Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики

РЕАЛІЗАЦІЯ ЗАСОБУ РОЗРОБКИ БАЗИ ДАНИХ PQL НА ОСНОВІ ТЕХНОЛОГІЇ PROTOBUFF

Текстова частина до курсової роботи
за спеціальністю
«Інженерія програмного забезпечення» 121

Керівник курсової роботи
ст. викладач Ющенко Ю.О.

_____

(підпис)

“____” _____ 2022 р.

Виконав студент
Чорнокозинський К. С.

“____” _____ 2022 р.

КИЇВ 2022

ЗМІСТ

АНОТАЦІЯ

У даній роботі розроблено прототип системи керування базами даних під назвою PQL (Proto Query Language). Цей сервіс призначений для взаємодії користувача з базою даних за допомогою протоколу RPC (Remote procedure call), який дозволяє програмі звертатися до функцій іншої віддалено запущеної програми.

Актуальність та доцільність цієї роботи обумовлюється обраним набором технологій, а особливо, використанням формату серіалізації даних Protocol Buffers, від якого додаток і запозичив частину назви. Використання цього протоколу дозволяє знизити затримки за рахунок уникнення зайвих трансформацій даних та пришвидшити інформаційний обмін у мережі.

Під час розробки були використані такі мови програмування та технології: Java 8, GraalVM, picocli, Redis, Docker, Gradle, gRPC, Protocol Buffers, Lettuce. Користь та переваги, у порівняні з іншими подібними аналогами, визначаються швидкістю роботи, простотою та елегантністю використання. В наступних розділах цієї роботи будуть детально описані використані технології, архітектура кінцевого проєкту і як його різні частини взаємодіють один з одним.

Ключові слова: Redis, Protobuf, gRPC, СКБД, системи керування базами данних, Java, SQL, NoSQL, PQL, адміністрування користувачів, CLI, інтерфейс командного рядка, Lettuce, Protocol Buffers, репозиторій, repository, схема, schema, сутність, entity.

ВСТУП

Система керування базами даних – це програмне забезпечення, яке приймає інформацію та перетворює її у різні види даних в єдиному контейнері для зберігання або об’єднує різноманітні дані в узгоджений ресурс, такий як база даних.

Наразі є актуальною проблема обробки надзвичайно великих об’ємів даних. Вже існує багато рішень для цього на базі реляційних баз даних, а також, так званих, NoSQL сховищ, але всі вони мають свої недоліки, тому весь час з’являються нові рішення. Під час цієї роботи я надихався ідеями таких технологій: FoundationDB (використання формату Protobuf для зберігання даних), MicroStreams (швидка серіалізація для обміну між Java клієнтами та віддаленими серверами сховищ даних). Мета даної роботи – демонстрація розробки прототипу системи керування базами даних під назвою PQL, використовуючи нестандартний набір технологій.

РОЗДІЛ 1. ОГЛЯД ВИКОРИСТАНИХ ТЕХНОЛОГІЙ

Вибір технологій для розробки проєкту є важливим підготовчим етапом, тому що вдало вибрані технології та інструменти можуть значно полегшити розробку проєкту або ж збільшити ефективність його роботи.

1.1. gRPC

gRPC – це сучасна високопродуктивна бібліотека, яка розвиває давній протокол віддаленого виклику процедур (RPC). На рівні програми gRPC оптимізує обмін повідомленнями між клієнтами та внутрішніми службами.

gRPC використовує HTTP/2 для свого транспортного протоколу. Хоча HTTP/2 сумісний з HTTP 1.1, він має багато переваг:

- протокол двійкового кадрування для транспортування даних – на відміну від HTTP 1.1, який базується на тексті.
- підтримка мультиплексування для надсилання кількох паралельних запитів через одне з’єднання , коли HTTP 1.1 обмежує обробку одним повідомленням запиту/відповіді за раз.

- двонаправлений повнодуплексний зв’язок для одночасної відправки як клієнтських запитів, так і відповідей сервера.
- вбудована потокова передача, що дозволяє запитам і відповідям асинхронно передавати великі набори даних.
- стиснення заголовків, що зменшує використання мережі.

gRPC легкий і високопродуктивний. Він може бути до 8 разів швидше, ніж серіалізація JSON з повідомленнями, які будуть значно меншими.

1.2. Protobuf Protocol

Protobuf Protocol забезпечують розширюваний механізм, який є нейтральним до мови та платформи, для серіалізації структурованих даних, сумісним із прямим і зворотним шляхом.

Protobuf Protocol – це комбінація мови визначення (яка використовується у файлах *.proto*), коду, який генерує компілятор protoc для взаємодії з даними, специфічних для мови бібліотек часу виконання та формату серіалізації для даних, які записуються у файл (або надсилаються). через мережеве підключення).

Буфери протоколу ідеально підходять для ситуації, коли потрібно серіалізувати структуровані, типізовані дані в розширюваний спосіб, нейтральний для мови та платформи. Найчастіше вони використовуються для визначення протоколів зв’язку (разом з gRPC) і для зберігання даних. Деякі з переваг використання буферів протоколу включають:

- компактне сховище даних;
- швидкий розбір;
- доступність на багатьох мовах програмування;
- оптимізована функціональність завдяки автоматично створеним класам.

1.3. GraalVM NativeImage

GraalVM – це високопродуктивний дистрибутив JDK, розроблений для прискорення виконання програм, написаних на Java та інших мовах JVM, а також надає середовище виконання для JavaScript, Python та низки інших популярних мов. Поліглотні можливості GraalVM дозволяють змішувати кілька мов програмування в одній програмі.

Native Image – це технологія для завчасної компіляції коду Java в автономний виконуваний файл, який називається нативним образом. Цей виконуваний файл включає класи програми, класи з його залежностей, класи бібліотеки часу виконання та статично пов’язаний рідний код з JDK. Він не працює на Java VM, але включає необхідні компоненти, такі як керування пам’яттю, планування потоків тощо з іншої системи виконання, яка називається «Substrate VM». Substrate VM – це назва компонентів середовища виконання (наприклад, деоптимізер, збирач сміття, планування потоків тощо). Отримана програма має швидший час запуску та менші витрати на пам’ять під час виконання порівняно з JVM.

1.4. Picocli

Picocli – це однофайлова платформа для створення програм командного рядка Java з майже нульовим кодом. Він підтримує різноманітні стилі синтаксису командного рядка, включаючи POSIX, GNU, MS-DOS тощо. Він генерує легко настроюванні довідкові повідомлення, які використовують кольори та стилі ANSI.

Програми на основі Picocli можуть завершувати команди командного рядка при нажаті клавіши TAB, показувати доступні параметри, параметри параметрів і підкоманди для будь-якого рівня вкладених підкоманд. Програми на основі Picocli можуть бути завчасно скомпільовані до нативного образу GraalVM з надзвичайно швидким часом запуску та меншими вимогами до пам’яті.

1.5. Gradle

Gradle – це система автоматизації збірки, яка повністю відкрита і побудована на концепціях подібних до тих, що використані в Apache Maven і Apache Ant. Gradle використовує предметну мову (англ. Domain Language) на основі мови програмування Groovy або Kotlin, що відрізняє його від Apache Maven, який використовує XML для конфігурації проєкту. Він також визначає порядок виконання завдань за допомогою орієнтованого ациклічного графа.

Gradle дозволяє інкрементні збірки, оскільки він перевіряє, які завдання оновлені чи ні. Якщо це так, то завдання не виконується, що дасть вам набагато менший час побудови. Інші відмінні характеристики продуктивності, які ви можете знайти на Gradle, включають:

- інкрементні компіляції для класів Java;
- уникнення компіляції для Java;
- використання API для інкрементальних підзадач;
- демон компілятора, який також робить компіляцію набагато швидшою.

1.6. Redis

Redis (розшифровується як Remote Dictionary Server) – це швидке сховище даних типу «ключ-значення» у пам’яті з відкритим вихідним кодом.

Redis забезпечує час відгуку на рівні часток мілісекунди і дозволяє програмам, що працюють у режимі реального часу, виконувати мільйони запитів на секунду. Такі програми потрібні у сферах ігор, рекламних технологій, фінансових сервісів, охорони здоров’я та IoT. Сьогодні Redis – одне з найпопулярніших сховищ з відкритим вихідним кодом.

