# **Tournament Scheduler** Task

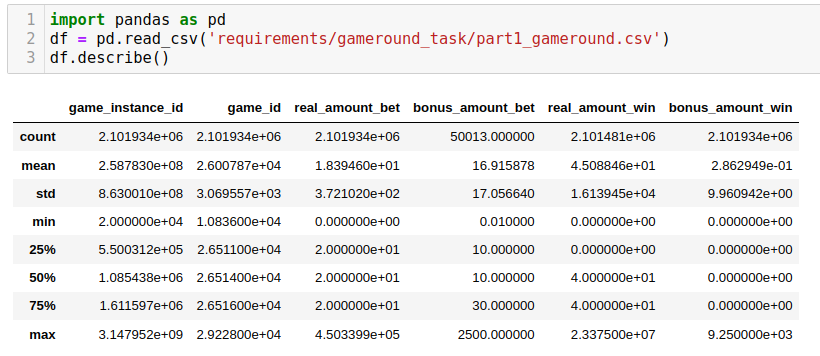
## Prerequisites

In order to complete the project, a virtual machine with a Linux operating system was installed and the following software was configured.

|  |  |
| --- | --- |
| **Software Description** | **Additional Information** |
| Installation of Linux machine on Virtualbox. | Login: Omar  Password omar123 |
| Install clickhouse and run it as server | Login: default  Pass: omar123 |
| Install dbeaver as a RDBMS to see the database. |  |
| Installed Apache NiFi. | URL: https://127.0.0.1:8443/nifi  Username: f9e86d4b-5181-4238-98e7-a9ef703e6a9a  Password: wpj4GH7KKHAh6uc1EzmowuQMo1VzmjLJ |
| Installed Eclipse for Java development. |  |
| Installed docker-compose. | sudo apt install docker-compose |
| Git hub repository | <https://github.com/ozammit/tournamentrepo> |

## Data Exploration

Before starting the provided data was loaded in a pandas DataFrame using python to analyse the given data and the distribution of values. The following information was extracted and the maximum values of each fields noted. This exercise helped determining the maximum size of the decimal number fields in the database.

It was also noted during data exploration that some **bonus\_amount\_bet** and the **real\_amount\_win** values had a null value instead of numerical.

## Task 1

The comma separated file headers were mapped to a suitable Clickhouse data type and a create script was created to store the file raw data before creating the dimension and fact tables. The SQL query used to create the table (located in *queries\staging\_script.sql*) is as follows:

