**GitHub Link:** https://github.com/Crystal0GMY/MultithreadPostWithRabbitMQ

**1. Architecture Overview**

The server follows a distributed, event-driven architecture utilizing **RabbitMQ** as the message broker to handle high throughput requests efficiently. It consists of the following key components:

* **Client**: Sends HTTP POST requests to the server.
* **Servlet (Producer)**: Receives and processes incoming HTTP requests, then publishes messages to RabbitMQ.
* **Message Queue (RabbitMQ)**: Acts as a buffer, decoupling the producer (Servlet) from the consumer (Worker).
* **Consumer (Worker Service)**: Listens for incoming messages, processes them, and stores data in a HashMap.

**2. Major Components and Classes**

**a. Client (/client1 folder)**

* **Sends HTTP POST requests** to the servlet.
* The same as in Assignment 1.
* The client runs locally by running the MultithreadClient class in the client1 folder.

**b. Servlet (/Assignment2Servlet folder)**

The producer is implemented as a Java Servlet, which handles HTTP POST requests, validates incoming data, and sends messages to **RabbitMQ** for asynchronous processing.

 **Initialize RabbitMQ Connection & Channel Pool** (on startup).

 **Handle POST requests** from clients, extract skier ride data, and validate input.

 **Publish messages** to RabbitMQ using a channel from the pool.

 **Return an appropriate HTTP response** (success or error).

**Key methods:**

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

* Creates a **single RabbitMQ connection**.
* Initializes a **pool of reusable channels** (channelPool) to improve performance.
* Declares the queue to ensure it exists before usage.

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

* **Extracts & validates URL parameters** (resortID, seasonID, dayID, skierID).
* **Reads & validates request body** (must contain time and liftID).
* **Creates a JSON message** for RabbitMQ.
* **Borrows a channel from the pool**, publishes the message, then **returns the channel to the pool**.
* **Sends an HTTP response** (201 Created on success, 400 Bad Request on error)

**3. sendMessageToQueue()**- **Publish Message to RabbitMQ**

* Uses basicPublish() to send messages to the queue.
* Logs success (✅ Sent to RabbitMQ) or failure (❌ Failed to send).

**Deployment:**

1. EC2 Instance Setup: The servlet is deployed on an EC2 instance running Apache Tomcat. Initially, the WAR file is uploaded to the EC2 instance, and Tomcat is configured to handle incoming HTTP requests on port 8080.
2. Configuration of Load Balancer: To distribute incoming traffic across multiple instances, an Elastic Load Balancer (ELB) is set up in front of the EC2 instances. This ensures higher availability and fault tolerance for the servlet application.
3. Update the MultithreadClient: In the MultithreadClient.java file, update the SERVER\_URL variable to point to the Load Balancer DNS (e.g., http://ServletLoadBalancer-1316451551.us-west-2.elb.amazonaws.com:8080/Assignment2\_war). This ensures that the client sends POST requests to the load balancer, which then routes the requests to one of the available EC2 instances.
4. Scaling with Multiple Instances: For high availability, two EC2 instances are used, and the Load Balancer is configured to distribute incoming requests to these instances based on traffic load. New EC2 instances can be launched from AMI “ImageWithTomcat”.

**c. RabbitMQ Message Queue**

RabbitMQ is deployed on a separate EC2 instance to handle message queuing independently from the servlet. After setting up RabbitMQ on this instance, the RABBITMQ\_HOST in both the producer and client must be updated to the public IPv4 address of the RabbitMQ EC2 instance to ensure proper communication.

* Handles asynchronous message processing.
* Uses **Direct Exchange** to route messages to a specific queue.

**Key configurations:**

* + channel.queueDeclare(queueName, true, false, false, null): Ensures queue persistence.
  + channel.basicPublish("", queueName, null, message.getBytes()): Publishes the message.

**d. Consumer (Worker)**

The consumer, Assignment2Consumer, is responsible for processing messages from RabbitMQ that contain skier lift ride data. It is designed to run in multiple threads, leveraging CPU parallelism for efficient message handling.

* Runs on a separate **EC2 instance**.
* Listens to the RabbitMQ queue.
* Processes messages and stores data in a **HashMap**.

Upon startup, the consumer initializes a fixed thread pool, where each thread runs an instance of ConsumerWorker. Each worker establishes a connection to RabbitMQ, declares the queue, and listens for incoming messages. When a message is received, it extracts skier and lift ride information and updates a concurrent hash map (skierLiftMap), which maintains a record of all lift rides for each skier. Messages are manually acknowledged after processing to ensure reliability.

The consumer also includes a **graceful shutdown mechanism**, which ensures that all threads are properly terminated when the application is stopped. Additionally, it enables **automatic connection recovery** and heartbeat monitoring to maintain stability in case of network issues.

**Deployment of Consumer**

The consumer is deployed on a separate **EC2 instance** to process messages independently. The JAR file containing the consumer application is copied to the EC2 instance using scp, and it is executed in the background using nohup to keep it running even after logging out:

nohup java -jar Assignment2-1.0-SNAPSHOT.jar > consumer.log 2>&1 &

**3. Message Queue Configuration**

**1. Queue Setup and Persistence**

Queue Name: "skiers\_queue"

Durable Messages: The queue is declared as durable to persist messages in case of server restarts.

Non-Exclusive, Non-Auto-Delete: Ensures multiple consumers can access it, and the queue persists even if no consumers are active.

Reason: Ensures reliability and prevents message loss in case of unexpected failures.

**2. Connection and Channel Pooling**

Connection Factory:

Uses factory.setHost(RABBITMQ\_HOST) to define the RabbitMQ server.

Authentication via setUsername() and setPassword().

Automatic Recovery Enabled: Ensures reconnection if the connection is lost.

Heartbeat (30s): Keeps the connection alive.

Channel Pooling (ArrayBlockingQueue):

Pre-initializes POOL\_SIZE channels to reduce the overhead of creating/destroying channels.

Channels are borrowed for publishing and returned after use.

Reason: Improves performance by reducing connection overhead and ensuring failover resilience.

**3. Consumer Configuration**

Initial Consumers:

NUM\_OF\_CONSUMERS = 5, dynamically adjusted based on workload.

Consumers are executed using a CachedThreadPool.

Message Handling (Fair Load Distribution):

channel.basicQos(10): Limits the number of unacknowledged messages per consumer to prevent overload.

basicConsume(..., false): Messages are manually acknowledged after processing.

Reason: Ensures controlled message consumption while maintaining flexibility for scaling.

**4. Dynamic Consumer Scaling**

**Scheduled Monitoring** (scheduleAtFixedRate):

Runs every few seconds to adjust consumer count.

Can scale up/down consumers based on queue length.

Reason: Prevents overloading while maintaining responsiveness to traffic spikes.

**5. Graceful Shutdown Handling**

**Shutdown Hook:**

Ensures the executor services (consumerExecutor, monitorExecutor) shut down cleanly.

Waits for consumers to finish before forcefully terminating.

Reason: Prevents message loss and ensures safe cleanup.

**4. Result**

**Test run results for a single servlet:**

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**Test run results for a load balanced servlet:**

**A screenshot of a computer

AI-generated content may be incorrect.**

A screenshot of a computer

AI-generated content may be incorrect.