Висновки

В цьому розділі була приділена особлива увага вибору інструментів розробки. Кожній вибраній технології для цієї роботи був даний короткий опис та перелік переваг над іншими засобами розробки.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА РОЗРОБКА

Перш за все, головною задачею було визначення функціоналу системи керування базами даних PQL. Тому був розроблений орієнтовний прикладний програмний інтерфейс, який визначає операції отримання схеми поточного репозиторію, збереження, додавання, оновлення, видалення та зчитування сутностей, записаний псевдокодом, схожим на мову protobuf.

```
service pql {
  getSchema() returns Schema;
  save(Entity entity) returns Entity;
  save(Entity[] entities) returns Entity;
  update(Entity entity) returns Entity;
  update(Entity[] entities) returns Entity[];
  saveOrUpdate(Entity entity) returns Entity;
  saveOrUpdate(Entity[] entities) returns Entity[];
  delete(Entity entity);
  delete(Entity[] entities);
  findAll(String tableName) returns Entity[];
}
```

Лістинг 2.1. Визначення приблизного інтерфейсу PQL (псевдокод)

Архітектура – це організація системи, втілена в її компонентах, їх відносинах між собою та з оточенням. Система – це набір компонентів, які об’єднані для виконання певної функції.

Наступним кроком систему PQL було вирішено поділити на 3 таких модулі:

- pql-external-api;
- pql-internal-api;
- pql-admin-cli.

pql-external-api

pql-external-api є обгорткою компонента pql-internal-api, задача якого описана нижче, та відповідає за надання певній особі або групі осіб право на

виконання певних дій; а також процес перевірки (підтвердження) даних прав під час спроби виконання цих дій. В основному, перевірка на наявність доступу користувача до репозиторіїв, представленням яких є різні працюючі екземпляри сервісу pql-internal-api.

pql-internal-api

pql-internal-api має єдину мету в житті – користуватися наданим репозиторієм в Redis: записувати, видаляти, оновлювати та зчитувати дані. Репозиторій в PQL представляє різновид облікового запису, дозвіл до якого дається користувачам. Кожний з них містить усі схеми, створені конкретним користувачем системи PQL. В свою чергу схема представляє набір таблиць, що описуються за допомогою мови protobuf.

pql-admin-cli

pql-admin-cli – це застосунок, що відповідає за адміністрування баз даних. Адміністратор, використовуючи інтерфейс командного рядка, може створювати аккаунти, давати їм права, додавати таблиці та схеми, та подібні операції, що будуть описані у підрозділі присвяченому модулю pql-admin-cli.

Під час проєктування системи було вирішено використати такі архітектурні шаблони проєктування: Посередник (англ. Mediator), Головний-Підпорядкований (англ. Master-Slave) та Замісник (англ. Proxy).

Посередник – шаблон проєктування, що застосовується, коли між модулями є залежність «багато до багатьох». Медіатор виступає як посередник у спілкуванні між модулями, діючи як центр зв’язку і позбавляє модулі необхідності явно посилатися один на одного. Тому зв’язки модулів «багато до багатьох» замінюється зв’язком «один до багатьох», де «один» – це модуль виконуючий роль посередника. В системі PQL компонентом-посередником є pql-external-api, який інкапсулює роботу кліенту з екземплярами модуля pql-internal-api, як було зазначено вище.

Також модуль pql-external-api було вирішено зробити Замісником, що обертає компонент pql-internal-api. Сурогат або Замісник, в даному контексті, це сервіс, інтерфейс якого ідентичний інтерфейсу іншого сервісу, який він обертає. При першому запиті клієнта заступник створює реальний об'єкт, зберігає його адресу і відправляє запит цьому об'єкту. Всі наступні запити просто переадресовуються інкапсульованому реальному сервісу.

Архітектурний шаблон проєктування Головний-Підпорядкований використовується для підвищення надійності та продуктивності системи шляхом розподілу роботи між головним і підлеглим компонентами. Кожен компонент має окремі обов'язки. Усі підпорядковані компоненти мають ідентичну або принаймні подібну роботу, і ця робота повинна бути визначена перед виконанням. Цей шаблон не є підходом «розділяй і володарюй» до архітектури; скоріше, це такий, де робота підпорядкованих компонентів заздалегідь визначена і повинна бути скоординована. В системі PQL pql-admin-cli є майстром, а екземпляри сервісу pql-internal-api його підлеглими. pql-admin-cli керує життєвий цикл внутрішнього сервісу та визначає з яким репозиторієм він працює. Нижче схематично продемонстровано взаємодію компонентів у системі PQL.

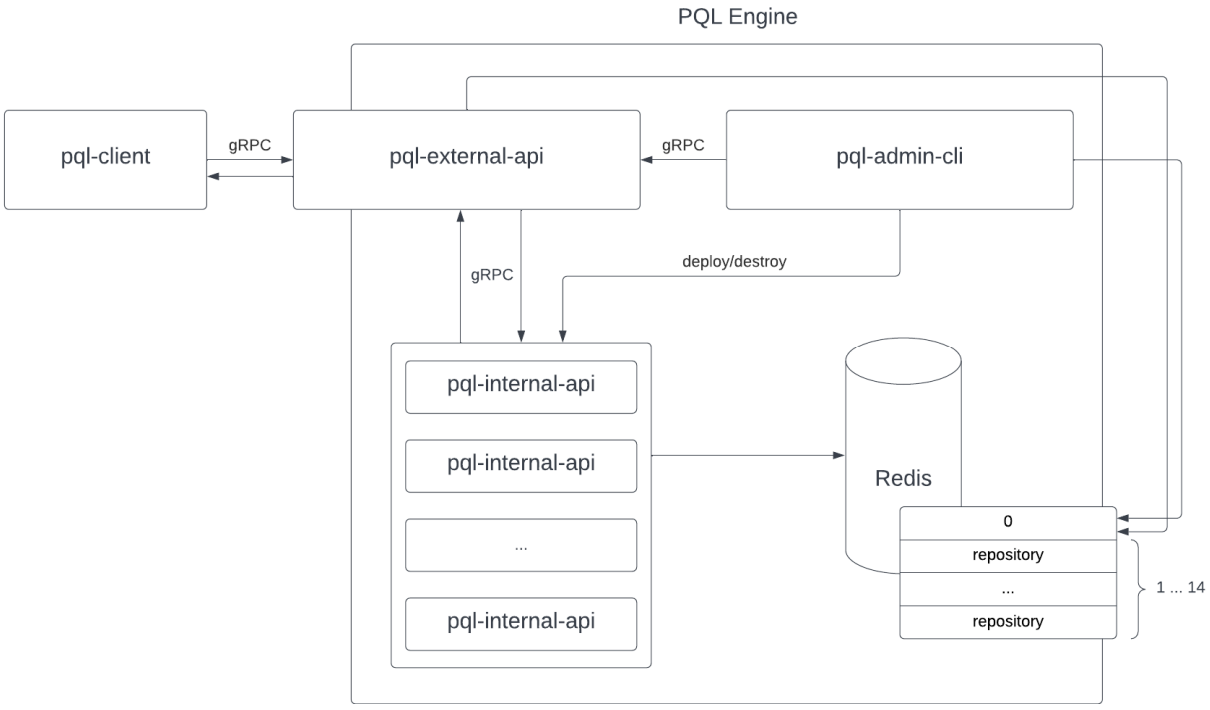


Рисунок 2.1.1. Архітектура системи PQL

У подальших підрозділах 2.1–2.3 детально описано розробку та проєктування архітектури основних компонентів.

