Highly available, highly scalable cluster and microservice architecture realize online shopping during live streaming

This work explores the resolution of latency, availability, and scalability challenges in live commerce, particularly under conditions of substantial access traffic within brief intervals. It proposes the construction of a multi-node, high-availability, load-balanced cloud service computing cluster, complemented by the implementation of a sophisticated microservice framework.

Preface

Without any exaggeration, I spent at least $350 on computing, networking, and storage resources to develop this project on two cloud platforms☁️☁️. Therefore, I don't think you would want to try running this program in a cloud environment by yourself. However, you can still find value in the screenshots and descriptions in my report.

Similarly, in the Getting Started section, I have thoroughly documented and verified the use of Docker container technology to run this complex distributed microservices project in a local environment. You can use this as proof of my integrity.

Zeang Gao Final Project

Introduction

There are three branches in total: `main`, `dev`, and `cloud_online`.

- `main`: This branch has been set up to run locally according to the instructions provided. It is designed to facilitate inspectors in testing the **rationality, authenticity, and completeness** of my project. The deployment environment primarily utilizes **containerization** technology, optimizing and eliminating much of the complex, redundant Maven code and intricate operational techniques. This truly achieves **cross-platform** one-click deployment.
- `cloud_online`: This branch is specifically designed for **cloud demonstration purposes**, to show inspectors a high-availability cluster deployment and cloud service techniques integrated with my microservices project.

The code and some files in the `cloud_online` branch differ from those in the `main` branch because the `cloud_online` branch contains code for a production environment that integrates a Kubernetes cluster, please refer to the `cloud_online` branch: https://git.cs.bham.ac.uk/projects-2023-24/zxg117/-/tree/cloud_online.

Backend

Combining Java's microservices framework Spring Cloud with Spring Boot offers many advantages and provides a powerful and efficient environment for developing microservices applications. The key features and benefits of using Spring Cloud and Spring Boot together in this project:

1. **Rapid Development:** Spring Boot is a framework for rapidly developing microservices, offering out-of-the-box features, including auto-configuration and embedded web servers, making it incredibly simple and fast to build microservices.
2. **Microservices Architecture:** Spring Cloud provides a set of tools and libraries for building and managing microservices architecture. It includes features like service registration and discovery, load balancing, circuit breakers, and configuration management, which help in developing and managing distributed systems.
3. **Distributed Configuration:** Spring Cloud Config allows you to centrally manage configurations and share them among multiple microservices. This makes configuration changes and updates more convenient and consistent.
4. **Service Registration and Discovery:** Spring Cloud Eureka provides the capability for service registration and discovery, allowing microservices to automatically discover other services and implement load balancing, thereby improving system scalability and availability.
5. **Load Balancing:** Spring Cloud Ribbon is a client-side load balancer that can automatically distribute requests to multiple microservice instances, enhancing system performance and stability.

Meanwhile, in this project, **interceptors** from Spring Boot have been utilized to:

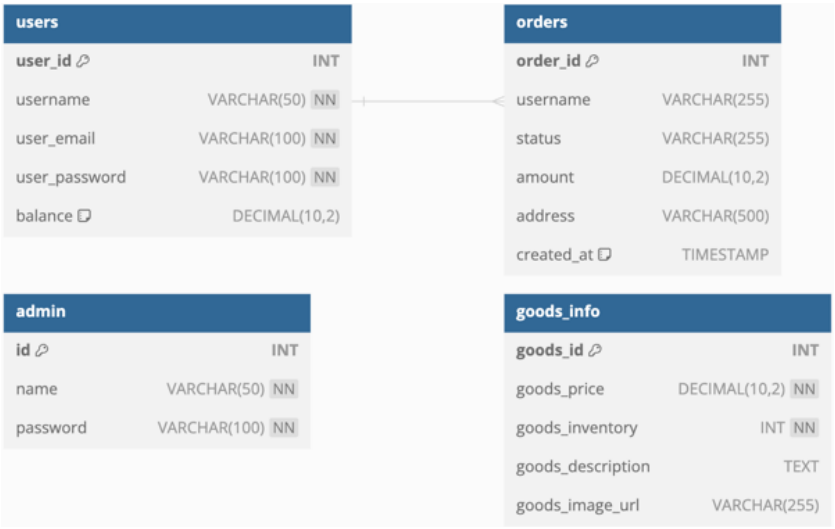
- Ensure that regular users cannot bypass login authentication or forcibly log in to the management side.
- Serve as a means of isolation, effectively separating regular user access from administrative access.

