CIS628 – Project # 2- SportsStats

Objective

The goal of this project was to expand on Project 1 by expanding on API communication. In this project, we will create a CLI that takes in a user input (NFL team name) and produces the team stats for 2020 season. The CLI is designed over two components, a client which represents user interactions and a server which takes user input (request) and creates the corresponding response by sending the request to Sports Nozzle API.

Client and Server requests

The interchange between the client and server happens via a queue based on Rabbit MQ in the following sequence –

1. The client (user) request is published to a queue
2. The server consumes the request from the queue, and –
    a. Parses the request and creates the API call
    b. Issues a GET request against the API call
    c. Parse the corresponding response, inject additional items (win/loss etc), and creates a payload
    d. The payload is then published back to same queue via a delivery callback
3. While #2, is happening the client is sleeping for 100 milliseconds, and consuming from the same queue via another delivery callback, that will –
    a. Parse the response and deserialize it as list of JSON
    b. Present the output to the end user

Every client request gets published with a unique UUID based correlation id that is used by the server to identify ***whose*** request is being processed.

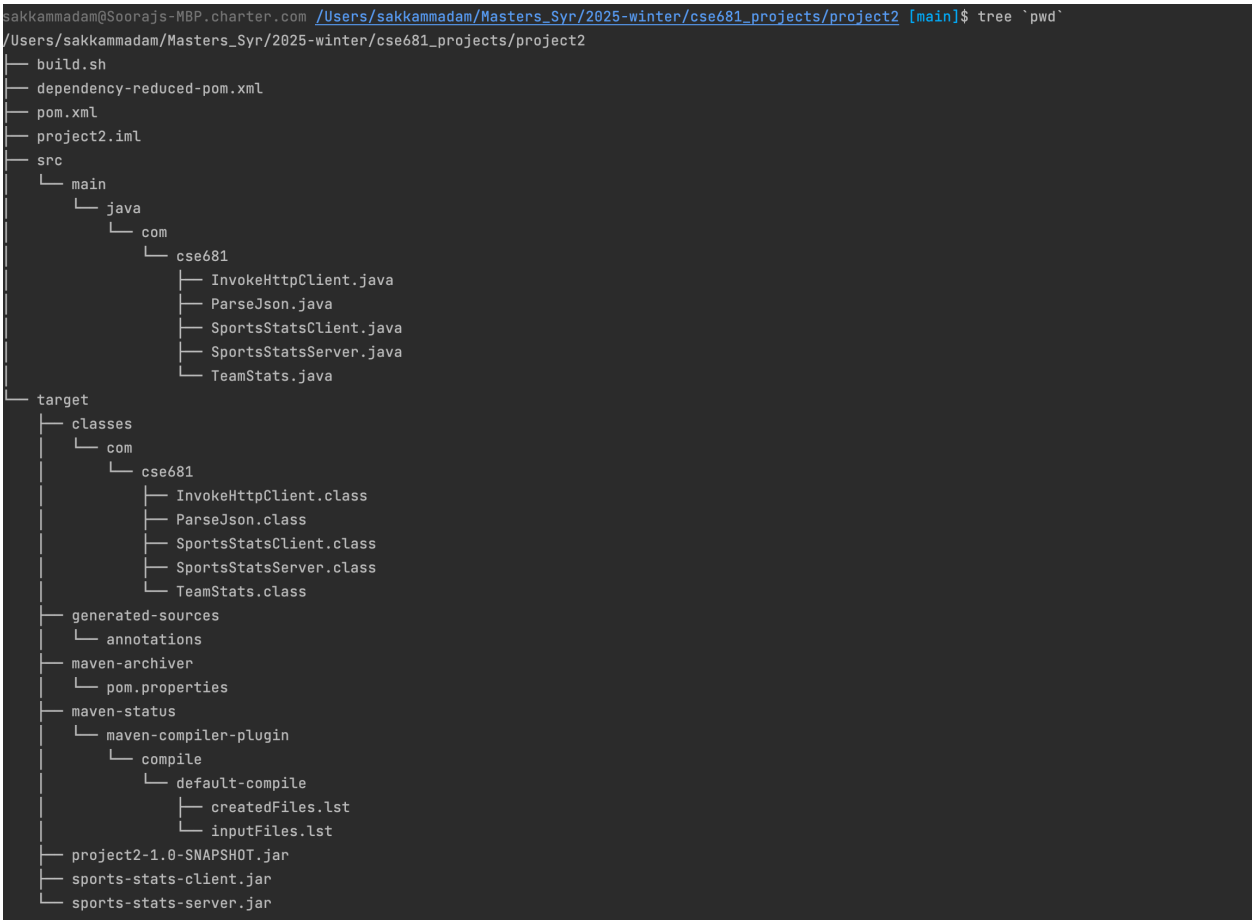
Design Choice

We have made the following design choices:

- Language: Java
    - Version: Java SDK 23
- IDE: IntelliJ IDEA
- Build Automation: Maven
- External Libraries:
    - FasterXML Jackson used to process JSON data
    - Rabbit AMQP client
        - SLF4J logging library (sub-dependency)

Project Structure

The project structure is as follows:



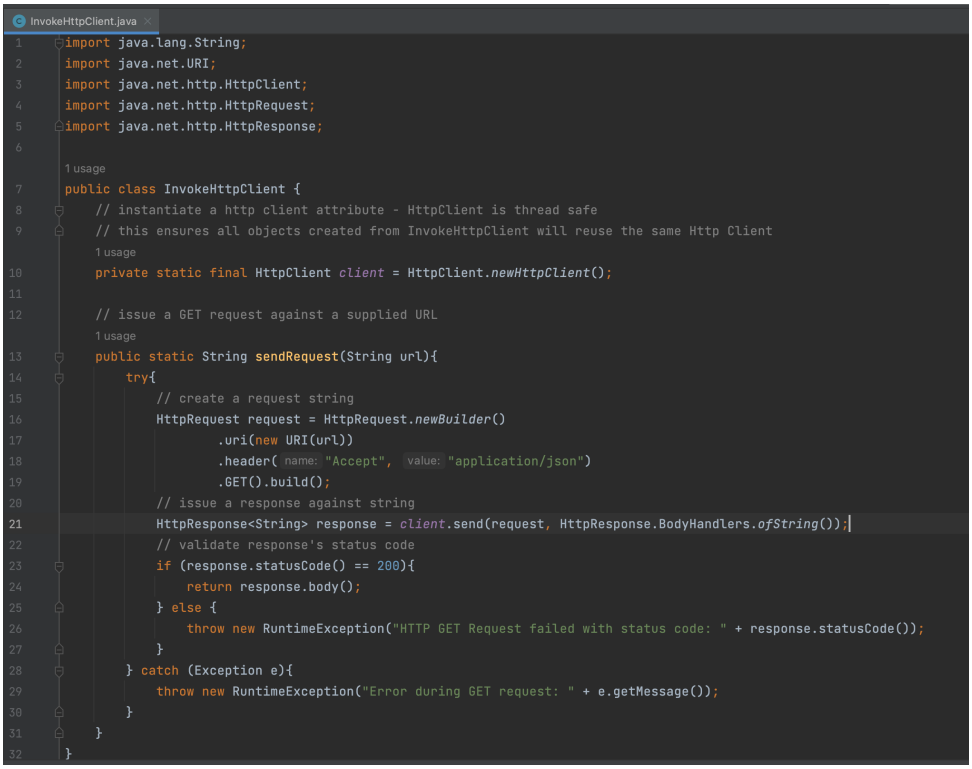
Points to consider:

- Root file `project2.iml` which contains module information for SportStats application.
- Mevan file `pom.xml` contains external library dependencies
- Src directory contains the Java code
    - InvokeHttpClient.java
    - ParseJson.java
    - TeamStats.java
    - SportsStatsClient.java
    - SportsStatsServer.java
- Target directory –
    - Contains compiled classes associated with the java code
    - FAT jar’s that can be used to run the program

Source Code

InvokeHttpClient.java

This Java class uses Java's net packages to instantiate an http client and make GET requests.