2.1. Розробка та проєктування компоненту pql-internal-api

Початком розробки будь-якого gRPC сервісу є визначення protobuf специфікації (задаються у файлах розширення .proto), за якою будуть генеруватися класи та інтерфейси, які надалі будуть імплементовані.

```
service InternalService {
  rpc getSchema(google.protobuf.Empty) returns (GetSchemaResponse);
  rpc save(SaveUpdateDeleteRequest) returns (SaveUpdateResponse);
  rpc update(SaveUpdateDeleteRequest) returns (SaveUpdateResponse);
  rpc saveOrUpdate(SaveUpdateDeleteRequest) returns (SaveUpdateResponse);
  rpc delete(SaveUpdateDeleteRequest) returns (google.protobuf.Empty);
  rpc findAll(FindAllRequest) returns (FindAllResponse);
}
```

Лістинг 2.1.1. Визначення прикладного програмного інтерфейсу PQL

Далі було було визначено типи, якими цей прикладний програмний інтерфейс буде оперувати. В роботі продемонстровано лише головні з них, які пронизують усю систему,: Schema, Repository та Account. Тип Schema містить два поля: просте ім’я файлу, в якому міститься визначення таблиць, та зміст цього файлу – саме визначення таблиць.

```
message Schema {
  string fileName = 1;
  string content = 2;
}
```

Лістинг 2.1.2. Визначення типу Schema

Наступний тип – це репозиторій. Він містить три поля: його ім’я, номер бази даних в Redis, яку цей репозиторій представляє, та ідентифікатор процессу в якому репозиторій був запущений.

```
message Repository {
  string name = 1;
  int32 redisDatabaseNumber = 2;
  int64 pid = 3;
}
```

Лістинг 2.1.3. Визначення типу Repository

Останній – Account, який є відображенням користувача в системі PQL. Цей тип має такі поля: логін користувача, кешований пароль, флаг, який вказує на те чи є аккаунт тимчасово призупинений, та список імен репозиторіїв, до яких користувач має доступ.

```
message Account {
  string login = 1;
  string cachedPassword = 2;
  bool isSuspended = 3;
  repeated string repositories = 4;
}
```

Лістинг 2.1.4. Визначення типу Account

Складені типи, що, власне, використовуються у визначенні специфікації сервісу, описують параметри запиту згідно з специфікацією вказаної у попередньому розділі. Наприклад, SaveUpdateDeleteRequest описує тип, який містить два поля: перший – це назва таблиці (за сумісництвом назва класу), а другий – сереалізований об’єкт або список об’єктів, що представляють сутність або список сутностей.

Наступною та певно визначальною задачею було вирішити як і у якому вигляді зберігати дані у Redis. Сутності було вирішено зберігати у сереалізованому вигляді як значення, а ключі це їх дещо видозмінений ідентифікатор UUID (розшифровується як універсальний унікальний ідентифікатор). Вигляд ключу, що відповідатиме сутності User, буде таким: *{ім’я класу}:entities:{uuid}*, наприклад, *«user:entities:87ae9041-ec1c-4725-91e5-6061a62446a8»*.

Наступною задачею було розробити архітектуру даного модулю. Він поділений на такі частини:

- service;
- common;
- exception.

В пакеті common визначені допоміжні функції, наприклад, утиліти для логування. В exception – описані типові виключення, які можуть бути викинуті під час роботи модуля. Найголовнішою частиною є пакет service, в ньому знаходиться імплементація інтерфейсу, який був вказаний вище.

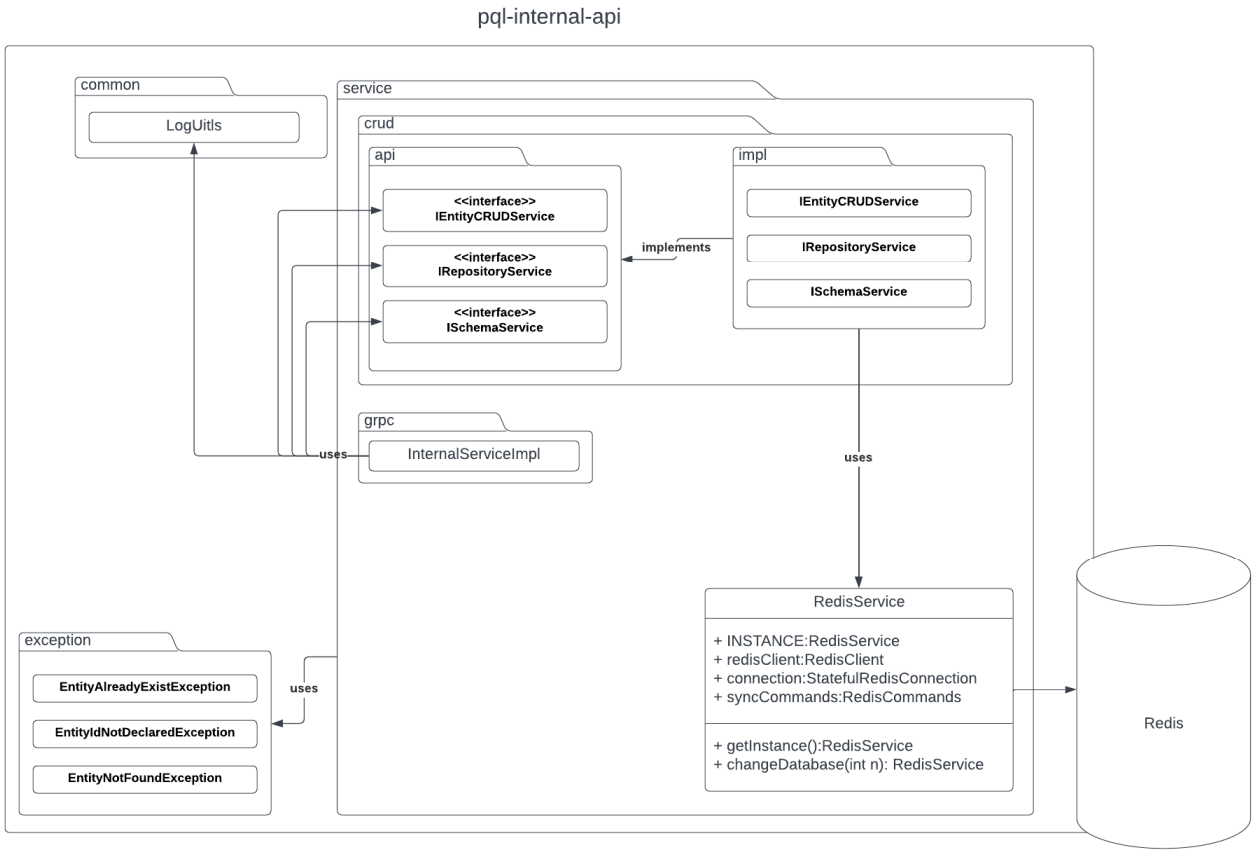


Рисунок 2.1.1. Архітектура pql-internal-api

Найоб'ємнішим пакетом є сервіс, вважаю доречним розібрати його детальніше. Першим кроком був визначений сервіс RedisService, який одночасно є реалізацією паттерну одинака (англ. Singletone). Він створює з'єднання з базою данних Redis та дозволяє виконувати команди і операції над даними.

Його використовують усі реалізації інтерфейсів IEntityCRUDService, IRepositoryService та ISchemaService.

2.1.1. Інтерфейс IEntityCRUDService та його реалізація

Метою інтерфейсу IEntityCRUDService, який знаходиться у пакеті service.crud.api, є виконання операцій на сутностями, таких як: збереження, видалення, зчитування і тому подібних. Нижче наведений рисунок, що може дати точне уявлення про інтерфейс.

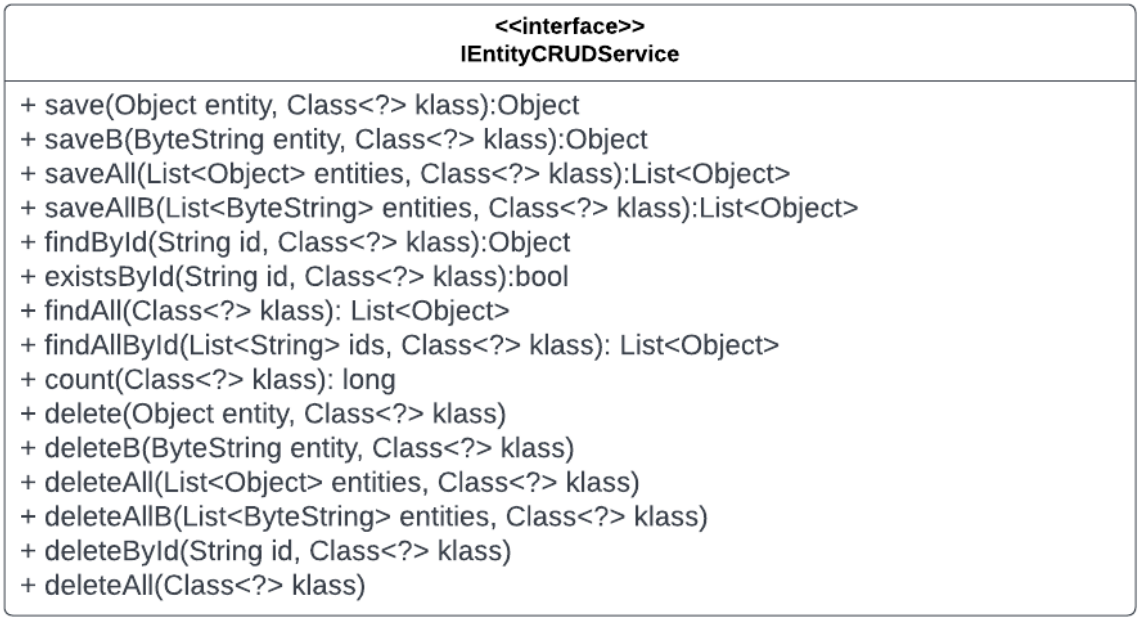


Рисунок 2.1.1.1. Інтерфейс IEntityCRUDService

Усі методи, що мають суфікс «B», приймають сутності у вигляді списку байтів, а якщо точніше, то у його обертці – ByteString. Параметр klass вказує таблицю з якою працює користувач (метод klass.getSimpleName() поверне назву таблиці, яка буде використовуватися при взаємодії з Redis). Коротке пояснення усіх методів:

- save(B), saveAll(B) – зберігає передані параметрами сутності або сутність;
- findById, findAllById – повертає сутність або список сутностей з переданими як параметр ідентифікатором або ідентифікаторами;

- existsById – перевіряє чи існує сутність з переданим як параметр ідентифікатором;
- count – повертає кількість сутносей у певній таблиці;
- delete(B), deleteAll(B) – видаляє передані параметрами сутності або сутність.

Реалізація цього інтерфейсу знаходиться у класі EntityCrudServiceImpl, який знаходиться у пакеті service.crud.impl. Спершу, важливо зазначити, що реалізація теж використовує шаблон проєктування одинака. Для імплементації було використано бібліотеку Java Reflection. Рефлексія – це особливість мови програмування Java, вона дозволяє програмі під час виконання (англ. run-time) самоаналізувати та маніпулювати внутрішніми властивостями. Нижче наведена імплементація одного з методів інерфейсу IEntityCRUDService в класі EntityCRUDServiceImpl.

```
@Override
public Object save(Object entity, Class<?> klass){

    if (getEntityUuid(entity, klass).isEmpty()) {
        entity = generateEntityUuidForEntity(entity, klass);
    }

    String value = getEntityUuid(entity, klass);

    RedisService
       .getInstance()
       .getSyncCommands()
       .set(getEntityPrefix(klass) + value,
       toBytes(entity).toByteArray());

    return entity;

}

public static String getEntityUuid(Object entity, Class<?> klass){
    Field field = klass.getDeclaredField(UUID_FIELD_NAME);
    field.setAccessible(true);
    String value = String.valueOf(field.get(entity));
    return value;
}

public static Object generateEntityUuidForEntity(Object entity,
Class<?> klass){

    Object newEntity = copy(entity, klass);

    Field field = klass.getDeclaredField(UUID_FIELD_NAME);
    field.setAccessible(true);
    String uuid = String.valueOf(UUID.randomUUID());
    field.set(newEntity, uuid);

    return newEntity;

}
```

Лістинг 2.1.1.1. Імплементація методу save

На лістингу 2.1.1.1 окрім імплементованого методу save є ще два допоміжних статичних методи getEntityUuid та generateEntityUuidForEntity, які написані завдяки рефлексії. Вони під час виконання програми змінюють визначення класів, а саме змінюють область видимості змінних. Ці методи повертають існуючий ідентифікатор сутності та генерують новий відповідно. Також можна помітити використання одинака RedisService, згаданий раніше, для того щоб наприкінці зберегти нову сутність до Redis. Таким чином перевизначені усі інші методи інтерфейсу.

Класи, що передаються параметром у всіх методах, визначають структуру таблиць. Ці класи генеруються з файлів розширення .proto, де і описуються у таблиці у вигляді повідомлень, з директорії з таким шляхом: *«/proto/entity»*. Це відбувається у результаті збирання проєкту за допомогою системи для автоматизації збирання застосунків Gradle.

2.1.2. Інтерфейс IRepositoryService та його реалізація

У сервіса RepositoryService є лише одна мета – отримати повну інформацію про репозиторій по його імені. Тому його інтерфейс дуже простий.

Рисунок 2.1.2.1. Інтерфейс IRepositoryService

Реалізація цього інтерфейсу знаходиться у класі RepositoryServiceImpl, який знаходиться у пакеті service.crud.impl. Як і попередній сервіс, ця імплементація теж реалізує шаблон проєктування одинака.

В базі даних під номером 0 у Redis знаходяться усі налаштування системи PQL, в тому числі і репозиторії. Вони зберігаються за ключами такого виду: *repositories:{ім'я репозиторію}* (наприклад, ідентифікатор репозиторію з іменем *«repo_name»* буде таким: *«repositories:repo_name»*). Нижче продемонстрований код імплементації.

```
@Override
public Repository getRepoByName(String name){

    RedisService.getInstance().ChangeDatabase(ADMIN_DB);
    String key = REPOSITORIES_PREFIX + name;
    Object repositoryResult = RedisService.getInstance()
       .getSyncCommands().get(key);
    RedisService.getInstance().setToDefault();

    return Repository.parseFrom((byte[]) repositoryResult);

} // де REPOSITORIES_PREFIX = "repositories:"
```

Лістинг 2.1.2.2. Імплементація методу getRepoByName

2.1.3. Інтерфейс ISchemaService та його реалізація

У сервіса SchemaService як і у RepositoryService є лише одна мета – отримати усі схеми та інформацію про них у певному репозиторії.

Рисунок 2.1.3.1. Інтерфейс ISchemaService

Реалізація цього інтерфейсу знаходиться у класі SchemaServiceImpl, який знаходиться у пакеті service.crud.impl. Імплементація сервісу теж реалізує шаблон проєктування одинака.

Схеми містяться у базі даних Redis під номер 0. Вони зберігаються за ключами такого виду: *{ім'я репозиторію}:schema:{ім'я файлу, у якому схема визначається}* (Наприклад, ідентифікатор схеми, визначеної у файлі *user.proto*, у репозиторії з іменем *«repo_name»* буде таким: *«repo_name:schema:user.proto»*). Нижче продемонстрований код імплементації.

```
@Override
public List<Schema> getSchemaByRepo(String repositoryName){

    RedisService.getInstance().ChangeDatabase(ADMIN_DB);
    String pattern = repositoryName + SCHEMA_PREFIX + "*";
    List<String> keys = RedisService
       .getInstance()
       .getSyncCommands()
       .keys(pattern);
    List<Schema> schemas = new ArrayList<>();
    for (String key : keys)
        schemas.add(Schema.parseFrom((byte[]) RedisService.get-
Instance().getSyncCommands().get(key)));

    RedisService.getInstance().setToDefault();
    return schemas;

}// де SCHEMA_PREFIX = ":schema:"
```

Лістинг 2.1.3.2. Імплементація методу getSchemaByRepo

2.2. Розробка та проєктування компоненту pql-external-api

Аналогічно до модуля pql-internal-api, розробку розпочато з визначення Protobuf специфікації, але цей компонент буде мати дві.

```
service ExternalService {
  rpc getSchema(google.protobuf.Empty) returns (GetSchemaResponse);
  rpc save(SaveUpdateDeleteRequest) returns (SaveUpdateResponse);
  rpc update(SaveUpdateDeleteRequest) returns (SaveUpdateResponse);
  rpc saveOrUpdate(SaveUpdateDeleteRequest) returns (SaveUpdateResponse);
  rpc delete(SaveUpdateDeleteRequest) returns (google.protobuf.Empty);
  rpc findAll(FindAllRequest) returns (FindAllResponse);
}
```

Лістинг 2.2.1. Визначення специфікації сервісу ExternalService

Визначення першого сервісу дуже схоже на те, що було у розділі про розробку та проєктуваня компоненту pql-internal-api. Насправді, специфікації цих модулів майже ідентичні, окрім одніє речі. До кожного параметра цих операцій додалося одне поле, а саме – ім’я репозиторію, з яким буде працювати користувач.

