**Row Match Rest API Documentation**

# Endpoints

## Root Path For Endpoints

### URL: localhost:8080/api/v1/

## User Related Endpoints

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| HTTP Method | URI | Description | Example Request Body | HTTP Status | Response Explanation | Example Response Body |
| POST | /users/create | Creates a new user | No Request Body Required | OK - 200 | Successful - User Created | {  "id": 1,  "level": 1,  "coinBalance": 5000 } |
| PUT | /users/updateLevel/{id} | Levels up the user | No Request Body Required | OK - 200 | Succesful - User Created | {  "level": 2,  "coinBalance": 5000 } |
| Not Found - 404 | User with specified id is  not found | {  "timestamp": "2023-03-10T20:48:50.914+00:00",  "errorDetails": "uri=/api/v1/users/updateLevel/0",  "errorCode": "resource\_withId\_not\_found",  "message": "User not found with id : '0'" } |

# Team Related Endpoints

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| HTTP Method | URI | Description | Example Request Body | HTTP Status | Response Explanation | Example Response Body |
| POST | /teams/create | User creates a new team | {  "id": 1, (Optional)  "name": "Team",  "memberCount": 1, (Optional)  "creatorId": 1  } | OK - 200 | Successful - Team created | {  "id": 1,  "name": "Team",  "memberCount": 1,  "creatorId": 1  } |
| Bad Request - 400 | Unsuccessful - Team name does not follow unique constraint | {  "timestamp": "2023-03-10T21:32:25.083+00:00",  "errorDetails": "uri=/api/v1/teams/create",  "errorCode": "unique\_field",  "message": "Team with same name already exists : “Team'"  } |
| Internal Server Error - 500 | Unsuccessful - User already has a team | {  "timestamp": "2023-03-10T21:33:16.857+00:00",  "errorDetails": "uri=/api/v1/teams/create",  "errorCode": "error",  "message": "User is already a member of a team"  } |
| Forbidden - 403 | Unsuccessful - User has insufficient balance to create team | {  "timestamp": "2023-03-10T21:35:29.789+00:00",  "errorDetails": "uri=/api/v1/teams/create",  "errorCode": "insufficient\_balance",  "message": "Insufficient balance to create team"  } |
| PUT | /teams/join | User joins a team | {  "userId": 2,  "teamId": 1  } | OK - 200 | Successful - User Joined Team | {  "id": 1,  "name": "Team",  "memberCount": 2,  "creatorId": 1  } |
| Not Found - 404 | Unsuccessful - User with specified id is  not found | {  "timestamp": "2023-03-10T20:48:50.914+00:00",  "errorDetails": "uri=/api/v1/users/updateLevel/0",  "errorCode": "resource\_withId\_not\_found",  "message": "User not found with id : '2'" } |
| Not Found - 404 | Unsuccessful - Team with specified id is  not found | {  "timestamp": "2023-03-10T21:46:05.018+00:00",  "errorDetails": "uri=/api/v1/teams/join",  "errorCode": "resource\_withId\_not\_found",  "message": "Team not found with id : '1'"  } |
| Internal Server Error - 500 | Unsuccessful - User already has a team | {  "timestamp": "2023-03-10T21:33:16.857+00:00",  "errorDetails": "uri=/api/v1/teams/create",  "errorCode": "error",  "message": "User is already a member of a team"  } |
| Internal Server Error - 500 | Unsuccessful – Team is full | {  "timestamp": "2023-03-10T21:51:06.141+00:00",  "errorDetails": "uri=/api/v1/teams/join",  "errorCode": "error",  "message": "Team is full"  } |
| GET | /teams/getTeams | Gets specified number of teams with open spots randomly | No Request Body Required | OK - 200 | Successful – Returned list of random teams | [  {  "id": 6,  "name": "Sixth team",  "memberCount": 1,  "creatorId": 2  },  {  "id": 9,  "name": "Ninth team",  "memberCount": 1,  "creatorId": 1  }  ] |
| Not Found - 404 | Unsuccessful – There are no teams which have an empty spot | {  "timestamp": "2023-03-10T22:09:15.328+00:00",  "errorDetails": "uri=/api/v1/teams/getTeams",  "errorCode": "no\_resources\_found",  "message": "There are no teams with empty spots"  } |

# One can test these endpoints by running the application and heading over to [Swagger UI](http://localhost:8080/api/v1/swagger-ui/index.html#/).

