ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ОЛЕСЯ ГОНЧАРА

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ

Кваліфікаційна робота

другий (магістерський) рівень вищої освіти

спеціальність 113 Прикладна математика

освітня програма Інформатика

АНАЛІЗ СУЧАСНИХ ТЕХНОЛОГІЙ ОТРИМАННЯ ДАНИХ З ХМАРНИХ ДЖЕРЕЛ

Виконавець

студент групи ПК-20м-1

Партала М.О\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис)

Керівник

доц., канд. фіз.-мат. наук

Степанова Н.І.\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис)

Завідувач кафедри

комп’ютерних технологій

д-р фіз.-мат. наук, проф.,

Гук Н.А. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис)

Дніпро – 2022

**Дніпровський національний університет імені Олеся Гончара**

Факультет прикладної математики

Кафедра\_\_комп’ютерних технологій

Рівень другий (магістерський)

Спеціальність 113 Прикладна математика

Освітня програма Інформатика

**ЗАТВЕРДЖУЮ**

**Завідувач кафедри**

комп’ютерних технологій

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  Гук Н.А.

**(підпис)**

« 20 » жовтня 2021 року

**З А В Д А Н Н Я**

**НА КВАЛІФІКАЦІЙНУ РОБОТУ**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Партала Михайло Олександрович\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(прізвище, ім’я, по батькові)

1. Тема роботи **\_\_**Аналіз сучасних технологій отримання даних з хмарних джерел**\_\_\_\_\_\_\_\_\_\_**\_\_**\_\_\_\_\_\_\_\_\_\_\_\_\_\_**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

керівник роботи \_\_Степанова Н.І., доц., канд. фіз.-мат. наук\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_,

( прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по Університету від « 20 » жовтня 2021 року № 1413с

2. Строк подання роботи 24 грудня 2020 року

3. Вхідні дані до роботи \_JSON файли різного обсягу, що містять інформацію про користувачів, Java бібліотеки для роботи з технологіями передачі даних, інтегроване середовище розробки Intelij Idea\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)\_\_розробка інформаційної та математичної моделі, створення веб-серверу за допомогою мови програмування Java з використанням сучасних технологій отримання даних, збір метрик завантаження даних з серверу, аналіз ефективності застосування технологій\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

Робота містить 18 рисунків, 3 графікиᅠᅠᅠᅠᅠᅠᅠᅠᅠᅠᅠᅠᅠᅠᅠᅠᅠᅠᅠᅠᅠᅠᅠᅠ

6. Консультанти розділів роботи

|  |  |  |  |
| --- | --- | --- | --- |
| Розділ | Прізвище, ініціали та посада  консультанта | Підпис, дата | |
| завдання видав | завдання  прийняв |
| Всі | Степанова Н.І., | 21.10.2021 | 10.12.2021 |
| розділи | доц., канд. фіз.-мат. наук |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

7. Дата видачі завдання 20 жовтня 2021 р.

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| №  з/п | Назва етапів роботи | Строк виконання етапів роботи | Примітка |
| 1. | Огляд літератури за темою дослід- | 21.10 – 26.10 | Виконано |
|  | ження. Визначення його основних |  |  |
|  | етапів |  |  |
| 2. | Визначення основних характеристик | 26.10 – 30.10 | Виконано |
|  | веб-технологій отримання даних |  |  |
| 3. | Генерування JSON файлу, для | 30.10 – 1.11 | Виконано |
|  | використання у якості вихідних |  |  |
|  | даних |  |  |
| 4. | Створення серверу з використанням | 1.11 – 15.11 | Виконано |
|  | розглянутих веб-технологій |  |  |
| 5. | Завантаження даних з серверу | 15.11 – 17.11 | Виконано |
|  | та збір метрик |  |  |
| 6. | Аналіз результатів. Графічна | 17.11 – 20.11 | Виконано |
|  | Візуалізація. |  |  |
| 7. | Оформлення роботи | 20.11 – 08.12 | Виконано |

Студент \_\_\_\_\_\_\_\_\_\_\_\_\_ Партала М.О.

( підпис )

Керівник роботи (проекту) \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Степанова Н.І.

( підпис )

РЕФЕРАТ

*Кваліфікаційна робота складається* з 54 стор., 18 рис., 3 таблиць, 2 додатків.

*Об’єктом дослідження даної кваліфікаційної роботи є* веб-сервер, який зберігає дані користувачів.

*Предмет дослідження:* розробка веб-серверу із застосуванням сучасних технологій клієнт-серверної комунікації, розробка моделей представлення запитів до серверу.

*Мета роботи:* проаналізувати використання сучасних технологій для отримання даних від віддаленого серверу.

*Методика дослідження:* створення інформаційної моделі предметної області, збір метрик, порівняльний аналіз отриманих даних, створення графіків.

*Одержані висновки та їх новизна:* за результатами проведеного аналізу сучасних технологій отримання даних з хмарних джерел визначено основні властивості технологій, їх ефективність використання при різних обсягах даних. Аналітичні дані стосовно ефективності технологій отримано у вигляді таблиць та діаграм.

*Результати досліджень можуть бути застосовані при* проектуванні архітектури проекту, а саме клієнт-серверної комунікації.

*Перелік ключових слів:* ОБ’ЄКТ ПЕРЕДАЧІ ДАНИХ, ЗАПИТ, ІНТЕРФЕЙС, СЕРВЕР, ПРОТОКОЛ, СЕРВІС

ANNOTATION

The diploma work of the 2th year student Partala Mykhailo (DNU, Faculty of Apply Mathematics, Department of Computer Technology) is devoted to the analysis of modern web technologies for obtaining data from the cloud. The object of the research of this qualification work is a web-server storing user data. Subject of the research: web-server development using modern client-server communication technologies, models development those aimed to represent requests to the server. Purpose of work: to analyze the use of modern technologies to obtain data from remote server. Research methodology: creating an information model of the subject area, collecting metrics, comparative analysis of the data obtained, creating graphs.

Conclusions and their novelty: the results of the analysis of modern technologies to obtain data from cloud sources helps to identify the main properties of technologies, their effectiveness at different volumes of data. The analytical data on the effectiveness of technologies have been obtained in the form of tables and charts. The results of the research can be applied in the project architecture design, mostly in client-server communication.

Bibliography – 20, pictures – 18, tables – 3

ЗМІСТ

[ВCТУП 7](#_Toc90414677)

[ПОСТАНОВКА ЗАДАЧІ 11](#_Toc90414678)

[1 ОГЛЯД ПРОБЛЕМИ 12](#_Toc90414679)

[1.1 Сучасний стан 12](#_Toc90414680)

[1.2 Інтерфейс REST API 16](#_Toc90414681)

[1.3 Мова запитів GraphQL 19](#_Toc90414682)

[1.4 Веб та TCP сокети 21](#_Toc90414683)

[1.5 Технологія gRPC 23](#_Toc90414684)

[1.6 Нові протоколи 26](#_Toc90414685)

[2 представлення даних задачі 28](#_Toc90414686)

[2.1 Інформаційна модель 28](#_Toc90414687)

[2.2 Математична модель 29](#_Toc90414688)

[3 ОПИС ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 35](#_Toc90414689)

[3.1 Вибір мови програмування та технологій 35](#_Toc90414690)

[3.2 Створення сервісу отримання масиву користувачів 37](#_Toc90414691)

[3.3 Створення REST API 38](#_Toc90414692)

[3.4 Створення gRPC серверу 40](#_Toc90414693)

[3.5 Створення GraphQL серверу 42](#_Toc90414694)

[3.6 Створення WebSocket серверу 43](#_Toc90414695)

[4 АНАЛІЗ РЕЗУЛЬТАТІВ 46](#_Toc90414696)

[ВИСНОВКИ 52](#_Toc90414697)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 53](#_Toc90414698)

[ДОДАТКИ 55](#_Toc90414699)

# ВCТУП

Розвиток індустрії інформаційних технологій супроводжується постійним збільшенням обсягів даних, у зв’язку з цим час пошуку і отримання клієнтом потрібної інформації збільшується. На даний час існує декілька підходів щодо вирішення даної проблеми, найбільш перспективним серед них є впровадження більш ефективних технологій взаємодії клієнтських програм з джерелами інформації.

Зазвичай клієнти і сервери обмінюються повідомленнями за шаблоном «запит-відповідь»: клієнт відправляє запит, а сервер повертає відповідь. Для реалізації такої взаємодії комп'ютери повинні мати спільну мову та правила спілкування, які визначаються протоколом зв'язку. Усі протоколи клієнт-серверної моделі працюють на рівні додатків. Протокол прикладного рівня визначає основні шаблони діалогу. Для більшої формалізації обміну даними, сервер може реалізувати інтерфейс прикладного програмування (API). API – це рівень абстракції доступу до сервісу. Обмежуючи зв'язок певним форматом контенту, синтаксичний аналіз спрощується, в той час як абстрагування полегшує міжплатформений обмін даними.

Одним із способів взаємодії компонентів розподіленого мережевого додатку є REST API (*Representational State Transfer*). REST API – узгоджений набір обмежень, які враховуються під час проектування системи. Використання таких обмежень призводить до підвищення продуктивності системи і спрощення її архітектури. REST API являє собою набір точок призначення, кожна з яких відповідає ресурсу. REST – це стандарт, придатний для широкого спектру API, але існують й інші стилі проектування API. Щоб допомогти розробникам API зрозуміти, який стиль проектування API слід використати і для якої ситуації, у даній роботі розглядаються сучасні технології, їх сильні та слабкі сторони.

Велика проблема REST API полягяє у численності точок призначення. Якщо клієнтові потрібні різні ресурси, йому доводиться робити декілька запитів, щоб зібрати всі необхідні дані. Крім того, REST API не пропонує клієнту мову запитів. Клієнт не впливає на те, які дані повертає сервер. Ще одна значуща проблема REST API – підтримка версій. Необхідність підтримувати кілька версій означає нові точки призначення. Це спричиняє додаткові труднощі у використанні і підтримці API і може стати причиною дублювання коду на сервері.

Для вирішення зазначених проблем був розроблений GraphQL, який являє собою мову запитів і маніпулювання даними з відкритим вихідним кодом для API. Одночасно з цим GraphQL є середовищем для виконання запитів з існуючими даними.

Технологія GraphQL – відмінний вибір для організацій з декількома командами і системами, які хочуть зробити свої дані легко доступними через єдиний уніфікований API. За допомогою GraphQL можна декларативно повідомляти бекенду про дані, які необхідно отримати. GraphQL дозволяє створювати API-інтерфейси, приховувати складність внутрішніх систем, використовуваних для отримання різних фрагментів даних, і використовувати систему типів, яка призводить до автоматичної і актуальної документації. Ці функції, поряд з інструментами і екосистемою, роблять GraphQL ефективним і дієвим інструментом як для розробників API, так і для розробників клієнтів.

Технологія вимагає певних вкладень, але це надає багато переваг в ситуаціях, коли існує безліч даних і послуг, які повинні бути доступні для різних існуючих і майбутніх клієнтів API.

Технологія gRPC – це новий підхід до старого підходу, відомого як RPC, або видалений виклик процедур. RPC – це метод виконання процедури на видаленому сервері, схожий на запуск програми на комп'ютері, що знаходиться на відстані милі від вашої робочої станції. Це має свої переваги і недоліки – саме ці недоліки були ключовими при розробці і впровадженні REST.

Ключова відмінність між gRPC і REST полягає в тому, яким чином RPC визначає переговори за контрактом. Тоді як REST визначає свої взаємодії через терміни, стандартизовані в запитах, RPC функціонує на основі ідеї контрактів, в яких обмін повідомленнями визначається і обмежується відносинами між клієнтом та сервером, а не самою архітектурою. RPC надає велику частину повноважень клієнтові для виконання, тоді як велика частина обробки і обчислень перекладається на видалений сервер, на якому розміщений ресурс. З цієї причини RPC дуже популярний для пристроїв, що вимагають спеціалізованих контрактних комунікацій для малопотужних пристроїв. REST часто вважають надмірно вимогливим до ресурсів, тоді як RPC можна використати навіть в ситуаціях з украй низьким енергоспоживанням.

Протокол WebSocket забезпечує можливість обміну даними між браузером та сервером через постійне з'єднання. WebSocket являє собою розширення протоколу HTTP. Синхронний протокол, побудований по моделі запит-відповідь, стає повністю асинхронним і симетричним. Дані передаються каналом зв'язку в обох напрямках у вигляді «пакетів», без розриву з'єднання та додаткових HTTP-запитів. Клієнт і сервер є двома рівноправними учасниками обміну даними. Кожен працює сам по собі, і коли потрібно відправляє дані іншому. Друга сторона може відповісти не відразу, а може і взагалі не відповісти. Тобто протокол дає повну свободу в обміні даними.

Сервер може отримувати запити від безлічі різних клієнтів за короткий період часу. У той самий час комп'ютер може виконувати тільки обмежену кількість завдань у будь-який момент і покладається на систему планування для визначення пріоритетів вхідних запитів від клієнтів для їх задоволення.

Тому є актуальним питання вибору технологій отримання даних при проектуванні архітектури веб додатків.

Кваліфікаційна робота складається з таких частин:

* вступ, у якому обґрунтовано актуальність роботи, визначено цілі проведення наукового дослідження, галузь дослідження, обрано методи дослідження;
* постановка задачі;
* основна частина, яка містить відомості щодо вибору мови програмування, створення серверу, розробки інтерфейсів отримання даних, тестування додатку, аналізу результатів;
* висновки;
* список використаних джерел;
* додатки.