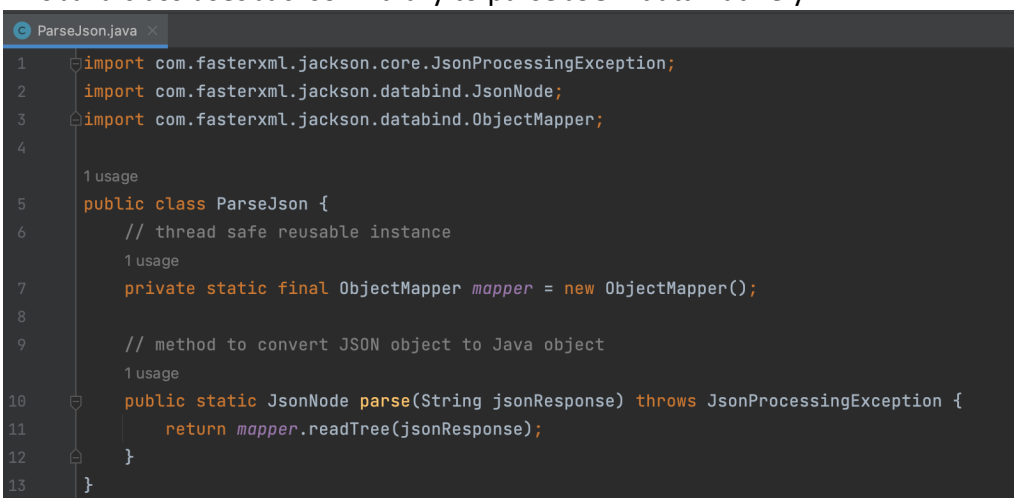


```java
import java.lang.String;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;

1 usage
public class InvokeHttpClient {
    // instantiate a http client attribute - HttpClient is thread safe
    // this ensures all objects created from InvokeHttpClient will reuse the same Http Client
    1 usage
    private static final HttpClient client = HttpClient.newHttpClient();

    // issue a GET request against a supplied URL
    1 usage
    public static String sendRequest(String url){
        try{
            // create a request string
            HttpRequest request = HttpRequest.newBuilder()
                    .uri(new URI(url))
                    .header( name: "Accept",  value: "application/json")
                    .GET().build();
            // issue a response against string
            HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
            // validate response's status code
            if (response.statusCode() == 200){
                return response.body();
            } else {
                throw new RuntimeException("HTTP GET Request failed with status code: " + response.statusCode());
            }
        } catch (Exception e){
            throw new RuntimeException("Error during GET request: " + e.getMessage());
        }
    }
}
```

This was carried over from Project1

ParseJson.java

This Java class uses Jackson library to parse JSON data natively.



```java
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;

1 usage
public class ParseJson {
    // thread safe reusable instance
    1 usage
    private static final ObjectMapper mapper = new ObjectMapper();

    // method to convert JSON object to Java object
    1 usage
    public static JsonNode parse(String jsonResponse) throws JsonProcessingException {
        return mapper.readTree(jsonResponse);
    }
}
```

This was carried over from Project1

TeamStats.java

This Java class acts as a POJO to describe a stats record returned from the API. Contains default constructor and getters to retrieve individual attributes. We also override toString method to depict the entire object as a string.
This was originally carried over from Project1 but we made the following additions:

- Added new fields –
  - `matchup` – Indicates if it was a home or away game
  - `opponentScore` – the score of the opponent
  - `matchResult` – Indicates whether it was a win or loss
- Added new getter methods –
  - `getMatchUp` – returns matchup
  - `getOpponentScore` – returns `opponentScore`
  - `getMatchResult` – returns `matchResult`

SportsStatsClient.java

This is the application class that represents the end-user interactions in an infinite while loop. It will publish a message to a queue where it gets processed by server-side code.
This class has the following maps –

- `teamMap` which is preloaded with team name and corresponding ids
  - When a user selects a team name this map is used to get corresponding id and is used to construct a message.
- `teamMapReverse` which is reverse of `teamMap` and is used for printing results

This class has the following methods –

- `centerText`
  - Used to pretty format the output
- `printResults`
  - Used to print out and format the results of array containing JSON responses from the server
  - Leverages the `TeamStats` class internally to represent data as a POJO
- `main`
  - Instantiates factory, connection and channel for Rabbit MQ
  - Instantiates runtime queue within Rabbit MQ
  - Instantiate Scanner object
  - Enters a while loop that will
    - Parse Scanner to take in user input that is validated against NFL teams
    - If input was exit, the loop will exit, and program will end
  - Parse user input and create a message with team name and id
  - The queue is assigned a correlation id to indicate a unique client request
  - Sets a response received flag as 0
  - Publish the message as bytes
    - Once the server processes the requests it sends a response back

    - Consume response the back and in process execute a delivery call back which will
        - Parse the output using `printResults`
        - Sets the response flag as 1
    - Call another inner while loop which instantiates a child thread that will wait for 100 milliseconds while waiting for the response flag to 1

