Assignment 2: SUSTech Merch Store

CS328 - Distributed and Cloud Computing

DDL: 23:59, December 1, 2024

1 Introduction

You are opening an online SUSTech Merch Store where exactly 3 products will be put for sale. The store is externally accessible from a RESTful API Service where customers can check product info, maintain their user info, and submit orders. The RESTful API Service will communicate with other gRPC microservices for different purposes.

2 Architecture Design

Figure 1 shows the general architecture design of SUSTech Merch Store. Besides, check the provided codebase together with this figure for better understanding.

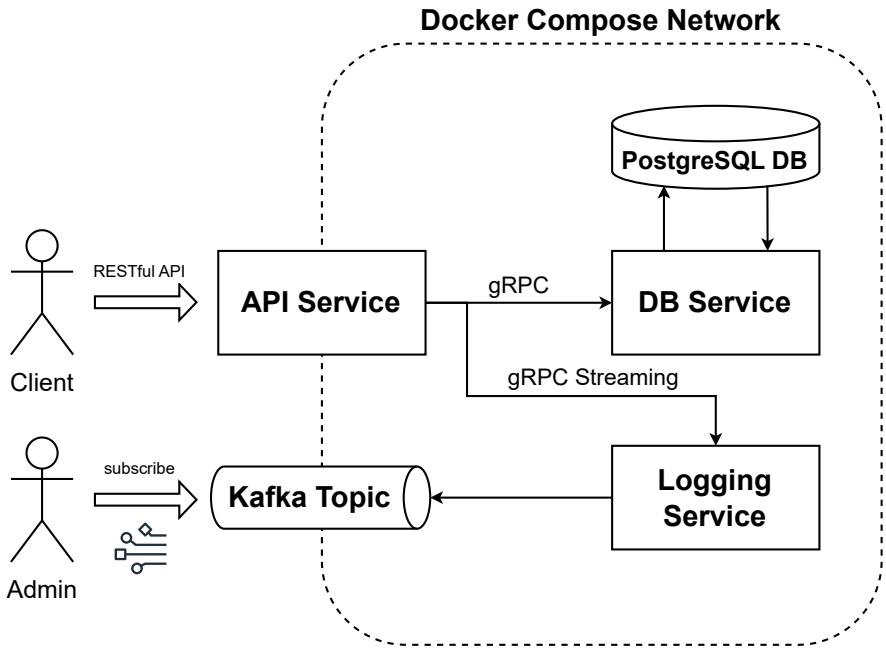

Figure 1: Architecture design of SUSTech Merch Store.

The 3 **products** for sale are stored in a **PostgreSQL Database**. Each product in the `products` table contains information like its name, category, price, stock, etc. Since this is a surprise event, all products are pre-added to the database and will remain unchanged.

The external customers need to register and log in to their accounts to be able to submit orders. The **user** and **order** information are also maintained in the database. Specifically, the `users` table maintains the user name and an encrypted password hash. The `orders` table records who (`user_id`) orders which product (`product_id`) and what are the order quantity and the total price. As a constraint, each order should not claim more than 3 product items. For simplicity, we do not implement the order payment logic. Check the initialization SQL file in the codebase for more information.

Each time the RESTful API service requires access to product, user, or order information, it needs to establish a new database connection, which can be resource-intensive. As a solution, a **gRPC Database (DB) Service** should be implemented to manage database operations on behalf of the RESTful API service. The DB service would maintain a connection pool, where multiple connections are created on demand and can be reused in the future. Check the local manager implementation in the codebase to understand how `psycopg2` handles this.

To implement a simplified and centralized monitoring service, a **gRPC Logging Service** will be implemented to collect operation logs from the RESTful API Service (and DB Service). Essentially, the Logging Service would define a client-side streaming RPC, enabling the API Service to continuously send log messages within a single RPC call. The Logging Service will then publish all log messages into a **Kafka topic**, allowing the administrator to monitor these messages by subscribing to the topic. Check the local publisher implementation in the codebase to understand how Confluent handles message publishing.

Specifically, the **RESTful API Service** includes certain user-specific operations that require authentication. Passing passwords to each of these APIs is cumbersome and insecure. To address this, JSON Web Token (JWT) is utilized for token-based authentication. An external codebase demonstrates the integration of JWT into a Python Flask server generated by OpenAPI Generator.

All the aforementioned components should be deployed in a Docker Compose Network. A **Compose file** in the codebase provides an initial configuration of the PostgreSQL Database and a single-node Kafka server (with a Zookeeper server). An environment file defines sensitive credentials used by the Compose file.

3 Tasks

In this assignment, you are required to implement the API Service, DB Service, and Logging Service from scratch. The PostgreSQL DB and the Kafka topic have been preconfigured for you. In detail, you should implement:

1. **API Service**:
   (a) A Greeting API that returns a welcome message at the base URL.
   (b) list-products and get-product operations for products.

(c) register, deactivate-user, get-user, update-user, login for users.

(d) place-order, cancel-order, get-order for orders.

(e) An OpenAPI specification YAML file defining the APIs above.

2. **DB Service**:

   (a) Relevant CRUD operations for products.

   (b) Relevant CRUD operations for users.

   (c) Relevant CRUD operations for orders.

   (d) A Proto file defining the RPCs above.

3. **Logging Service**:

   (a) Client-side streaming RPC to collect log messages.

   (b) A Proto file defining the RPCs above.

4. Updated **Docker Compose File**.

The general requirements are:

1. API/RPC definitions should be reasonable. For example, get-user should not fetch user-sensitive information like password.
2. User-specific APIs should introduce JWT-based authentication.
3. Field data types should be consistent across the OpenAPI specification file, Proto file, and database schema.
4. gRPC error handling (status code and detail message) should be implemented when necessary.
5. The implementation should adhere to standard style guidelines and include comments where needed.

By default, you should use Python Flask to implement the API Service and Python gRPC to implement the DB Service and Logging Service. You **might** want to use OpenAPI Generator to generate some code for the API Service. Then, you should write a report to answer the following questions:

1. What are the procedures of your implementation for each component? Explain how you set up the environment, generate code, and implement the business logic.
2. For your API Service, which APIs require authentication? How do you implement the authentication logic?
3. How do you select field data types for different definitions?

4. For gRPC-based services, select an arbitrary Proto message from your definition and analyze how it is encoded into binary format. Use Protobuf to programmatically verify the encoding result.

5. For your Logging Service, explain how the server-side streaming RPC works.

6. How do you configure Docker and Docker Compose so that these services can communicate with one another?

7. How do you run the experiment? Which tool (i.e., cURL, Postman, Swagger UI) do you use to test your API Service? How do you monitor the log messages from the Kafka topic?

4 Bonus

We introduce 2 bonuses for this assignment:

1. **Cross-Language**: Use different languages (i.e., Go and Python) for the gRPC client and gRPC server.

2. **Load-Balancing**: Use NGINX to load balance requests to multiple duplicates of RESTful API Servers. To try this bonus, you need to first have a complete and correct implementation of the API Service.

Each bonus offers 10 extra points to your assignment. **If you get more than 100 points in total, your bonus points will be scaled and added to the overall score of your homework.**

5 Submission

Be sure to include in your submission: source code files and a report (using a provided template) in **PDF** format. Pack all files into `SID_NAME_A2.zip`, where `SID` is your student ID and `NAME` is your pinyin name (e.g., `11710106_ZhangSan_A2.zip`). Any text should be in English only. Plagiarism is strictly prohibited.