# ПОСТАНОВКА ЗАДАЧІ

У роботі розглядаються сучасні засоби і технології отримання даних з метою визначення найбільш оптимальних варіантів клієнт серверної взаємодії для побудови ефективної архітектури. Для досягнення поставленої у даному дослідженні мети необхідно ознайомитися з існуючими клієнт-серверними технологіями, визначити їх особливості, обрати відповідні засоби для програмної реалізації клієнт-серверної взаємодії, провести порівняльний аналіз різних способів організації такої взаємодії.

Таким чином, у даній роботі ставляться такі задачі:

* дослідити та провести аналіз предметної області, а саме – провести розгляд існуючих способів взаємодії клієнт-серверних додатків;
* розглянути особливості роботи і можливості сучасних технологій;
* визначити моделі і методи, з використанням яких можливо прискорити отримання даних від серверу;
* розробити інформаційну модель системи тестування технологій;
* визначити функціональні вимоги до програмного забезпечення, яке може бути застосовано у роботі;
* обрати мову програмування для розробки серверної частини і виконати програмну реалізацію;
* реалізувати інтерфейс взаємодії з додатком;
* протестувати розроблений програмний додаток;
* зробити висновки з дослідження, визначити умови і  шляхи для подальшого удосконалення системи тестування.

# ОГЛЯД ПРОБЛЕМИ

## Сучасний стан

Від правильного вибору архітектури програмного забезпечення залежить успіх і швидкість розробки проекту, розуміння структури програми та складність його підтримки. З цієї причини розробники, намагаючись знайти найкращі рішення проблеми, зазвичай використовують нові архітектурні засоби для створення програмного забезпечення.

Деякий час тому поширеною практикою розробки програм була розробка монолітних додатків, що передбачало побудову архітектури проекту як єдиного програмного комплексу, який відповідає за весь необхідний функціонал. Даний підхід мав свої переваги, такі, наприклад, як простота розробки, тестування та розгортання. Але зі зростанням проекту монолітна архітектура починає вносити деякі недоліки в процес розробки, наприклад:

* додавання певної нової функціональності супроводжується збільшенням бази коду;
* розмивається дотримання меж відповідальності внутрішніх модулів;
* компіляція та розгортання займає набагато більше часу.

Враховуючи вищенаведені недоліки монолітної архітектури, розробники почали використовувати інші підходи. Одним з таких підходів є мікросервісна архітектура. Мікросервісна архітектура – ​​варіант модульного підходу до розробки програмного забезпечення, орієнтованого на взаємодію якомога менших за розміром, слабо пов’язаних і легко замінних модулів – мікросервісів. При такій архітектурі проект являє собою набір невеликих сервісів, зв’язок між якими здійснюється за допомогою технологій передачі даних. Кожна з цих послуг має виконувати своє чітко визначене бізнес-завдання.

Мікросервісна архітектура має такі переваги:

* ​​краще масштабування;
* незалежна розробка та розгортання модулів;
* краща можливість повторного використання тощо.

Архітектура розподіленої системи вимагає глибокого розуміння доступних технологій. Кожна технологія може бути використана як інструмент для вирішення реальних проблем.

Таким чином, постає питання, яку технологію варто розглядати при проектуванні веб-системи з використанням мікросервісної архітектури.

Хмарні сховища даних дозволяють кардинально підвищити доступність клієнтських даних та необхідних мережевих елементів для їх надійного передавання і зберігання. Географічний розподіл клієнтських застосувань, їх мобільність та потреба у збереженні цілісності даних породжують протиріччя, яке полягає у необхідності підвищення пропускної здатності існуючої комунікаційної складової розподілених сховищ даних в умовах підвищення вимог щодо їх доступності, а також відносно несанкціонованого доступу та захисту від пошкоджень.

Отже, наукова задача розроблення моделей та методів підвищення пропускної здатності розподілених систем на основі нових протоколів доступу є актуальною і своєчасною.

Однак на даний час існує ряд недостатньо опрацьованих наукових питань, що стають на заваді ефективній організації хмарних джерел даних, та, відповідно, розподілених обчислювальних систем на їх основі, наприклад:

* недостатньо розвинена теоретична база, яка могла б стати заміною класичній теорії масового обслуговування при побудові сучасних комунікаційних систем розподілу інформації;
* недостатньо розвинуті методи та алгоритми, які впливають на якість обслуговування;
* недостатньо опрацьовані питання визначення показників якості функціонування систем розподілу та передачі інформації у розподіленому мережному середовищі.

Складність розподілених систем для передачі даних через хмарні джерела даних на сьогоднішній день в значній мірі визначається складністю протоколів та їх поєднанням. Протоколи представляють собою набір правил, у відповідності до яких взаємодіють системи. Під час проектування та розробки систем передачі даних протоколи реалізуються програмно або апаратно. У зв’язку з тим, що впровадження технологій відбувається поверх наявного апаратного обладнання та створювати і замінювати його досить проблематично, для хмарних технологій, в основному використовують програмну реалізацію протоколів. Специфікації протоколів представлені в стандартах, які повинні забезпечувати ефективну взаємодію та бути коректними.

Для поняття продуктивності системи існує багато аспектів, серед яких у випадку хмарної системи зберігання даних головним є процес переміщення даних між користувачем і віддаленим постачальником послуг. Проблема криється в транспортному протоколі TCP, який є основним робочим протоколом Інтернету. TCP керує потоком даних на основі підтвердження прийому пакетів з віддаленого вузла. Втрата або затримка пакетів дозволяють управляти перевантаженням, що ще більше обмежує продуктивність для уникнення глобальних мережевих проблем. TCP ідеально підходить для пересилання невеликих за розміром даних через глобальну мережу Інтернет, але є мало придатним для доставки великих обсягів інформації – у цьому випадку час обміну даними значно збільшується.

Процес розробки протоколів та їх комбінацій стає все більш динамічним, крім того, зростає складність самих протоколів, що побічно підтверджується зростанням об’єму їх стандартних специфікацій. Таким чином, формальне доведення коректності протоколів та їх поєднань представляє собою важливу наукову проблему. Як відомо, для організації надання будь-якого сервісу в хмарних технологіях і доступу до хмарних джерел, зокрема, необхідна наявність відповідного сховища. Тобто, це сервер, або мережа серверів, через які для клієнтів реалізовано інтерфейс доступу до сховища. Самий звичайний спосіб надання сервісу клієнту – це обслуговування його запитів. Обслужити запит – означає отримати запит і направити відповідь стороні, яка створила цей запит.

Сервер хмарного сховища має обмежені обчислювальні ресурси, тобто може виконувати лише обмежену кількість операцій за одиницю часу. Але ж для створення відповіді на запит потрібно провести деяку кількість операцій. Відповідно, сервер хмарного джерела може обслужити за одиницю часу лише обмежену кількість запитів, яка визначається обчислювальною потужністю сервера.

Якщо за одиницю часу кількість запитів, які поступають від клієнтів, перевищує обчислювальні можливості сервера, то певна кількість запитів залишиться не обслуженими. Щоб зменшити кількість необроблених запитів, необхідно збільшити обчислювальну потужність сервера джерела.

Найбільш прямолінійний підхід до збільшення потужності сервера сховища – використовування більш потужних комп’ютерів в якості серверів. Цей

підхід має певні недоліки:

* потужність комп’ютерів обмежена;
* вартість сховища за рахунок використовуваних комп’ютерів зростає швидше ніж його продуктивність, тобто вартість в два рази більш потужного комп’ютера для сховища зросте більше ніж у два рази.

Інший підхід, який виключає вказані недоліки, полягає в тому, щоб використовувати для обслуговування клієнтських запитів декілька серверів сховища. При цьому вартість рішення буде прямо пропорційна його потужності, і верхня межа на обчислювальну потужність визначається не рівнем апаратних технологій, а кількістю серверів сховища. Проте, при великій розподіленості сховища зростають затрати на передачу даних між сереверами сховища, дані яких повинні копіюватися одночасно на всі сервери сховища.

## Інтерфейс REST API

REST API – це найпростіший і поширеніший спосіб взаємодії між сервером і клієнтом. Використовуючи протокол HTTP, клієнти можуть використати різні методи, такі як GET, PUT, POST, PATCH та DELETE для завантаження або оновлення вмісту на віддаленій машині. Важливо відмітити, що REST API не мають стану із-за природи протоколу HTTP, тобто кожного разу, коли клієнт робить запит, він повинен ідентифікувати себе, як правило, за допомогою токена або ідентифікатора сесії, щоб сервер міг ідентифікувати і обробити запит.

RESTful API були розроблені на основі протоколу HTTP. Це означає, що розробникам не треба встановлювати додаткове програмне забезпечення або бібліотеки при створенні REST API.

Однією з ключових переваг REST API є те, що вони забезпечують велику гнучкість. Дані не прив’язані до ресурсів або методів, тому REST може обробляти безліч типів викликів, повертати різні формати даних і навіть змінюватися структурно. Така гнучкість дозволяє розробникам створювати API, що відповідають вашим потребам, а також потребам різних клієнтів.

Більшість сучасних API використовують формат Json або XML для надсилання і отримання даних, і, якщо тіло повідомлення не зашифроване, вміст запиту та відповіді легко прочитати для тестування і налагодження.

REST API чудово підходять, коли потрібно просто отримати доступ або записати дані в певний момент часу, але вони не призначені для потокового контенту чи комунікацій в реальному часі.

Також REST API визначаються на основі конкретних контрактів. Це означає, що додавання і видалення параметрів в запиті або відповіді вимагає змін як на сервері, так і на боці клієнта. У великих компаніях зі значною кількістю різних API, підтримка і забезпечення зворотної сумісності є складною задачею.

Одним з кращих безкоштовних інструментів для тестування REST API є середовище Postman. Postman дозволяє зробити запит або організувати набір запитів для імітації поведінки клієнта без його створення або присутності.

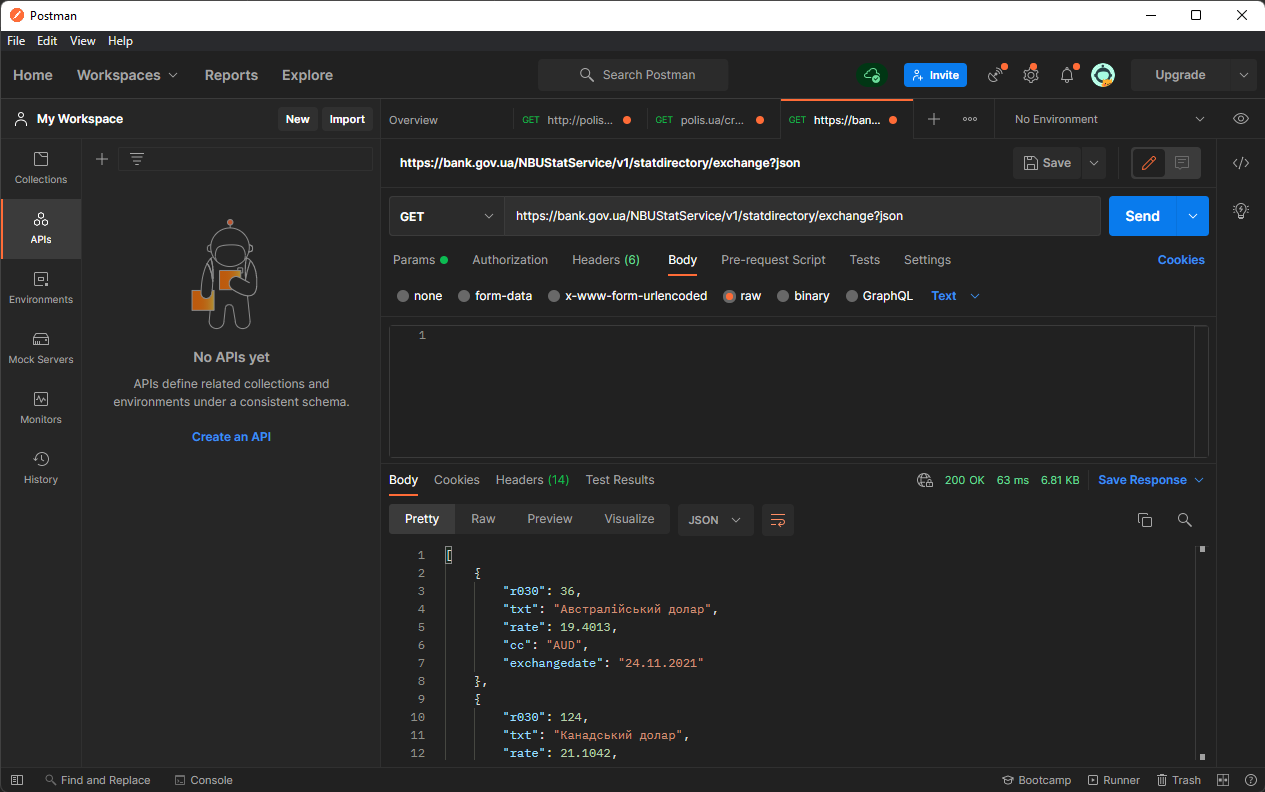


Рисунок 1.1 – Створення REST запиту за допомогою Postman

Написання і супровід контрактів, розробка об'єктів передачі даних (DTO) і їх документування завжди були складними задачами для виконання будь якими командами розробників. Для спрощення цих процесів у 2010 році було представлено проект Swagger, який пізніше був перейменований у стандарт Open API.

