**Assignment 4 Doc**

Team:

Member:

Grace (Jingshu) Zhou: 002859188

Git repo: https://github.khoury.northeastern.edu/gracezhou0713/CS6650-HW4

**Application Setup**

Project Structure:

* Server
* Client1
* Client2
* Consumer

Steps to run the application:

1. Update RabbitMQ host/username/password and Redis host for server and consumer
   1. In Server – Config.class
   2. In Consumer – Config.class
2. Build server and consumer
   1. Server: build artifacts – Server:war
   2. Consumer: build artifacts – Consumer:jar
3. Deploy the server
   1. War file path: Server/out/artifacts/Server\_war/Server\_war.war
4. Deploy the consumer
   1. Jar file path: Consumer/out/artifacts/Consumer\_jar/Consumer.jar
5. Update Server URL for Client 2
   1. In MultiThreadClient.java, replace the URL at the beginning of the class
      1. EC2: http://<ec2-ip>:8080/Server\_war/
6. Run client1 or 2
   1. Output will be printed
   2. In RabbitMQ management website, the queue information will be displayed
7. JMeter test
   1. Open JMeter and import “GET Request Test.jmx”
   2. Run the test and the result will be in listener

**Project Architecture**

**Major Classes:**

Server

1. LiftRide
   1. Purpose: data model class
   2. Includes information of time and lift ID
2. LiftRideMessage
   1. Purpose: data model class
   2. Encapsulates the full lift ride message data sent to RabbitMQ
   3. Includes information of resortID, seasonID, dayID, skierID, and a LiftRide object.
3. Config
   1. Purpose: Configuration class
   2. Including RabbitMQ and Redis constants
4. Servlet
   1. Purpose: servlet; handle POST and GET requests
   2. Process POST requests and publish messages to RabbitMQ
   3. Process GET requests by querying Redis for skier and resort data.
5. RMQChannelPool
6. RMQChannelFactory
7. RedisClient

Consumer:

1. Application
   1. Purpose: main application
   2. Establishes a RabbitMQ connection, creating a fixed-size thread pool
   3. Consumer Task Execution: Each thread executes a MessageConsumer to handle messages from the assignment2\_queue queue.
2. MessageConsumer
   1. Purpose: consume messages from RabbitMQ
   2. Queue Declaration: Declares the assignment2\_queue queue and starts consuming messages using basicConsume with auto-acknowledgment enabled.
   3. Asynchronous Processing: Each message is processed in a separate thread using CompletableFuture to improve scalability and reduce potential blocking
3. MessageHandler
   1. Processes each message, updating in-memory data structures based on received data.
   2. Concurrent Hash Map: Stores and aggregates lift ride information in a ConcurrentHashMap keyed by skierID to support safe concurrent access by multiple threads.

**Client1:**

Major Classes:

1. LiftEvent
   1. Purpose: data model class
   2. Includes information of Lift Ride, skier ID, resort ID season ID, and day ID
2. DataGenerator
   1. Purpose: produce a specified number of LiftEvent objects with randomized data, simulating ski lift rides.
   2. The generated LiftEvent objects are stored in a blocking queue
3. ApiPostClient
   1. Purpose: take objects from the queue and post them to the SkierApi concurrently
   2. The class counts down the latch upon completion
   3. For each event, the postEvent method attempts to send the request to the API endpoint with a maximum retry count
   4. Two AtomicInteger counters, successfulRequests and failedRequests are incremented based on whether the API call successes or fails.
4. MultiThreadClient
   1. Purpose: manage and execute 200,000 concurrent requests to SkierApi endpoint by using multiple threads
   2. Initialization: define URL, total request number, initial threads number, requests per thread, etc.
   3. Data generation: create an instance of DataGenerator to generate 200,000 LiftEvent objects; Start a separate thread and wait until it completes; Retrieve the events from the blocking queue.
   4. Initial Thread Execution: create a fixed thread pool with 32 threads using ExecutorService; submit 32 ApiPostClient tasks to the thread pool; await the completion of the initial latch for at least one thread to finish
   5. Additional Thread Execution: create a new thread pool for calculated additional threads; submit the remaining tasks to the thread pool; await the completion of all additional threads using the latch.
   6. Shutdown and Statistics: shut down both the initial and additional thread pools; print out the statistics

**Client2:**

Added or edited classes:

1. Record
   1. Purpose: data model class
   2. Store details of each API request, including the start time, request type, latency, and status code
2. ThroughputData
   1. Purpose: data model class
   2. Store information about throughput at specific time intervals
3. CsvWriter
   1. Purpose: log API request details into a CSV file, including information such as request start time, type, latency, and status code
   2. The constructor verifies the directory and CSV file, or creates the new ones if they don’t exist
   3. The writeRecordsToCsv method accepts a list of Record objects and writes them to the CSV file. The file is overwritten with each call to this method
4. ThroughputPlotWriter
   1. Purpose: plot the throughput over time
5. MultiThreadClient
   1. A List<Record> recordList has been added to store detailed information about each API request
   2. A ScheduledExecutorService is used to record throughput (requests/second) every second, storing the data in ThroughputPlotWriter, which is responsible for collecting and plotting this throughput data.
   3. The calculateAndPrintStatistics method uses DescriptiveStatistics to calculate and display detailed performance metrics

**Message Flow:**

**POST**

1. Client to Server:
   1. The client sends a POST request with lift ride data
   2. Servlet validates and parses the request, creating a LiftRideMessage object.
   3. The message is serialized to JSON and published to the assignment2\_queue in RabbitMQ.
2. Server to RabbitMQ:
   1. RabbitMQ receives the message, storing it in assignment2\_queue until a consumer is ready to process it.
3. RabbitMQ to Consumer:
   1. Application starts multiple MessageConsumer threads, each consuming messages from assignment2\_queue.
   2. Each MessageConsumer asynchronously processes the message using MessageHandler.
4. Consumer Handling:
   1. MessageHandler deserializes the message, extracts the info and generates the Redis key, stores information into Redis

**GET**

1. Client:
   1. The client sends a GET request
      1. /skiers/{skierID}/vertical
         * resort parameter required; season parameter optional
      2. /skiers/{resortID}/seasons/{seasonID}/days/{dayID}/skiers/{skierID}
      3. /resorts/{resortID}/seasons/{seasonID}/days/{dayID}/skiers
2. Server
   1. Servlet validates and parses the request, creating corresponding Redis Key
   2. The response is capsulated to JSON and sent back to client
3. Redis
   1. Return the query results according to the key

**Database Design**

Because the project involves large-scale data write and read operations, we need a database that supports high-throughput data storage and retrieval. Redis would be a great solution.

As it requires, the data model in the database should enable queries like:

**/skiers/{skierID}/vertical** (resort parameter required; season parameter optional)

Structure

* Key: resort:{resortID}:skier:{skierID}:vertical
* Type: Hash
* Purpose:
  + Tracks the total vertical distance covered by a skier across multiple seasons and all time.
  + Separates the data by season and provides an aggregate (all).

Example

* Key: resort:12:skier:67890:vertical
* Fields:
  + 2024: 8000 (total vertical distance for season 2024)
  + 2023: 7000 (total vertical distance for season 2023)
  + all: 15000 (cumulative vertical distance across all seasons)

**/skiers/{resortID}/seasons/{seasonID}/days/{dayID}/skiers/{skierID}**

Structure

* Key: resort:{resortID}:season:{seasonID}:day:{dayID}:skier:{skierID}
* Type: Hash
* Purpose:
  + Tracks the total vertical distance covered by a specific skier (skierID) at a specific resort, season, and day.
  + Stores details for each skier's daily activity.

Example

* Key: resort:12:season:2024:day:1:skier:67890
* Fields:
  + vertical: 4500 (total vertical distance covered on this day)

**/resorts/{resortID}/seasons/{seasonID}/days/{dayID}/skiers**

Structure

* Key: resort:{resortID}:season:{seasonID}:day:{dayID}:skiers
* Type: Set
* Purpose:
  + Tracks all unique skier IDs for a specific resort (resortID), season (seasonID), and day (dayID).
  + Allows querying how many unique skiers visited a resort on a specific day during a season.

Example

* Key: resort:12:season:2024:day:1:skiers
* Value: {12345, 67890, 11122} (a set of skier IDs)

**Test Run Results**

**Post:**

Bottleneck:

Firstly, I used a t2 micro instance to host Redis, but the messages quickly accumulated in the queue, and Redis eventually stopped accepting requests.

A screenshot of a computer

Description automatically generated

Solution1: upgrade the instance to t3 medium

Solution2: implemente a pipeline for writing data to Redis

This configuration successfully handled 200,000 requests efficiently. Though there is still some messages in the queue, but it is much fewer than before

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**A screenshot of a computer

Description automatically generated**

**GET: JMeter**

**A screenshot of a computer

Description automatically generated**