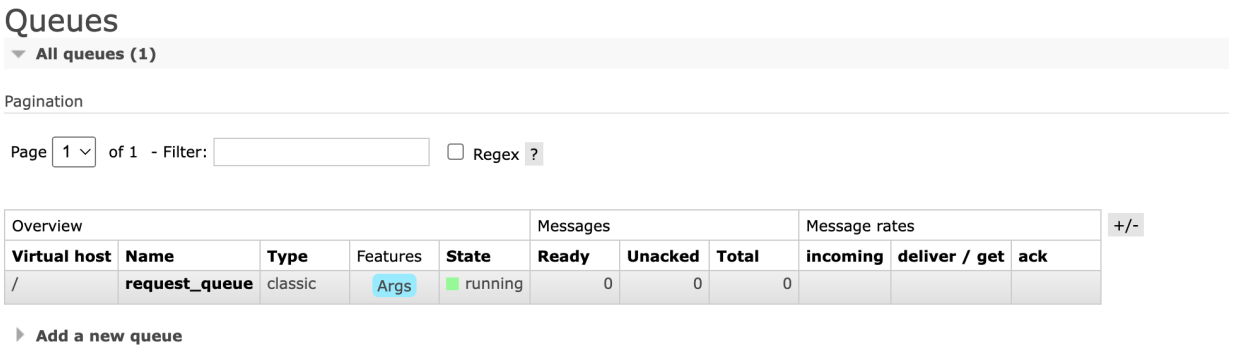
SportsStatsServer.java

This is the application class that represents the server-side code which processes the user requests, converts them to API calls and sends the response back to a queue. This class has the following methods –

- `buildStats`
    - This method is used to enrich a JSON object with details about scores, results
- `main`
    - Instantiates factory, connection and channel for Rabbit MQ
    - Instantiates runtime queue within Rabbit MQ
    - References the main as the current thread and will keep it running forever to handle multiple requests
    - Consume messages from the same queue and run a callback method which will –
        - Parse message and get corresponding team name and id
        - Read the unique correlation id
        - Make an http request using the InvokeHttpClient class
        - Parse the HTTP response using ParseJSON class
        - Instantiate an array of enriched JSON responses using buildStats method
        - Publish the array of JSON as string to the queue
    - Implements a shutdown hook (ctrl + c) on a separate thread which will –
        - Close out the channel and connection associated with RabbitMQ
        - Notify the main thread and gracefully shuts down the program

Rabbit MQ setup

We set up a Rabbit MQ broker running on localhost. This is used to setup a Queue by the code



During client-server interactions we can see 3 channels, (two clients and one server)-

Channels

All channels (3)