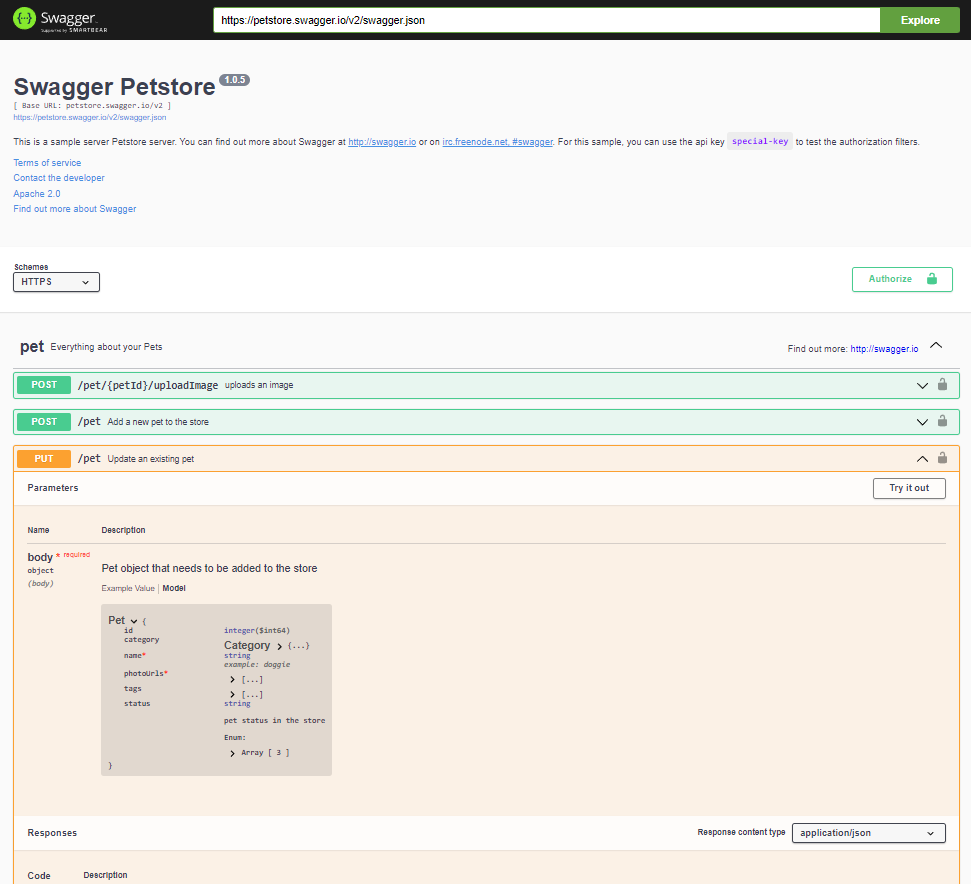


Рисунок 1.2 – Відображення технічної специфікації (Swagger UI)

Використовуючи Open API, бекенд-команди можуть розробити технічну специфікацію API у форматі YAML або JSON, яка буде містити усю інформацію про маршрути, параметри, безпеку, а також всю необхідну документацію. Найважливіше у використанні Open API – це генератори коду. Після визначення специфікацій API, як бекенд, так і клієнтські команди можуть генерувати початковий код для мережевого рівня. Це скорочує час, витрачений розробниками на реалізацію, тестування і налагодження системи зв'язку між обома сторонами.

## Мова запитів GraphQL

Для початку необхідно розібратися – що таке взагалі GraphQL та як з ним можна працювати. Як вказано на офіційному сайті <https://graphql.org>, GraphQL – це мова запитів, а також середовище виконання цих запитів для вже існуючих даних. Ключовим моментом є те, що клієнт може запитувати саме ті дані, які йому необхідні, що може значно прискорити виконання запитів:

*“GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools”.*

На рис.1.3 і 1.4 наведено відповідно приклади запиту мовою GraphQL і відповіді на цей запит.

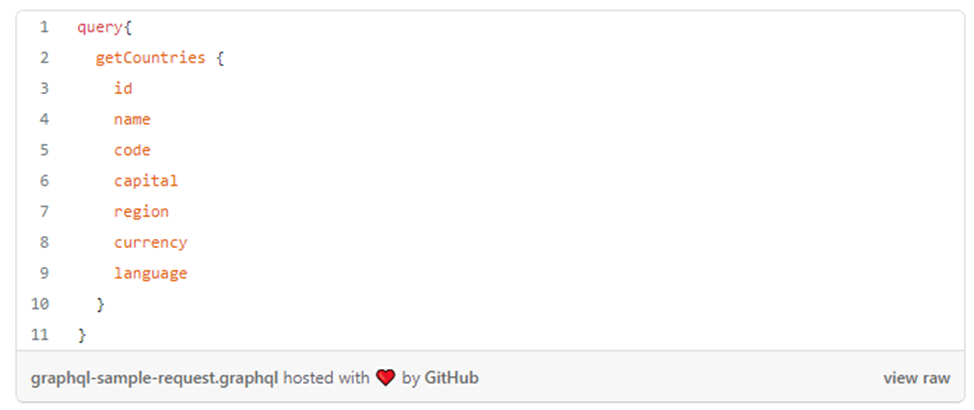


Рисунок 1.3 – GraphQL запит



Рисунок 1.4 – GraphQL відповідь

Хоча GraphQL має багато переваг для спрощення запиту, він додає накладні витрати на продуктивність, які необхідно враховувати. Складні представлення можуть як і раніше викликати безліч звернень до бази даних, що, у свою чергу, може привести до повільного завантаження. Існує декілька підходів до вирішення цих проблем, таких як використання завантажувачів даних, кешування і агрегація запитів.

Існує безліч мов програмування, які можуть бути використані для роботи з GraphQL. Технологія не накладає будь-яких обмежень на використання джерел даних.

GraphQL був створений на основі RestAPI. Це означає, що він має усі переваги REST API і може бути легко відлагоджений.

Postman має підтримку GraphQL і дозволяє тестувати GraphQL - запити аналогічно REST API.

## Веб та TCP сокети

Веб-сокети є відмінним вибором для чат-додатків, додатків для фінансових ринків і браузерних ігор, які вимагають обміну даними у реальному часі між сервером і клієнтом за протоколом HTTP. Накладні витрати HTTP (такі як заголовки, кукі і т.д) обмежені, що дозволяє передавати дані з набагато більшою швидкістю.

WebSocket – це комунікаційний протокол, який забезпечує двонаправлений, повнодуплексний зв'язок через постійне TCP-з'єднання (рис. 1.5).

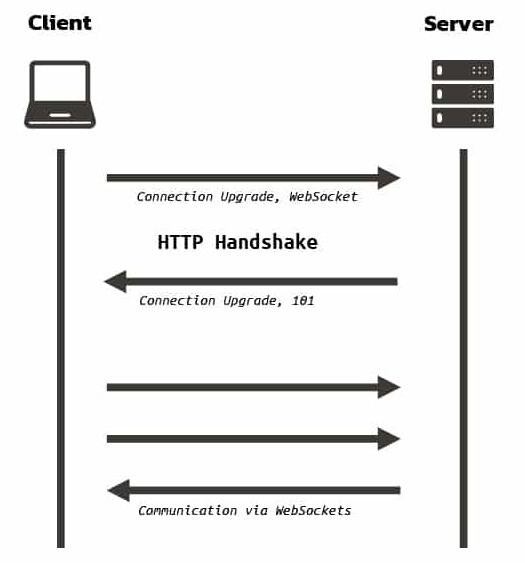


Рисунок 1.5 – Клієнт-серверна комунікація з використанням веб-сокетів

На відміну від HTTP, з'єднання через веб-сокети можуть бути ненадійними через проблеми зі з'єднанням. У реальних сценаріях завжди корисно мати рішення для відновлення втрачених повідомлень.

На відміну від HTTP-серверів, де можливе горизонтальне масштабування, для серверів які використовують веб-сокети не існує стандартного засобу масштабування. Це пов'язано з тим, що кожному клієнтові для спілкування потрібне постійне з'єднання з одним сервером. Існують спеціальні реалізації, такі як N-CHAN, які дозволяють подолати цю проблему.

З розвитком протоколу HTTP веб-сокети не підтримуються в HTTP/2.0. Для вирішення цієї проблеми скоро з'являться нові протоколи, такі як WebTransport. WebTransport може запропонувати стандартизовані живі комунікації через різні версії протоколів HTTP.

У мережевих рівнях OSI (рис.1.6) сокет TCP працює на сеансовому рівні, розташованому нижче рівня представлення. Це забезпечує велику гнучкість, коли продуктивність є чинником для комунікації.

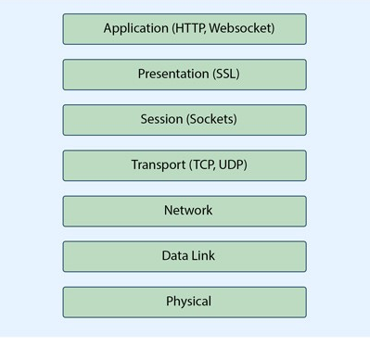


Рисунок 1.6 – Мережева модель OSI

На відміну від RESTful API, в TCP сокетах цілісність з’єднання може бути перевірена в процесі початку сеансу без повторної перевірки особи в кожному повідомленні. Цей процес може тривати до того часу, поки TCP-з’єднання залишається відкритим. Оскільки у TCP-з’єднаннях відсутні HTTP заголовки, TCP-сокет може забезпечити високу пропускну здатність і низьку затримку зв'язку між сервером і клієнтами.

Важливо відмітити, що рівень TCP гарантує доставку мережевих пакетів. Сокет TCP – це надійний протокол, який чудово підходить для точних застосувань, таких як онлайн-ігри, системи моніторингу, датчики тощо.

Хоча TCP має високу продуктивність, масштабування TCP-сокетів і балансування навантаження можуть бути складними.

В TCP-сокетах повідомлення є бінарними, і, на відміну від REST API, у даному випадку нелегко переглядати і редагувати тіло запиту.

## Технологія gRPC

Специфікація gRPC (Google – RPC) – це новий стандарт CNCF, що грунтується на HTTP/2. Він був введений компанією google для забезпечення високошвидкісного бінарного обміну даними між клієнтом і сервером.

У разі використання gRPC клієнтський додаток може безпосередньо викликати метод серверного додатку, який працює у цей час на іншій машині, начебто це локальний об'єкт (рис.1.7). Вказана властивість полегшує створення розподілених застосувань і сервісів. Як і у багатьох інших системах RPC, в основі gRPC лежить ідея визначення служби, в якій вказуються методи з їх параметрами і типами, що повертаються. Ці методи можуть бути викликаними віддалено. Сервер реалізує даний інтерфейс і запускає gRPC-сервер для обробки клієнтських викликів. Клієнт має заглушку, яка надає ті ж самі методи, що і сервер.

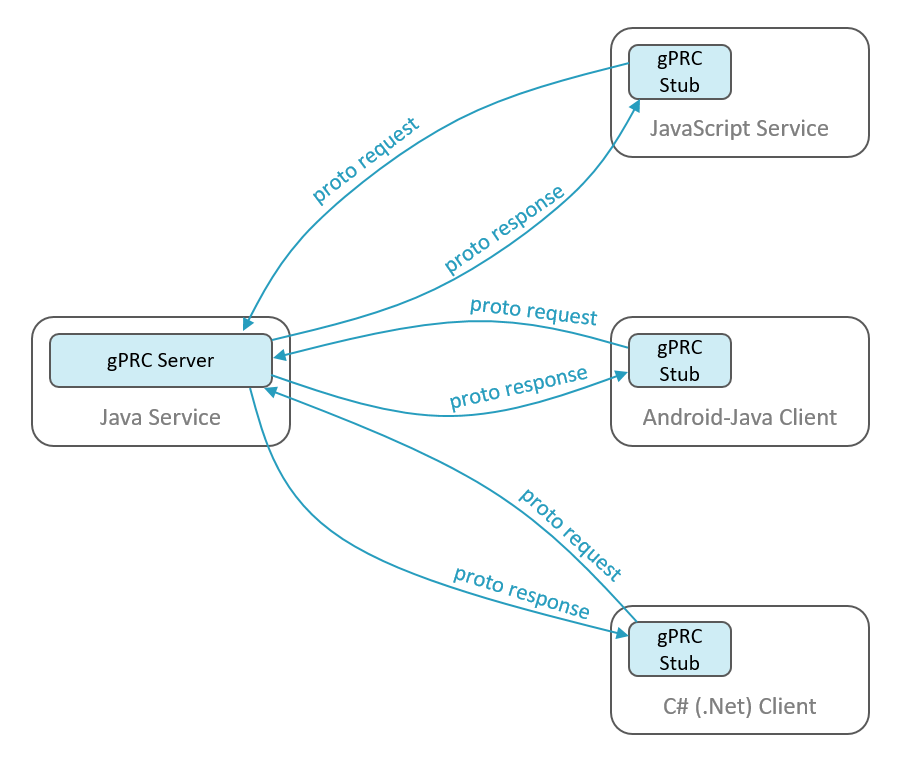


Рисунок 1.7 – Клієнт-серверна комунікація з використанням gRPC

Протокол gRPC володіє безліччю чудових можливостей, таких як двонаправлена потокова передача і відмінна масштабованість. Подібно до Open API, після визначення файлу специфікації, клієнт і сервер можуть легко використати генератор коду для реалізації мережевого рівня додатків.

