	1
LastEditedBy	DeliveryMethods_Archive	int	0	1	false	2024-12-10 18:04:17.405441	1
DeliveryMethodID	DeliveryMethods_Archive	int	0	1	false	2024-12-10 18:04:17.405441	1
ValidFrom	DeliveryMethods_Archive	datetime2	0	1	false	2024-12-10 18:04:17.405441	1
DeliveryMethodName	DeliveryMethods_Archive	nvarchar	0	1	false	2024-12-10 18:04:17.405441	1
ValidTo	StateProvinces_Archive	datetime2	0	1	false	2024-12-10 18:04:17.268897	1
LastEditedBy	StateProvinces_Archive	int	0	1	false	2024-12-10 18:04:17.268897	1
SalesTerritory	StateProvinces_Archive	nvarchar	0	1	false	2024-12-10 18:04:17.267896	1
Border	StateProvinces_Archive	geography	0	1	false	2024-12-10 18:04:17.267896	1
LatestRecordedPopulation	StateProvinces_Archive	bigint	0	1	false	2024-12-10 18:04:17.267896	1
StateProvinceName	StateProvinces_Archive	nvarchar	0	1	false	2024-12-10 18:04:17.267896	1
CountryID	StateProvinces_Archive	int	0	1	false	2024-12-10 18:04:17.267896	1
StateProvinceID	StateProvinces_Archive	int	0	1	false	2024-12-10 18:04:17.267896	1

Рисунок 5.4. Результати аналізу колонок

У файлі routines_analysis.html можна побачити оцінку складності усіх функцій та процедур що є в базі даних (рис. 5.5). У цій таблиці можна побачити атрибути кожної рутини, а також їх оцінку (колонка Complexity). Рутини рахуються складними, якщо мають динамічний SQL, або багато залежностей.

Databases Analysis

Детальний аналіз метрик бази даних, включаючи схеми, таблиці, індекси та складність.

ROUTINE NAME	SCHEMA NAME	ROUTINE TYPE	COMPLEXITY	HAS DYNAMIC SQL	DYNAMIC SQL CATEGORY	LINE COUNT CATEGORY	LINE COUN
SearchForStockItemsByTags	Website	P	1	null	1	4	null
Configuration_EnableInMemory	Application	P	1	null	1	4	null
DeactivateTemporalTablesBeforeDataLoad	DataLoadSimulation	P	1	null	1	4	null
Configuration_ApplyDataLoadSimulationProcedures	DataLoadSimulation	P	1	null	1	4	null
DetermineCustomerAccess	Application	F	1	null	1	4	null
ActivateWebsiteLogon	Website	P	1	null	1	4	null
CalculateCustomerPrice	Website	F	1	null	1	4	null
ChangePassword	Website	P	1	null	1	4	null
RecordVehicleTemperature	Website	P	1	null	1	4	null
SearchForSuppliers	Website	P	1	null	1	4	null
SearchForCustomers	Website	P	1	null	1	4	null
SearchForStockItems	Website	P	1	null	1	4	null
InsertCustomerOrders	Website	P	1	null	1	4	null
RecordColdRoomTemperatures	Website	P	1	null	1	4	null

Рисунок 5.5. Результати аналізу рутин

У файлі views_analysis.html можна побачити оцінку складності усіх функцій та процедур що є в базі даних (рис. 5.5). У цій таблиці можна побачити атрибути кожного представлення, а також їх оцінку (колонка Complexity). Представлення також рахуються складними якщо мають багато залежностей (джоінів).

Databases Analysis

Детальний аналіз метрик бази даних, включаючи схеми, таблиці, індекси та складність.

VIEW NAME	SCHEMA NAME	COMPLEXITY	HAS UDF	UDF CATEGORY	LINE COUNT CATEGORY	LINE COUNT	CHAR COUNT	SELECT STATEMENT COUNT	MERGE STATEME
Customers	Website	1	false	1	4	null	null	null	null
VehicleTemperatures	Website	1	false	1	4	null	null	null	null
MyView_no_source2	dbo	1	false	1	4	null	null	null	null
Suppliers	Website	1	false	1	4	null	null	null	null
MyView_no_source	dbo	1	false	1	4	null	null	null	null
MyView	dbo	1	false	1	4	null	null	null	null
MyView3	dbo	1	false	1	4	null	null	null	null
MyView2	dbo	1	false	1	4	null	null	null	null

Рисунок 5.5. Результати аналізу представлень

5.2 Оцінка ефективності програми

Програма дозволяє детально проаналізувати структуру бази даних, щоб зрозуміти, наскільки складним буде її перенесення. Вона аналізує різні об'єкти бази даних, такі як таблиці, схеми, колонки, функції, процедури, представлення та інші, і надає розширені технічні показники, які дозволяють визначити складність міграції. Складність бази визначається за кількістю та типами її об'єктів. Наприклад, програма може виявити, що в базі даних міститься 250 таблиць, з яких 40% мають понад 10 колонок і понад 15 залежностей. Кількість індексів оцінюється окремо, і, наприклад, у базі може бути 500 індексів, із яких 30% мають складну структуру. Програма також аналізує тригери; якщо з 150 тригерів 40% мають складну логіку, це підвищує загальний рівень складності бази.