Pagination

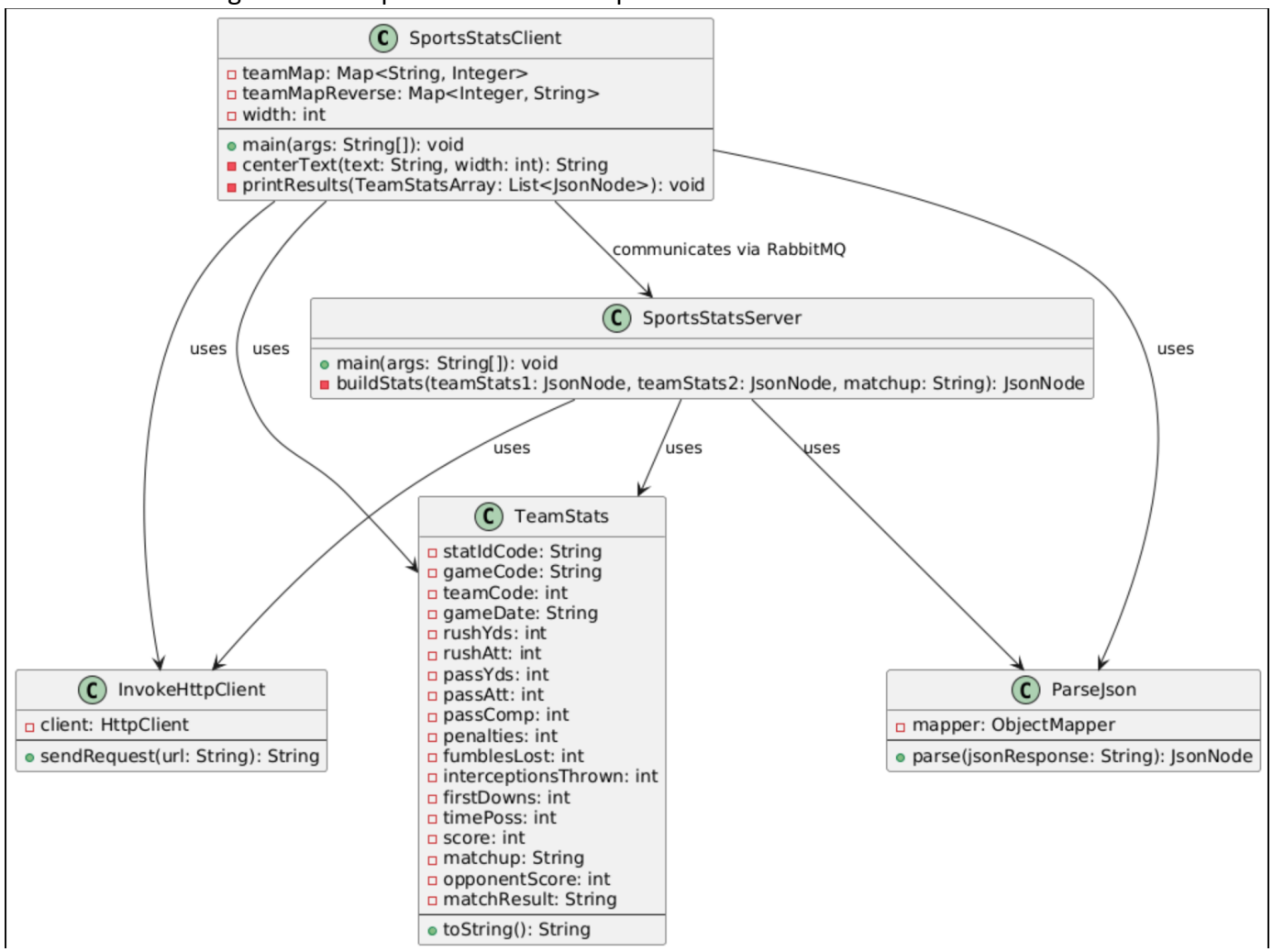
Page 1 of 1 - Filter: Regex ?

Overview				Details			Message rates					+/-
Channel	**User name**	Mode ?	**State**	**Unconfirmed**	Prefetch ?	**Unacked**	**publish**	**confirm**	**unroutable (drop)**	**deliver / get**	**ack**	
127.0.0.1:51883 (1)	guest		idle	0		0						
127.0.0.1:52737 (1)	guest		idle	0		0						
127.0.0.1:52740 (1)	guest		idle	0		0						

AMQP protocol is used to communicate and is built into Rabbit MQ client which is used by SportStats client and server classes.

UML

This is the UML diagram that depicts the relationship of the classes:



Functional Requirements

These were the original functional requirements:

1. `Shall communicate with other servers.`
    a. This is done by two separate processes (`SportsStatsClient` and `SportsStatsServer`)
    b. Every selection by an end-user using `SportsStatsClient` is routed to a `queue` managed within RabbitMQ
    c. The `SportsStatsServer` pulls requests from the same queue, issues a response against the request and sends it back to the same queue
    d. A callback on the client side will parse the response and print to end-user screen
2. `Shall support printing a JSON object in a user-friendly manner to the console or a file with printing of team name, team number, and season record`
    a. This is accomplished by the private method `printResults` in SportsStatsClient.java
    b. This method uses `centerText` method for pretty formatting
3. `Shall store JSON response data object`
    a. The callback within `SportsStatsServer` invokes `buildStats` for enriching information in form of JSON object. It adds all JSON records into a list and converts the entire list as an array and sends this as a final response to the queue.
    b. The callback within `SportsStatsClient` uses object mapper to deserialize string-based array and converts back into list of JSON nodes
    c. It then uses `printResults` to print results in pretty format.