> In the cloud environment deployment, **Kubernetes** will be employed to enhance the **resilience** of the cluster.

Database

SQL

In this project, a total of two databases comprising four tables have been established using MySQL. For more details, please refer to the `db_schema.sql` file. Or please see the **ER Diagram** below:



Chose MySQL as the database, due to:

1. **Open Source Nature:** MySQL's open-source nature makes it an economical choice, particularly suitable for startups and projects, as it does not require the purchase of expensive licenses.
2. **Cross-Platform Compatibility:** MySQL's cross-platform compatibility allows it to run on multiple operating systems. This means you can use MySQL in different environments without worrying about platform compatibility issues.
3. **High Performance:** MySQL excels in handling queries and large datasets. It boasts optimized performance and can meet the demands of high-concurrency connections and large data volumes, which is crucial for high-traffic applications.

> Meanwhile, implemented **fuzzy search** functionality to enable users to search for products using keywords.

NoSQL

Chose Redis as the cache because Redis is a high-performance, multi-functional and easy-to-use in-memory database, and can cache commonly used data, which greatly reduces the traffic pressure on the back-end MySQL database.
In this project, Redis serves two primary purposes:

1. **User Login State Management:** Redis is utilized to store the login states of users. Additionally, an expiration time is set for these states to ensure the security of user login sessions. This mechanism helps in automatically invalidating the session after a certain period, enhancing the overall security posture.
2. **Email Verification Code Storage:** Another critical role of Redis is in the email verification feature of the project. It stores the verification codes sent to users and imposes a validity period on these codes. This ensures that the verification codes are only valid for a short duration, thereby maintaining the security and integrity of the verification process.

Database Connection Pool

Alibaba's Druid is an open-source database connection pool and SQL monitoring tool that is widely used in Java applications to enhance database access performance and management capabilities. Druid is not a cache pool but rather a **connection pool** designed to manage database connections, not for caching data.

The main features and functionalities of the Druid connection pool include in this project:

1. **High Performance:** The Druid connection pool is highly performant, effectively managing database connections in high-concurrency environments, reducing the overhead of connection creation and destruction.
2. **Monitoring and Statistics:** Druid provides rich monitoring and statistics capabilities, allowing real-time monitoring of database connection usage, SQL execution performance, and aiding in identifying slow queries and performance issues.
3. **SQL Injection Attack Defense:** Druid includes an SQL firewall that can detect and defend against SQL injection attacks, improving the security of your applications.

Frontend

In the front end, conventional technologies such as HTML, CSS, and JavaScript have been employed. Additionally, the project incorporates several mainstream front-end libraries, including but not limited to **SweetAlert** and **Bootstrap**, to ensure that the web pages are not only aesthetically pleasing and dynamic but also accessible.

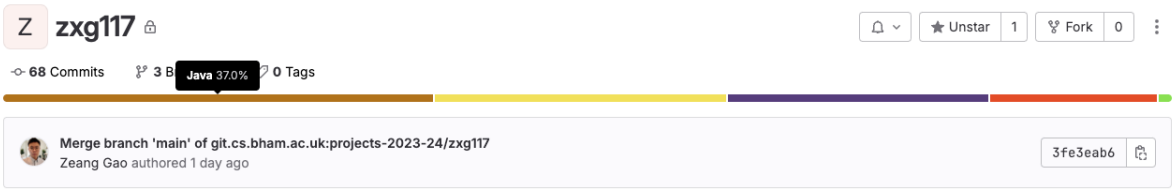
Getting Started

These instructions will get you a copy of the project up and running on your local machine for development and testing purposes.