Крім того, gRPC здатний забезпечити на порядок швидший зв'язок в порівнянні з REST API завдяки природі бінарного зв'язку і перевагам HTTP2. У 2019-2020 роках багато великих компаній інвестували на підтримку gRPC і міграцію REST API. Очікується, що це буде великою тенденцією найближчими роками.

Оскільки виклики RPC мають бінарний формат, тестування, налагодження та редагування запиту може бути ускладнено.

Proto-файл – це каталог, що визначає метод і аргументи, необхідні для ініціації виклику gRPC між клієнтом і сервером (рис. 1.8).

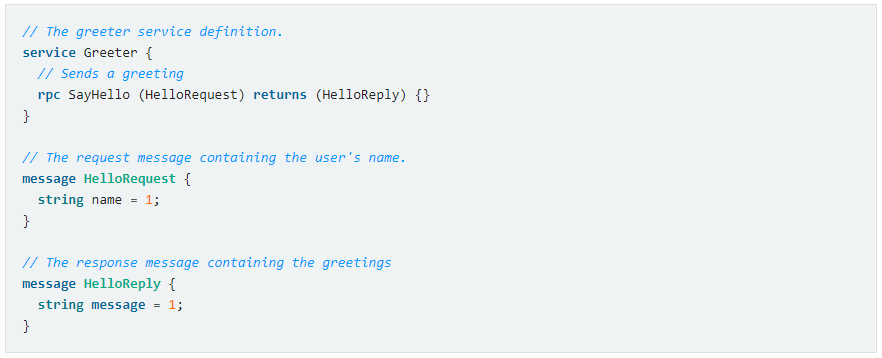


Рисунок 1.8 – Proto-файл

Слід зазначити, що gRPC – це все ще дуже новий протокол. Поки ще не так багато інструментів для тестування, але очікується, що в найближчому майбутньому в нього буде внесено багато вкладу з боку співтовариства.

Bloom RPC – відмінний інструмент з відкритим початковим кодом для тестування і відладки викликів RPC (рис.1.9). Він пропонує функції, аналогічні Postman, для ініціації і тестування викликів gRPC.

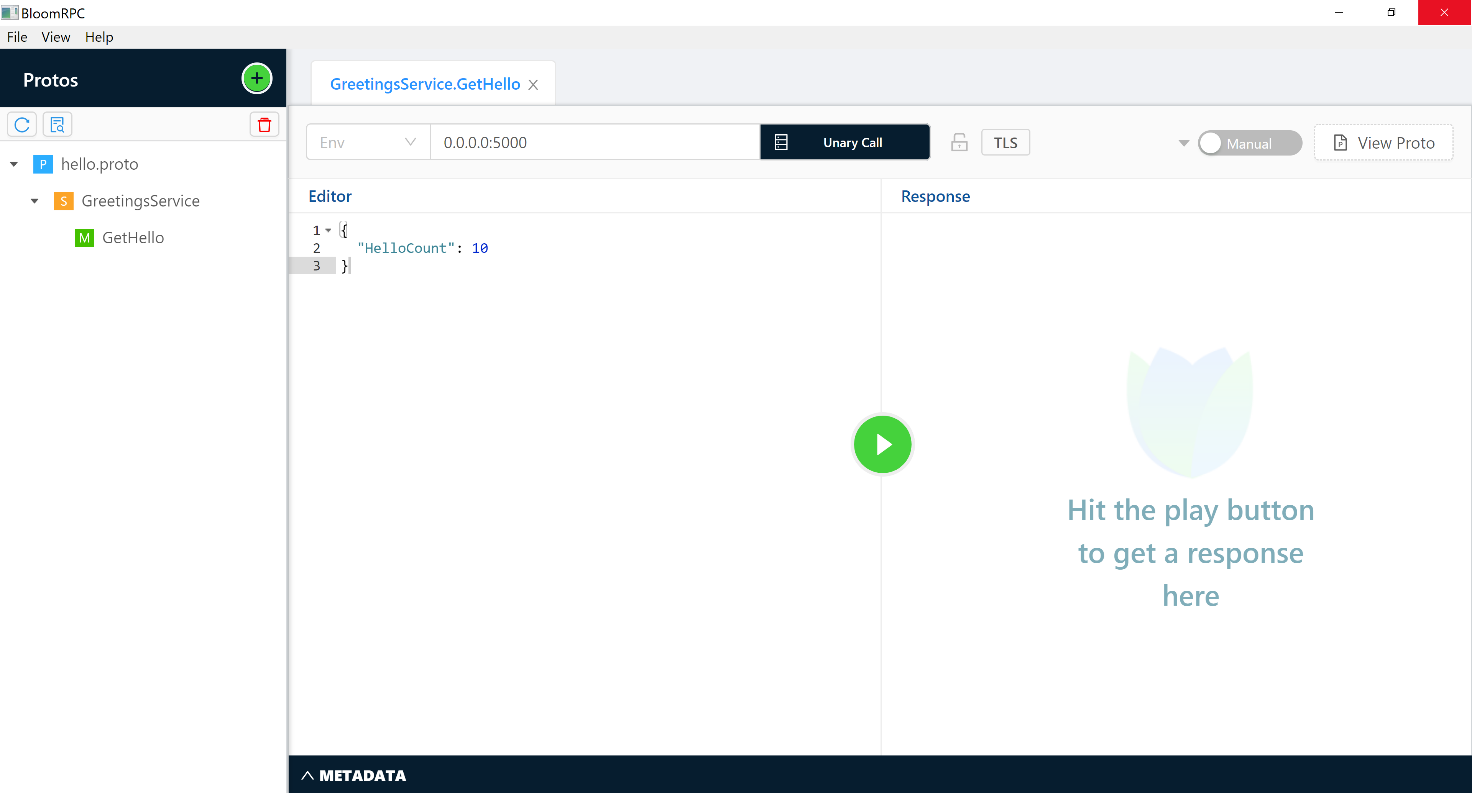


Рисунок 1.9 – Bloom RPC

## Нові протоколи

WebTransport – це новий протокол, який був представлений для заповнення пропусків між HTTP/2, Websockets і UDP- з'єднаннями. Він також дозволяє веб-додаткам встановлювати інтерактивні, двонаправлені, мультиплексовані мережеві з'єднання. WebTransport сумісний з HTTP3 і пропонується сумісність з протоколом QUIC.

WebTransport підтримує безліч функцій, деякі з яких приведені нижче:

* *Однонапрямлені потоки* – це невизначено довгі потоки байтів в одному напрямі із зворотним тиском на відправника. Це відбувається, коли або одержувач не може читати досить швидко, або його обмежує пропускна здатність мережі або перевантаження.

Однонапрямлені потоки можуть бути корисні для відправки повідомлень, які не чекають відповіді. Послідовна і надійна передача повідомлень може бути досягнута шляхом відправки безлічі повідомлень в одному потоці. Позапорядковий обмін повідомленнями може бути досягнутий шляхом відправки одного повідомлення в одному потоці.

* *Двонаправлені потоки* – це повнодуплексні потоки. Двонаправлений потік фактично є парою однонапрямлених потоків.
* *Датаграми* – це окремі пакети даних, розмір яких обмежений максимальною одиницею передачі (MTU) базового з'єднання, і які можуть бути успішно передані або не передані. Якщо вони передаються, то можуть бути передані в довільному порядку. Ці характеристики роблять API датаграм ідеальним для передачі даних з малою затримкою, з найкращими зусиллями. Датаграми можна розглядати як повідомлення протоколу призначених для користувача датаграм (UDP), але зашифровані і з контролем перевантаження.

Веб-транспорт чудово підходить для багатьох застосувань. Ось деякі з них:

* відправка або отримання мультимедіа з низькою затримкою;
* пересилання файлів;
* відправка поточного стану гри з мінімальною затримкою на сервер у вигляді безлічі невеликих, ненадійних, неврегульованих повідомлень через регулярний інтервал часу;
* отримання медіафайлів, що надсилаються невпорядковано з сервера з мінімальною затримкою.
* отримання повідомлень, що відправляються з сервера;
* запит по HTTP, що передаються без порядку і ненадійно, через одне і те ж саме мережеве з’єднання.

Веб-транспорт все ще знаходиться на ранніх стадіях і вимагає реалізації і підтримки на різних мовах.

QUIC (Quick UDP Internet Connections) – це мережевий протокол транспортного рівня загального призначення, схожий на UDP. Даний протокол був публічно представлений компанією Google в 2013 році. Поки він ще є інтернет-чернеткою, але вже використовується більш ніж в половині усіх з'єднань браузеру Chrome з серверами Google. Microsoft Edge, Firefox і Safari підтримують його, навіть якщо він не є обраним за умовчанням.

QuicTransport відображає безпосередньо потоки і датаграми QUIC, що дозволяє легко підключатися до серверів, що спілкуються за допомогою QUIC, з мінімальними накладними витратами.

Існують і інші протоколи, які менш поширені, але широко використовуються в реальних застосуваннях. Протокол транспортного рівня, який чудово підходить для потокового відео і комунікацій, де часткова втрата даних не дуже важлива. WebRTC: подібно gRPC дозволяє здійснювати RPC- комунікації через HTTP.

# представлення даних задачі

## 2.1 Інформаційна модель

Концепція інформаційної мережі в якій основна частина її ресурсів зосереджена в серверах, обслуговуючих своїх клієнтів, називається клієнт-серверною архітектурою.

Клієнт-серверна архітектура визначає такі типи компонентів:

* множина серверів, що надають інформацію додаткам, які звертаються до них;
* множина клієнтів, що використовують сервіси, які надаються серверами;
* мережа, яка забезпечує взаємодію між клієнтами та серверами.

Правила взаємодії між сервером і клієнтом визначаються протоколом обміну. У першу чергу, модель клієнт-серверної взаємодії визначається розподілом обов’язків. Таким чином, можна відокремити такі рівні операцій:

* рівень представлення даних, який відповідає за представлення даних користувачу і введення від нього керуючих команд, тобто цей рівень представляє собою інтерфейс користувача;
* прикладний рівень, на якому здійснюється обробка інформації. Зазвичай на цьому рівні реалізується основна логіка додатку;
* рівень керування даними, цей рівень забезпечує зберігання даних, а також доступ до них.

Зв’язок між клієнтами і сервером реалізовано за типом “один до багатьох”. Ініціатором діалогу з сервером зазвичай є клієнт. Після отримання запиту на послугу від клієнта, сервер вирішує, у який спосіб має бути виконана ця послуга.

Керування програмами і загальними даними відбувається централізовано, що знижує вартість обслуговування і захищає цілісність даних. В той же час, дані клієнтів залишаються персоналізованими і незалежними.

Сервер може виконуватися на тій же машині, що і клієнт, або на іншій машині, підключеній по мережі. Програмне забезпечення «клієнт-сервер» зазвичай приховує місце розташування сервера від клієнтів шляхом перенаправлення запиту на послуги через мережу.

Клієнти і сервер є нежорстко пов'язаними процесами, які обмінюються повідомленнями: запитами на послуги і відповідями на ці запити.

Модульний дизайн програмної платформи надає їй стійкість до відмов, тобто, відмова в якомусь модулі не викликає відмови усього додатку. У такій системі один або більше серверів можуть відмовити без зупинки усієї системи в цілому, до того часу, поки послуги сервера, що відмовив, можуть бути надані з резервного сервера. Інша перевага модульності полягає в тому, що додаток «клієнт-сервер» може автоматично реагувати на підвищення або зниження навантаження на систему, шляхом додавання або відключення серверів.

Клієнти і сервери можуть розгортатися на різних апаратних платформах і різних операційних системах.

Системи можуть масштабуватися як горизонтально (по числу серверів і клієнтів), так і вертикально (по продуктивності і спектру послуг).

Один сервер може надавати послуги безлічі клієнтів одночасно, і регулювати їх доступ до спільно використовуваних ресурсів.

## 2.2 Математична модель

Математичну модель представляє модель функціонування мережі передачі даних, яка використовує конкуруючий доступ до середовища передачі даних на основі комутованого пристрою з обмеженим розміром буфера.

Для розв’язку поставленої задачі необхідно моделювати систему роботи з чергами. Нехай існує n джерел заявок, у кожному джерелі заявки формуються з інтенсивністю , розмір буферу комутатора містить N елементів, обробка заявок здійснюється з інтенсивністю . Припустимо, що при досягненні рівня К елементів у черзі, знову сформована заявка залишається в джерелі до тих пір, поки не з’явиться можливість відправити її в буфер комутатора, при цьому наступна заявка даному джерелі не формується.

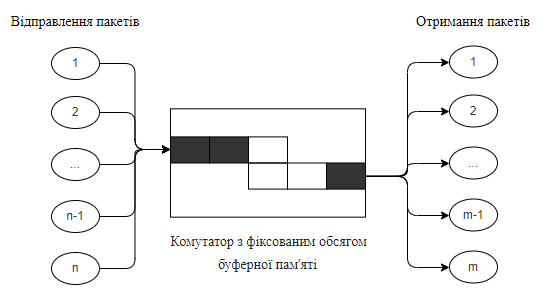


Рисунок 2.1 – Схема мережі передачі даних

Для визначення закону розподілу часу отримання пакета сформуємо вектор, що характеризує стан системи: , де – кількість пакетів у черзі комутатора, , а – кількість джерел, що мають пакети .