Друга специфікація, що описує лише одну операцію (про неї детальніше у розділі про компонент pql-admin-cli). Вона виглядає так:

```
service RestartConnectionService {
  rpc restartConnection(Repository) returns (google.protobuf.Empty);
}
```

Лістинг 2.2.2. Визначення специфікації сервісу RestartConnectionService

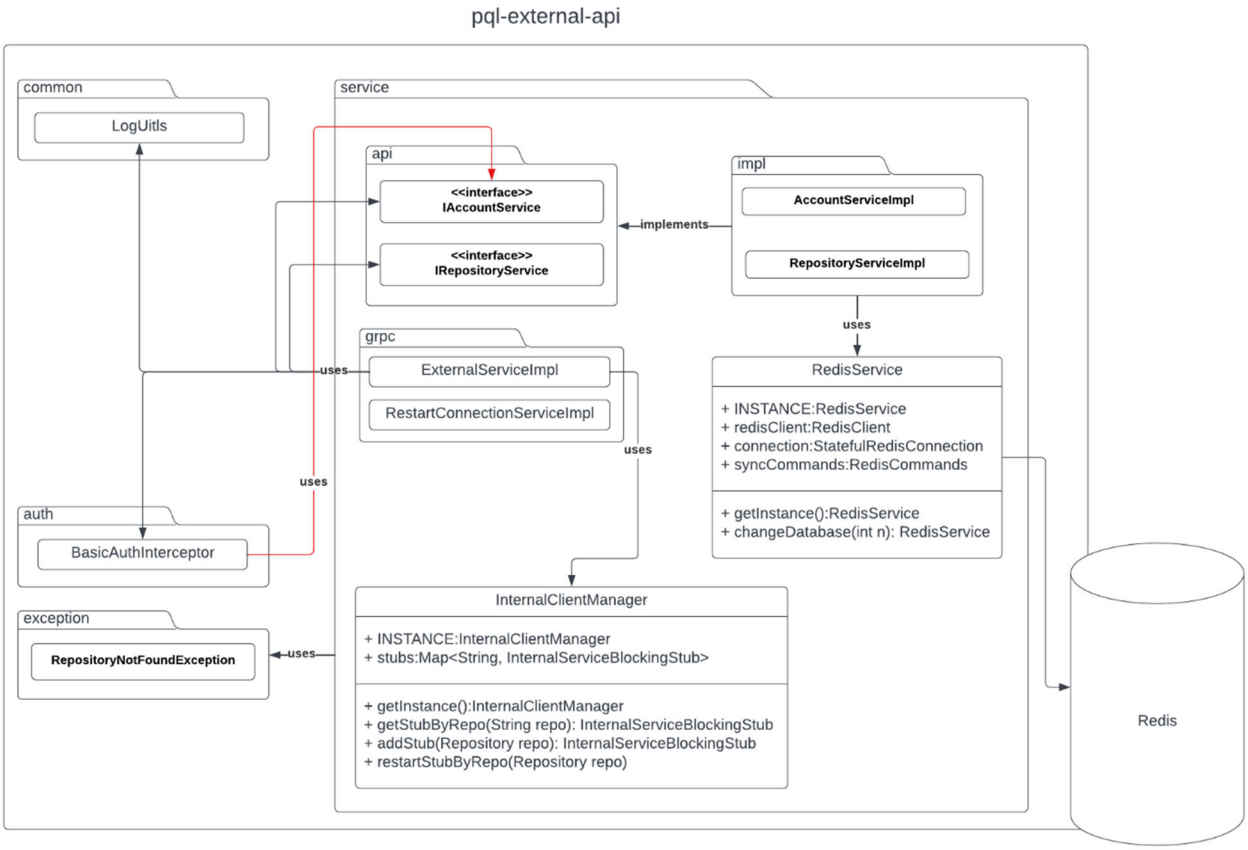
Задача цього модуля – інкапсулювати логіку реалізовану у внутрішньому компоненту від користувача та забезпечити авторизацію.

Архітектура має досить багато спільного з тою, що було продемонстровано в попередньому підрозділі.

Він поділений на такі частини:

- service;
- common;
- exception;
- config;
- auth.

Цього разу додалося два підмодуля: config та auth. Схематично архітектура виглядає так:



Лістинг 2.1.4. Архітектура pql-external-api

2.2.1. Аутентифікація

Наразі була використана базова аутентифікація. Базова аутентифікація HTTP – це простий механізм виклику та відповіді, за допомогою якого сервер може запитувати інформацію для аутентифікації (ідентифікатор користувача та пароль) від клієнта. Клієнт передає інформацію автентифікації на сервер у HTTP заголовку авторизації. Інформація для аутентифікації використовує кодування base64.

Реалізація аутентифікації знаходиться у пакеті auth у классі BasicAuthInterceptor. Якщо користувачу вдалося аутентифікуватися, то контекст системи запам'ятовує його логін, якому відповідає набір прав, що має користувач. Нижче наведена імплементація валідації користувача:

```
private boolean validUser(String basicAuthString) {
    if (basicAuthString == null) {
        return false;
    }
    String token = basicAuthString.substring("Basic
".length()).trim();
    String auth = new String(Base64.getDecoder().decode(token.get-
Bytes()));
    Account account = null;
    try {
        account = AccountServiceImpl.getInstance().getAc-
countByLogin(getLogin(basicAuthString));
    } catch (InvalidProtocolBufferException e) {
        e.printStackTrace();
    }
    if (auth == null)
        return false;

    return BCrypt.checkpw(auth, account.getCachedPassword());
}
```

Лістинг 2.2.1.1. Імплементація методу validUser

В останній стрічці программа хешує пароль та перевіряє на співпадіння з вже існуючим у базі даних. Якщо співпадає, то поверає істину, а якщо ні – хибність.

2.2.2. Інтерфейс IAccountService та його реалізація

У сервіса AccountService є лише одна мета – отримати всю інформацію про обліковий запис по його логіну.

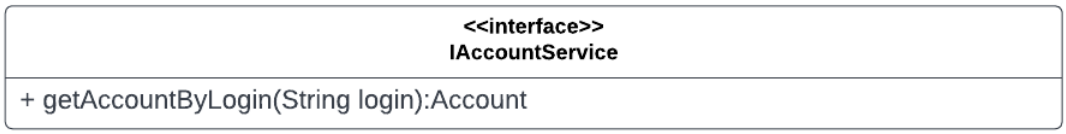


Рисунок 2.2.2.1. Інтерфейс IAccountService

Реалізація цього інтерфейсу знаходиться у класі AccountServiceImpl, який знаходиться у пакеті service.impl. Імплементація сервісу теж реалізує шаблон проєктування одинака.

Облікові записи містяться у базі даних Redis під номером 0. Вони зберігаються за ключами такого виду: *accounts:{логін}* (наприклад, ідентифікатор облікового запису з логіном *«login»* буде таким: *«accounts:login»*). Нижче продемонстрований код імплементації.

```
@Override
public Account getAccountByLogin(String login){
    String key = AppConfig
       .INSTANCE.getConfig()
       .getString("redis.accountsPrefix") + login;
    if (RedisService.getInstance().getSyncCommands().exists(key) == 0)
        return null;
    Object accountResult = RedisService
       .getInstance()
       .getSyncCommands()
       .get(key);
    return Account.parseFrom((byte[]) accountResult);
}
```

Лістинг 2.2.2.1. Імплементація методу getAccountByLogin

2.2.3. Клас InternalClientManager

Клас одинака InternalClientManager (знаходиться у пакеті service) відповідальний за керування підключеннями до сервісів pql-internal-api. Він має поле словник з ключом, який відповідає назві репозиторію, та значенням типу InternalServiceBlockingStub. Заглушка (англ. Stub) – це згенерований клас за protobuf специфікацією для використання клієнтами gRPC для виклику методів, визначених сервісом. Кожна заглушка є обгорткою каналу (англ. Channel), який використовує для надсилання віддалених викликів процедур (англ. RPC) до сервісу. Схематичне представлення класу наведено нижче:

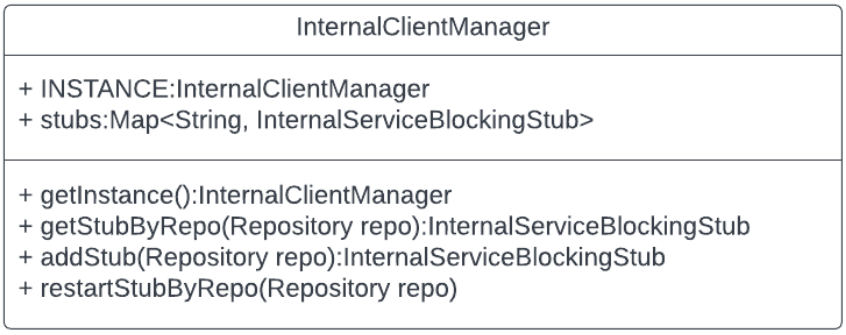


Рисунок 2.2.3.1. Клас InternalClientManager

Коротке пояснення усіх методів:

- getInstance – повертає екземпляр класу, якщо його немає, то створить новий і тільки потім поверне його;
- getStubByRepo – повертає ту заглушку, що відповідає за контроль заданим репозиторієм;
- addStub – додає у словник новостворену заглушку, що буде відповідати за репозиторій переданий параметром;
- restartStubByRepo – перезапускає заглушку, якає відповідає даному репозиторію.