Prerequisites (For Running Locally)

This project requires Linux, MAC, or Windows 10 and above (Special note: Docker has not been tested on Windows systems due to device limitations, so it is not guaranteed to work on Windows).

Additionally, the primary programming languages used in this project are **Java (37%)** and JavaScript (25%).



To avoid unnecessary complications, please run it using JDK 17 (Download Link). For local testing of the project's logical consistency, it is highly recommended to use an editor tool like IntelliJ for running Maven projects, to prevent issues such as the project not running. However, if you encounter any problems related to running the software, please email: zxg117@student.bham.ac.uk or zeang.gao2002@outlook.com, and cc zeang.gao@gmail.com. I will address your concerns. The entire project has been tested on two different computers to ensure all logic and configurations **work correctly**, so if there are any questions, please feel free to ask. Thank you for your time and effort!

> 💪😎 Of course, if you are someone without engineering experience or unfamiliar with Java programming, I will also provide an alternative method below that does not require running through an editor. This method involves using Docker images and containers for execution.⬇️

Run This Project Without Engineering Experience

First, you will need a cloud server, or you may refer to it as a virtual machine (I believe you should understand how to connect to your own server via SSH). I am assuming you are using a popular Linux operating system, such as **Ubuntu** (if you are using CentOS, the download commands are more or less similar, as it ultimately boils down to downloading). Then follow the next steps:

- **Docker Installation**:
  - For Linux (using Ubuntu as an example):

    ```
    apt update && sudo apt upgrade -y
    apt install -y docker.io
    systemctl enable --now docker
    ```

  - For Mac, download the Docker Desktop version (Download Link).
- **Git Clone to Your Server:**

  Before you clone, you need add SSH **public key** to your GitLab first and then clone my project. Then, cd into the project directory.

  ```
  ssh-keygen -t rsa -C "yourEmail@example.bham.ac.uk"
  ```

  ```
  git clone git@git.cs.bham.ac.uk:projects-2023-24/zxg117.git
  cd /project/path
  ```

- **Creating Database Containers in Docker**:
  - Please run the `docker_create_dbs.sh` script to create a MySQL container and a Redis container. This will expose ports mapped to the host's target ports, so please check for any **port overlap**.

    ```
    chmod +x docker_create_dbs.sh
    ./docker_create_dbs.sh
    ```

    In the docker_create_dbs.sh, I help you create the template of containers related to the databases in advance:

    ```
    # MySQL -- 3306 PORT
    # docker run --detach --name=mydb --env="MYSQL_ROOT_PASSWORD=root" --publish
    3306:3306 \
    # --volume=/root/docker/mydb/conf.d:/etc/mysql/conf.d mysql
    # Redis -- 6379 PORT
    # docker run --name redisdb -p 6379:6379 -d redis
    ```

- **Database Connection**:

- Use database connection tools such as Navicat or TablePlus (Download Link for Navicat, Download Link for TablePlus) to connect to the database using the password set in the script. Then, add the database structure `db_schema.sql` to create two databases with a total of four tables.
- Or you can use the script below if you don't know how to use the database connection tools:

```
# Path to your db_schema.sql file on your host machine (Change the path)
# e.g. SQL_FILE_PATH="./db_schema.sql"
SQL_FILE_PATH="/path/to/your/db_schema.sql"

# Copy db_schema.sql into the running MySQL container (mydb)
docker cp "${SQL_FILE_PATH}" mydb:/db_schema.sql

# Execute the SQL file inside the MySQL container
docker exec -i mydb sh -c 'exec mysql -uroot -p"root" < /db_schema.sql'
```