**CREATE** **TABLE** default.Staging\_GameRound

(

created\_timestamp **DateTime**,

game\_instance\_id **Int64**,

user\_id **VARCHAR**(100),

game\_id **Int64**,

real\_amount\_bet Nullable(**Decimal**(10, 2)),

bonus\_amount\_bet Nullable(**Decimal**(10, 2)),

real\_amount\_win Nullable(**Decimal**(10, 2)),

bonus\_amount\_win Nullable(**Decimal**(10, 2)),

game\_name **VARCHAR**(100),

provider **VARCHAR**(100),

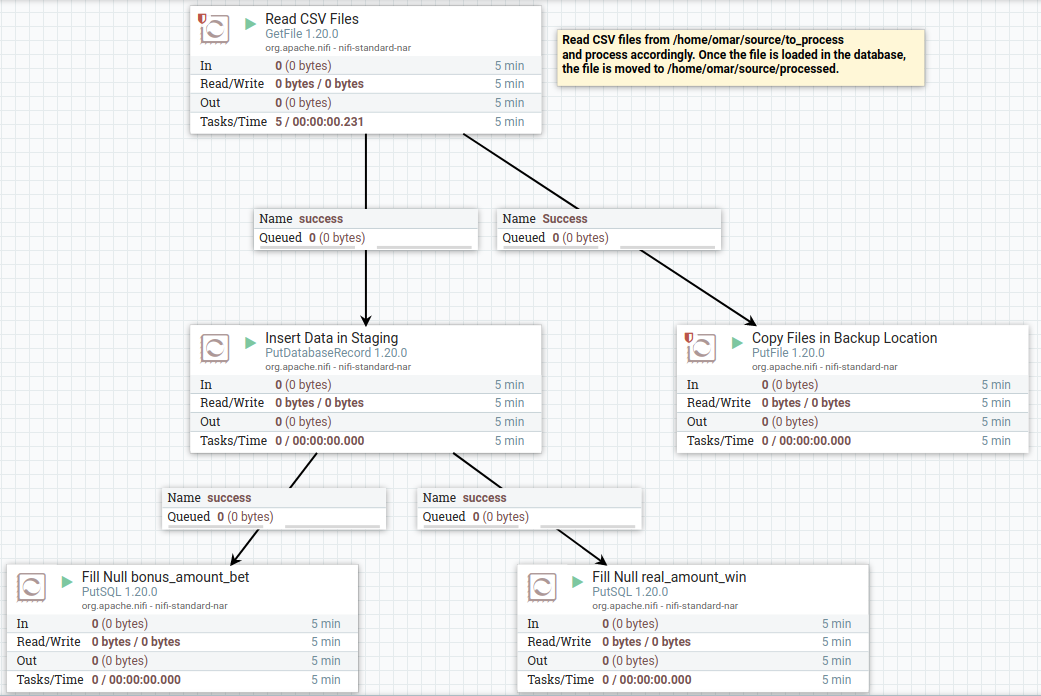
dw\_status **Int8**

)

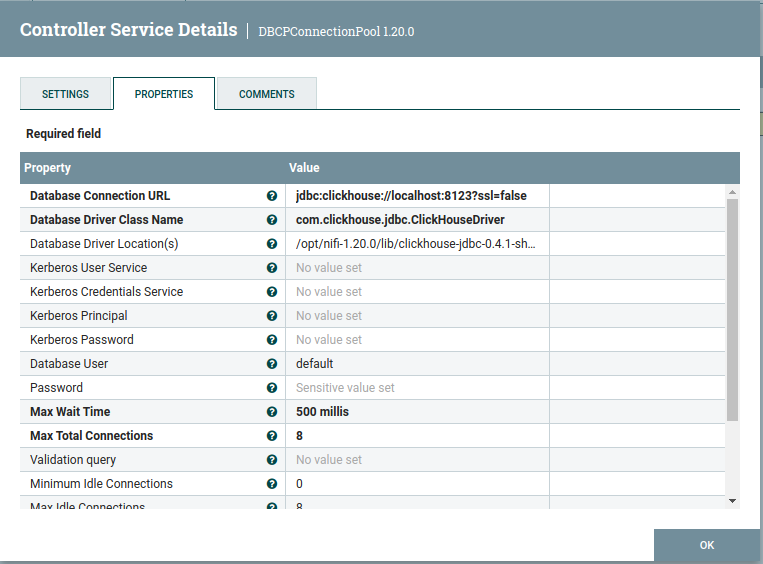
ENGINE = AggregatingMergeTree()

