Row Match Rest API Documentation

1. Endpoints

1.1. Root Path For Endpoints

URL: localhost:8080/api/v1/

1.2. 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'" }

1.3. 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	{ "userId": 1 "name": "Team" }	OK - 200	Successful - Team created	{ "id": 1, "name": "Team", "memberCount": 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" }
				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_withField_not_found", "message": "User not found with id : '1'" }
			{ "userId": 1 }	Bad Request - 400	Unsuccessful – Team name is empty	{ "name": "name must be sent and cannot be empty" }
PUT	/teams/join	User joins a team	{ "userId": 2, "teamId": 1 }	OK - 200	Successful - User Joined Team	{ "id": 1, "name": "Team", "memberCount": 2, }
				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_withField_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_withField_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,
						},
						{
						"id": 9,
						"name": "Ninth team",
						"memberCount": 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.

2. Implementation Details and Design Choices

(The design choices and reasons for these choices are italicized for easier recognition)

2.1. 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 so that there is no need for a new query to database for every time these configurations are needed. 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.

2.2. Models

2.2.1. 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.

2.2.2. 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 and Member Count. *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.

2.2.3. 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. *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.* In the model, it looks like the user entity has a team object however ORM converts this to have teamId field in the users table which is a foreign key. Also, since one user may have one team at most, but one team may have many users, it is marked as @ManyToOne. It is designed in the way that if the user is not a member of a team, the team is set to null. *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.*

2.3. Controllers

The controller classes are 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 these classes are controllers, and they should return a response body in the HTTP response.

2.3.1. User Controller

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 for creating a new user (user progress) which handles HTTP post requests and another for leveling up a user which handles HTTP put requests. Example inputs and possible outputs with respective status codes can be found in the 1.2. User Related Endpoints section. Also, these endpoints can be tried with the help of Swagger.

2.3.2. Team Controller

The base URL is set to localhost:8080/api/v1/teams for the endpoints under this class. The team controller depends on the team service which is injected using constructor injection. The controller includes three endpoints, one for creating a new team which handles HTTP post requests, one for joining a team which handles HTTP put requests and another for getting x amount (specified in the configurations) of teams with open spots in them randomly which handles HTTP get requests. Example inputs and possible outputs with respective status codes can be found in the 1.3. Team Related Endpoints section. Also, these endpoints can be tried with the help of Swagger.

2.4. Exceptions

Some custom exceptions are created in cases which these exceptions can be used for multiple different actions and results. For example, Insufficient Balance Exception is one of these custom exceptions and it can be used for any action and resource when the users balance in insufficient to perform that action with the specified resource. For the rest, Runtime Exception is thrown. Both custom and default exceptions are handled in the Global Exception Handler class and formatted to output a descriptive error response.

2.4.1. Insufficient Balance Exception

Insufficient Balance Exception is thrown when a user tries to perform an action on a particular resource, but performing such task requires a payment and the user's balance is insufficient. This exception takes the action name and resource name as parameters to have multiple use cases and not create separate exceptions for every action that needs payment. For example, if the user wants to create a team but does not have the sufficient balance "Insufficient balance to create a team" with Forbidden status code will be returned.

2.4.2. No Resources Found Exception

No Resources Found Exception is thrown when a list of records is queried but there are no records which apply to the query. This exception takes resource name as a parameter to provide a general exception for all kinds of resources. For example, if user tries to get 10 teams with empty spots but there is none, "There are no teams with empty spots" with Not Found Status code will be returned.

2.4.3. Resource With Field Not Found

Resource With Field Not Found exception is thrown when a specific resource is queried by id but it is not found. This exception takes resource name, field name and field value as a parameter to provide a general exception for all kinds of resources. For example, if the user with id=3 is queried but there is no such user, then "User not found with id: '3'" with Not Found Status code will be returned.

