URL: https://github.com/YeZhang125/cs6650-assginment2.git

Deployment:

Configuration: Upload WAR file to EC2 instance, and modify the URL path in the MultiThreadedLiftRideClient class: private static final String **SERVER_URL**

Configuration of Load Balancer: An Elastic Load Balancer (ELB) is deployed in front of two EC2 instances to distribute incoming traffic efficiently. This enhances the servlet application's availability and fault tolerance. Modify the URL path in the MultiThreadedLiftRideClient class: private static final String **SERVER_URL to point to the Load Balancer DNS**

To run the project, start the **ServerAPI** project, the **Client** project and **Consumer** project separately, as they are built independently using Maven. To run **Consumer,** upload jar file to EC2 instance and run java -jar Consumer-1.0-SNAPSHOT.jar. Change HOST in **SkierConsumer and ServerAPI** accordinging to EC2 instance ip address that host Rabbitmq.

Server Design Overview

The server architecture utilizes a servlet-based API and a message queue system for asynchronous processing. The key components are outlined below:

1. SkierServlet (HTTP API Entry Point)

- **What it is:**
  A servlet that acts as the main HTTP API entry point in a Java-based web application.
- **Role:**
  It receives incoming HTTP requests, processes them, and routes them to the appropriate service.
- **major classes**(SkierServlet)**:**

  The SkierServlet class is a Java Servlet that processes skier-related requests and interacts with a RabbitMQ messaging queue for handling skier events asynchronously. It is mapped to the /skiers/* URL pattern and provides both POST and GET handlers.

  Methods:

  1. init() - Initialize RabbitMQ Connection & Channel Pool

2. doPost() - Process HTTP Requests & Send Messages

- **Common Responsibilities:**
  - Accepting HTTP requests (e.g., `POST /skiers/{skierID}/vertical` to update skier stats).
  - Validating request URL parameters.
  - Returning appropriate HTTP responses (e.g., JSON data, status codes).
- **Example Flow:**
  - A client sends a `POST` request.
  - `SkierServlet` receives it and calls a backend service to fetch skier data.
  - The response is returned as JSON.

---

2. Consumer (Message Processor)

- **What it is:**
  A background process that listens to messages from a message queue RabbitMQ
- **Role:**
  It processes asynchronous tasks that do not require immediate user interaction.
- **major classes**(SkierConsumer)**:**

  The `SkierConsumer` class is a multi-threaded consumer that connects to a RabbitMQ server and consumes messages from specified queues. It is designed to handle high throughput by utilizing a thread pool for concurrent message consumption. At startup, the consumer initializes a fixed thread pool, with each thread executing an instance of `ConsumerTask`. Each `ConsumerTask` connects to RabbitMQ, declares the queue, and listens for incoming messages. Upon receiving a message, it extracts skier and lift ride details, updating a concurrent hash map (`messagestore`) that tracks all lift rides for each skier. To ensure reliability, messages are manually acknowledged after processing.

- **Common Responsibilities:**
  - Consuming messages from a queue.
  - Processing messages and store to hash map.
  - Ensuring reliability and error handling in message processing.
- **Example Flow:**
  - The `SkierServlet` receives a skier activity update and sends it to RabbitMQ.
  - The `Consumer` picks up the message

---

3. Client (Handles HTTP Requests to RabbitMQ)

- **What it is:**
  A service or module that sends HTTP requests to RabbitMQ.
- **Role:**
  It acts as an API consumer, fetching data or triggering actions in external services.
- **major classes:**

1. MultiThreadedLiftRideClient.java

- **Purpose:** Acts as the program's main entry point, orchestrating the load testing process.
- **Functions:**
  - Configures parameters such as server URL, number of threads, and requests per thread.
  - Utilizes `ExecutorService` to manage concurrent execution.
  - Monitors thread completion and delegates task execution to `HTTPClientThread`.