Схеми оцінюються за кількістю об’єктів, що до них входять, та їхньою взаємозалежністю. Наприклад, у базі є 10 схем, з яких 3 мають понад 50 об’єктів і складні залежності між компонентами. Таблиці оцінюються за структурою: наприклад, середня кількість атрибутів на таблицю становить 8, а 15% таблиць мають більше 5 зовнішніх ключів. Це дозволяє оцінити складність таблиць і визначити, які з них потребують додаткової уваги.

Колонки аналізуються на наявність обмежень. Наприклад, у базі є 5000 колонок, із яких 60% мають первинні чи зовнішні ключі, а 25% використовують специфічні типи даних, такі як JSON чи ARRAY. Колонки з такими типами даних вважаються складними для міграції.

Функції та процедури оцінюються за залежностями та складністю логіки. Наприклад, база може містити 120 функцій, із яких 30% використовують динамічний SQL, а 20% мають понад 10 викликів інших об’єктів. Представлення оцінюються за складністю їхніх джоінів та обчислень. Наприклад, із 80

представлень 40% мають понад 3 джоіни, а 25% використовують складні обчислення, такі як агрегація чи вкладені підзапити.

Програма також генерує кількісні показники продуктивності. Наприклад, середній час виконання аналітичного запиту на представлення становить 0.5 секунд, але для представлень із великою кількістю джоінів цей час може сягати 2 секунд. Це дає уявлення про необхідність оптимізації після міграції.

Прогнозується, що завдяки автоматизації витрати на міграцію знизяться до 30-40%, а продуктивність бази після перенесення зросте на 25-30% за рахунок оптимізації індексів та представлень. Програма надає всі ці технічні показники у зручній формі, дозволяючи ефективно планувати ресурси, мінімізувати ризики й забезпечити успішну міграцію.

ВИСНОВКИ І ПРОПОЗИЦІЇ

Розробка програмного забезпечення для автоматизованого оцінювання складності міграції реляційних баз даних стає все більш актуальною у зв'язку зі зростанням обсягів даних та необхідністю у точних і ефективних інструментах для планування міграцій. Існуючі інструменти, такі як AWS Database Migration Service, Azure Migrate та Fivetran, надають певні переваги в автоматизації перенесення даних, але не забезпечують достатньо детальної оцінки складності міграції для складних, кастомізованих баз даних з високими рівнями залежностей. Ці сервіси більше орієнтовані на автоматичний процес перенесення даних, а не на точне планування і оцінку складності міграцій.

Тому створення спеціалізованого програмного забезпечення для автоматизації оцінки складності міграції реляційних баз даних є надзвичайно важливим. Задоволення цієї потреби дозволить знизити витрати на міграцію, скоротити час на виконання ручних операцій та підвищити точність оцінок.

Програма буде використовувати Docker для створення ізольованого середовища, що забезпечить портативність та легкість розгортання на різних платформах.

Серверна частина буде розроблена на основі Spring Boot, що забезпечить масштабованість та зручність у підтримці.

Для зберігання метаданих баз даних буде використано PostgreSQL, що дозволить зберігати дані про структуру бази, об'єкти та їхні залежності, що в подальшому допоможе точніше оцінити складність міграції. Механізм динамічного підключення конфігурацій та SQL запитів дозволить отримати метадані з різних баз даних та створювати аналітичні представлення для детального оцінювання складності.

Програма також надасть консольний інтерфейс для зручного доступу до функціоналу, а результати будуть експортовані у форматі HTML, що полегшить прийняття рішень щодо міграції.

Для покращення програми, можна додати графічний інтерфейс користувача (GUI), що дозволить зробити взаємодію з програмою більш інтуїтивно зрозумілою і доступною для ширшого кола користувачів. Додавання REST API дозволить інтегрувати програму з іншими системами і автоматизувати процеси міграції, що стане корисним для великих організацій, де важливо мати можливість автоматично взаємодіяти з іншими інструментами та сервісами.