**ORDER** **BY** (created\_timestamp, game\_instance\_id, user\_id, game\_id);

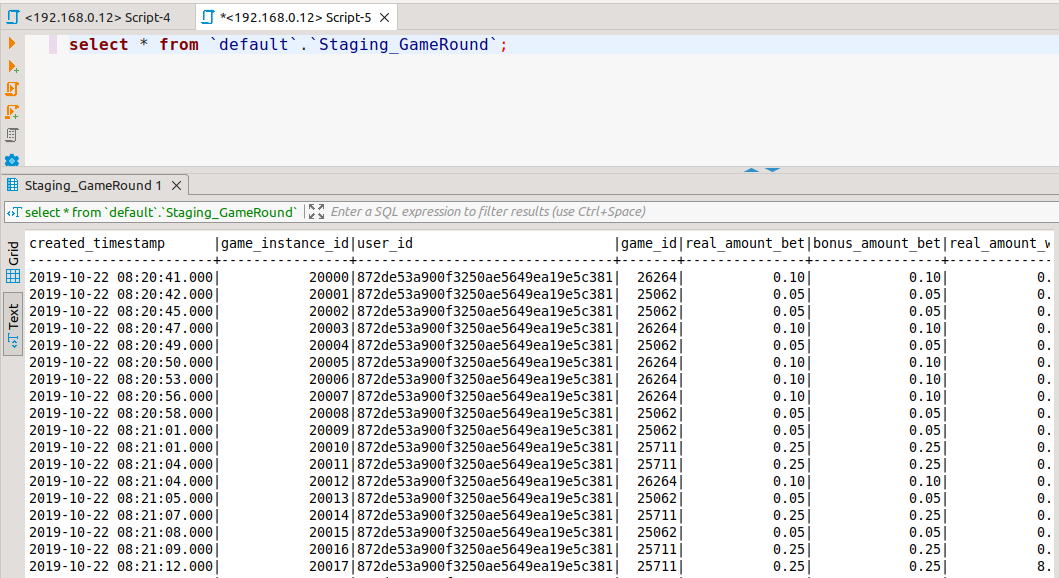
The **Read CSV Files** process reads all files within a specific folder and extracts their contents. The data is inserted into the staging table and nullable fields are updated with a default value.



The following configuration show how connection to Clickhouse was configured.

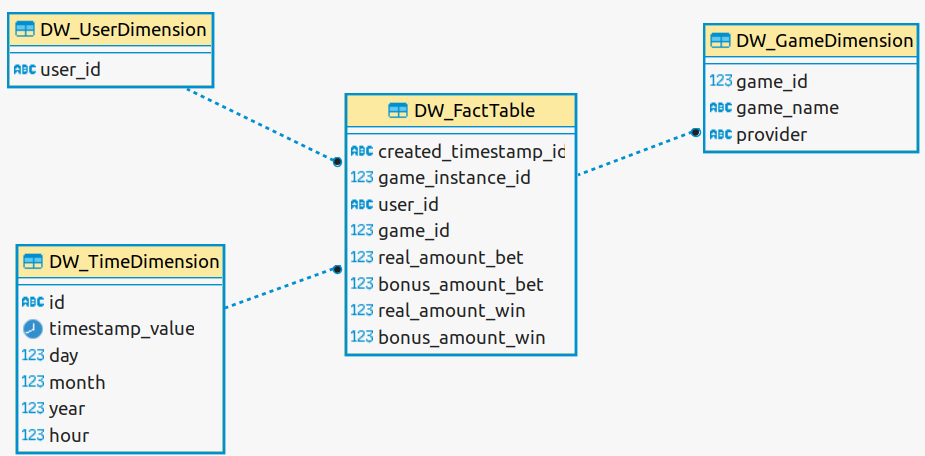


The following is a screenshot of the **default.Staging\_GameRound** table after a process run.