Таким чином, за допомогою графу Марковського процесу загибелі та розмноження з безперервним часом можна представити всі стани системи.

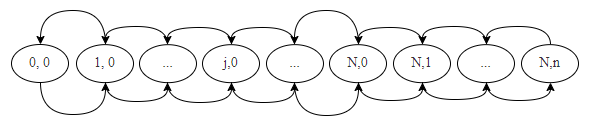


Рисунок 2.2 – Граф станів системи

Математичне опис Марківських процесів зазвичай представляється як систем диференціальних або алгебраїчних рівнянь, вирішення яких, у загальному випадку, одержати у явному вигляді не вдається. Це зумовлює необхідність застосування чисельних методів розв'язання систем диференціальних або рівнянь алгебри.

Система рівнянь Колмогорова для графа матиме вид:

|  |  |
| --- | --- |
|  | (2.1) |

Вирішуючи цю систему рівнянь із початковими умовами , , можна обчислити ймовірності через .

Розглядаються стаціонарні ймовірності станів системи. Якщо , таким чином, тоді . Отже, . Нехай , , тоді ймовірність .

Визначимо з рівняння нормування :

, де .

З урахуванням виділених станів шуканий закон розподілу часу обслуговування заявки можна подати у вигляді:

Таким чином,  і – це умовні закони розподілу. Закон розподілу можна уявити, що:

|  |  |
| --- | --- |
| , | (2.2) |

де – ймовірність того, що в момент приходу пакета в системі знаходилося пакетів, тобто, пакет, що прийшов, зайняв місце в черзі комутатора. – закон розподілу часу обслуговування пакета, розташованого на місці в буфері. Оскільки закон обслуговування комутатора має експоненційний характер, то можна записати у вигляді , тобто бслуговування відбуватиметься за законом Ерланга -го порядку.

Для визначення законів розподілу виділимо з джерел один і визначимо для нього час обслуговування пакета. Для цього введемо наступний вектор , N – розмір заповненого буфера, i – кількість джерел, що мають пакети, крім виділеного, , k – наявність або відсутність пакета у виділеному пристрої , або .

Розглянувши частину графа, представленого на рис. 2.2, починаючи з , можна побудувати новий граф з урахуванням введення нового вектора стану, а також для цього графу скласти рівняння Колмогорова. Якщо скористатися рівністю , то отримаємо рівняння, що збігаються з відповідними рівняннями системи (2.1). Оскільки всі джерела в системі однакові, то для стаціонарних ймовірностей маємо:

*,,.*

Закон розподілу можна записати наступним чином:

|  |  |
| --- | --- |
| *,* | (2.3) |

де – ймовірність того, що в момент надходження пакету від виділеного пристрою в системі вже пристроїв мають пакети; – закон розподілу часу обслуговування пакета, що надійшов -м у систему, коли буфер був зайнятий. Для знаходження даного закону збудуємо новий граф, де стани будуть поглинаючими станами. Склавши систему рівнянь Колмогорова та вирішивши їх щодо відповідних початкових умов, отримаємо, що закон розподілу в цьому випадку набуде вигляду:

Цей закон розподілу характеризує час очікування пакета від моменту надходження в систему до моменту попадання в буфер комутатора на місце . Таким чином, закон розподілення, визначаючий час від моменту попадання пакета в систему до його виходу визначатиметься композицією двох законів:

Для визначення закону розподілу необхідно знайти ймовірності та , що беруть участь у формулах (2.2) та (2.3) відповідно. Ці ймовірності характеризують стан системи до моменту попадання в неї пакета, так само ймовірності умовні, а їх сума повинна дорівнювати 1.

Виразимо ймовірності через відповідні стаціонарні ймовірності станів графа на рисунку 2.2, використовуючи формулу Байєса.

Позначивши , отримаємо та .

Таким чином, закон розподілу можна представити:

*,*

де – ймовірність того, що в черзі є пакет, та жодне джерело не відправляє пакетів, – розмір черги комутатора, – ймовірність того, що буфер заповнений та джерел у системі формують пакети, – закон розподілу, що характеризує час очікування пакета від моменту надходження до системи до моменту надходження у буфер комутатора на місце , – закон Ерланга N-порядку.

Таким чином, розроблена математична модель мережі, що використовує конкуруючий доступ до середовища передачі даних на основі комутатора з обмеженим буфером, що дозволяє визначати ймовірнісні характеристики мережі, часові характеристики та закон розподілу часу обслуговування пакетів.

# ОПИС ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## 3.1 Вибір мови програмування та технологій

Для розробки проекту було використано мову програмування Java. На сьогоднішній день Java є однією з найбільш розповсюджених і найпопулярніших мов програмування. Сильними сторонами цієї мови є висока надійність роботи, так як вона розроблялася як об’єктно-орієнтована мова високого рівня з жорсткою типізацію, а також кросплатформенність, що дозволяє запускати додатки Java на більшості операційних систем. Парадигма ООП надає мові можливість неодноразово розширювати розроблену систему. Об’єктно-орієнтований підхід до написання коду дозволяє оперувати поняттями, взятими із реального життя за допомогою абстракцій. Ще одною перевагою є можливість додавання нових компонентів, без зміни вже існуючих, така властивість називається розширюваністю, а багаторазове використання написаного коду значно скорочує кількість написаного коду.

Для більш швидкої реалізації серверу, а також для дотримання сучасних методів розробки використовується фреймворк Spring. Spring Framework задає архітектуру проекту, а також надає велику кількість стандартних інструментів.

У основі Spring лежать наступні стратегії:

* інверсія управління, яка дозволяє писати незалежні компоненти. Незалежність компонентів надає переваги при розробці додатку в команді, заміні або перенесенні модулей;
* Spring виступає у ролі контейнера для об’єктів, що дозволяє йому управляти життєвим циклом об’єктів;
* слабке зв’язування, яке досягається шляхом інтегрування залежностей через конструктор, а також взаємодії через інтерфейси. Такий підхід дозволяє координувати роботу кожного об’єкта в системі від третьої сторони;
* декларативне програмування через аспекти, що полегшує сприйняття коду;
* використання простих Java об’єктів без необхідності реалізовувати специфічні для фреймворку інтерфейси, а також без наслідування відповідних класів.

Ін’єкція залежностей – це шаблон проектування та архітектурна модель, що описує ситуацію, коли один об’єкт реалізує свій функціонал через інший об’єкт. Об’єкт передає керування створенням необхідних йому залежностей зовнішньому, спеціально призначеному для цього механізму. Наприклад, з’єднання з базою даних передається конструктору об’єкта через аргумент, замість встановлення з’єднання у конструкторі. Внаслідок таких дій отримаємо гнучкість в розробці додатку, тому що це дозволяє відділити об’єкти від реалізації механізмів, котрі він використовує. Spring складається з модулів, які є незалежними один від одного, а також модуля, до якого приєднуються інші модулі.

За допомогою пакетного менеджеру Maven були додані бібліотеки для використання технологій клієнт-серверної комунікації, а саме gRPC, GraphQL, WebSocket. Maven – це фреймворк для автоматизації збірки проектів на основі їх декларативного опису структури в файлах мовою POM. У файлах опису проекту не містяться окремі команди виконання, лише його специфікація. За допомогою завдань, описаних в специфікації Maven обробляє їх використовуючи вбудовані і зовнішні плагіни. Для підключення бібліотек був створений xml файл з декларативним описом необхідних бібліотек, їх версій, а також деталі побудови проекту.

У якості даних було згенеровано JSON файли різних розмірів, у яких містяться масиви даних користувачів у кількості 1, 100, 500, 1000, 5000, 10000, 25000, 50000, 75000 відповідно. JSON-об'єкт – це невпорядкована безліч пар   
“ключ – значення”.

Як значення в JSON можуть бути використані:

* JSON-об'єкт;
* масив;
* число;
* логічні значення;
* null – відсутнє значення;
* символьний рядок.

Для більш точних вимірювань часу отримання даних було прийнято рішення використовувати StopWatch таймер. StopWatch – більш ефективна абстракція, тому що вона розкриває лише відносні значення, а не абсолютні.

Клієнтом для серверу буде Postman. Postman – це API-платформа для створення та використання API. Даний інструмент спрощує кожен крок життєвого циклу API та спрощує співпрацю, щоб ви могли створювати кращі API швидше.

У якості середовища розробки було використано Intelij Idea, даний програмний продукт дає зручні інструменти для розробки додатків та перегляду файлів.

## 3.2 Створення сервісу отримання масиву користувачів

З урахуванням того, що дані користувачів зберігаються у JSON файлах, потрібно реалізувати сервіс, який здатен завантажити даний файл з ресурсів додатку, та перетворити його у масив користувачів. Для цього спочатку необхідно створити клас для представлення даних користувача – UserData. Далі за допомогою ObjectMapper перетворити JSON у масив користувачів. ObjectMapper – це клас у пакеті jackson-databind, що забезпечує функцію зчитування та запису JSON, який може легко перетворювати об'єкти та JSON:

ObjectMapper objectMapper = new ObjectMapper()

.configure(DeserializationFeature.FAIL\_ON\_UNKNOWN\_PROPERTIES, false);

InputStream inputStream = UsersService.class.getResourceAsStream("/static/users1.json");

UserData[] users = objectMapper.readValue(inputStream, UserData[].class);

Масив користувачів буде використовуватися у якості тестових даних для дослідження ефективності технологій.

## 3.3 Створення REST API

Для взаємодії з сервером з архітектурним підходом REST зазвичай використовують наступні методи:

* GET – використовується для запиту вмісту заданого ресурсу, або для старту якогось процесу. При використанні цього метода можна використовувати параметри, які додаються до кінця посилання після знаку `?`, параметри розділяються знаком `&`;
* POST – використовується для передачі даних користувача вказаній адресі. Використовується для додавання нових об’єктів/ресурсів;
* PUT – використовується для завантаження вмісту запита на вказаний URI.

Зазвичай використовується для оновлення об’єктів. Різниця з POST полягає в тому, що ресурс в випадку POST запиту буде оброблювати дані, передані клієнтом, в той час як з методом PUT клієнт передбачає, що завантажуваний вміст вже відповідає тому, що знаходиться за даним URI ресурсу;

* DELETE – слугує методом для видалення даних.

За допомогою Spring Framework створимо клас-контролер DemoRestController, який буде приймати HTTP-запити, та передавати інформацію з запиту у клас-компонент DemoRestComponent. Контролер містить метод loadUsers з анотацією @GetMapping("load-users"), яка буде виконуватися у разі отримання запиту від клієнтів:

private DemoRestComponent demoRestComponent;

@GetMapping("load-users")

public List<UserData> loadUsers() {

return demoRestComponent.loadUsers();

}

Зазвичай контролер не містить логіки обробки даних, а делегує цю роботу іншому прошарку серверу – компоненту, де будуть виконуватись певні дії над даними, в залежності від методу та параметрів. Дані будуть представлені у вигляді екземплярів класу UserData, які зберігаються у JSON файлах у ресурсах. Після того, як DemoRestComponent виконає роботу, результат виконання функції буде передано до контролеру, який поверне його клієнту у відповідь.

При запуску програми, Spring за замовчуванням, буде використовувати стандартний порт 8080, тому HTTP-запити необхідно виконувати використовуючи адресу localhost:8080.

У результаті запуску додатку ми отримаємо мікросервіс, який містить вхідну точку, за допомогою якої можна отримати інформацію про користувачів:

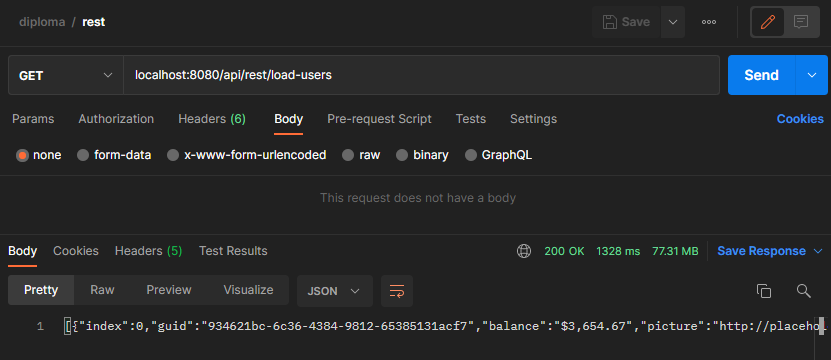


Рисунок 3.1 – Отримання даних від RESTful API

Статус 200 повернений від серверу означає, що дані були завантажені успішно, їх можна побачити нижче у вигляді JSON.

## 3.4 Створення gRPC серверу

Перед початком створення gRPC серверу необхідно створити .proto файл, який буде містити інформацію про сервіси та формат повідомлень. Після компіляції gRPC сформує необхідні класи для створення серверу. gRPC використовує протокол HTTP версії 2.0, який має бінарний формат, тому є необхідність перетворення об’єктів у бінарний код. Для більш ефективного перетворення об’єктів у масив байтів, та навпаки використовується Protobuf. Оскільки gRPC використовує бібліотеку Protobuf, усі запити та відповіді повинні бути описані у файлі. За допомогою синтаксу proto3 був створений .proto файл (додаток Б).