- **Starting Microservices**:
  - **Special Note:** In the `Authentication` module's `application.properties` file, change the service email account and password. You can use your own Outlook account, as Outlook by default has **SMTP** enabled. Configuration example was:

```
spring.mail.username=${MAIL}
spring.mail.password=${MAIL_PWD}
```

  - You can use the Docker and the script I provided without Java Env:

```
chmod +x docker_create_image.sh
./docker_create_image.sh
chmod +x start.sh
./start.sh
```

    In the startup script, I've added a **Docker Network Configuration** to ensure that communication between services is maintained normally while meeting isolation requirements.

    > Or you can start the five microservice modules (`admin8200`, `Authentication`, `Eureka7001`, `order_history`, `products_display`) either through the Java (JDK 17) command line or preferably using IntelliJ (Download Link).

- **Accessing From The User Side:**

  http://localhost:8081/html/user_login.html

  Note: If you don't have an account, you should register an account first.

- **Accessing From The Management Side:**

  http://localhost:8081/html/admin_login.html

  Note: Then use username `admin` and password `admin` to login the management panel.

🏝️ This setup ensures that all necessary components are in place for the project's deployment **in a local environment.**🏝️

Running On Cloud

In this section, I will endeavor to demonstrate how this project should be deployed in a cloud environment, and also, please combine with **my final report** to read for more details. However, due to the integration of **highly specialized cluster designs** and cloud service configurations, including but not limited to

- IAM
- Security Groups
- VPC Networks
- EIP Network Interface Settings
- Image Packaging/Cloning

this represents a highly customized case. Nevertheless, if you possess knowledge and skills in cloud computing, you can implement it on platforms such as AWS, Azure, or Huawei Cloud. During the actual deployment process, I utilized two different cloud platforms:

- **Microsoft Azure**: Served as the primary deployment environment for the project.
- **Huawei Cloud**: Acted as the experimental environment for testing and validation purposes.

In the cloud, I have set up a Kubernetes cluster consisting of **three nodes**:

- One master node (at least 2vCPU + 2G RAM)
- Two worker nodes (Worker1 and Worker2)

> For the specific setup process, please refer to **my final report.**

Implementation

If the purpose is solely to test the project's logic, please follow the instructions and guidance for running it locally provided above. In my GitLab repository, there are three branches in total: `main`, `dev`, and `cloud_online`.

- **`main`**: This branch has been set up to run locally according to the instructions provided.
- **`cloud_online`**: This branch is specifically designed for **demonstration purposes**, to show inspectors a high-availability cluster deployment and cloud service techniques integrated with my microservices project. In this branch, there are two files, `worker1.yaml` and `worker2.yaml`, using