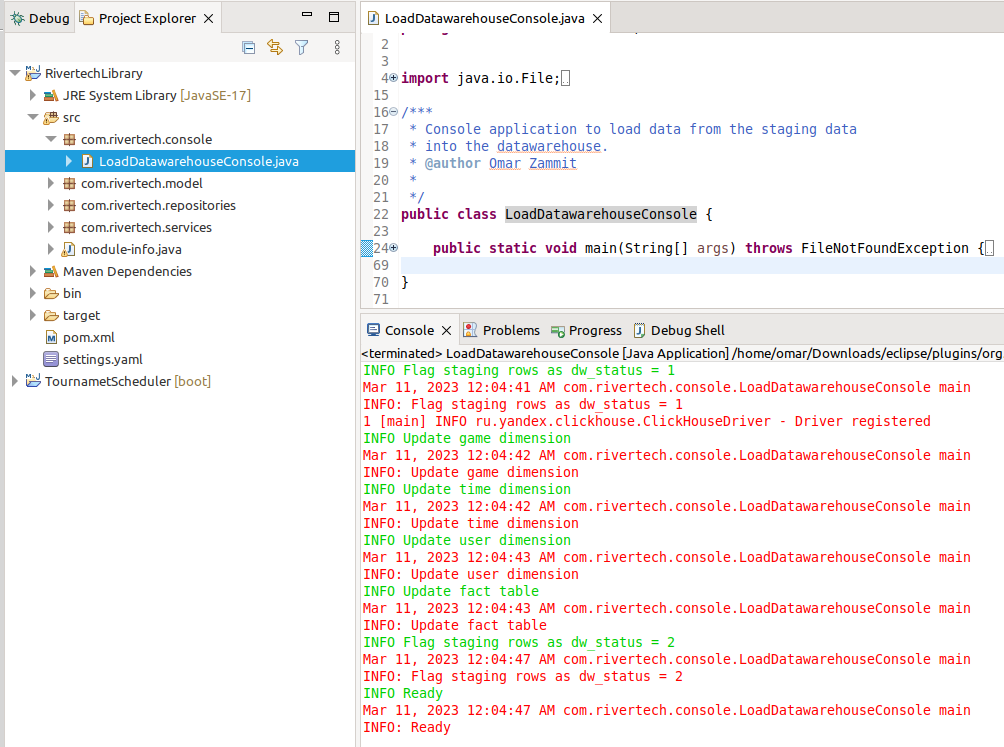


## Task 2

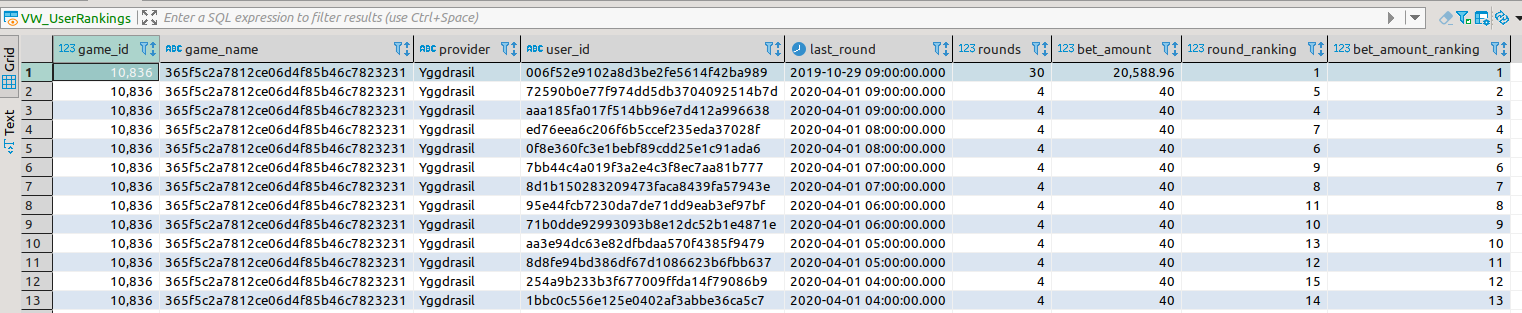
The data warehouse was designed using 3 dimension tables and a fact table as shown below:



The data warehouse queries are located in file *queries\datawarehouse\_script.sql.* In order to populate the data warehouse a Java Maven project (*RivertechLibrary*) was created to act as a library and to provide all the required repositories to manage the data, A console application *LoadDatawarehouseConsole.java* was created to execute various SQL scripts to populate the data warehouse as shown below. **The queries to load the data are located in the *DatawarehouseRepository.java.***



In order to facilitate querying the data, a view was created to compute the rounds and the maximum bets played by each user. The view script is located in *queries\vw\_userratings.sql* and its aim is to group the data accordingly as shown below.