# Implementation Details and Design Choices

# General Aim and Architecture

The project is a backend implementation for a game called Row Match using Spring Boot and MySQL. The software is structured into multiple layers, including the Controller layer for handling endpoints,

the Service layer for implementing the business logic, and the Repository layer for access to database and ORM.

* The Controller layer receives HTTP requests, makes calls to the corresponding methods in the Service layer, and returns appropriate responses.
* The Service layer implements game related business logic, such as creating users, joining teams and leveling up users. It also interacts with the Repository layer for reading and writing data from the database.
* The Repository layer uses Spring Data JPA to provide CRUD (Create, Read, Update, Delete) operations to the database. It defines interfaces for data access and provides implementations automatically based on the defined models.
* The Model layer defines data models for entities such as User, Team, and Configuration. These models include properties, relationships, and validation constraints that are used across the application.

By splitting the code into these distinct layers, it is possible to have a project which is easier to understand, maintain, and extend. The application’s business logic can be modified in the Service layer without affecting the Controller layer or the Model and Repository layers. Similarly, it is possible to swap out the database technology used in the Repository layer without affecting the rest of the application. Overall, this architecture provides flexibility.

In order to provide further flexibility and scalability, a set of configurations such as “Gained Coin Per Level Win” and “Maximum Capacity Of A Team” are held in the database and fetched at the beginning of the application. This way, multiple instances running on different systems can be set with the same configurations and without the risk of inconsistency whereas if the configurations were hardcoded or fetched from a file it would be more complicated to make changes and would create risk of forgetting to change in one of the instances. In this implementation, the application must be restarted for the changes to take effect. However, some simple put endpoints could be implemented to change these configurations instead of changing them manually from the database which would allow the application to invalidate the cached configuration object and fetch it from the database again. As this is out of the scope of this project, this functionality is not implemented.

In this project, dependency injection is used to avoid having hard-dependencies and obtain more modular, testable and extensible code. This way, in unit tests it is possible to mock the dependencies of a class instead of using the actual class which helps to achieve isolation. Moreover, dependency injection provides decoupling since if the code is wanted to be extended in the future with other implementations of some dependencies, dependency injection would allow the developers to use the new implementations without any changes in the injected class.

* 1. Models  
     1. Configuration  
          
        The configuration model represents the configurations table in MySQL, and it is implemented to store the configurations of the game. It includes “Starting Coin Balance”, “Coin Gained Per Level Win”, “Max Capacity of a Team”, “Team Creation Cost” and “Number of Teams with Empty Spot to Get”. The id field is hardcoded since there will be only one configuration object. All fields have not-null constraint as they are all essential.
     2. Team  
          
        The team model represents the teams table in MySQL, and it is implemented to represent the attributes of a team in Row Match. The attributes are as such: Id, Team Name, Member Count and Creator Id. The member count is included in the team entity since it is predicted to be needed frequently due to the Get Teams endpoint which fetches all the teams with an empty spot as the first step and in a game, there can be millions of teams. If this attribute was not required this frequently, the same functionality could be achieved by using sum function of MySQL. The id field is automatically handled by the database and incremented one by one.
     3. User (User Progress)  
          
        The team model represents the teams table in MySQL, and it is implemented to represent the attributes of a team in Row Match. The attributes are as such: Id, Level, Coin Balance and Team Id. The requirements did not have the team id or any real relationship between user and team however it is added to bring more functionality and logic. The team id is kept in the User entity instead of a separate table for user-table relationship since traditionally every user can only have one team at a time. However, such model still could be implemented in order to add additional information or metadata to the relationship itself, such as the date the user joined the team, their role in the team, or any other relevant information.
  2. Controllers  
     1. User Controller

The user controller class is responsible for handling the incoming HTTP requests and providing the appropriate responses to the client. The @RestController annotation is added to indicate to Spring that this class is a controller and it should return a response body in the HTTP response. The base URL is set to localhost:8080/api/v1/users for the endpoints under this class. The user controller depends on the user service which is injected using constructor injection. The controller includes two endpoints, one to create