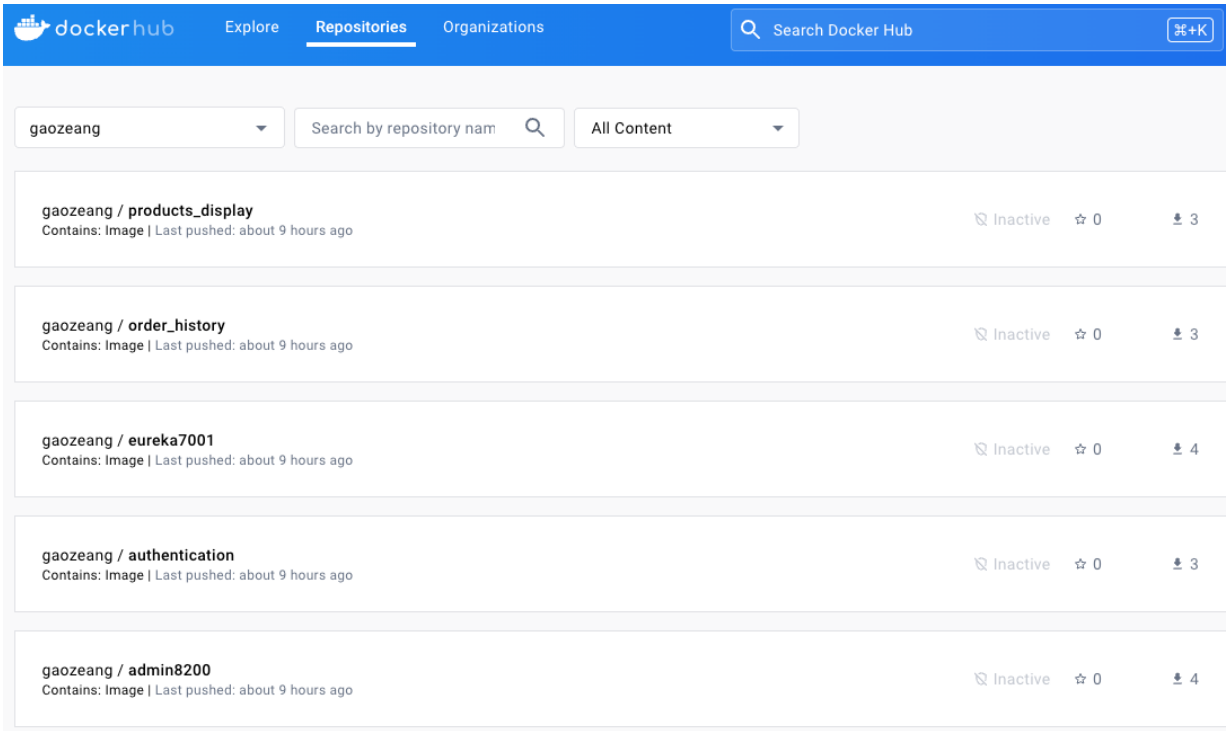
  ```
  kubectl apply -f workerX.yaml
  ```

  to deploy five Kubernetes deployments and five Kubernetes services on different nodes, ensuring that the microservices modules are deployed across these two worker nodes, fulfilling automatic scaling and load balancing requirements.

```
root@master:~/zxg117# kubectl get pods
NAME                                      READY   STATUS    RESTARTS      AGE
admin-deployment-69fc67cdb5-4bqph         1/1     Running   0             74s
authentication-deployment-6bf74b7795-kpnxh   1/1     Running   0             57s
display-deployment-86985fd85b-x9k26       1/1     Running   0             57s
eureka-deployment-668f4cbf55-26rqg        1/1     Running   3 (51s ago)   74s
order-deployment-67987c98c7-24fcd         1/1     Running   0             57s
```

> The screenshot above illustrates the deployment of five microservice modules across three distributed nodes on Microsoft Azure's cloud computing service. Eureka acts as a service registry and discovery center for microservices.