Класи генеруються за допомогою опису сервісу та формату повідомлень у .proto файлі, після компіляції класи стають доступними для використання.

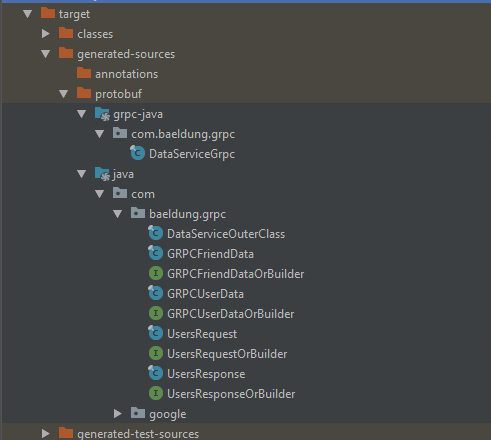


Рисунок 3.2 – Згенеровані файли після компіляції

Використовуючи згенеровані класи необхідно створити сервіс, що буде відповідати за обробку gRPC запиту та слугувати вхідною точкою. Даний сервіс повинен реалізовувати згенерований інтерфейс DataServiceImplBase, а саме метод loadUsers, що був описаний в .proto файлі:

public void loadUsers(UsersRequest request, StreamObserver<UsersResponse> responseObserver) {

System.out.println("Request received from client:\n" + request);

List<UserData> users = usersService.loadUsers();

List<GRPCUserData> grpcUserDataList = users.stream()

.map(UsersConverter::toGRPCData)

.collect(Collectors.toList());

UsersResponse response = UsersResponse.newBuilder()

.addAllUsers(grpcUserDataList)

.build();

responseObserver.onNext(response);

responseObserver.onCompleted();

}

Після створення сервісу необхідно налаштувати gRPC сервер, та додати сервіс:

Server server = ServerBuilder.forPort(8081)

.addService(new GrpcDemoService(new UsersService()))

.maxInboundMessageSize(100 \* 1024 \* 1024)

.build();

try {

log.info("[gRPC]: starting server...");

server.start();

log.info("[gRPC]: server started!");

server.awaitTermination();

} catch (Exception ex) {

log.info("[gRPC]: server throw error!", ex);

}

## 3.5 Створення GraphQL серверу

Перед створенням серверу був створений файл users.graphql (додаток В). Даний файл містить інформацію про методи та поля об’єктів. Далі необхідно описати схему з використанням users.graphql:

URL url = Resources.getResource("graphql/users.graphql");

String sdl = Resources.toString(url, Charsets.UTF\_8);

GraphQLSchema graphQLSchema = buildSchema(sdl);

Метод buildSchema використовується для зв’язування схеми з сервісом отримання даних, такий сервіс повинен реалізовувати інтерфейс DataFetcher, а саме метод get:

public List<UserData> get(DataFetchingEnvironment environment) throws Exception {

return usersService.loadUsers();

}

Після запуску додатку, Spring за вказаними налаштуваннями створить вихідні точки GraphQL, за допомогою Postman можна створити запит для перевірки роботи методу (рис. 3.3). При використанні GraphQL необхідно обов’язково вказувати поля, які клієнт бажає отримати, навіть якщо клієнтові необхідні усі поля.

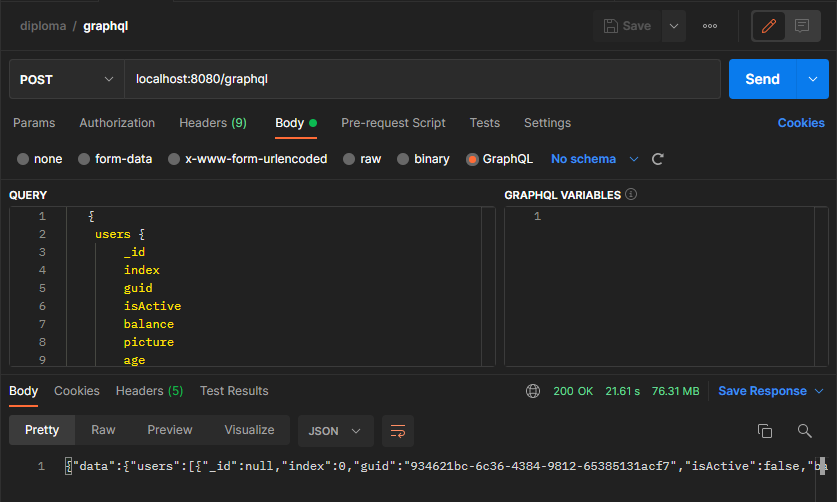


Рисунок 3.3 – Отримання даних від GraphQL серверу

## 3.6 Створення WebSocket серверу

Для створення WebSocket серверу необхідно створити конфігурацію з анотацією @EnableWebSocketMessageBroker, конфігурація повинна наслідувати клас AbstractWebSocketMessageBrokerConfigurer та перевизначати його методи:

@Override

public void configureMessageBroker(MessageBrokerRegistry config) {

config.enableSimpleBroker("/topic", "/queue");

config.setApplicationDestinationPrefixes("/app");

config.setUserDestinationPrefix("/user");

}

Перший метод конфігурує простий брокер повідомлень у пам'яті з однією адресою. Адреси з префіксом “/app” призначені для повідомлень, оброблюваних методами анотації @MessageMapping.

@Override

public void registerStompEndpoints(StompEndpointRegistry registry) {

registry.addEndpoint("/ws");

registry.addEndpoint("/ws").withSockJS();

}

Другий метод реєструє кінцеву точку STOMP “/ws”. Ця кінцева точка використовуватиметься клієнтами для підключення до STOMP-сервера. Тут також включається резервний SockJS, який використовуватиметься, якщо WebSocket буде недоступний.

Зазвичай веб-сокети використовуються для комунікації повідомленнями невеликого розміру. Так як планується передавати великий обсяг даних, то необхідно збільшити ліміти розміру повідомлення, буферу, а також час передачі даних:

@Override

public void configureWebSocketTransport(WebSocketTransportRegistration registration) {

registration.setMessageSizeLimit(100 \* 1024 \* 1024);

registration.setSendTimeLimit(20 \* 10000);

registration.setSendBufferSizeLimit(3\* 512 \* 1024);

}

Після створення серверу створимо необхідно створити контролер, що буде слідкувати за вхідними повідомленнями:

@MessageMapping("/users")

@SendTo("/topic/users")

public List<UserData> send() {

return usersService.loadUsers();

}

За допомогою анотації @MessageMapping ми налаштовуємо, що при надсиланні повідомлення за шляхом “/app/users” викликається метод send(). До вхідної точки додається налаштований раніше префікс “/app”.

Для початку роботи з сервером необхідно відправити повідомлення для створення з’єднання (рис. 3.4).

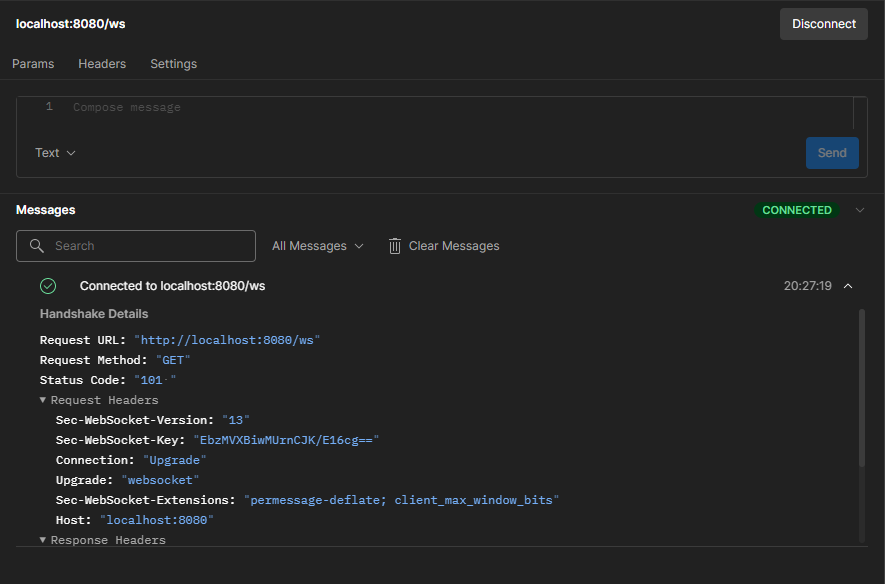


Рисунок 3.4 – Створення WebSocket з’єднання за допомогою Postman

Після успішного з’єднання можна відправити повідомлення, щоб створити підписку на вхідну точку, після цього сервер буде розуміти до якого саме клієнту відправляти дані. Після цього можна створити повідомлення для серверу, які дані необхідні клієнтові. Таким чином, при отримані повідомлення сервером буде ініційоване завантаження користувачів, а також відправлення повідомлень до всіх клієнтів, підписаних на даний метод.

# АНАЛІЗ РЕЗУЛЬТАТІВ

Після створення додатку були проведені заміри часу отримання даних від серверу. Кожний файл був завантажений 10 разів за допомогою розглянутих технологій для більш точної оцінки результатів вимірів. Отримані значення були використані для розрахунку середнього часу отримання даних від серверу (табл. 4.1).

Таблиця 4.1 – Середні значення часу отримання даних від серверу

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Обсяг даних (кб) | Середній час завантаження даних (мс) | | | |
|  |
| REST | gRPC | GraphQL | WebSocket |  |
| 1,48 | 3,3 | 3,5 | 9 | 7,2 |  |
| 137 | 8,7 | 11,3 | 39,5 | 11,4 |  |
| 681 | 21,1 | 18,9 | 154,3 | 37,7 |  |
| 1348 | 29 | 31 | 295,7 | 59,6 |  |
| 6742 | 92 | 84,4 | 1433,9 | 233,5 |  |
| 13482 | 180,7 | 137 | 2871 | 458,8 |  |
| 33716 | 437,6 | 359,9 | 7372 | 1098 |  |
| 67496 | 866,7 | 636,6 | 13954 | 2173 |  |
| 101180 | 1303,2 | 1022,7 | 21488 | 3212,2 |  |

На основі даних з таблиці 4.1 було побудовано діаграму залежності часу передачі даних від обсягу даних (рис. 4.1). На цій діаграмі можна побачити, що під час пересилання невеликого обсягу даних різниця у часі майже не залежить від вибору технології. Але при збільшенні обсягів даних можна помітити, що технологія graphQL працює дуже повільно. Найшвидше були отримані дані, за допомогою gRPC, у той час як REST працює трохи повільніше. Технологія WebSocket займає третє місце, що також є непоганим результатом у порівнянні з graphQL.

Рисунок 4.1 – Залежність часу пересилання даних від їх обсягу

Також було розраховано стандартні відхилення часу для кожного обсягу даних (табл. 4.2) за формулою:

,

де – середній час отримання даних, n – кількість елементів вибірки.

Таблиця 4.2 – Середньоквадратичне відхилення вибірки

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Обсяг даних (кб) | Стандартне відхилення мс | | | |
| REST | gRPC | GraphQL | WebSocket |
| 1,48 | 0,7 | 1,9 | 3,6 | 5,3 |
| 137 | 2,5 | 3,3 | 5,5 | 3,5 |
| 681 | 5,6 | 6 | 7,7 | 7 |

*Продовження табл 4.2*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1348 | 12 | 8,4 | 9,3 | 5,1 |
| 6742 | 6,3 | 26,4 | 41,6 | 9,8 |
| 13482 | 17,3 | 44,6 | 82,1 | 31,7 |
| 33716 | 8,1 | 57,4 | 265,9 | 14,4 |
| 67496 | 21 | 111 | 270,4 | 39 |
| 101180 | 27 | 107,5 | 250 | 55,4 |

Використовуючи дані з таблиці було отримано діаграму (рис. 4.2), яка репрезентує залежність стандартного відхилення часу отримання даних від обсягу даних. З даного графіку можна побачити, що graphQL має найбільше середньоквадратичне відхилення, в той час як інші технології мають наближені значення один до одного.

Рисунок 4.2 – Залежність стандартного відхилення часу від обсягу даних

За допомогою середніх значень часу та обсягу даних було розраховано середню швидкість передачі даних за формулою:

, де – обсяг даних, – середній час отримання даних.

Таблиця 4.3 – Середня швидкість завантаження даних

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Обсяг даних (кб) | Середня швидкість завантаження даних кб/с | | | |
| REST | gRPC | GraphQL | WebSocket |
| 1,48 | 448,5 | 422,9 | 164,4 | 205,6 |
| 137 | 15747,1 | 12123,9 | 3468,4 | 12017,5 |
| 681 | 32274,9 | 36031,8 | 4413,5 | 18063,7 |
| 1348 | 46482,8 | 43483,9 | 4558,7 | 22617,5 |
| 6742 | 73282,6 | 79881,5 | 4701,9 | 28873,7 |
| 13482 | 74609,9 | 98408,8 | 4696 | 29385,4 |
| 33716 | 77047,5 | 93681,6 | 4573,5 | 30706,7 |
| 67496 | 77877,0 | 106025,8 | 4837,0 | 31061,2 |
| 101180 | 77639,7 | 98934,2 | 4708,7 | 31498,7 |

З отриманих даних було отримано графік залежності швидкості передачі даних від обсягу даних (рис. 4.3).