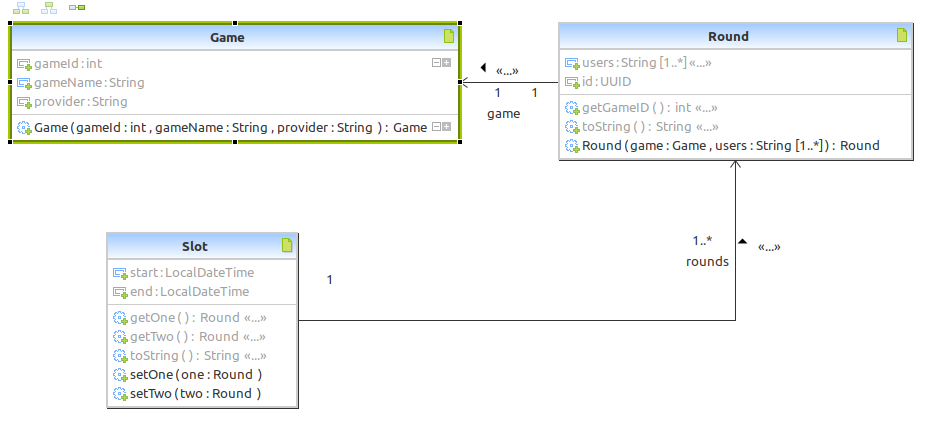
The following image shows all tables populated with data.

## **Task 3**

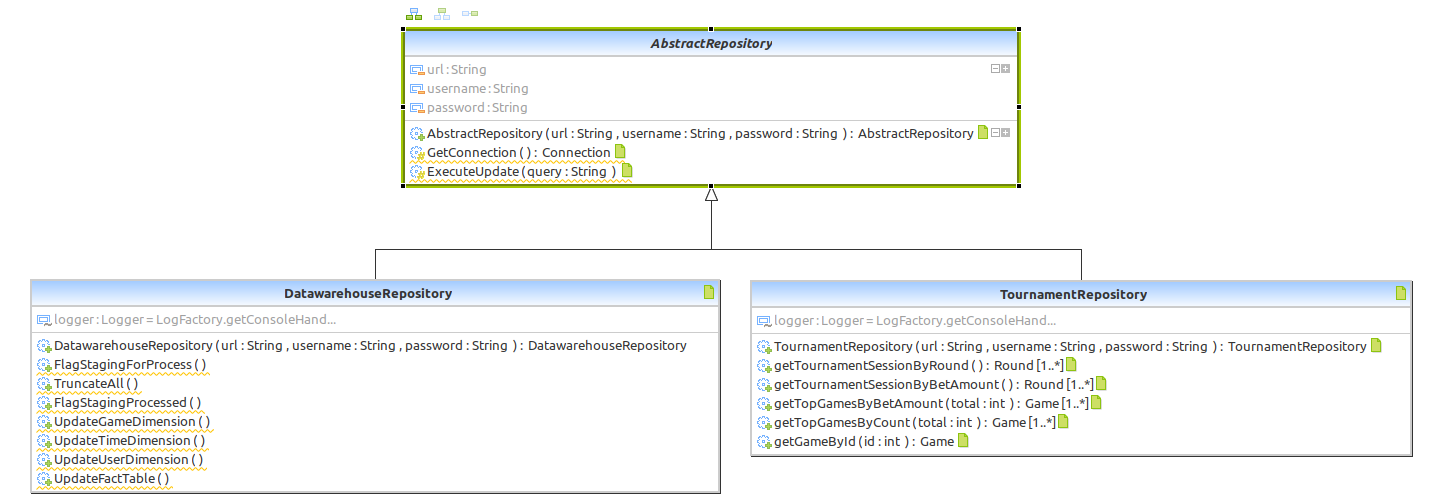
The **RivertechLibrary** was extended to provide the required functionality to get the data from the database and to plan the tournament schedules. In addition a new Spring Boot application (**Tournament Scheduler)** was created to act as an API and provide the data. All Java files are located in the *workspace* folder.