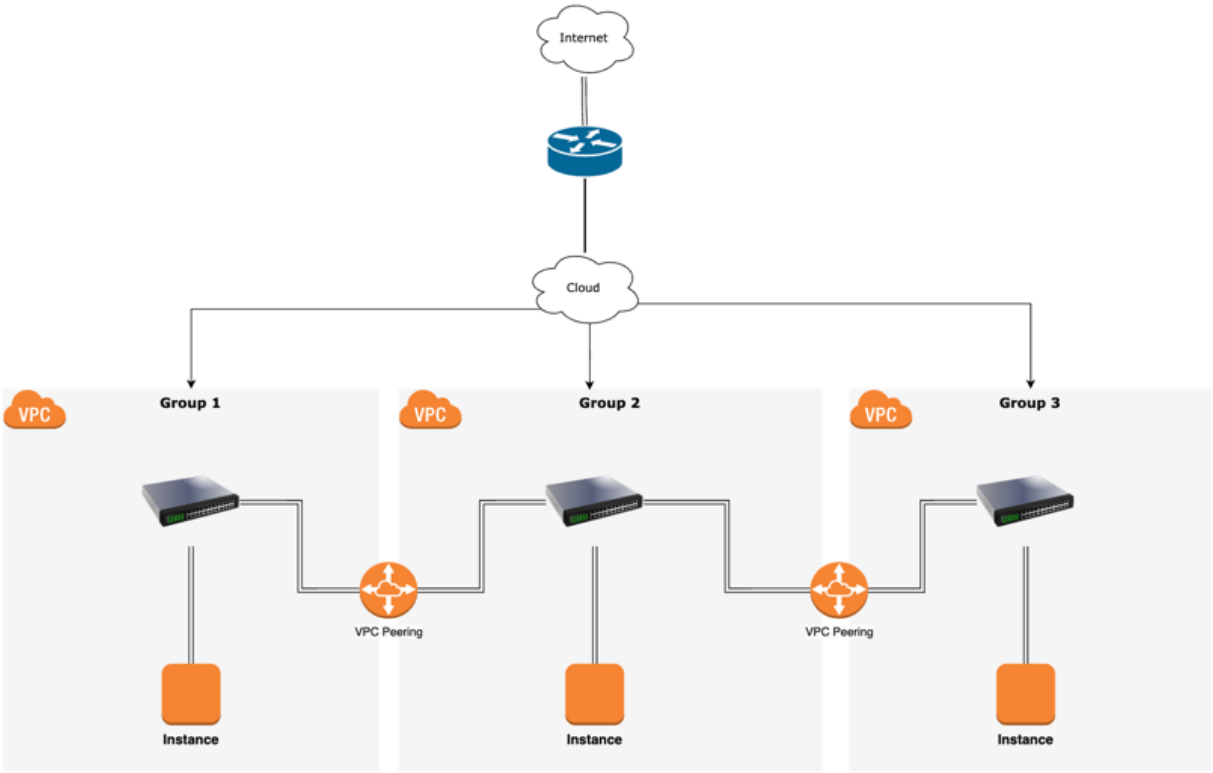
It's noteworthy that both configurations actually pull microservice module images from my pre-configured private repository. I utilize DockerHub to store images that are configured to meet production-level requirements. See the **repository** below:



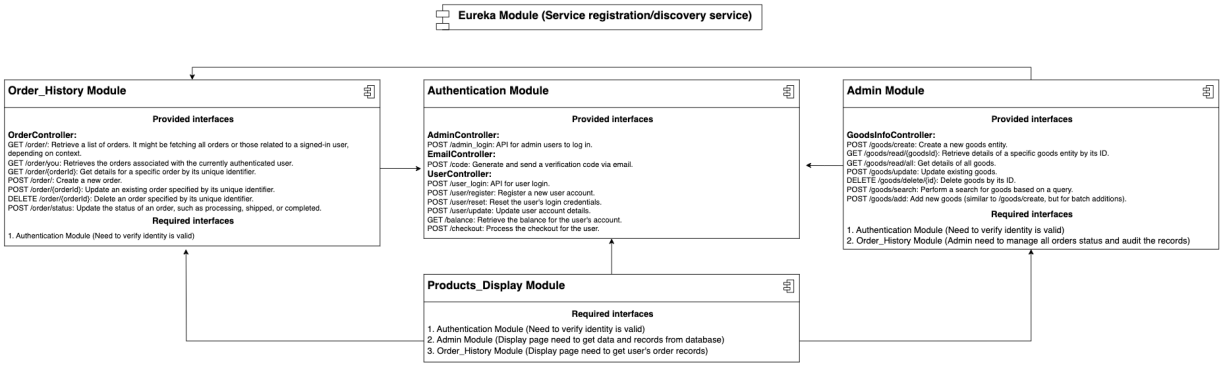
Architecture Diagrams

For an overall project architecture diagram on the cloud, see below:

1. Cloud Network Deployment



2. Application Component



3. Application Deployment