2.4.4. Unique Field Exception

Unique Field Exception is thrown when the unique constraints of the database are not followed. This exception takes resource name, field name and field value as a parameter to provide a general exception for all kinds of resources. For example, when a team with name "First Team" is tried to be created but there already is a team with the same name, "Team with same name already exists: 'First Team'" with Bad Request Status Code.

2.5. Services

The services are implemented in two layers, interfaces and the implementations to achieve abstraction which provides a more modular, extensible and decoupled code.

2.5.1. User Service

User Service has 6 methods, 2 of them are main functions directly related with the endpoints and the remaining 4 are the helper functions.

2.5.1.1. Create User

Creates a new user object. Since the id field is automatically managed by the database and Level / Team fields are initialized with the default values which are 1 and null only the coin balance is set according to the configurations set in the database. After setting up the object, it is saved into the database. *Then, in order to exclude the Team field from the response, which is not wanted according to the requirements, the user's respective fields are mapped to a New User DTO and the DTO is returned to the controller.*

2.5.1.2. Update Level

Levels up the user with the passed user id. First finds the user with the specified id, if no such user is found a Resource With Field Not Found Exception is thrown. If the user is found, the user's level is incremented by 1 and the coin balance is incremented by the specified amount in the configurations. After the user object is set, the user is saved to the database. *Then, in order to exclude the User Id and the Team fields from the response, the user's respective fields are mapped to a Progress DTO and the DTO is returned to the controller.*

2.5.1.3. Check Balance

Checks if the user with the passed id has more coins than the passed cost value. First finds the user with the specified id, if no such user is found a Resource With Field Not Found Exception is thrown. If the user is found, and the user's balance is more than the cost, returns true; returns false otherwise.

2.5.1.4. Is Member Of Team

Checks if the user with the passed id is enrolled in a team. First finds the user with the specified id, if no such user is found a Resource With Field Not Found Exception is thrown. If the user is found, and the team field of the user is different than null, returns true; returns false otherwise.

2.5.1.5. Set Team

Sets the Team of the user with passed user id to passed Team object. First finds the user with the specified id, if no such user is found a Resource With Field Not Found Exception is thrown. If the user is found, the Team object of the user is set to the passed team and with the save function only the teamId of the team is written to the user's record. It does not return anything thanks to the Transaction Management in the team service's methods. No checks need to be made to understand whether the operation was successful or not in the calling team service method. *This method is also marked with @Transactional to prevent inconsistency due to concurrency. In case of multiple rapid requests to the server, the data read from the database may be outdated and both requests may be processed with the same initial data. This would result in only one request's result being committed and the other's result being discarded. So, @Transactional annotation is necessary to achieve atomicity.*

2.5.1.6. Deduct From Balance

Deducts the cost value from the user with the passed user id. First finds the user with the specified id, if no such user is found a Resource With Field Not Found Exception is thrown. If the user is found, then the cost is deducted from the balance and the user object with the new balance is saved to the database. *This method is marked with @Transactional to prevent inconsistency due to concurrency. In case of multiple rapid requests to the server, the data read from the database may be outdated and both requests may be processed with the same initial data. This would result in only one request's result being committed and the other's result being discarded. So, @Transactional annotation is necessary to achieve atomicity.*

2.5.2. Team Service

Team Service has 3 methods directly related to the endpoints.

2.5.2.1. Create Team

Creates a team for a user with the passed Id and team name. Three checks are made:

- Check if there already is a team with the same name -> Throw Unique Field Exception
- Check if the user is already a member of a team -> Throw Runtime Exception
- Check if the user has sufficient balance to create a team -> Throw Insufficient Balance Exception

Also, an implicit check is done for user's existence functions this method calls. If the check fails it throws Resource With Field Not Found Exception.

If all checks are passed, save the new team to the database, get the created team object, set the users team with this team and deduct the cost of creating team which is specified in the configuration from the user's balance. Created team is returned.

Since even one of these operations failing when others are successful would create an inconsistency in the database, @Transactional annotation is used. If all operations are successful, then the changes are committed. If even one of them is unsuccessful, all successful operations are rolled back.