Таким чином, при зверненні користувача до pql-external-api, викликаючи певну процедуру, клієнт отримає можливість спілкуватися з певним екземпляром компонента pql-internal-api в залежності від переданого ним імені репозиторія.

2.3. Розробка та проєктування компоненту pql-admin-cli

Модуль pql-admin-cli дозволяє користувачу адмініструвати систему PQL за допомогою інтерфейсу командного рядка. Було вирішено додати до модуля такий функціонал:

- для роботи з обліковим записом:
  - реєстрація;
  - призупинення роботи облікового запису та його відновлення;
  - видалення;
  - виведення усіх або певних облікових записів;
  - надання прав обліковому запису та їх конфіскація;

- для роботи з репозиторієм:
  - створення;
  - видалення;
  - запуск да зупинення;
  - виведення інформації;
- для роботи з схемою:
  - створення.

Програму було розбито на частини як вказано на схемі нижче:

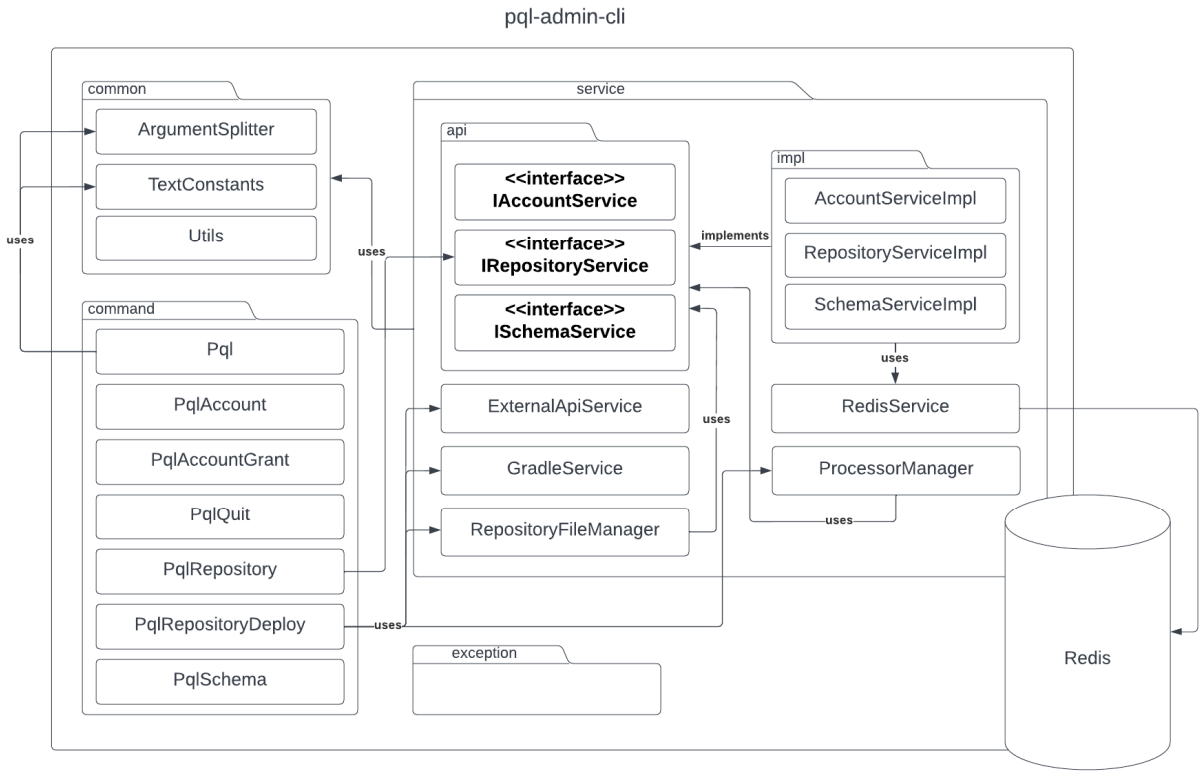


Рисунок 2.3.1. Архітектура pql-admin-cli

Інтерфейси IAccountService, IRepostitoryService та ISchemaService були оглянуті у попередніх підрозділах. Тут вони залишилися тими самими, але додалося більше функціоналу до кожного з інтерфейсу, таких як видалення, зберігання і тому подібне.

RedisService не потребує змін. Його так само використовують усі сервіси, що взаємодіють з Redis.

2.3.1. Сервіси GradleService, ExternalApiService, RepositoryFileManager, ProcessManager.

Ці сервіси потрібні для збірки, запуску та підключення екземпляру *pql-internal-api* відповідно до певного репозиторію.

Сервіс RepositoryFileManager відповідальний за розархівування файлу *pql-internal-server.zip*, який знаходиться у ресурсах проєкту, у папку *PQL_HOME*, яка визначається адміністратором, та зчитування усіх схем з заданого репозиторію, та записування їх у вигляді прото файлів у директорію *PQL_HOME/{repository_name}/proto/entity*. Його схема виглядає так:

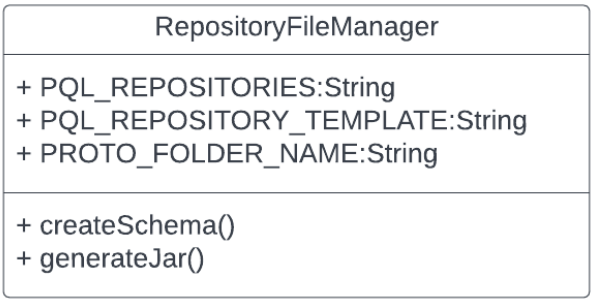


Рисунок 2.3.1.1. Клас RepositoryFileManager

Наступним кроком GradleService збирає новостворений проєкт та створює його Graal нативним образом. При його розробці було використано прикладний програмний інтерфейс gradle-tooling-api, який дозволяє підключатися до проєктів, що мають файл gradle.build, та виконувати визначені задачі, в тому числі збірку проєкту. Його схема виглядає так:

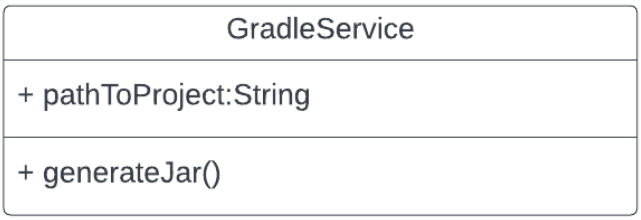


Рисунок 2.3.1.2. Клас GradleService

ProcessManager запускає створений GradleService нативний образ з певними параметрами, які визначаються репозиторієм, для якого цей застосунок був створений.

І останнім кроком ExternalApiService віддалено викликає процедуру з сервісу pql-external-api RestartConnectionService, який перестворює заглушку до pql-internal-api. Як було вказано у розділі про компонент pql-external-api, RestartConnectionService має лише одну процедуру – restartConnection.

Бізнес логіка, описана вище, викликається командою: «pql repo deploy repo_name».