Рисунок 4.3 – Залежність швидкості передачі даних від їх обсягу

Можна зробити висновок, що швидкість передачі даних найбільша у gRPC та REST, найменша – при використанні технології graphQL або WebSocket, останні технології працюють на основі протоколу HTTP1, що знижує швидкість передачі даних, в додаток до цього graphQL витрачає багато часу на перетворення даних, відповідно до запитаних полів.

На рис.4.3 також можна побачити, що усі розглянуті технології мають свій ліміт швидкості. Тому можна оцінити максимальну швидкість передачі даних для кожної з досліджених технологій.

Максимальна швидкість з використанням REST складає 77877 кб/с, gRPC – 106025 кб/с, GraphQL – 4837 кб/с, WebSocket – 31498 кб/с.

На мою думку дані технології варто розглядати не лише за швидкістю передачі даних, а також за легкістю використання, рівнем володіння технологіями, додатковими інструментами. Вибір повністю залежить від поставленої задачі, архітектури проекту та рівня знань інженерів.

Як правило, для спрощення і швидкого виведення додатка на ринок, використання Restful API є найбільш простим.

Якщо продуктивність є проблемою або частина вашої системи має велику кількість клієнтів, а корисне навантаження відносно велике, ви можете реалізувати gRPC і розробити специфікації .proto файлів для досягнення високої продуктивності.

GraphQL варто розглядати у випадку, коли у проекті існує багато API і постійно додаються нові, або якщо необхідно отримувати лише вказані поля з великих об’єктів, але необхідно пам’ятати що дана технологія працює на основі протоколу HTTP1, а також те, що GraphQL витрачає багато часу на обробку схеми, тому її ефективність буде помітно менше у порівнянні з REST та gRPC.

Веб-сокети ідеально підійдуть якщо необхідна двонапрямлена комунікація між клієнтом та сервером. Веб-сокети часто демонструють як технологію передачі даних при розробці чатів, а також додатків де необхідно постійно відправляти клієнтові повідомлення з оновленою інформацією, наприклад у CRM системах. Так як веб-сокети працюють на основі TCP з’єднання, то виникають деякі складності із масштабуванням системи.

Для більш специфічних задач, наприклад для реалізації надійного обміну даними на дуже високій швидкості з низькою затримкою, можна використати протоколи більш низького рівня – TCP або UDP.

# ВИСНОВКИ

За результатами виконання кваліфікаційної роботи можна зробити такі висновки:

* проведено аналітичний огляд літературних джерел за обраною темою, визначено найбільш перспективні технології обміну даними у клієнт-серверних системах;
* розроблено і досліджено математичну модель клієнт-серверної взаємодії;
* побудовано інформаційну модель для створення веб-серверу;
* розроблено веб-сервер з використанням розглянутих технологій обміну
* даними, мови програмування Java і фреймворку Spring;
* проведено експериментальне дослідження клієнт-серверного обміну з використанням пакетів даних різного розміру;
* отримано і проаналізовано результати дослідження, побудовано відповідні таблиці і графіки;
* зроблено висновки щодо ефективності розглянутих технологій.

Результати роботи можуть бути застосовані при проектуванні архітектури проекту, а саме клієнт-серверної комунікації.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sharma, S. Modern API Development with Spring and Spring Boot : Design highly scalable and maintainable APIs with REST, gRPC, GraphQL, and the reactive paradigm / S. Sharma. – Birmingham, UK : Packt Publishing Ltd, 2021. – 582 с.
2. Walls, C. Spring in Action / C. Walls. – Shelter Island, NY : Manning Publications, 2018. – 520 с.
3. Кирпичников, А.П. Методы прикладной теории массового обслуживания / А.П. Кирпичников. – Казань : Казанский ун-т, 1999. – 2011 с.
4. Porcello, E. Learning GraphQL: Declarative Data Fetching for Modern Web Apps / E. Porcello, A. Banks. – Cebastopol, CA : O`Reilly Media, 2018. – 181 с.
5. Коваленко, И.Н. Теория массового обслуживания / И.Н. Коваленко. – Москва, Россия : Либроком, 2012. – 304 с.
6. A query language for your API [Електронний ресурс] // graphql.org. Режим доступу: <https://graphql.org>.
7. Вентцель, Е.С. Теория вероятностей / Е.С. Вентцель. – Москва : Наука, 1969. – 576 с.
8. Lombardi, A. WebSocket: Lightweight Client-Server Communications / A. Lombardi. – Sebastopol, CA : O`Reilly Media, 2015. – 125 с.
9. Richardson, С. Microservices Patterns: With examples in Java / С. Richardson. – Manning Publications : Shelter Island, NY, 2018. – 520 с.
10. Сказ о том, как разбирался в gRPC и Websocket [Електронний ресурс] // dev.by. Режим доступу: <https://dev.by/blogs/godel-technologies-europe/articles/skaz-o-tom-kak-razbiralsya-v-grpc-i-websocket>.
11. Костробій, П.П. Елементи теорії випадкових процесів / П.П. Костробій, Б.М. Маркович, М.В. Токарчук. – Львів : Растр-7, 2015. – 180 с.
12. When to Use What: REST, GraphQL, Webhooks, & gRPC [Електронний ресурс] // nordicapis.com. Режим доступу: <https://nordicapis.com/when-to-use-what-rest-graphql-webhooks-grpc/>.
13. Гардинер, К.В. Стохастические методы в естественных науках / К.В. Гардинер. – СССР : Мир, 1986. – 528 с.
14. What Is Microservice Architecture? Microservices Explained [Електронний ресурс] // bmc.com. Режим доступу :

<https://www.bmc.com/blogs/microservices-architecture>.

1. Ньюмен, С. Создание микросервисов / С. Ньюмен. – СПб : Питер, 2018. – 304 с.
2. Ludin, S. Learning HTTP/2: A Practical Guide for Beginners / S. Ludin, J. Garza. – Sebastopol, CA : O’Reilly Media, 2017. – 156 с.
3. Абрамов, Г. В. Проектирование информационных систем / Г. В. Абрамов, И. Е. Медведкова, Л. А. Коробова. – Воронеж : Воронежский государственный университет инженерных технологий, 2012. – 172 с.
4. An Architect's guide to APIs: SOAP, REST, GraphQL, and gRPC Explained [Електронний ресурс] // redhat.com. Режим доступу: <https://www.redhat.com/architect/apis-soap-rest-graphql-grpc>.
5. Parashu, R. Cloud Computing. Strategies for Improving the Performance and Security / R. Parashu, O. Priyanka. – Norderstedt, Germany : Grin Verlag, 2019. – 120 с.
6. Томашевський, В.М. Моделювання систем / В.М. Томашевський. – Київ : Видавнича група BHV, 2005. – 352 с.

# ДОДАТКИ

Додаток А

Лістинг програми

// UserData.java

@Data

@Accessors(chain = true)

public class UserData implements Serializable {

@JsonProperty("\_id")

private String id;

private int index;

private String guid;

private boolean isActive;

private String balance;

private String picture;

private int age;

private String eyeColor;

private String name;

private String gender;

private String company;

private String email;

private String phone;

private String address;

private String about;

private String registered;

private Double latitude;

private Double longitude;

private List<String> tags;

private List<FriendRef> friends;

private String greeting;

private String favoriteFruit;

}

// FriendRef.java

@Data

@Accessors(chain = true)

public class FriendRef implements Serializable {

private int id;

private String name;

}

// UsersService.java

@Slf4j

@Service

public class UsersService {

private ObjectMapper objectMapper = new ObjectMapper()

.configure(DeserializationFeature.FAIL\_ON\_UNKNOWN\_PROPERTIES, false);

public List<UserData> loadUsers() {

try {

InputStream inputStream = UsersService.class.getResourceAsStream("/static/users75000.json");

UserData[] users = objectMapper.readValue(inputStream, UserData[].class);

return Arrays.asList(users);

} catch (Exception ex) {

log.error("Can't load events from file", ex);

return new LinkedList<>();

}

}

}

// DemoRestController.java

@RestController

@AllArgsConstructor

@RequestMapping("api/rest")

public class DemoRestController {

private DemoRestComponent demoRestComponent;

@GetMapping("load-users")

public List<UserData> loadUsers() {

return demoRestComponent.loadUsers();

}

}

// DemoRestComponent.java

@Component

@AllArgsConstructor

public class DemoRestComponent {

private UsersService usersService;

public List<UserData> loadUsers() {

return usersService.loadUsers();

}

}

// DemoRestClient.java

@Slf4j

public class DemoRestClient {

public static void main(String[] args) {

RestTemplate restTemplate = new RestTemplate();

StopWatch timer = new StopWatch();

timer.start();

List<UserData> users = restTemplate.exchange("http://localhost:8080/api/rest/load-users", HttpMethod.GET,

new HttpEntity<>(new HttpHeaders()),

new ParameterizedTypeReference<List<UserData>>() {

}

).getBody();

timer.stop();

log.info("[REST]: time: " + timer.getLastTaskTimeMillis());

}

}

// GrpcServer.java

@Slf4j

public class GrpcServer {

public static void main(String[] args) {

Server server = ServerBuilder.forPort(8081)

.addService(new GrpcDemoService(new UsersService()))

.maxInboundMessageSize(100 \* 1024 \* 1024)

.build();

try {

log.info("[gRPC]: starting server...");

server.start();

log.info("[gRPC]: server started!");

server.awaitTermination();

} catch (Exception ex) {

log.info("[gRPC]: server throw error!", ex);

}

}

}

// GrpcDemoService.java

@Service

@AllArgsConstructor

public class GrpcDemoService extends DataServiceGrpc.DataServiceImplBase {

private UsersService usersService;

@Override

public void loadUsers(UsersRequest request, StreamObserver<UsersResponse> responseObserver) {

System.out.println("Request received from client:\n" + request);

List<UserData> users = usersService.loadUsers();

List<GRPCUserData> grpcUserDataList = users.stream()

.map(UsersConverter::toGRPCData)

.collect(Collectors.toList());

UsersResponse response = UsersResponse.newBuilder()

.addAllUsers(grpcUserDataList)

.build();

responseObserver.onNext(response);

responseObserver.onCompleted();

}

}

// UsersConverter.java

public class UsersConverter {

public static GRPCUserData toGRPCData(UserData userData) {

return GRPCUserData.newBuilder()

.setId(userData.getId())

.setIndex(userData.getIndex())

.setGuid(userData.getGuid())

.setIsActive(userData.isActive())

.setBalance(userData.getBalance())

.setPicture(userData.getPicture())

.setAge(userData.getAge())

.setEyeColor(userData.getEyeColor())

.setName(userData.getName())

.setGender(userData.getGender())

.setCompany(userData.getCompany())

.setEmail(userData.getEmail())

.setPhone(userData.getPhone())

.setAddress(userData.getAddress())

.setAbout(userData.getAbout())

.setRegistered(userData.getRegistered())

.setLatitude(userData.getLatitude())

.setLongitude(userData.getLongitude())

.addAllTags(new LinkedList<>(userData.getTags()))

.addAllFriends(userData.getFriends().stream()

.map(UsersConverter::toGRPCData)

.collect(Collectors.toList()))

.build();

}

public static GRPCFriendData toGRPCData(FriendRef friendRef) {

return GRPCFriendData.newBuilder()

.setId(friendRef.getId())

.setName(friendRef.getName())

.build();

}

public static UserData toData(GRPCUserData grpcUserData) {

return new UserData()

.setId(grpcUserData.getId())

.setIndex(grpcUserData.getIndex())

.setGuid(grpcUserData.getGuid())

.setActive(grpcUserData.getIsActive())

.setBalance(grpcUserData.getBalance())

.setPicture(grpcUserData.getPicture())

.setAge(grpcUserData.getAge())

.setEyeColor(grpcUserData.getEyeColor())

.setName(grpcUserData.getName())

.setGender(grpcUserData.getGender())

.setCompany(grpcUserData.getCompany())

.setEmail(grpcUserData.getEmail())

.setPhone(grpcUserData.getPhone())

.setAddress(grpcUserData.getAddress())

.setAbout(grpcUserData.getAbout())

.setRegistered(grpcUserData.getRegistered())

.setLatitude(grpcUserData.getLatitude())

.setLongitude(grpcUserData.getLongitude())

.setTags(grpcUserData.getTagsList())

.setFriends(grpcUserData.getFriendsList().stream()

.map(UsersConverter::toData)

.collect(Collectors.toList()));

}

public static FriendRef toData(GRPCFriendData grpcFriendData) {

return new FriendRef()

.setId(grpcFriendData.getId())

.setName(grpcFriendData.getName());

}

}

// GrpcClient.java

@Slf4j

public class GrpcClient {

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

ManagedChannel channel = ManagedChannelBuilder.forAddress("localhost", 8081)

.usePlaintext()

.maxInboundMessageSize(100 \* 1024 \* 1024)

.build();

DataServiceGrpc.DataServiceBlockingStub stub = DataServiceGrpc.newBlockingStub(channel);

StopWatch timer = new StopWatch();

timer.start();

UsersResponse usersResponse = stub.loadUsers(UsersRequest.newBuilder().build());

List<UserData> users = usersResponse.getUsersList().stream()

.map(UsersConverter::toData)

.collect(Collectors.toList());

timer.stop();

log.info("[gRPC]: time: " + timer.getLastTaskTimeMillis());

channel.shutdown();

}

}

// GraphqlProvider.java

@Component

@AllArgsConstructor

public class GraphqlProvider {

private GraphqlUsersFetcher graphqlUsersFetcher;