2.5.2.2. Join Team

Makes a user with the passed user id member of an existent team with the passed team id. Three checks are made:

- Check if there is a team with the specified team id -> Resource With Field Not Found Exception
- Check if the user trying to join is already a member of a team -> Runtime Exception
- Check if the team to be joined is full -> Runtime Exception

Also, an implicit check is done for user's existence functions this method calls. If the check fails, it throws Resource With Field Not Found Exception.

If all checks are passed users team is set to the joined team, team's member count is incremented and updated team is written to the database. Joined team is returned.

Since even one of these operations failing when others are successful would create an inconsistency in the database, @Transactional annotation is used. If all operations are successful, then the changes are committed. If even one of them is unsuccessful, all successful operations are rolled back.

2.5.2.3. Get Teams

Gets specified number of teams with open spots randomly. *How many teams to get is set in the configurations. This is designed like that since the most likely use case of this endpoint is when a user without a team wants to join a team and gets in the Teams view. Some number of teams with empty spots will be shown to the user in case the user is not looking for a specific team. So, it makes sense to change the number of teams to show in this screen based on some analytics etc.*

All teams with at least one empty spot are pulled into a list. There are three possibilities:

- If there are more teams which have an empty spot than the number of teams to get:
  - Pick specified number of teams with empty spots randomly from the gathered list. *An easy and popular implementation would be to shuffle this list using Collections.shuffle() and get the first k number of teams to fulfil the desired functionality. However, considering in a game, there may be millions of teams this implementation would not be efficient as Collections.shuffle()'s time complexity is O(n) where n is the number of teams. Instead, picking random k indices and picking the teams from those indices without replacement gives the desired output without duplicate teams in O(k) time complexity where k is the number of teams to return. As for the most times, expected scenario is O(n) >> O(k) which shows this implementation is far more efficient.* The list of teams with picked indices are returned.
- If there are less teams which have an empty spot than the number of teams to get:
  - Randomize the list gathered from the last query and return it as it is. *Here, Collections.shuffle() is used instead of the algorithm explained above since in this scenario it is guaranteed to have k=n. So both algorithms work efficiently. To prevent clutter, simpler implementation is used.*

- If there are no teams with empty spots -> Throw No Resources Found Exception

2.5.3. Configuration Service

2.5.3.1. Fetch Configurations

Gets the configuration record from the database for initialization of game configurations. *The top first record is read instead of the record with id = 1 to decrease the possibility of an error in case the record gets a different id by mistake. Since the configuration record is very essential for the game as it is not possible for the game to run without them, if the configurations are not set, a default configurations record is loaded to the database and returned to the configurations initializer.* If the record already exists, the method returns that configurations object.

2.6. Repositories

Implements the data access layer. Responsible from CRUD operations and ORM.

2.6.1. Configurations Repository

This class extends the JPA Repository which provides basic functions for CRUD operations such as findAll(), save() etc. Also, one finder method is implemented which helps to get the first record by id. It is also implemented automatically by JPA.

2.6.2. Team Repository

This class extends the JPA Repository which provides basic functions for CRUD operations such as findAll(), save() etc. Also, two finder methods are implemented which help to get the teams with member count less than the passed argument and get a team by name field. They are also implemented automatically by JPA.

2.6.3. User Repository

This class extends the JPA Repository which provides basic functions for CRUD operations such as findAll(), save() etc. No custom methods are implemented.

2.7. Configurations

2.7.1. Configuration Initializer

This class is used to get the configurations from the database right after Spring application runs by implementing CommandLineRunner. The configurations are pulled from the database using Fetch Configurations from configuration repository and they are set to a configuration object which then be used application wide.