2.3.2. Імплементація команд

Для реалізації інтерфейсу командного рядка була використана бібліотека picocli, яка дозволяє створювати прості інтерфейси командного рядка. Для більшого розуміння нижче продемонстрована імплементація однієї з команд, а саме *«pql schema»*:

```
CommandLine.Command(name = "schema", header = "Add schemas, list sche-
mas", subcommands = {
        CommandLine.HelpCommand.class
})
public class PqlSchema implements Callable<Integer> {

    ISchemaService schemaService = new SchemaServiceImpl();

    @CommandLine.Parameters(index = "0", arity = "1")
    String repoName;

    @CommandLine.Parameters(index = "1", arity = "1")
    String filePath;

    @Override
    public Integer call() throws Exception {
        Schema schema = Schema
                .newBuilder()
                .setContent(getFileContent(filePath))
                .setFileName(getFileName(filePath))
                .build();
        schemaService.uploadSchema(schema, repoName);
        clean();
        return 0;
    }

    private void clean() {
        repoName = null;
        filePath = null;
    }
}
```

Лістинг 2.3.2.1. Імплементація команди «pql schema»

Висновки

В розділі був описаний процес створення архітектури системи PQL, її розбиття на компоненти та детальний опис їх розробки та залежностей один від одного за допомогою схем.

РОЗДІЛ 3. ІНСТРУКЦІЯ КОРИСТУВАННЯ СИСТЕМОЮ

3.1. Інструкція для користувача

Спершу користувач повинен додати необхідні залежності до проєкту. У випадку якщо ви працюєте з Apache Maven, то потрібно внести зміни до файлу pom.xml. В цьому розділі буде наведено приклад роботи з системою автоматизації збірки Gradle. Додайте код наведений нижче у файл *build.gradle*.

```
apply plugin: 'java'
apply plugin: 'com.google.protobuf'
repositories {
    mavenCentral()
}

def grpcVersion = '1.44.0'

dependencies {
    compile "io.grpc:grpc-netty:${grpcVersion}"
    compile "io.grpc:grpc-protobuf:${grpcVersion}"
    compile "io.grpc:grpc-stub:${grpcVersion}"
    implementation 'javax.annotation:javax.annotation-api:1.3.2'
    compile 'com.google.protobuf:protobuf-java:3.9.2'
}
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'com.google.protobuf:protobuf-gradle-plugin:0.8.10'
    }
}
protobuf {
    protoc {
        artifact = 'com.google.protobuf:protoc:3.9.2'
    }
    plugins {
        grpc {
            artifact = "io.grpc:protoc-gen-grpc-java:${grpcVersion}"
        }
    }
    generateProtoTasks {
        all()*.plugins {
            grpc {}
        }
    }
}
sourceSets {
    main {
        java {
            srcDirs 'build/generated/source/proto/main/grpc'
            srcDirs 'build/generated/source/proto/main/java'
        }
    }
}
```

Лістинг 3.1. Зміст файлу build.gradle

Наступним кроком буде створення директорії *{ім'я проєкту}/src/main/proto*, а потім створення ще двух в ній: entity і service. У папку *service* треба розархівувати файл *pql-setup-proto.zip*. Після цього проєкт повинен виглядати приблизно так:

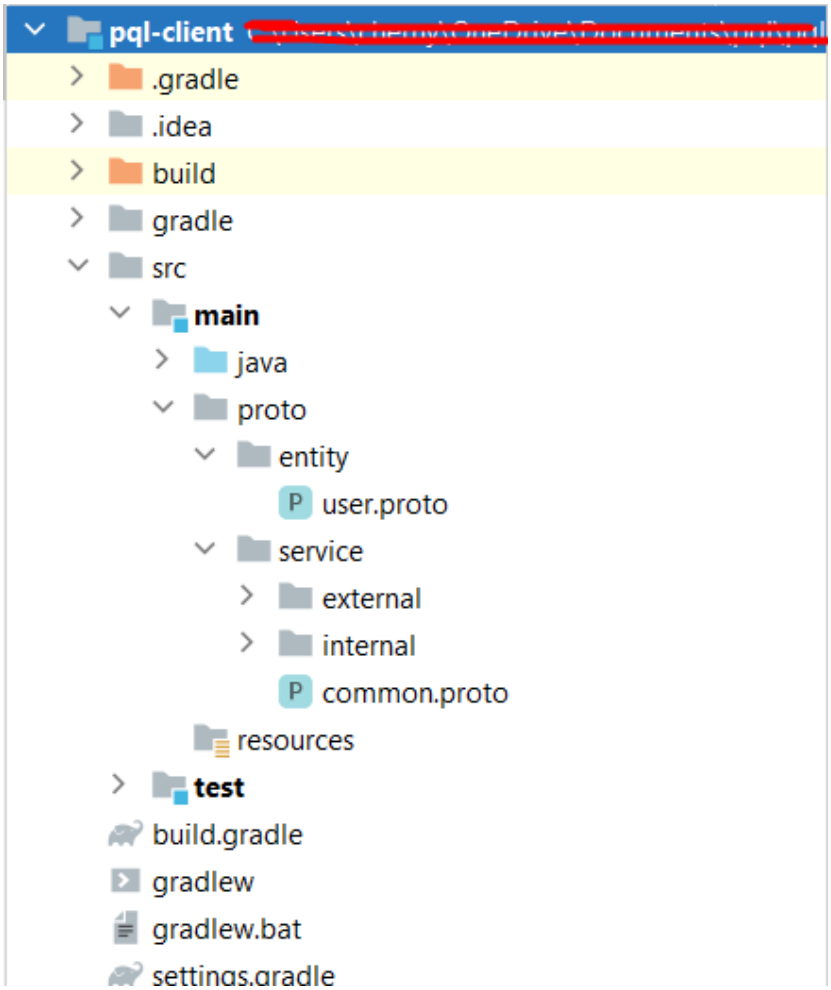


Рисунок 3.1.1. Структура проєкту

У директорію entity користувач може додати файли .proto, в яких будуть визначені схеми, наприклад, *user.proto*, який видно на рисунку вище:

```
syntax = "proto3";
package pql.entity;

message User {
  string uuid = 1;
  string name = 2;
}
```

Лістинг 3.1.2. Визначення схеми User

Далі потрібно зібрати проєкт, якщо ви працюєте з Gradle, то в директорії *build/generated/source* повинні з'явитися згенеровані класи згідно до прото файлів у директорії proto.

Тепер щодо коду. Створіть клас BasicCallCredentials та скопіюйте в нього такий контент:

```
public class BasicCallCredentials extends CallCredentials {

    private final String credentials;

    public BasicCallCredentials(String username, String password) {
        this.credentials = username + ":" + password;
    }

    @Override
    public void applyRequestMetadata(RequestInfo requestInfo, Executor
executor, MetadataApplier applier) {
        executor.execute(() -> {
            try {
                Metadata headers = new Metadata();
                Metadata.Key<String> authKey
       = Metadata.Key.of("authorization", Metadata.ASCII_STRING_MAR-
SHALLER);
                headers.put(authKey, "Basic "
       + Base64.getEncoder().encodeToString(credentials.getBytes()));
                applier.apply(headers);
            } catch (Throwable e) {
                applier.fail(Status.UNAUTHENTICATED.withCause(e));
            }
        });
    }

    @Override
    public void thisUsesUnstableApi() {

    }
}
```

Лістинг 3.1.3. Визначення класу BasicCallCredentials

Цей клас знадобиться для проходження аутентифікації. Тепер черга створити клас який буде клієнтом, тобто комунікувати з системою PQL. В даному прикладі він буде називатися PqlClient. Виглядає він так:

```
public class PqlClient {

    public static void main(String[] args)
throws InvalidProtocolBufferException {

        ManagedChannel channel = NettyChannel-
Builder.forAddress({host}, {port})
                .usePlaintext()
                .build();

        BasicCallCredentials callCredentials = new BasicCallCreden-
tials({login}, {password});

        ExternalServiceGrpc.ExternalServiceBlockingStub stub =
            ExternalServiceGrpc.newBlockingStub(channel)
                                .withCallCredentials(callCredentials);

        }

}
```

Лістинг 3.1.4. Визначення класу PqlClient

Перед тим як використовувати ті схеми, що зараз знаходяться у директоії *proto/entites* треба дізнатися чи додані вони до системи PQL у адміністратора. Якщо так то можемо додати до методу main() перший запит на додавання сутності User:

```
UserOuterClass.User user = UserOuterClass.User
        .newBuilder()
        .setName("UserName")
        .build();

SaveUpdateDeleteRequest saveUpdateDeleteRequest =
SaveUpdateDeleteRequest
        .newBuilder()
        .setSingularRequest(SingularSaveUpdateDeleteRequest
                .newBuilder()
                .setEntity(user.toByteString())
                .setKlass(UserOuterClass.User.class.getTypeName())
                .build())
        .setRepo("repo3")
        .build();

SaveUpdateResponse saveUpdateResponse = stub.save(saveUpdateDelete-
Request);
```