Team numbers

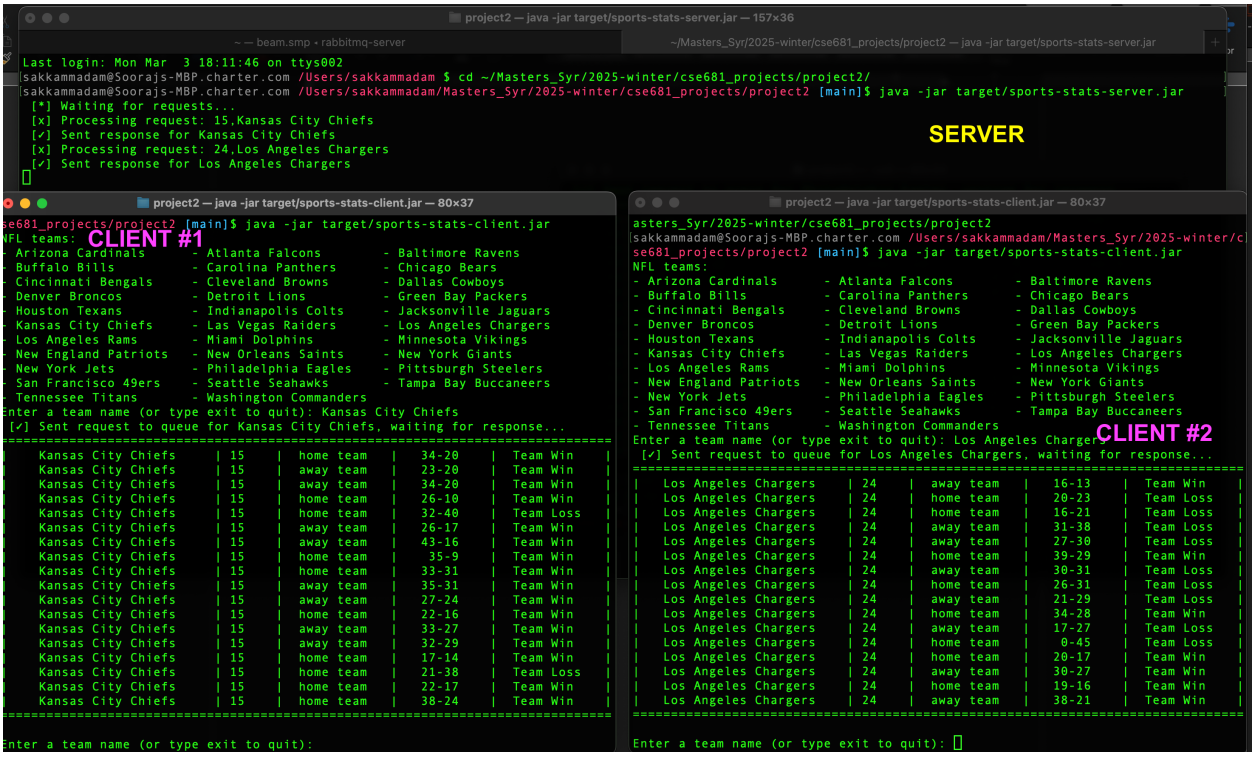
We set up a Tree map to maintain order and load all the different 32 teams.

```
private static final Map<String, Integer> teamMap = new TreeMap<>();
// load the teamMap
static {
        teamMap.put("Arizona Cardinals",1);
        teamMap.put("Atlanta Falcons",2);
        teamMap.put("Baltimore Ravens",3);
        teamMap.put("Buffalo Bills",4);
        teamMap.put("Carolina Panthers",5);
        teamMap.put("Chicago Bears",6);
        teamMap.put("Cleveland Browns",7);
        teamMap.put("Dallas Cowboys",8);
        teamMap.put("Denver Broncos",9);
        teamMap.put("Detroit Lions",10);
        teamMap.put("Green Bay Packers",11);
        teamMap.put("New York Giants",12);
        teamMap.put("Indianapolis Colts",13);
        teamMap.put("Jacksonville Jaguars",14);
        teamMap.put("Kansas City Chiefs",15);
        teamMap.put("Miami Dolphins",16);
        teamMap.put("Minnesota Vikings",17);
        teamMap.put("New England Patriots",18);
        teamMap.put("New Orleans Saints",19);
        teamMap.put("New York Jets",20);
        teamMap.put("Las Vegas Raiders",21);
```

```
        teamMap.put("Philadelphia Eagles",22);
        teamMap.put("Pittsburgh Steelers",23);
        teamMap.put("Los Angeles Chargers",24);
        teamMap.put("Seattle Seahawks",25);
        teamMap.put("San Francisco 49ers",26);
        teamMap.put("Los Angeles Rams",27);
        teamMap.put("Tampa Bay Buccaneers",28);
        teamMap.put("Tennessee Titans",29);
        teamMap.put("Washington Commanders",30);
        teamMap.put("Cincinnati Bengals",31);
        teamMap.put("Houston Texans",32);
    }
```

Based on the values (for the most part), the ids correspond to team names in alphabetical order.

Appendix – Output



This screen shot shows a side-by-side terminal interactions between client#1 and server, and client#2 and server.