### **Library Classes**

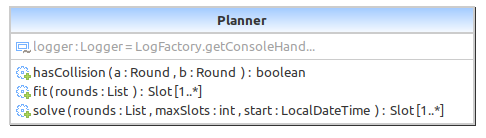
The library model is made up of three classes as shown below:



The repository provides the functionality to connect to the database as shown below:



The planner class is a custom implementation of the scheduler algorithm that was designed and implemented for this project.



Planner Class **fit** method logic:

1. Create a variable *fittedSlots* to store all slots that are filled with rounds.
2. Create a variable *fittedRounds* to store rounds that are already assigned into a slot.
3. For each *roundA* in all available rounds
   1. Check if *roundA* is already fitted in *fittedRounds.*
   2. If fitted ignore and move to the next round.
   3. Create a *slot* variable to store both rounds.
   4. Assign *roundA* to *slot* as the first game.
   5. Add *roundA* to the list of *fittedRounds*.
   6. For each *roundB* in all available rounds
      1. Check if *roundB* is already fitted in *fittedRounds.*
      2. If fitted ignore and move to the next round.
      3. Compare *roundA* and *roundB* using the *hasCollision* method.
      4. If it has a collision criteria ignore and more to the next.
      5. If there is not collision criteria assign *roundB* to *slot* as the second game.
   7. Add *slot* to the list of *fittedSlots.*

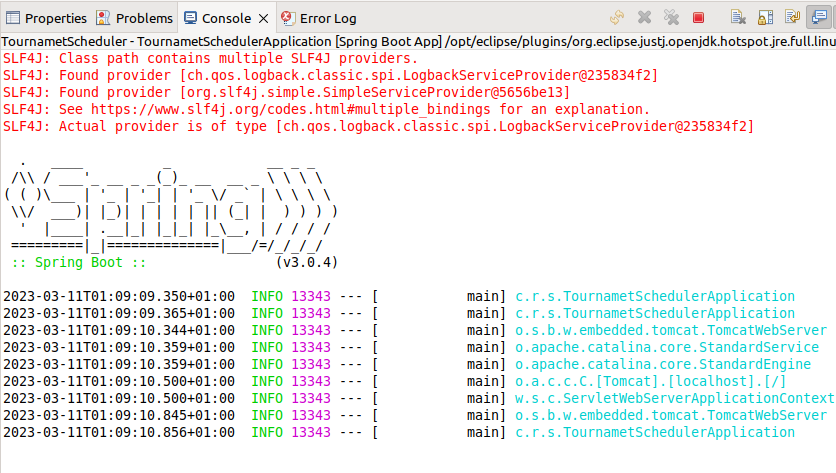
Planner Class **hasCollision** method logic:

1. Given two rounds *a* and *b.*
2. If the unique ID of *a* is equal to *b* then return true.
3. If the game ID of *a* is equal to the game ID of *b* then return true.
4. If the length of the intersection users between *a* and *b* is more than 0 then return true.
5. Return false.

Planner Class **solve** method logic:

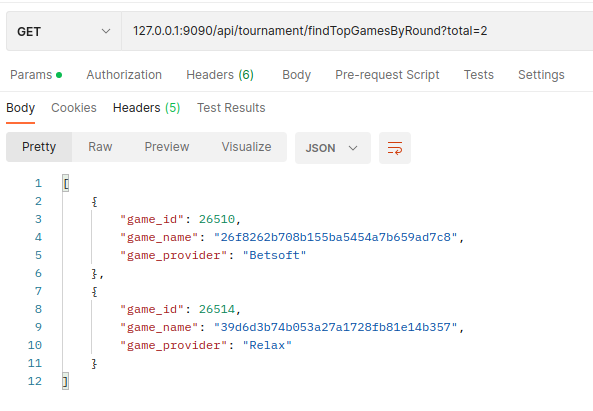
1. Given a list of rounds.
2. Create a variable *slots* to store fitted slots.
3. Loop from 0 to the length of all the available rounds and store iteration into *i*.
   1. Swap round at index 0 with round at index *i*.
   2. Call the *fit* method with the current round list.
   3. If the fitted size is less than or equal to the maximum time slot set *slots* as the current list.
4. Loop through each slot and assign a start time and end time.