Таким чином, автоматизація оцінки складності міграції реляційних баз даних знизить ризик помилок, зменшить час, необхідний для оцінки складності, і підвищить точність планування, що має важливе значення для організацій, які стикаються з необхідністю міграції великих та складних баз даних.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Mike K. and Harpreet V., AWS Prescriptive Guidance: Choosing a migration tool for rehosting databases. (Електронний ресурс). Режим доступу: https://docs.aws.amazon.com/pdfs/prescriptive-guidance/latest/migration-database-rehost-tools/migration-database-rehost-tools.pdf#dms, 2022. pp 8-12. (дата звернення: 3.11.2024).
2. Azure. What is Azure Database Migration Service, 2024. (Електронний ресурс). Режим доступу: https://learn.microsoft.com/en-us/azure/migrate/migrate-services-overview (дата звернення: 3.11.2024).
3. Fivetran. The ultimate guide to data integration, 2024. pp 22-32 (Електронний ресурс). Режим доступу: https://get.fivetran.com/rs/353-UTB-444/images/the-ultimate-guide-to-data-integration.pdf. (дата звернення: 3.11.2024).
4. Oracle. What is a Relational Database (RDBMS), 2022 (Електронний ресурс). Режим доступу: https://www.oracle.com/ua/database/what-is-a-relational-database/ (дата звернення: 3.11.2024).
5. Bilal K., Saifullah J., Wahab K. and Muhammad I. C., An Overview of ETL Techniques, Tools, Processes and Evaluations in Data Warehousing, 2023. pp. 1-4. (Електронний ресурс). Режим доступу: https://file.techscience.com/files/jbd/2024/TSP_JBD-6/TSP_JBD_46223/TSP_JBD_46223.pdf (дата звернення: 3.11.2024).
6. Poonam D., JDBC FUNDAMENTALS, 2023, pp. 1-10. (Електронний ресурс). https://gcgldh.org/downloads/e-Content/Learning-Material/Computer-Science/JDBC.pdf (дата звернення: 3.11.2024).
7. EPAM, Database Migration Explained, 2021. (Електронний ресурс). Режим доступу: https://solutionshub.epam.com/blog/post/what-is-database-migration (дата звернення: 3.11.2024).

8. Chang-Yang L., Migrating to Relational Systems: Problems, Methods, and Strategies, 2008. pp. 6-9. (Електронний ресурс). Режим доступу: https://www.cmr-journal.org/article/download/1127/2278 (дата звернення: 3.11.2024).
9. IBM, Database objects, 2021. (Електронний ресурс). Режим доступу: https://www.ibm.com/docs/en/db2-warehouse?topic=database-objects (дата звернення: 3.11.2024).
10. Airbyte, Top 12 Database Migration Tools, 2024. (Електронний ресурс). Режим доступу: https://airbyte.com/top-etl-tools-for-sources/top-data-migration-tools (дата звернення: 3.11.2024).
11. Alibaba, The Future of Database Migrations: Trends to Watch, 2024. (Електронний ресурс). Режим доступу: https://www.alibabacloud.com/tech-news/a/database_migration/guh6vm5187-the-future-of-database-migrations-trends-to-watch (дата звернення: 3.11.2024).
12. Deloitte, Data is the new gold. The future of real estate service providers, 2018, pp 1-5 (Електронний ресурс). Режим доступу: https://www2.deloitte.com/content/dam/Deloitte/global/Documents/Public-Sector/gx-real-estate-data-new-gold.pdf (дата звернення: 3.11.2024).
13. Dimitri Y., Edgar Y. W. and Andreas S. T., DataJoint: A Simpler Relational Data Model, 2018. pp. 4-13. (Електронний ресурс). Режим доступу: https://arxiv.org/pdf/1807.11104. (дата звернення: 3.11.2024).
14. Jim B., Dan C., Jim D., SQL Procedures, Triggers, and Functions on IBM DB2, 2016. pp. 92-98. (Електронний ресурс). Режим доступу: https://hightouch.com/sql-dictionary/sql-stored-procedures. (дата звернення: 3.11.2024).
15. James T., The Docker Book, 2016. pp. 7-16 (Електронний ресурс). Режим доступу: http://lsi.vc.ehu.es/pablogn/docencia/manuales/The%20Docker%20Book.pdf. (дата звернення: 3.11.2024).
16. Pierre W., Maksymilian S., Novice use of the Java programming language, 2022. pp. 1-5. (Електронний ресурс). Режим доступу: https://twistedsquare.com/Novice-Java-Use.pdf (дата звернення: 3.11.2024).

17. Pratik P., Java Database Programming with JDBC, 1997, pp. 28-33. (Електронний ресурс). Режим доступу: https://www.marcobehler.com/guides/java-databases (дата звернення: 3.11.2024).

18. G Stewart I. and Mark J., On Implementing High Level Concurrency in Java, 2003, pp. 1-10. (Електронний ресурс). Режим доступу: https://jenkov.com/tutorials/java-concurrency/index.html (дата звернення: 3.11.2024).