@Bean

public GraphQL graphQL() throws IOException {

URL url = Resources.getResource("graphql/users.graphql");

String sdl = Resources.toString(url, Charsets.UTF\_8);

GraphQLSchema graphQLSchema = buildSchema(sdl);

return GraphQL.newGraphQL(graphQLSchema).build();

}

private GraphQLSchema buildSchema(String sdl) {

TypeDefinitionRegistry typeRegistry = new SchemaParser().parse(sdl);

RuntimeWiring runtimeWiring = buildWiring();

SchemaGenerator schemaGenerator = new SchemaGenerator();

return schemaGenerator.makeExecutableSchema(typeRegistry, runtimeWiring);

}

private RuntimeWiring buildWiring() {

return RuntimeWiring.newRuntimeWiring()

.type(newTypeWiring("Query")

.dataFetcher("users", graphqlUsersFetcher))

.build();

}

}

// GraphqlUsersFetcher.class

@Slf4j

@Component

@AllArgsConstructor

public class GraphqlUsersFetcher implements DataFetcher<List<UserData>> {

private UsersService usersService;

@Override

public List<UserData> get(DataFetchingEnvironment dataFetchingEnvironment) throws Exception {

return usersService.loadUsers();

}

}

// WebSocketConfig.class

@Configuration

@EnableWebSocketMessageBroker

public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer {

@Override

public void configureMessageBroker(MessageBrokerRegistry config) {

config.enableSimpleBroker("/topic", "/queue");

config.setApplicationDestinationPrefixes("/app");

config.setUserDestinationPrefix("/user");

}

@Override

public void registerStompEndpoints(StompEndpointRegistry registry) {

registry.addEndpoint("/ws");

registry.addEndpoint("/ws").withSockJS();

}

@Override

public void configureWebSocketTransport(WebSocketTransportRegistration registration) {

registration.setMessageSizeLimit(100 \* 1024 \* 1024); // default : 64 \* 1024

registration.setSendTimeLimit(20 \* 10000); // default : 10 \* 10000

registration.setSendBufferSizeLimit(3\* 512 \* 1024); // default : 512 \* 1024

}

}

// WebSocketController.java

@Slf4j

@AllArgsConstructor

@RestController

public class WebSocketController {

private UsersService usersService;

@MessageMapping("/users")

@SendTo("/topic/users")

public List<UserData> send() {

return usersService.loadUsers();

}

}

// WebSocketDemoClient.java

@Slf4j

@Component

public class WebSocketDemoClient {

@PostConstruct

public void test() {

WebSocketContainer container = ContainerProvider.getWebSocketContainer();

container.setDefaultMaxBinaryMessageBufferSize(100 \* 1024 \* 1024);

container.setDefaultMaxTextMessageBufferSize(100 \* 1024 \* 1024);

WebSocketClient transport = new StandardWebSocketClient(container);

WebSocketStompClient stompClient = new WebSocketStompClient(transport);

stompClient.setMessageConverter(new MappingJackson2MessageConverter());

stompClient.setInboundMessageSizeLimit(Integer.MAX\_VALUE);

StompSessionHandler sessionHandler = new StompSessionHandler() {

StopWatch timer = new StopWatch();

StompSession session;

@Override

public void afterConnected(StompSession stompSession, StompHeaders stompHeaders) {

stompSession.subscribe("/topic/users", this);

timer.start();

session = stompSession;

stompSession.send("/app/users", "");

}

@Override

public void handleException(StompSession stompSession, StompCommand stompCommand, StompHeaders stompHeaders, byte[] bytes, Throwable throwable) {

}

@Override

public void handleTransportError(StompSession stompSession, Throwable throwable) {

}

@Override

public Type getPayloadType(StompHeaders stompHeaders) {

return UserData[].class;

}

@Override

public void handleFrame(StompHeaders stompHeaders, Object o) {

UserData[] users = (UserData[]) o;

timer.stop();

log.info("[WebSocket]: time " + timer.getLastTaskTimeMillis());

}

};

String url = "ws://localhost:8080/ws";

stompClient.connect(url, sessionHandler);

}

}

//pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.6.0</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

<groupId>com.example</groupId>

<artifactId>demo</artifactId>

<version>1.0-SNAPSHOT</version>

<name>demo</name>

<description>Demo project for Spring Boot</description>

<properties>

<graphql.version>11.0.0</graphql.version>

<graphql-java-tools.version>6.2.0</graphql-java-tools.version>

<java.version>11</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-websocket</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-configuration-processor</artifactId>

<optional>true</optional>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.18.10</version>

<optional>true</optional>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>io.grpc</groupId>

<artifactId>grpc-netty</artifactId>

<version>1.16.1</version>

</dependency>

<dependency>

<groupId>io.grpc</groupId>

<artifactId>grpc-protobuf</artifactId>

<version>1.16.1</version>

</dependency>

<dependency>

<groupId>io.grpc</groupId>

<artifactId>grpc-stub</artifactId>

<version>1.16.1</version>

</dependency>

<dependency>

<groupId>com.graphql-java</groupId>

<artifactId>graphql-java</artifactId>

<version>11.0</version>

</dependency>

<dependency>

<groupId>com.graphql-java</groupId>

<artifactId>graphql-java-spring-boot-starter-webmvc</artifactId>

<version>1.0</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-websocket</artifactId>

<version>5.2.2.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-messaging</artifactId>

<version>5.2.2.RELEASE</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-core</artifactId>

<version>2.10.2</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.10.2</version>

</dependency>

</dependencies>

<build>

<extensions>

<extension>

<groupId>kr.motd.maven</groupId>

<artifactId>os-maven-plugin</artifactId>

<version>1.6.1</version>

</extension>

</extensions>

<plugins>

<plugin>

<groupId>org.xolstice.maven.plugins</groupId>

<artifactId>protobuf-maven-plugin</artifactId>

<version>0.6.1</version>

<configuration>

<protocArtifact>

com.google.protobuf:protoc:3.3.0:exe:${os.detected.classifier}

</protocArtifact>

<pluginId>grpc-java</pluginId>

<pluginArtifact>

io.grpc:protoc-gen-grpc-java:1.4.0:exe:${os.detected.classifier}

</pluginArtifact>

</configuration>

<executions>

<execution>

<goals>

<goal>compile</goal>

<goal>compile-custom</goal>

</goals>

</execution>

</executions>

</plugin>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<excludes>

<exclude>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

</exclude>

</excludes>

</configuration>

</plugin>

</plugins>

</build>

</project>

Додаток Б

.proto файл

// DataService.proto

syntax = "proto3";

option java\_multiple\_files = true;

package com.baeldung.grpc;

message UsersRequest {

}

message UsersResponse {

repeated GRPCUserData users = 1;

}

message GRPCUserData {

string \_id = 1;

int32 index = 2;

string guid = 3;

bool isActive = 4;

string balance = 5;

string picture = 6;

int32 age = 7;

string eyeColor = 8;

string name = 9;

string gender = 10;

string company = 11;

string email = 12;

string phone = 13;

string address = 14;

string about = 15;

string registered = 16;

double latitude = 17;

double longitude = 18;

repeated string tags = 19;

repeated GRPCFriendData friends = 20;

string greeting = 21;

string favoriteFruit = 22;

}

message GRPCFriendData {

int32 \_id = 1;

string name = 2;

}

service DataService {

rpc loadUsers(UsersRequest) returns (UsersResponse);

}

Додаток В

.graphql файл

//users.graphql

type Query {

users: [User]

}

type User {

\_id: ID

index: Int

guid: String

isActive: Boolean

balance: String

picture: String

age: Int

eyeColor: String

name: String

gender: String

company: String

email: String

phone: String

address: String

about: String

registered: String

latitude: Float

longitude: Float

tags: [String]

friends: [Friend]

greeting: String

favoriteFruit: String

}

type Friend {

id: String

name: String

}

Додаток Г

Отримані результати

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Дані 1.48 КБ (1 об'єкт) | | | | | | | | | | |
| Технологія | Час (мс) | | | | | | | | | |
| REST | 3 | 3 | 3 | 3 | 5 | 3 | 3 | 3 | 3 | 4 |
| gRPC | 7 | 2 | 3 | 2 | 5 | 4 | 2 | 2 | 2 | 6 |
| GraphQL | 8 | 8 | 18 | 9 | 6 | 7 | 12 | 6 | 9 | 7 |
| WebSocket | 17 | 5 | 4 | 14 | 4 | 3 | 6 | 13 | 3 | 3 |
| Дані 137 КБ (100 об'єктів) | | | | | | | | | | |
| Технологія | Час (мс) | | | | | | | | | |
| REST | 13 | 12 | 10 | 8 | 9 | 7 | 7 | 7 | 5 | 9 |
| gRPC | 18 | 13 | 13 | 10 | 14 | 11 | 8 | 8 | 7 | 11 |
| GraphQL | 47 | 43 | 49 | 39 | 36 | 37 | 32 | 37 | 34 | 41 |
| WebSocket | 12 | 16 | 9 | 9 | 17 | 8 | 8 | 8 | 15 | 12 |
| Дані 681 КБ (500 об'єктів) | | | | | | | | | | |
| Технологія | Час (мс) | | | | | | | | | |
| REST | 34 | 21 | 20 | 20 | 27 | 16 | 19 | 20 | 15 | 19 |
| gRPC | 35 | 18 | 19 | 17 | 16 | 15 | 16 | 17 | 15 | 21 |
| GraphQL | 166 | 147 | 158 | 149 | 151 | 150 | 147 | 152 | 169 | 154 |
| WebSocket | 43 | 48 | 42 | 35 | 31 | 30 | 36 | 31 | 33 | 48 |
| Дані 1348 КБ (1000 об'єктів) | | | | | | | | | | |
| Технологія | Час (мс) | | | | | | | | | |
| REST | 42 | 44 | 27 | 26 | 21 | 19 | 20 | 20 | 20 | 51 |
| gRPC | 44 | 36 | 29 | 34 | 28 | 31 | 23 | 20 | 22 | 43 |
| GraphQL | 307 | 293 | 296 | 307 | 302 | 297 | 290 | 292 | 298 | 275 |
| WebSocket | 72 | 65 | 59 | 58 | 57 | 59 | 56 | 57 | 55 | 58 |
| Дані 6742 КБ (5000 об'єктів) | | | | | | | | | | |
| Технологія | Час (мс) | | | | | | | | | |
| REST | 103 | 90 | 96 | 89 | 87 | 88 | 87 | 92 | 86 | 102 |
| gRPC | 139 | 111 | 94 | 72 | 65 | 65 | 68 | 63 | 63 | 104 |
| GraphQL | 1485 | 1396 | 1442 | 1414 | 1411 | 1433 | 1405 | 1503 | 1376 | 1474 |
| WebSocket | 256 | 228 | 232 | 231 | 237 | 232 | 231 | 239 | 217 | 232 |
| Дані 13482 КБ (10000 об'єктів) | | | | | | | | | | |
| Технологія | Час (мс) | | | | | | | | | |
| REST | 226 | 176 | 170 | 173 | 181 | 183 | 176 | 167 | 167 | 188 |
| gRPC | 241 | 166 | 133 | 116 | 105 | 109 | 104 | 126 | 98 | 172 |
| GraphQL | 2860 | 2820 | 3000 | 2860 | 2870 | 2750 | 2840 | 3000 | 2790 | 2920 |
| WebSocket | 519 | 491 | 458 | 482 | 446 | 423 | 431 | 436 | 429 | 473 |
| Дані 33716 КБ (25000 об'єктів) | | | | | | | | | | |
| Технологія | Час (мс) | | | | | | | | | |
| REST | 442 | 432 | 432 | 433 | 448 | 455 | 430 | 436 | 434 | 434 |
| gRPC | 494 | 398 | 356 | 372 | 370 | 332 | 310 | 314 | 293 | 360 |
| GraphQL | 7940 | 7120 | 7450 | 7520 | 7420 | 7090 | 7040 | 7330 | 7290 | 7520 |
| WebSocket | 1109 | 1113 | 1079 | 1102 | 1079 | 1083 | 1093 | 1119 | 1096 | 1107 |
| Дані 67496 КБ (50000 об'єктів) | | | | | | | | | | |
| Технологія | Час (мс) | | | | | | | | | |
| REST | 836 | 834 | 858 | 863 | 892 | 883 | 873 | 889 | 858 | 881 |
| gRPC | 844 | 708 | 713 | 638 | 661 | 551 | 519 | 504 | 523 | 705 |
| GraphQL | 13720 | 13810 | 14090 | 13970 | 14090 | 13630 | 14010 | 13970 | 13690 | 14560 |
| WebSocket | 2212 | 2152 | 2152 | 2149 | 2149 | 2172 | 2167 | 2269 | 2161 | 2147 |
| Дані 101180 КБ (75000 об'єктів) | | | | | | | | | | |
| Технологія | Час (мс) | | | | | | | | | |
| REST | 1264 | 1276 | 1312 | 1358 | 1321 | 1309 | 1319 | 1290 | 1282 | 1301 |
| gRPC | 1267 | 1027 | 995 | 1028 | 872 | 1046 | 902 | 1073 | 972 | 1045 |
| GraphQL | 21590 | 21520 | 21210 | 21500 | 21570 | 21470 | 21350 | 21540 | 22030 | 21100 |
| WebSocket | 3205 | 3137 | 3140 | 3226 | 3218 | 3211 | 3195 | 3311 | 3289 | 3190 |