Лістинг 3.1.5. Будівництво запиту на додавання сутності

3.2. Інструкція для адміністратора

По-перше, треба відкрити консоль PQL. Запустіть файл *pql.exe*. Вітаюче вікно повинно виглядати так:

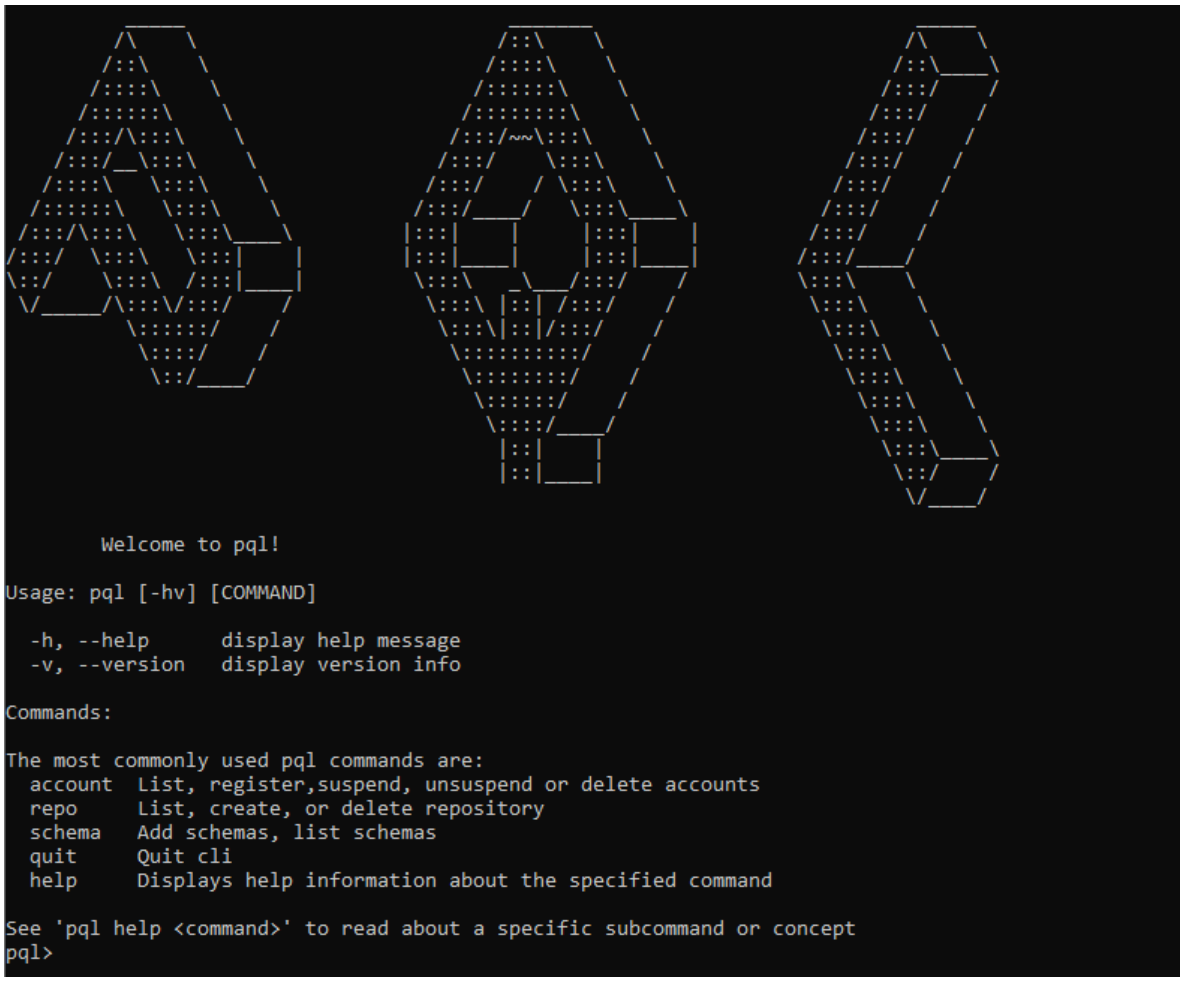


Рисунок 3.2.1. Вітаюче вікно PQL

В цій інструкції буде продемонстровано створення облікового запису, репозиторію, завантаження схеми, надання користувачу права на користування репозиторієм та запуск репозиторію.

Для того щоб створити обліковий запис, треба виконати таку команду: *pql account -r {логін} {пароль}*.

Для того щоб створити репозиторій, треба виконати таку команду: *pql repo -c {назва репоизторію} {номер бази даних в Redis}*.

Для того щоб дати обліковому запису права на користування репозиторієм, треба виконати таку команду:

pql account grant { логін облікового запису } { назва репоизторію }.

Для того щоб завантажити схему на репозиторій, треба виконати таку команду:

pql schema { назва репозиторію } { шлях до файлу .proto }.

Для того щоб запустити роботу репозитрію, треба виконати таку команду:

pql repo deploy { назва репозиторію }.

Результат в консолі повинен виглядати так:

```
pql> account -r account_login account_password
account_login is registered
pql> repo -c repo_name 6
repo_name created
pql> account grant account_login repo_name
pql> schema repo_name C:\Users\cherny\OneDrive\Documents\pql\pql-client\src\main\proto\entit
y\user.proto
pql> repo deploy repo_name
java -jar C:/Users/cherny/OneDrive/Documents/pql-home/repositories/repo_name/build/libs/pql-
internal-server.jar repo_name 8088
repo_name deployed
pql>
```

Рисунок 3.2.2. Результат виконання команд

Тепер користувач з логіном *«account_login»* може користуватися новоствореним репозиторієм *«repo_name»* з однією схемою user.proto. Для того щоб вивчити команди краще, можно використати команду *help*, вона виведе інформацію про існуючі команди.

ВИСНОВКИ

У даній роботі було розроблено прототип системи керування базами даних під назвою PQL (Proto Query Language), детально продемонстровано процес розробки, особливості його архітектури та роботи.

Початковий задум був набагато ширший ніж, на жаль, вдалося реалізувати. Первинно було заплановано ще і така функціональність:

- повноцінна генерація лямбда-виразів з урахуванням вимог безпеки (унеможливлення запуску неверифікованого коду) з їх виконанням на віддаленому сервері ;
- розгортка всіх екземплярів (pql-internal-api, pql-external-api) у Docker контейнерах та організація їх взаємодії за допомогою платформи Kubernetes.

Однак базовий концепт був реалізований, а саме розроблено та протестовано власну функціональну мультиплатформену систему керування базами даних.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Redis [Електронний ресурс] – Режим доступу до ресурсу: https://redis.io/.
2. gRPC [Електронний ресурс] – Режим доступу до ресурсу: https://grpc.io/.
3. Picocli – a mighty tiny command line interface [Електронний ресурс] – Режим доступу до ресурсу: https://picocli.info/.
4. Lettuce Reference Guide [Електронний ресурс] – Режим доступу до ресурсу: https://lettuce.io/core/release/reference/.
5. GraalVM: Native Image overview [Електронний ресурс] – Режим доступу до ресурсу: https://www.graalvm.org/22.0/reference-manual/native-image/.
6. Protocol Buffers [Електронний ресурс] – Режим доступу до ресурсу: https://developers.google.com/protocol-buffers.
7. Configuration library for JVM languages [Електронний ресурс] – Режим доступу до ресурсу: https://lightbend.github.io/config/.
8. Revolutionizing Java with GraalVM Native Image [Електронний ресурс] – Режим доступу до ресурсу: https://www.infoq.com/articles/native-java-graalvm/.
9. FoundationDB [Електронний ресурс] – Режим доступу до ресурсу: https://www.foundationdb.org/.
10. MicroStream [Електронний ресурс] – Режим доступу до ресурсу: https://microstream.one/.
11. Design patterns [Електронний ресурс] – Режим доступу до ресурсу: https://refactoring.guru/uk.
12. Head First Design Patterns Brain Friendly [Електронний ресурс] – Режим доступу до ресурсу: https://www.amazon.com/Head-First-Design-Patterns-Brain-Friendly/dp/0596007124.
13. Gradle overview [Електронний ресурс] – Режим доступу до ресурсу: https://gradle.org/.