19. Patrick P., Nick H, Hibernate Quickly, 2005, pp. 74-78. (Електронний ресурс). Режим доступу: https://dotline sistemas.com/livros/Hibernate%20Quickly%20(2005).pdf (дата звернення: 3.11.2024).

20. Kamalakant L K., Design and implementation of massive MYSQL data intelligent export system to excel by using Apache – POI libraries, 2015. (Електронний ресурс). Режим доступу: http://paper.ijcsns.org/07_book/201509/20150917.pdf (дата звернення: 3.11.2024).

21. Mohannad A., Qiben Y., Towards Best Secure Coding Practice for Implementing SSL/TLS, 2018. (Електронний ресурс). Режим доступу: https://www.wolfssl.com/use-tls-java/ (дата звернення: 3.11.2024).

22. Craig W., Spring Boot in action, 2015. pp. 9-15. (Електронний ресурс). Режим доступу: https://moldstud.com/articles/p-what-is-spring-boot-and-why-is-it-popular-among-developers (дата звернення: 3.11.2024).

23. Rod J, Juergen H., Keith D., Spring Framework Reference Documentation: Step-by-Step Guide for Developers, 2013, pp 281-282. (Електронний ресурс). Режим доступу: https://docs.spring.io/spring-framework/docs/3.2.17.RELEASE/spring-framework-reference/pdf/spring-framework-reference.pdf (дата звернення: 3.11.2024).

24. John I., Stefanos G., Diomidis S., Towards Secure Downloadable Executable Content: The JAVA Paradigm, 1998. (Електронний ресурс). Режим доступу:

https://www.icsd.aegean.gr/publication_files/125664052.pdf (дата звернення: 3.11.2024).

25. Korry S., Susan S., PostgreSQL: a comprehensive guide to building, programming, and administering PostgresSQL databases, 2005. pp. 6-10. (Електронний ресурс). Режим доступу:https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d69c4352f497d832d2b095f909c8ec2432a8ecde (дата звернення: 3.11.2024).
26. Leigh H., Unleash the power of storing JSON in Postgres, 2018. (Електронний ресурс). Режим доступу: https://personalinterests.lipingyang.org/wp-content/uploads/2017/05/Postgres-JSON_-Unleash-the-Power-of-Storing-JSON-in-Postgres.pdf (дата звернення: 3.11.2024).
27. Neil M. and Richard S., Beginning Databases with PostgreSQL From Novice to Professional, 2005. pp. 491-510. (Електронний ресурс). Режим доступу: https://tembo.io/docs/getting-started/postgres_guides/connecting-to-postgres-with-java#connecting-to-postgres-with-java-using-jdbc (дата звернення: 3.11.2024).
28. Jetbrains. IntelliJ IDEA overview, 2024. (Електронний ресурс). Режим доступу: https://www.jetbrains.com/help/idea/discover-intellij-idea.html (дата звернення: 3.11.2024).
29. Brian W., How Linux works: What every superuser should know, 2021. pp. 8-12. (Електронний ресурс). Режим доступу: https://dl.ebooksworld.ir/motoman/NSP.How.Linux.Works.What.Every.Superuser.Should.Know.2nd.Edition.www.EBooksWorld.ir.pdf (дата звернення: 3.11.2024).
30. Mendel C., Advanced Bash-Scripting Guide, 2014. pp. 3-6. (Електронний ресурс). Режим доступу: https://www.iitk.ac.in/LDP/LDP/abs/abs-guide.pdf (дата звернення: 3.11.2024).
31. James R. G and Paul N. W., SQL: The Complete Reference, 1999. pp. 8-50 (Електронний ресурс). Режим доступу: https://englishonlineclub.com/pdf/SQL%20-

%20The%20Complete%20Reference%20[EnglishOnlineClub.com].pdf (дата звернення: 3.11.2024).

32. Oracle, Coding Dynamic SQL Statements, 2002. (Електронний ресурс). Режим доступу: https://docs.oracle.com/cd/A97630_01/appdev.920/a96590/adg09dyn.htm (дата звернення: 3.11.2024).

33. PostgreSQL, PL Matrix, 2024. (Електронний ресурс). Режим доступу: https://wiki.postgresql.org/wiki/PL_Matrix (дата звернення: 3.11.2024).

34. Jolana G., Petra M., User-Defined Financial Functions for MS SQL Server, 2018 (Електронний ресурс). Режим доступу: https://pdfs.semanticscholar.org/3b3b/2b821a01e6b5c9fddc14b1473b818b03ebd2.pdf (дата звернення: 3.11.2024).

35. Закон України "Про охорону праці". (Електронний ресурс). Режим доступу: https://zakon.rada.gov.ua/laws/show/2694-12#Text (дата звернення: 3.11.2024).