2.7.2. Open API Configuration

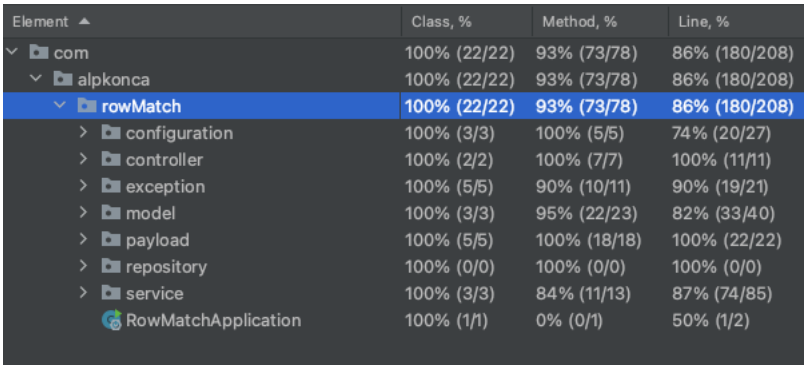
This class is used to configure Swagger.

2.7.3. SQL Initializer

Details of this class will be discussed in 3. Migration part.

2.8. Unit Tests

There are 34 different Unit Tests. These tests are written for testing the controller layer and the service layer. Coverage report can be seen in the Figure 1 below or in the interactive coverage report html.



Element	Class, %	Method, %	Line, %
com	100% (22/22)	93% (73/78)	86% (180/208)
alpkonca	100% (22/22)	93% (73/78)	86% (180/208)
rowMatch	100% (22/22)	93% (73/78)	86% (180/208)
configuration	100% (3/3)	100% (5/5)	74% (20/27)
controller	100% (2/2)	100% (7/7)	100% (11/11)
exception	100% (5/5)	90% (10/11)	90% (19/21)
model	100% (3/3)	95% (22/23)	82% (33/40)
payload	100% (5/5)	100% (18/18)	100% (22/22)
repository	100% (0/0)	100% (0/0)	100% (0/0)
service	100% (3/3)	84% (11/13)	87% (74/85)
RowMatchApplication	100% (1/1)	0% (0/1)	50% (1/2)

Figure 1

2.8.1. Controller Tests

The unit tests are split into three parts: Arrange, Act, Assert. In the arrange part, necessary arrangements are done such as creating objects necessary for the unit test to run. In the act part, a request is sent to the endpoint using MockMvc. In the assert part, the actual output is compared with the desired output.

Each endpoint in the controllers is tested in isolation using mock objects of its dependencies. Using when/then expected outputs from the dependencies are specified and calls to these dependencies are resulted with these outputs. The tests are done for every logical scenario which are successful or have a dedicated exception / are expected. The tests check the following:

- For successful operations -> Check if the returned object(s) is(are) as desired
- For unsuccessful operations -> Check the status code and the error message

2.8.2. Service Tests

The unit tests are split into three parts: Arrange, Act, Assert. In the arrange part, necessary arrangements are done such as creating objects necessary for the unit test to run. In the act part, a call is made to the method being tested. In the assert part, the actual output is compared with the desired output. There are no unit tests for void methods since it is pointless to check if setTeam is working with a mock repository layer as all it does is update the database.

Each method in the services is tested in isolation using mock objects of its dependencies. Using when/then expected outputs from the dependencies are specified and calls to these dependencies are resulted with these outputs. The tests are done for every logical scenario which are successful or have a dedicated exception / are expected. The tests check the following:

- For successful operations -> Check if the returned object(s) is(are) as desired
- For unsuccessful operations -> Check if the thrown exception is of right kind

3. Migration

3.1. Database and Table Creation

Database and Tables are automatically created if they do not exist due to JPA and createDatabaseIfNotExists=true query parameter.

3.2. Populating Sample Data

Except the configuration record, which is very essential for the game to run, sample data is populated if the users and teams table are empty at the start of the application. The SqlInitializer which implements Application Runner starts at the beginning of the program and checks if the tables are empty. If both are empty, it reads the insert.sql script line by line and executes the insert queries.