**Running the Spring Boot Application**



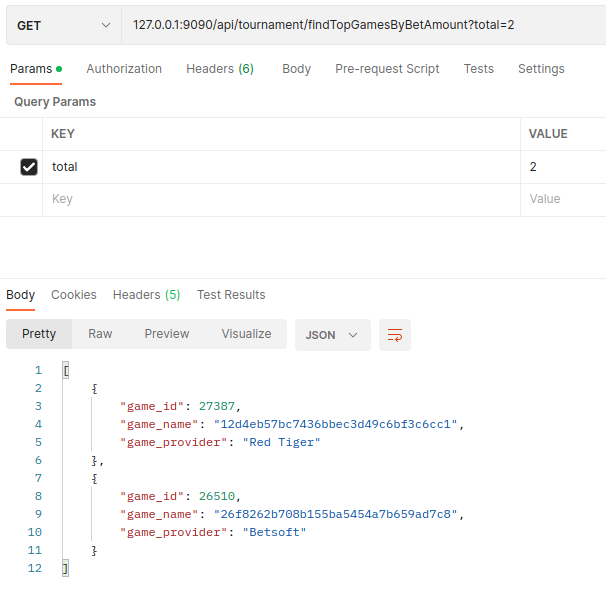
### **API: Get Top Games By Count**

An response example is located in *response/findtopgamebyround.json.*



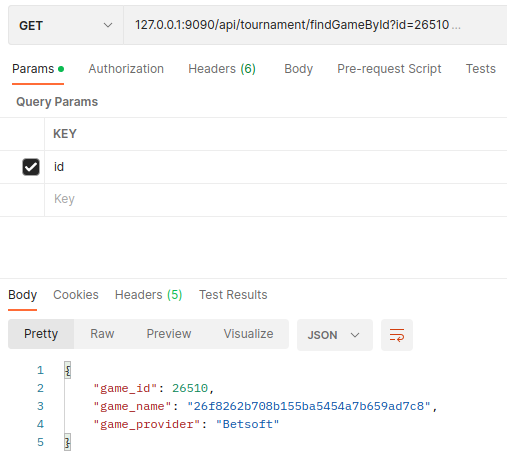
**API: Top Games By Bet Amount**

An response example is located in *response/findgamebybetamount.json.*

****

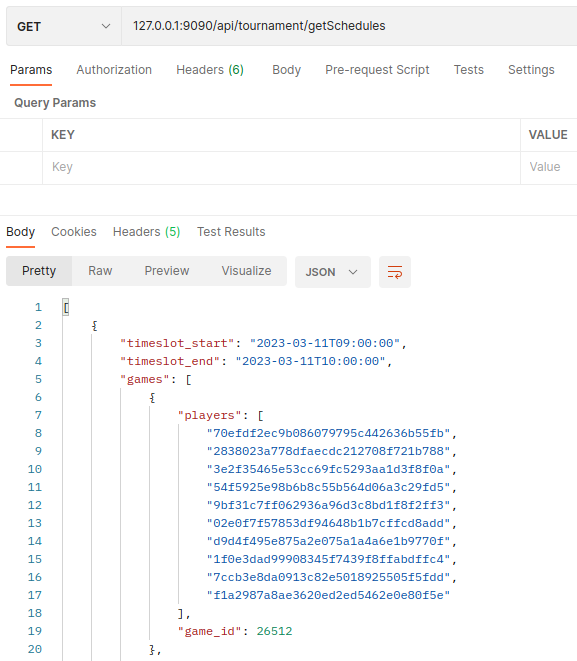
**API: Find a Game**

An response example is located in *response/findgamebyid.json.*

****

**API: Tournament Session**

An response example is located in *response/getschedules.json.*

****