2. HTTPClientThread.java

- **Purpose:** Handles HTTP request execution and logs responses from simulated clients.
- **Functions:**
  - Sends HTTP POST requests via `HttpClient`.
  - Logs request details, including timestamp, type, latency, response code, and throughput, to a CSV file.
  - Implements retry logic for failed requests.
  - Calculates and stores response times for further evaluation.

3. EventProducer.java

- **Purpose:** Generates randomized event data for HTTP POST requests, simulating skier lift events.
- **Functions:**
  - Produces synthetic data representing ski resort activities with randomized attributes (e.g., resort ID, season ID, skier ID).
  - Ensures varied request simulation to enhance realism in load testing.

- **Common Responsibilities:**
  - Handling retries, timeouts, and error responses.
  - Parsing and processing API responses.
- **Example Flow:**
  - The `Consumer` processes a skier activity update and store it to hash map.

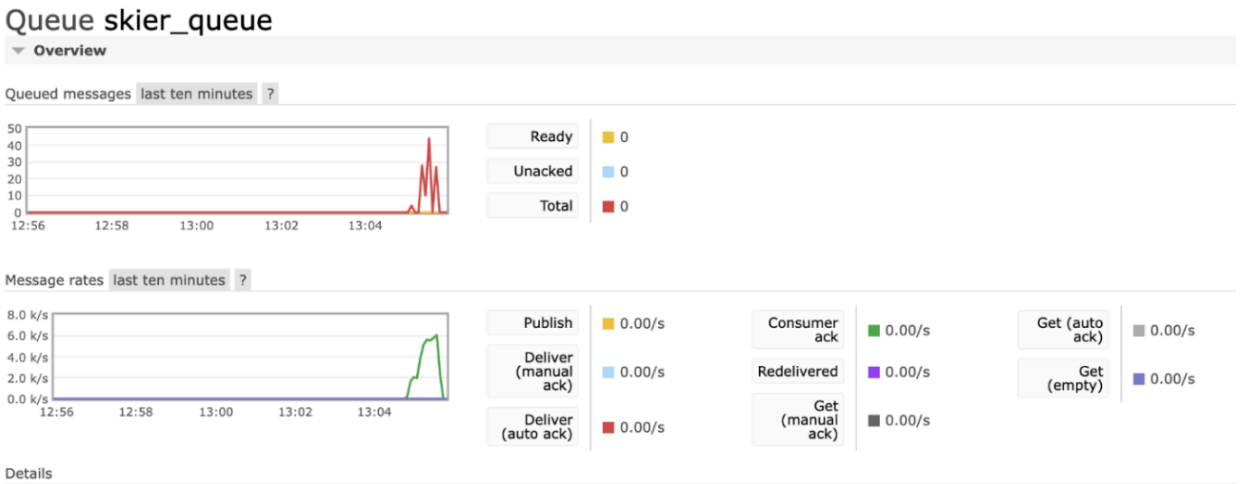
- The `Client` makes an HTTP request, receives a response, and returns it to the message queue.

Single Servlet Throughput:

```
/Users/yezhang/Library/Java/JavaVirtualMachines/corretto-17.0.14/Contents/Home/bin
Deleted request_logs.csv
Starting 32 threads...
Finished 1 of 32 threads!
Starting additional threads...
======= Client 1 Output =======
Number of Threads: 452
Successful requests: 200000
Failed requests: 0
Total requests sent: 200000
Total response time: 40943 ms
Throughput: 4884.8398993722985 requests per second

Process finished with exit code 0
```

RMQ management windows showing queue size, send/receive rates:



Load Balanced Servlet Throughput:

```
Deleted request_logs.csv
Starting 32 threads...
Finished 1 of 32 threads!
Starting additional threads...
======= Client 1 Output =======
Number of Threads: 452
Successful requests: 200000
Failed requests: 0
Total requests sent: 200000
Total response time: 35003 ms
Throughput: 5713.795960346256 requests per second

Process finished with exit code 0
```

RMQ management windows showing queue size, send/receive rates: