Row Match Software Documentation

By Ata Ernam

March 28, 2023

Usage And Requirements

1. Background Information and Pre-Requisites
2. Code Implementation Overview
    a. Architecture
    b. Code Design & Database Design
    c. Repositories
    d. Models
    e. Services
    f. Controllers
3. Current Bugs and Problems

Chapter One

BACKGROUND INFORMATION AND PRE-REQUISITES

Description:

Row Match is a casual mobile game played by millions of users worldwide. This Rest API s designed to keep players of this game progress secure and have a fast performance for a better player experience.

The backend services are written in Java with Spring Boot. As for the database, MySQL has been used for data storage and the H2 database engine was used for in-memory storage which was crucial for the implementation of unit tests.

PRE-REQUISITES:

External Apps:

The use of external apps such as Postman and MySQL Workbench is highly encouraged since it makes testing our code while implementing much easier. Postman can be used for quickly testing API requests and organizing our request routes.
MySQL Workbench can be used for easy access to our data and to keep in track of the data that we will create during the implementation

Dependencies:

Firstly, Spring Web libraries are needed to develop the base of our code. Spring Data JPA libraries will be used for the connection between the database and the application. For databases, MySQL will be used as the main data storage tool and H2 database will be used for in-memory processes which will be the unit tests. For unit testing, JUnit Jupiter dependencies will be used.

Chapter Two

CODE IMPLEMENTATION OVERVIEW

Architecture:

The model-view-controller (MVC) software architectural paradigm is used to develop the backend. This design option was taken because the architectural pattern made development easier by providing clearly defined tiers of web activities for each item. This layer separation is:

- **Models**: Entities that we will be making transactions on and which hold the data. In our case, these entities are Team and User.

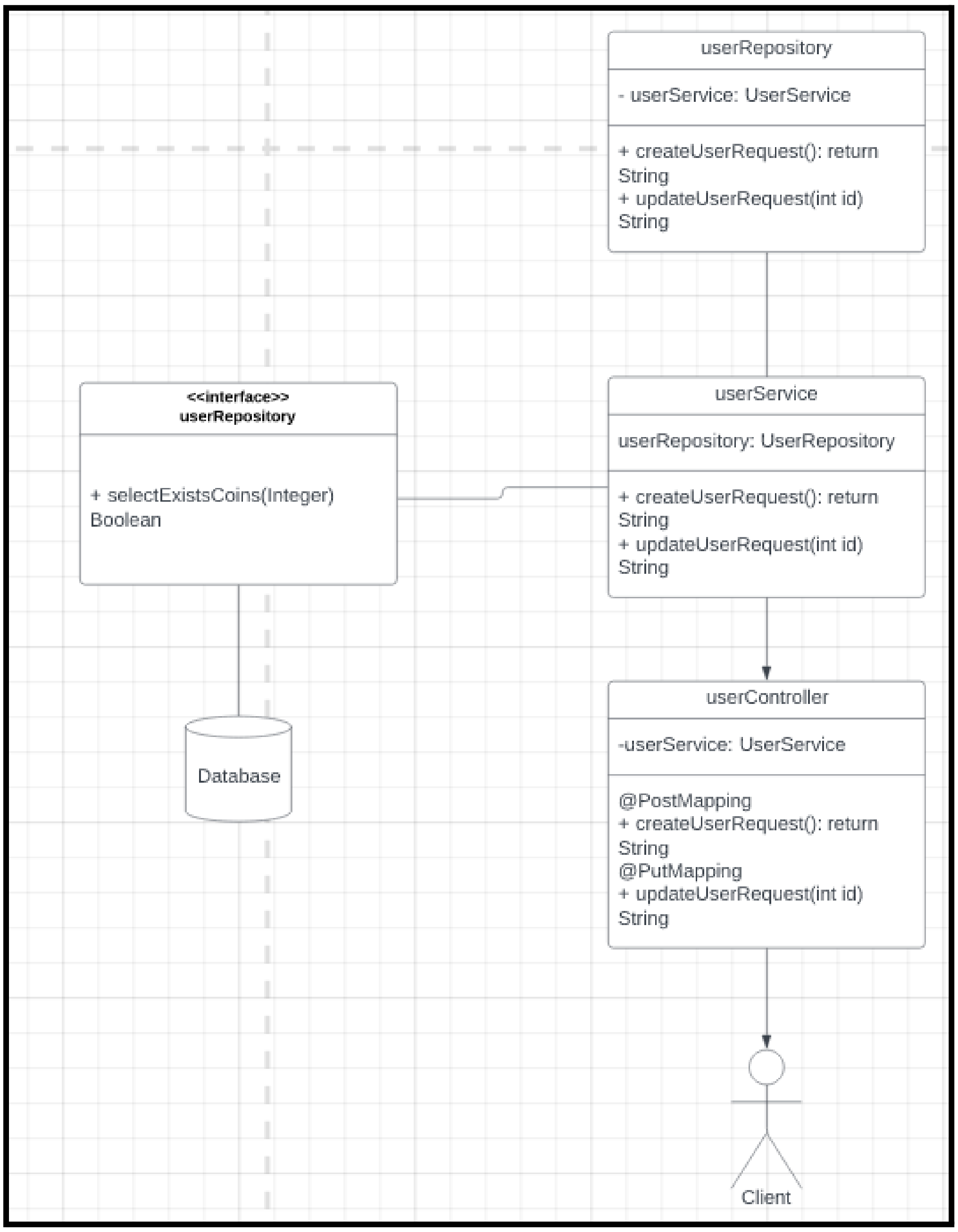
- **Repositories**: JpaRepository is a Repository extension that is JPA-specific. It includes a full API CrudRepository as well as a PagingAndSortingRepository. So, basically, Jpa Repository contains APIs for basic CRUD operations, pagination APIs, and sorting APIs. In our case, we can define different repositories for each entity. So, team and user entities can have their own CRUD operations and more specific operations which we can define for our requirements.

- **Services**: Services are CRUD operations that we can define which then can interact with our repositories to give or transform the necessary information that we need. In our case, both team and user have their own services with operations intended for both the players and the admins.

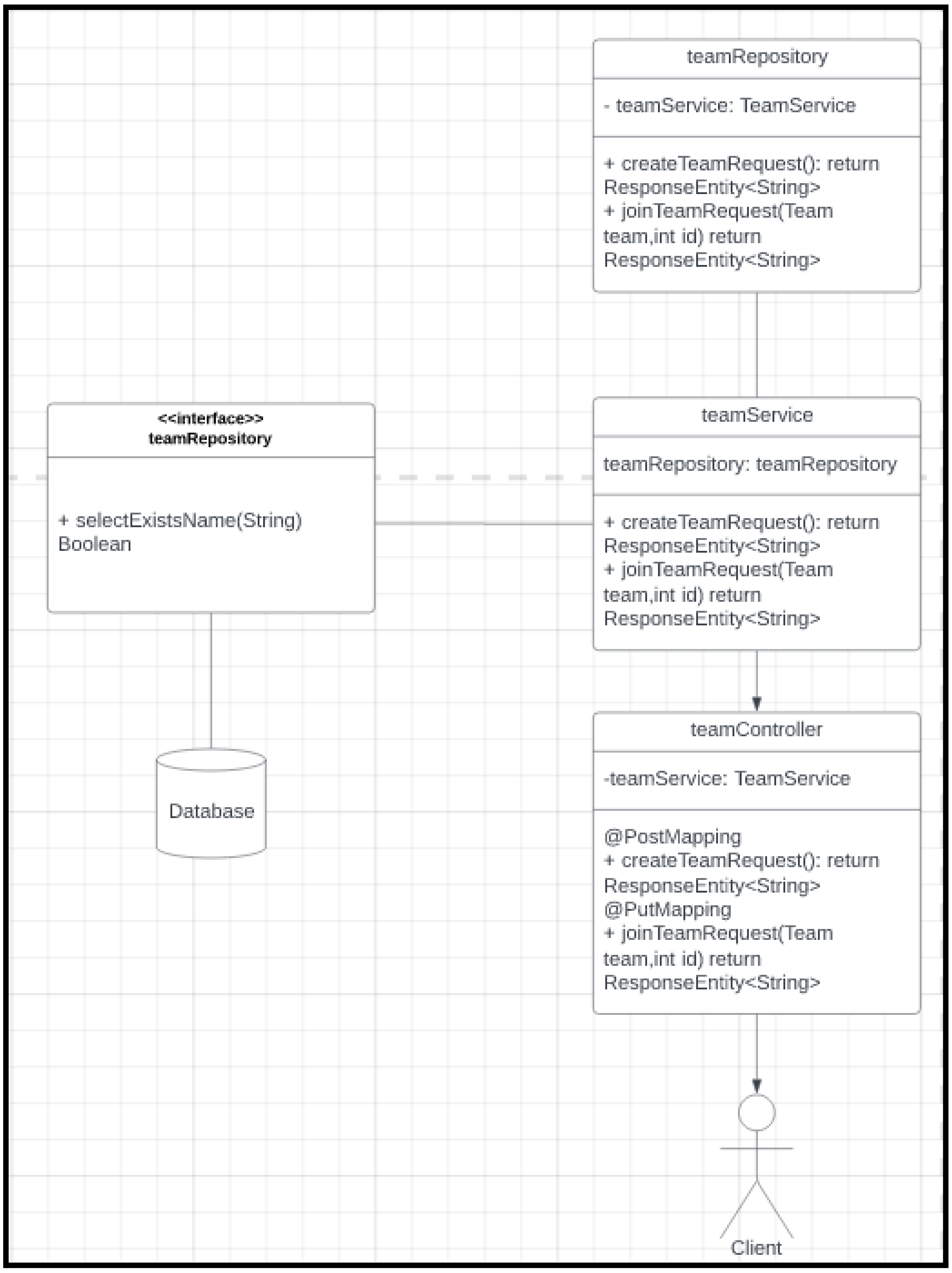
- **Controllers**: Controllers are the layer that we use to assign routes to our operations in the services. In our case, we can use implement controllers to determine which operations in our services are equivalent to which CRUD operations and how these operations will accept inputs.

Backend Design and Database Schema:

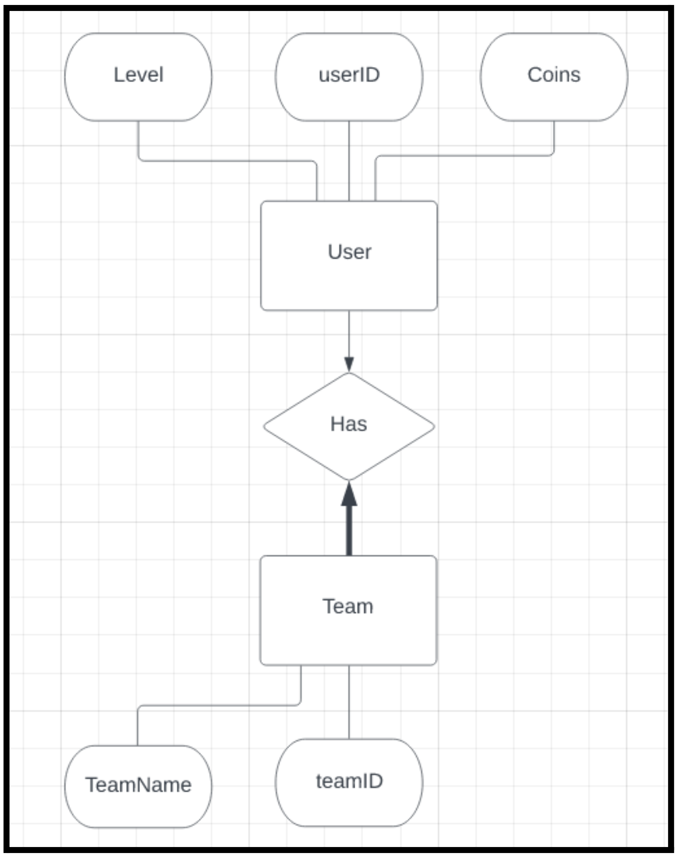
User Design


userRepository
- userService: UserService
+ createUserRequest(): return String
+ updateUserRequest(int id) String
<<interface>>
userRepository
+ selectExistsCoins(Integer) Boolean
userService
userRepository: UserRepository
+ createUserRequest(): return String
+ updateUserRequest(int id) String
Database
userController
-userService: UserService
@PostMapping
+ createUserRequest(): return String
@PutMapping
+ updateUserRequest(int id) String
Client

Team Design


teamRepository
- teamService: TeamService
+ createTeamRequest(): return ResponseEntity<String>
+ joinTeamRequest(Team team,int id) return ResponseEntity<String>
<<interface>>
teamRepository
+ selectExistsName(String) Boolean
teamService
teamRepository: teamRepository
+ createTeamRequest(): return ResponseEntity<String>
+ joinTeamRequest(Team team,int id) return ResponseEntity<String>
teamController
-teamService: TeamService
@PostMapping
+ createTeamRequest(): return ResponseEntity<String>
@PutMapping
+ joinTeamRequest(Team team,int id) return ResponseEntity<String>
Database
Client

Database Schema:



Repositories:

In our implementation, we have two different repositories for each team and user entity. They have unique operations specific to our requirements and needs.

userRepository:

```
package com.example.RowMatch.repository;

import com.example.RowMatch.Models.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;


@Repository
public interface userRepository extends JpaRepository<User, Integer> {

   @Query("" +
           "SELECT CASE WHEN COUNT(s) > 0 THEN " +
           "TRUE ELSE FALSE END " +
           "FROM User s " +
           "WHERE s.coins = ?1"
   )
   Boolean selectExistsCoins(Integer coins);

}
```

- SelectExistsCoins:
  Returns true if a user with a specific number of coins exists or else returns false. This function doesn't have a practical task on the client side, it is used for unit testing the userRepository

teamRepository:

```
package com.example.RowMatch.repository;
import com.example.RowMatch.Models.Team;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;
import java.util.Optional;
@Repository
public interface teamRepository extends JpaRepository<Team, Integer> {
   Optional<Team> findByName(String name);

   @Query("" +
           "SELECT CASE WHEN COUNT(t) > 0 THEN " +
           "TRUE ELSE FALSE END " +
           "FROM Team t " +
           "WHERE t.name = ?1"
   )
   Boolean selectExistsName(String name);
}
```

- SelectExistsName:
  Returns true if a team with a specific name exists or else returns false. This function doesn't have a practical task on the client side, it is used for unit testing the teamRepository
- findByName:
  Returns Optional<Team> if it finds a team with the given name. This function is used in the teamService for finding teams with their names.

Models:

User:

```java
package com.example.RowMatch.Models;

import jakarta.persistence.*;

@Entity
@Table(name = "user_tbl")
public class User {

   @Id
   @GeneratedValue
   private int id;
   @Column(nullable = false)
   private Integer coins;
   @Column(nullable = false)
   private Integer level;

   public User() {
   }

   public User(Integer coins, Integer level) {
       this.coins = coins;
       this.level = level;
   }
   // Getters and Setters


}
```

- int id: gives every user a unique ID
- Integer coins: Keeps the number of coins the user has which can be used for creating a team
- Integer level: Keeps the level of the user which can increase as the user progresses in the game

Team:

```
package com.example.RowMatch.Models;
import jakarta.persistence.*;
import java.util.List;

@Entity
@Table(name = "TEAM_TBL")
public class Team{

   @Id
   @GeneratedValue
   private int id;
   private String name;

   @ElementCollection
   private List<Integer> members;

   public Team() {
   }

   public Team(String name, List<Integer> members) {
       this.name = name;
       this.members = members;
   }

   public Team(int id,String name, List<Integer> members) {
       this.id = id;
       this.name = name;
       this.members = members;
   }

   public String getInfo() {
       StringBuilder retVal = new StringBuilder("Name:" + name + "\n");
       retVal.append("Members\n");

       for (Integer member : members) {
           retVal.append(member);
       }

       return retVal.toString();
   }
   // Getters and setters
```

- int id: Gives every team a unique ID
- String name: Keeps the name of the team which is a distinct property in the database
- List<Integer> members: Keeps the ids of the team members as a list
- String getInfo(): returns information about the team such as team name and the members' ids

Services:

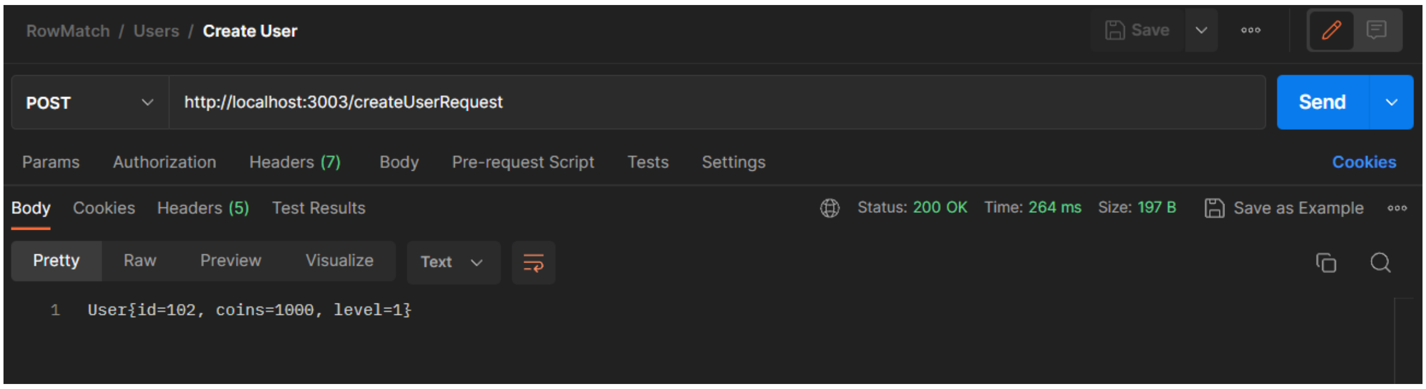
User Services

● User Specific Services

createUserRequest:

```
public String createUserRequest(){
   User userReq = new User(1000,1);
   repository.save(userReq);
   return userReq.toString();
}
```

- String createUserRequest(): creates a new user with the request from the player. initializes a new user which is the userReq with the base coins and level which 1000 coins and level 1. Then userReq is saved to the userRepository. Finally, the user information is returned as a string.



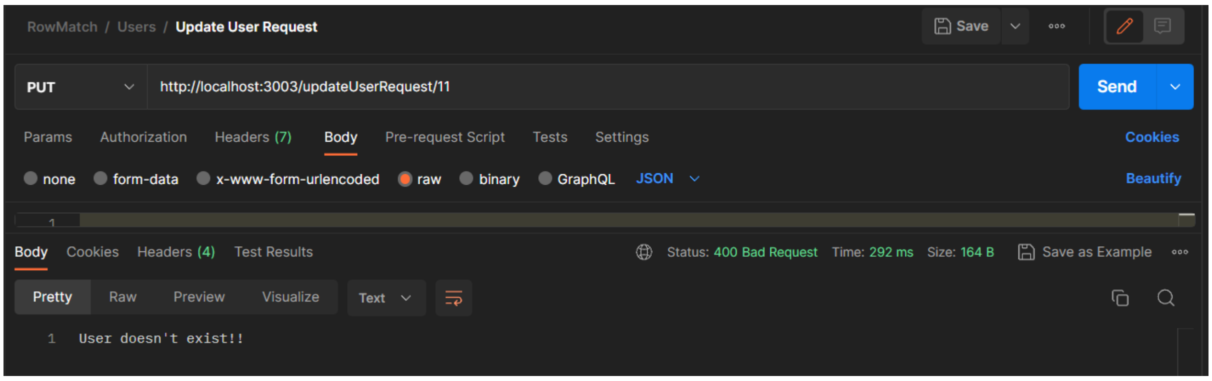
```
User{id=102, coins=1000, level=1}
```

- With the help of Postman software, we can see that when we send the request to our route, A new user will be created with a generated id.

updateLevelRequest:

```
public ResponseEntity<String> updateLevelRequest(int id){
   User existingUser = repository.findById(id).orElse(null);
   if (existingUser == null){
       return ResponseEntity.status(400).body("User doesn't exist!!");
   }
   existingUser.setCoins(existingUser.getCoins() + 10000);
   existingUser.setLevel(existingUser.getLevel() + 1);

   repository.save(existingUser);
   return ResponseEntity.status(200).body(existingUser.toString());
}
```

- String updateLevelRequest(int id): takes int id as input which should be a user id. The function will look for the user with the given Id, if the id is not found in the repository, it will throw a 400 bad request status code with the relevant message

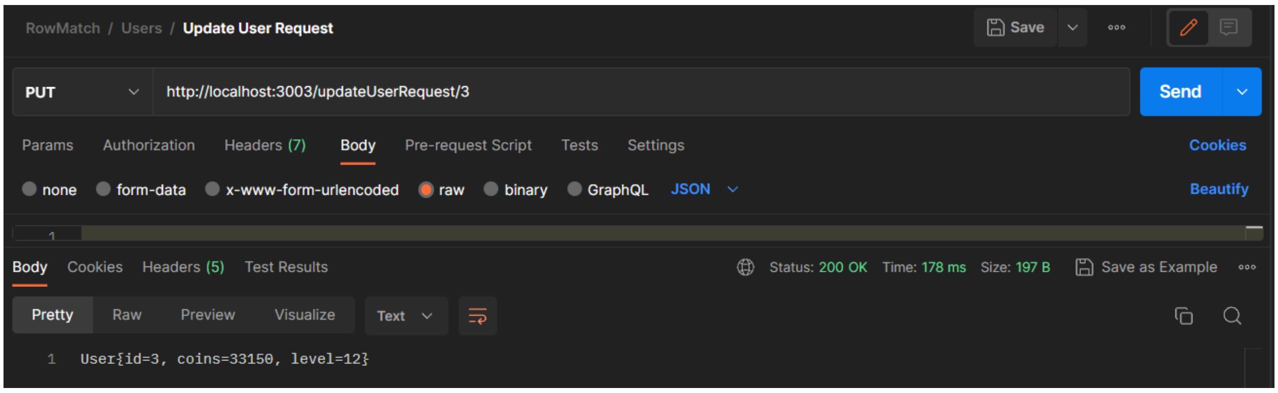

RowMatch / Users / Update User Request
Save
PUT
Send
Params Authorization Headers (7) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
Body Cookies Headers (4) Test Results
Save as Example
Pretty Raw Preview Visualize Text

http://localhost:3003/updateUserRequest/11

Status: 400 Bad Request Time: 292 ms Size: 164 B

```
User doesn't exist!!
```

- We can see that the given user id in the URL does not exist in the userRepository which gives us the error message.
- If the userId exists, the function will increase the coins and level the user has and save the user to the userRepository


RowMatch / Users / Update User Request
Save
PUT
Send
Params Authorization Headers (7) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL JSON
Beautify
Body Cookies Headers (5) Test Results
Save as Example
Pretty Raw Preview Visualize Text

http://localhost:3003/updateUserRequest/3

Status: 200 OK Time: 178 ms Size: 197 B

```
User{id=3, coins=33150, level=12}
```

- We can see that user 3 does exist and their coins and levels were increased.

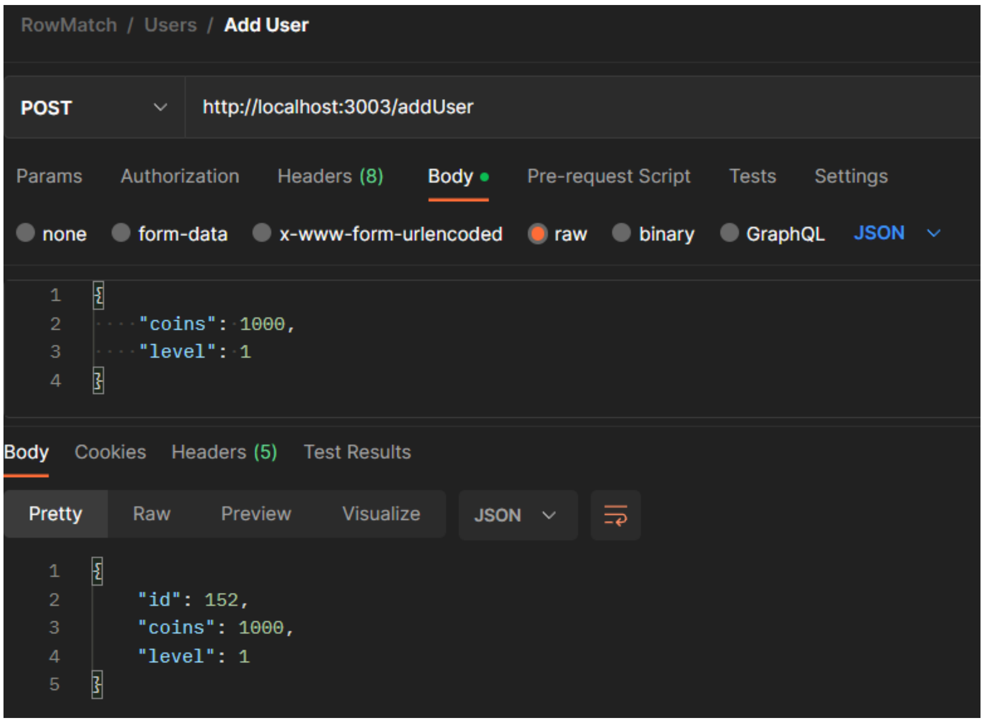
• Admin Services

These service functions are meant for developers which can use see or edit information

updateLevelRequest:

```
public User saveUser(User user){
   return  repository.save(user);
}
```

- User saveUser(User user): The function takes a User object as the input. It is used for directly saving users into userRepository from the admin side.



RowMatch / Users / **Add User**

POST http://localhost:3003/addUser

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```
{
    "coins": 1000,
    "level": 1
}
```

Body Cookies Headers (5) Test Results

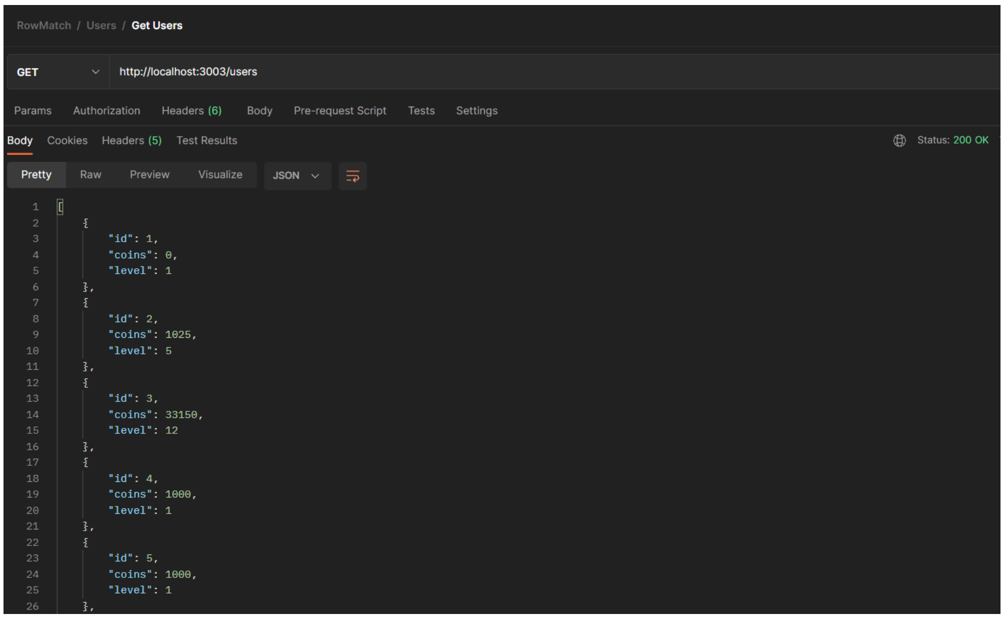
Pretty Raw Preview Visualize JSON

```
{
    "id": 152,
    "coins": 1000,
    "level": 1
}
```

getUsers:

```
public List<User> getUsers(){
   return repository.findAll();
}
```

- List<User> getUsers(): It is used for directly getting all users from the userRepository which the admins can use to see the data.



RowMatch / Users / **Get Users**

GET http://localhost:3003/users

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Body Cookies Headers (5) Test Results Status: 200 OK

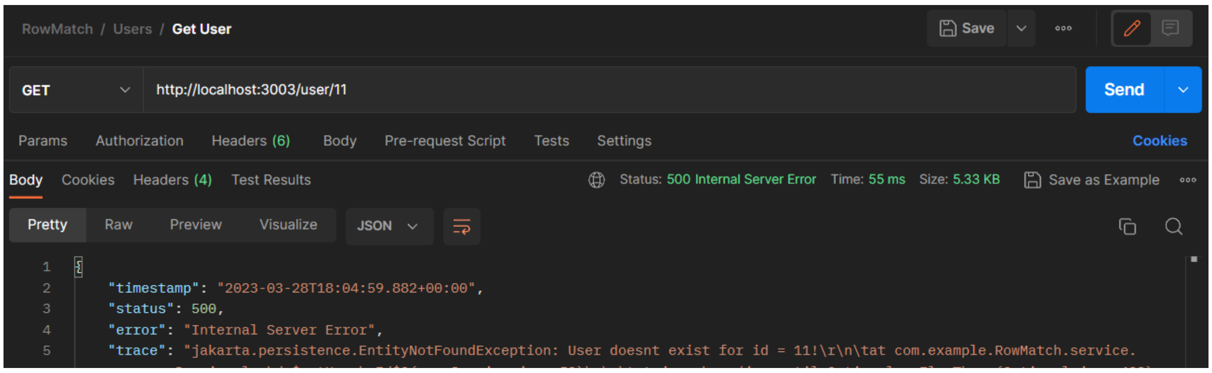
Pretty Raw Preview Visualize JSON

```
[
    {
        "id": 1,
        "coins": 0,
        "level": 1
    },
    {
        "id": 2,
        "coins": 1025,
        "level": 5
    },
    {
        "id": 3,
        "coins": 33150,
        "level": 12
    },
    {
        "id": 4,
        "coins": 1000,
        "level": 1
    },
    {
        "id": 5,
        "coins": 1000,
        "level": 1
    },
```

getUsersbyId:

```java
public User getUserbyId(int id){
   return repository.findById(id).orElseThrow(() -> new
EntityNotFoundException(String.format("User doesnt exist for id = %s!", id)));
}
```

- User getUsersbyId(int id): The function takes an integer id as input which is used for finding a specific user for the admin to see. If the user is not found. The function will throw a EntityNotFoundException with the relevant message

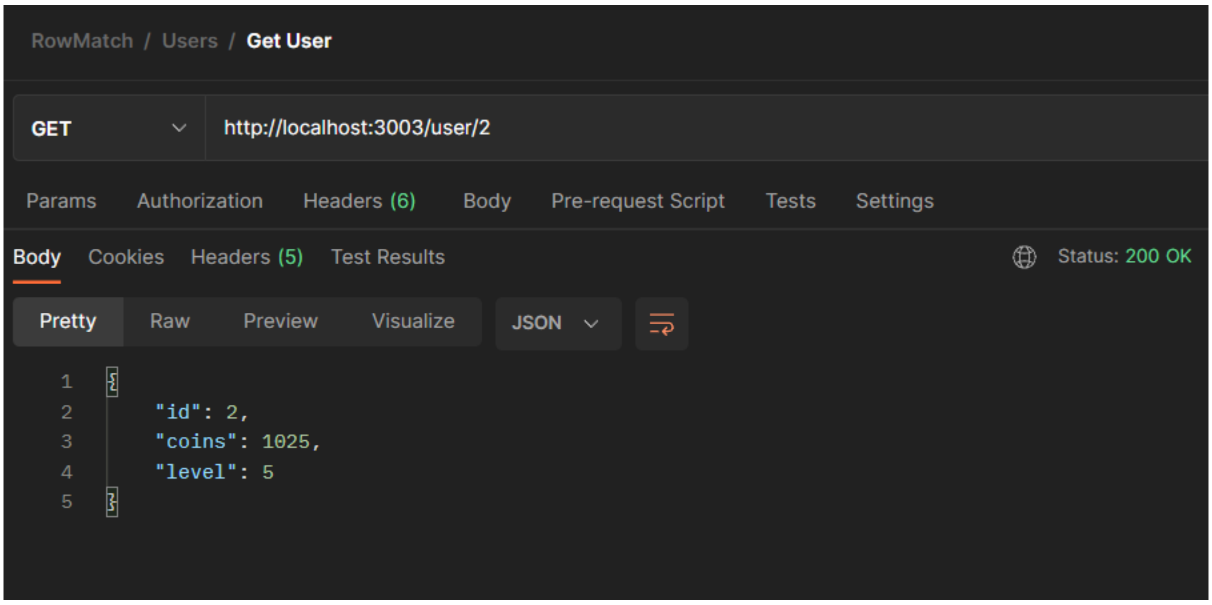


GET http://localhost:3003/user/11

Status: 500 Internal Server Error Time: 55 ms Size: 5.33 KB

```
{
    "timestamp": "2023-03-28T18:04:59.882+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "trace": "jakarta.persistence.EntityNotFoundException: User doesnt exist for id = 11!\r\n\tat com.example.RowMatch.service.
```

- If the user is found, It will return the information about the user.



GET http://localhost:3003/user/2

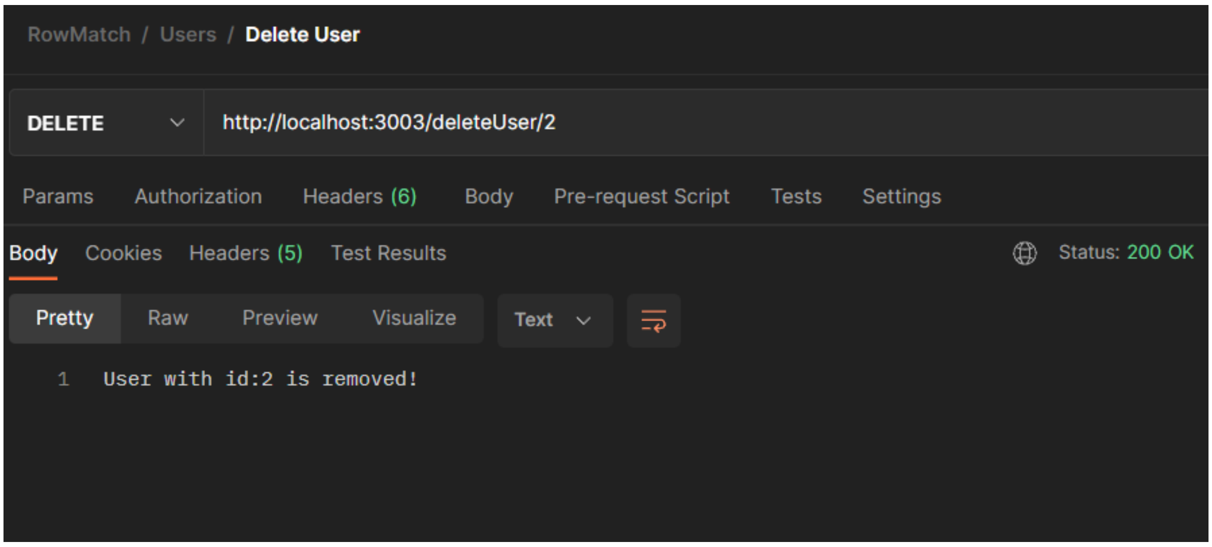
Status: 200 OK

```
{
    "id": 2,
    "coins": 1025,
    "level": 5
}
```

deleteUser:

```java
public String deleteUser(int id){
   repository.deleteById(id);
   return "User with id:" + id + " is removed!";
}
```

- String deleteUser(int id): This function deletes the user of the given id and returns the relevant message



http://localhost:3003/deleteUser/2

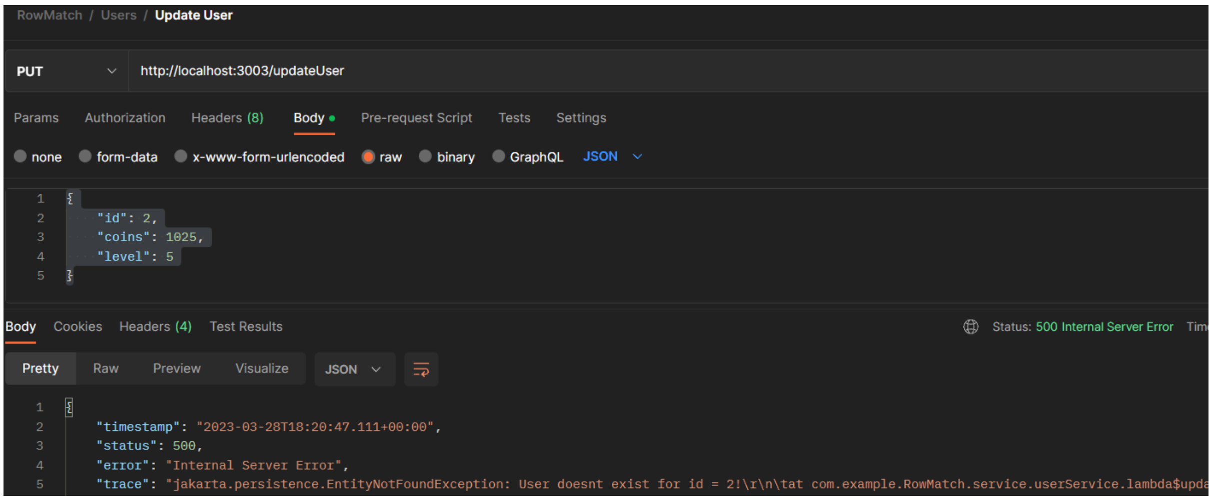
Status: 200 OK

```
User with id:2 is removed!
```

updateUser:

```java
public User updateUser(User user){
   User existingUser = repository.findById(user.getId()).orElseThrow(() -> new
EntityNotFoundException(String.format("User doesnt exist for id = %s!",
user.getId())));
   existingUser.setCoins(user.getCoins());
   existingUser.setLevel(user.getLevel());

   return repository.save(existingUser);
}
```

- String updateUser(User user): the function is responsible for updating an existing user's coins and level attributes in a database using the UserRepository object. It first finds the existing user with the same id as the input user, throws an exception if no user is found, updates the user's coins and level attributes, saves the changes to the database, and returns the updated User object.



RowMatch / Users / Update User

PUT http://localhost:3003/updateUser

Params Authorization Headers (8) Body Pre-request Script Tests Settings

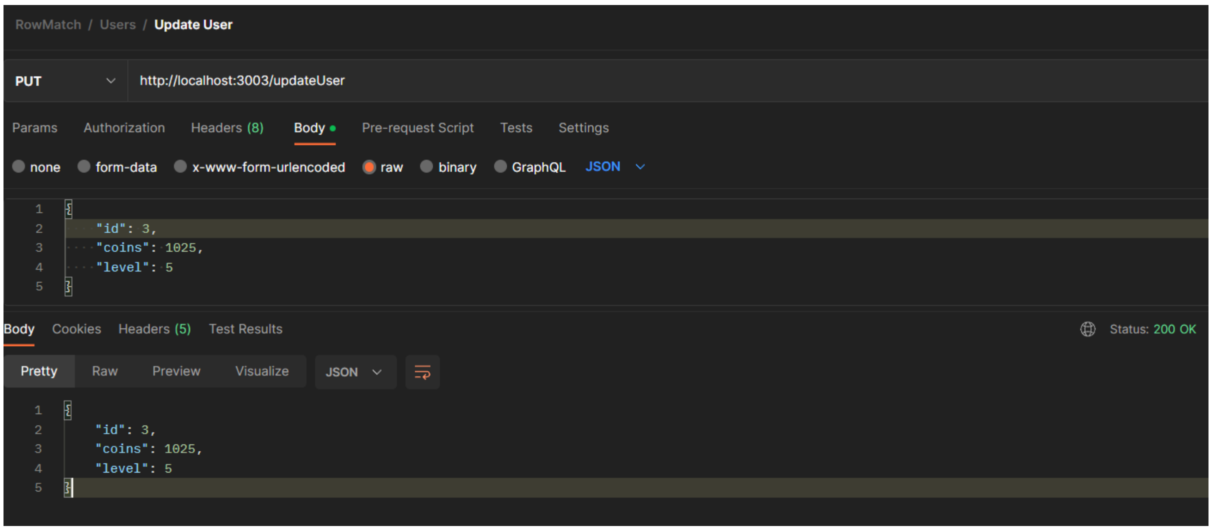
none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
{
    "id": 2,
    "coins": 1025,
    "level": 5
}
```

Body Cookies Headers (4) Test Results Status: 500 Internal Server Error

Pretty Raw Preview Visualize JSON

```
{
    "timestamp": "2023-03-28T18:20:47.111+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "trace": "jakarta.persistence.EntityNotFoundException: User doesnt exist for id = 2!\r\n\tat com.example.RowMatch.service.userService.lambda$upda
```



RowMatch / Users / Update User

PUT http://localhost:3003/updateUser

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
{
    "id": 3,
    "coins": 1025,
    "level": 5
}
```

Body Cookies Headers (5) Test Results Status: 200 OK

Pretty Raw Preview Visualize JSON

```
{
    "id": 3,
    "coins": 1025,
    "level": 5
}
```

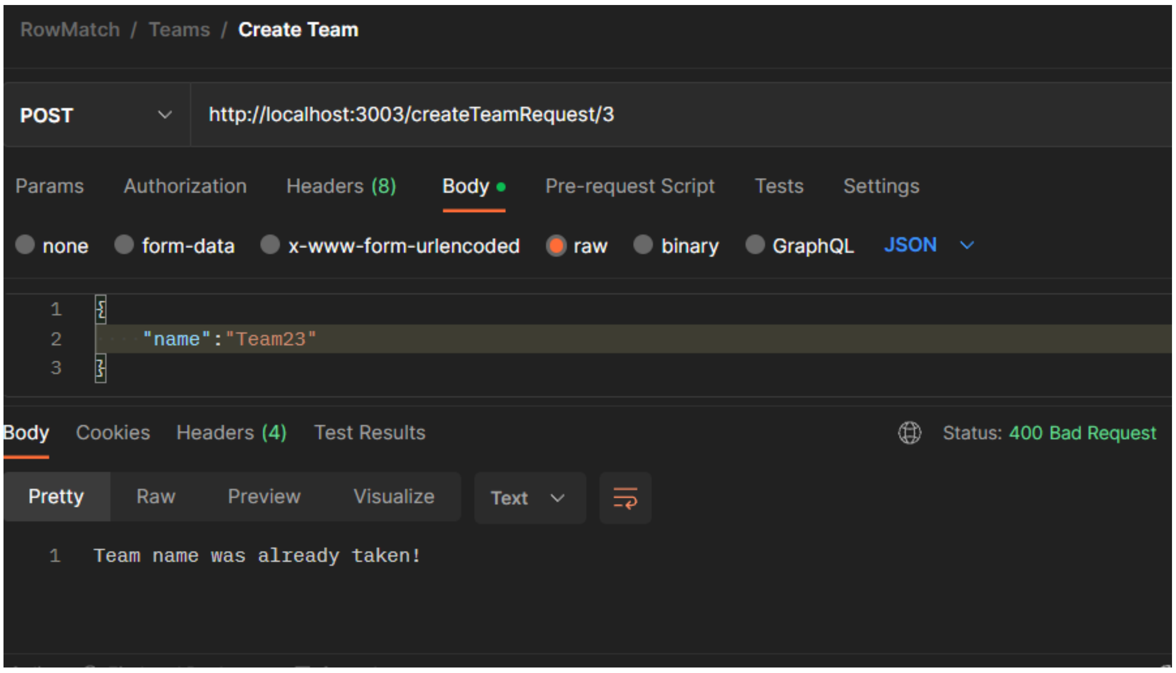
Team Services

- **User Specific Services**

createTeamRequest:

```java
public ResponseEntity<String> createTeamRequest(Team team, int userId) {
   if (repository.findByName(team.getName()).orElse(null) != null) {
       return ResponseEntity.status(400).body("Team name was already taken!");
   }

   User existingUser = userRepository.findById(userId).orElse(null);
   if (existingUser.getCoins() < 1000) {
       return ResponseEntity.status(200).body("You don't have enough coins!!");
   }
   existingUser.setCoins(existingUser.getCoins() - 1000);

   List<Integer> newTeamList = new
java.util.ArrayList<>(Collections.emptyList());
   newTeamList.add(userId);
   Team createdTeam = new Team(team.getName(), newTeamList);

   userRepository.save(existingUser);
   repository.save(createdTeam);
   return ResponseEntity.status(200).body("Team " + team.getName() + " was
created successfully!");
}
```

- createTeamRequest(Team team, int userId): This function is responsible for creating a new team with the specified name, adding the user with the given userId as a member of the team, and updating the user's coins attribute to reflect the cost of creating a team.
- It checks whether a team with the same name already exists and whether the user has enough coins to create a team.
- If either of these conditions is not met, an appropriate error message is returned in the response body.
- If the team is successfully created, a success message is returned in the response body.

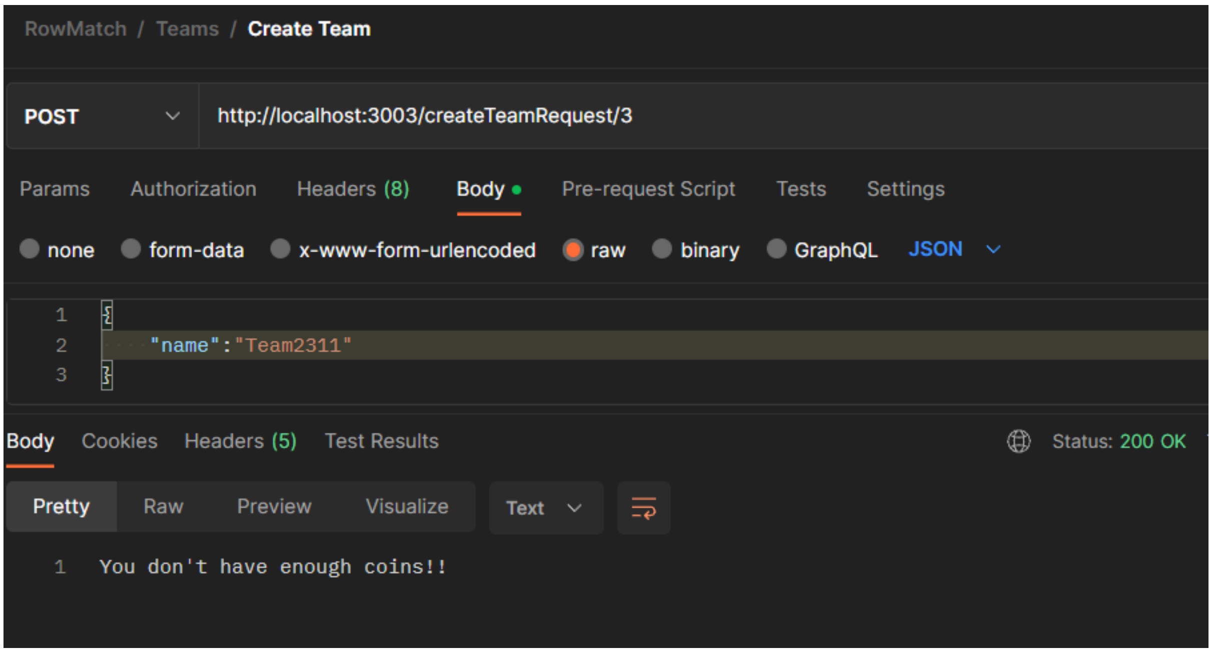


POST http://localhost:3003/createTeamRequest/3

```
{
    "name":"Team23"
}
```

Status: 400 Bad Request

```
Team name was already taken!
```


RowMatch / Teams / Create Team

POST http://localhost:3003/createTeamRequest/3

Params Authorization Headers (8) Body Pre-request Script Tests Settings

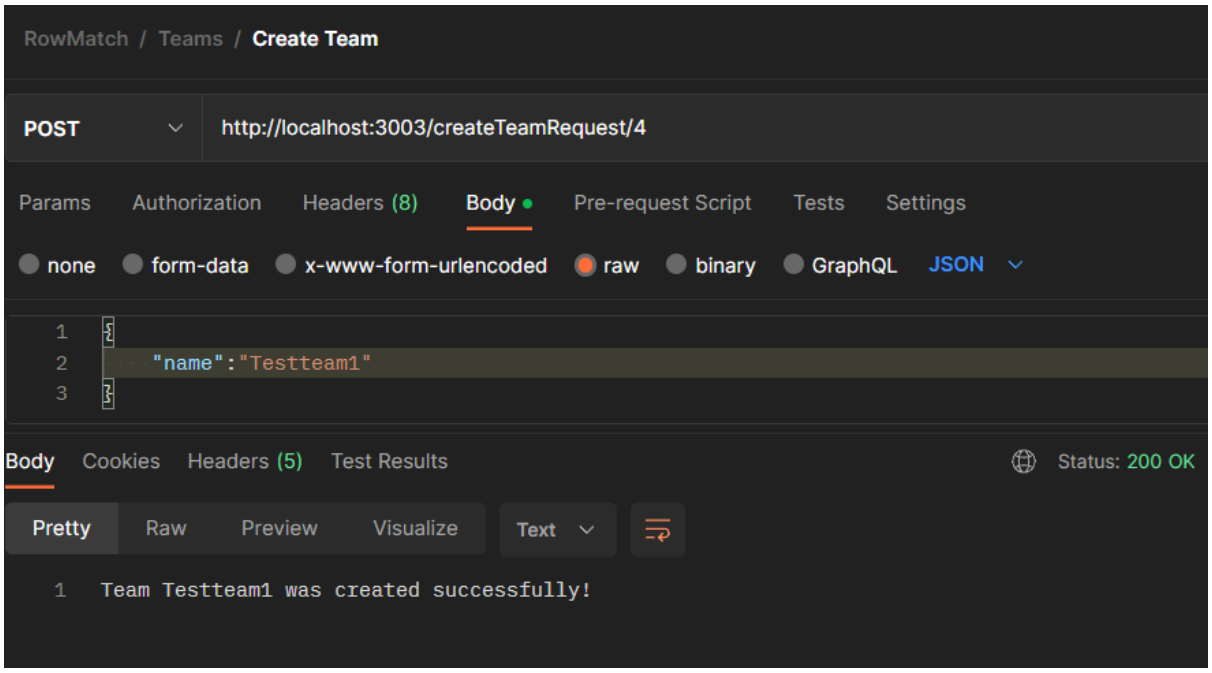
none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```
{
    "name":"Team2311"
}
```

Body Cookies Headers (5) Test Results Status: 200 OK

Pretty Raw Preview Visualize Text

```
You don't have enough coins!!
```


RowMatch / Teams / Create Team

POST http://localhost:3003/createTeamRequest/4

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```
{
    "name":"Testteam1"
}
```

Body Cookies Headers (5) Test Results Status: 200 OK

Pretty Raw Preview Visualize Text

```
Team Testteam1 was created successfully!
```

joinTeamRequest:

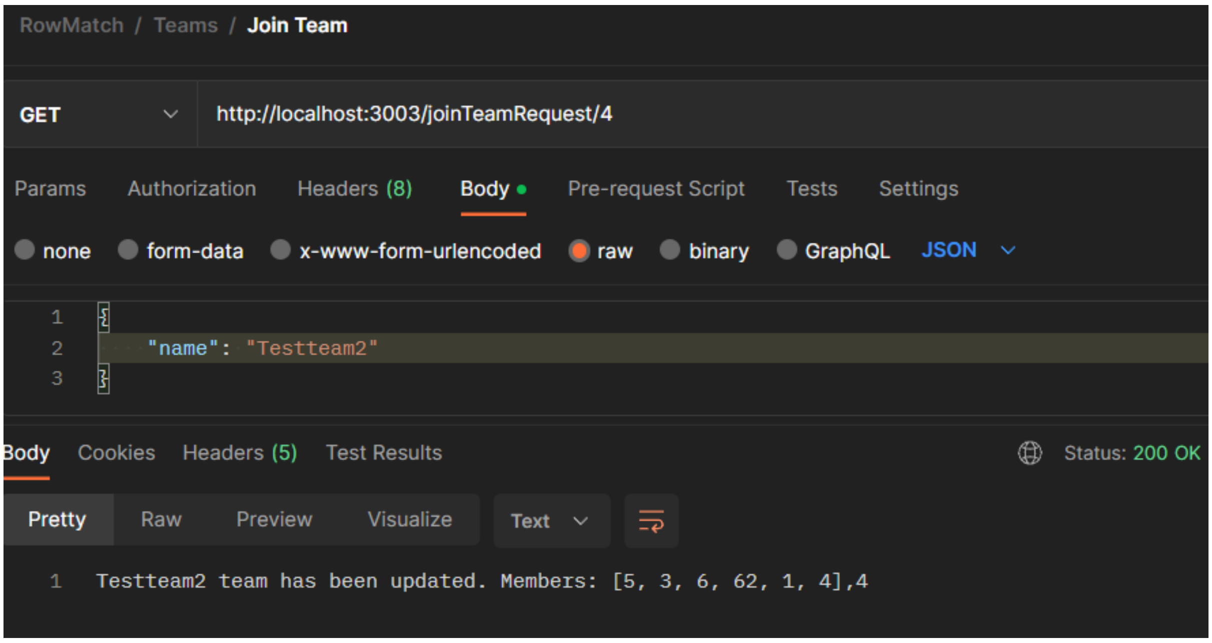
```java
public ResponseEntity<String> joinTeamRequest(Team team, int userId) {
   Team existingTeam = repository.findByName(team.getName()).orElse(null);
   if (existingTeam == null) {
       return ResponseEntity.status(400).body("Team doesn't exist!!");
   }

   List<Integer> updatedTeam = null;
   if(existingTeam.getMembers() == null){
       updatedTeam = new ArrayList<>();
   }
```

```
    else {
        updatedTeam = existingTeam.getMembers
    }
    if (updatedTeam.contains(Integer.valueOf(userId))) {
        return ResponseEntity.status(400).body("You are already in this team");
    } else if (updatedTeam.size() >= 10) {
        return ResponseEntity.status(200).body("This team is full!");
    } else {
        updatedTeam.add(userId);
        existingTeam.setMembers(updatedTeam);

        repository.save(existingTeam);
        return ResponseEntity.status(200).body(team.getName() + " team has been
updated. Members: " + team.getMembers().toString() + "," + userId);
    }
}
```

- joinTeamRequest(Team team, int userId): Overall, This function is responsible for adding a user with the given userId to the specified team if the team exists and has room for more members.
- It checks whether the team exists and whether the user is already a member of the team or whether the team has room for more members. If either of these conditions is not met, an appropriate error message is returned in the response body.
- If the team is successfully updated with the new member, a success message is returned in the response body along with the updated list of team members.

```
List<Integer> updatedTeam =
existingTeam.getMembers();

if
(updatedTeam.contains(Integer.valueOf(userId)))
{
    return ResponseEntity.status(400).body("You
are already in this team");
} else if (updatedTeam.size() >= 10) {
    return
ResponseEntity.status(200).body("This team is
full!");
} else {
    updatedTeam.add(userId);
    existingTeam.setMembers(updatedTeam);
```

- These lines retrieve the member list of the existingTeam object and check whether the user with the given userId is already a member of the team or whether the team has room for more members.
- If the user is already a member of the team, a ResponseEntity with a 400 status code and a response body containing an error message is returned.
- If the team is already full, a ResponseEntity with a 200 status code and a response body containing an error message is returned.
- If the team has room for more members, the user's userId is added to the member list of the team.

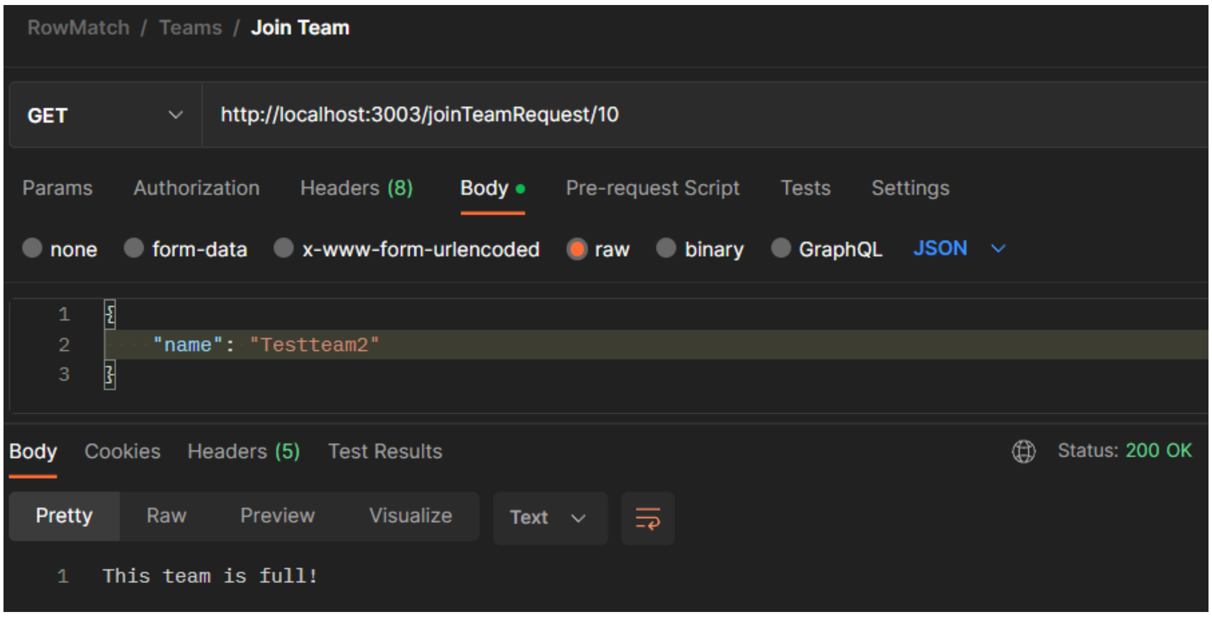

RowMatch / Teams / Join Team

GET http://localhost:3003/joinTeamRequest/4

```
{
    "name": "Testteam2"
}
```

Status: 200 OK

```
Testteam2 team has been updated. Members: [5, 3, 6, 62, 1, 4],4
```

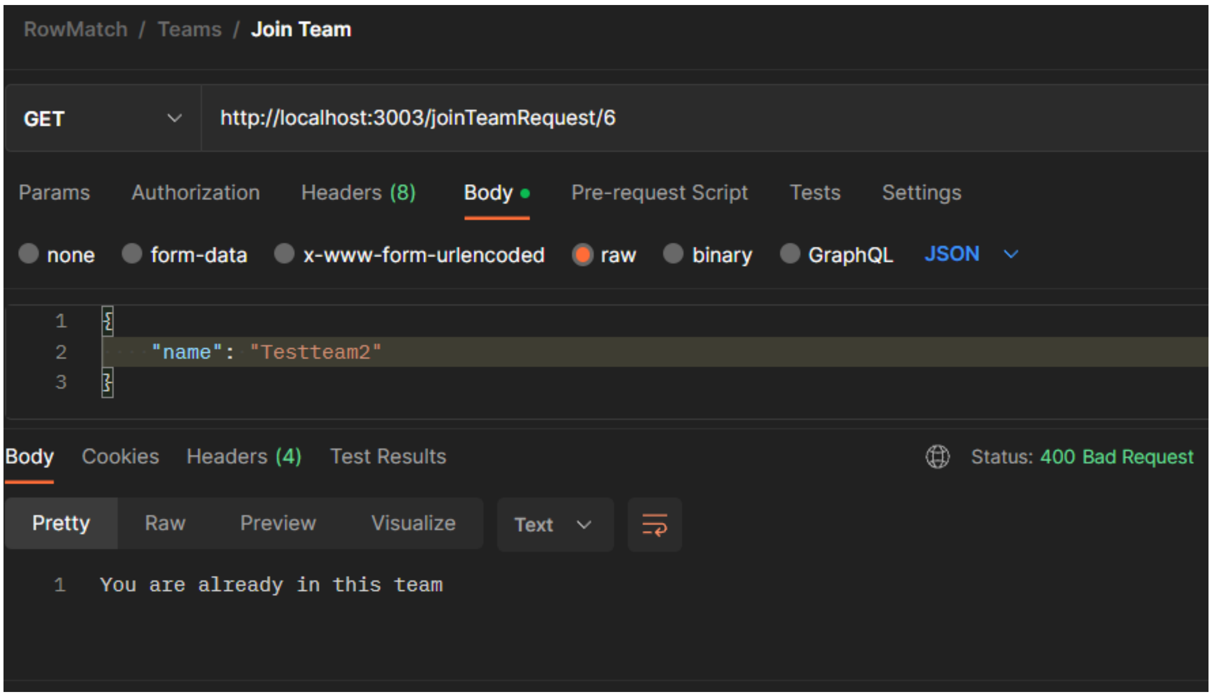


GET http://localhost:3003/joinTeamRequest/10

```
{
    "name": "Testteam2"
}
```

Status: 200 OK

```
This team is full!
```

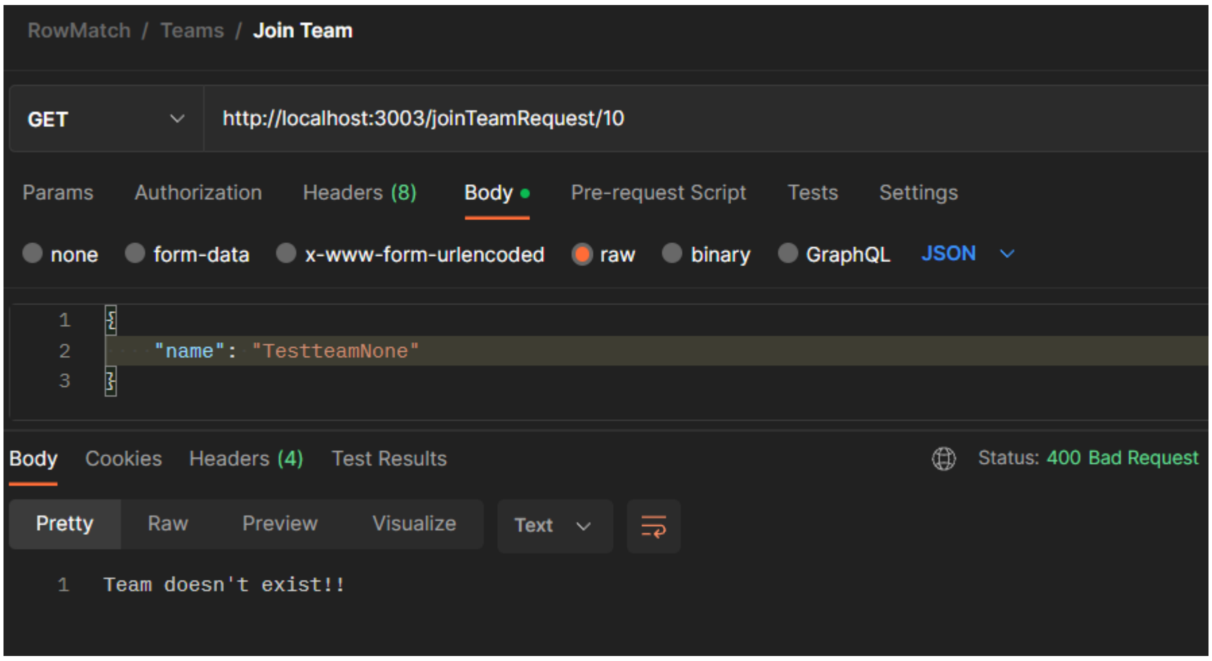


GET http://localhost:3003/joinTeamRequest/6

```
{
    "name": "Testteam2"
}
```

Status: 400 Bad Request

```
You are already in this team
```



http://localhost:3003/joinTeamRequest/10

```
{
    "name": "TestteamNone"
}
```

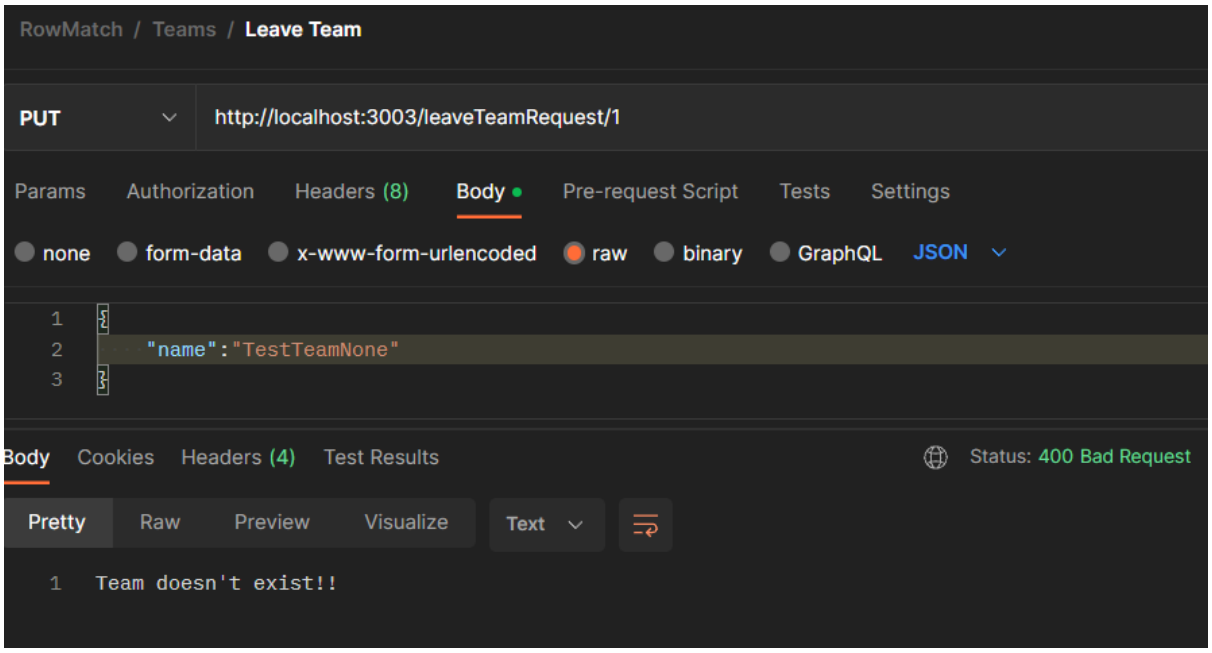
Status: 400 Bad Request

```
Team doesn't exist!!
```

leaveTeamRequest:

```
public ResponseEntity<String> leaveTeamRequest(Team team, int userId) {
   Team existingTeam = repository.findByName(team.getName()).orElse(null);
   if (existingTeam == null) {
       return ResponseEntity.status(400).body("Team doesn't exist!!");
   }
   List<Integer> updatedTeam = existingTeam.getMembers();

   updatedTeam.remove(Integer.valueOf(userId));
   existingTeam.setMembers(updatedTeam);

   repository.save(existingTeam);
   return ResponseEntity.status(200).body("You have successfully left the
team");
}
```

- leaveTeamRequest(Team team, int userId): This function is used for leaving the current team the user is in. Firstly, It will check if the team exists by using the given team name. If the team doesn't exist it will throw Bad Request status and show the appropriate message. If everything checks out, It will remove the user from the team and save the team back to the repository. After that, it will give 200 status with the relevant message.



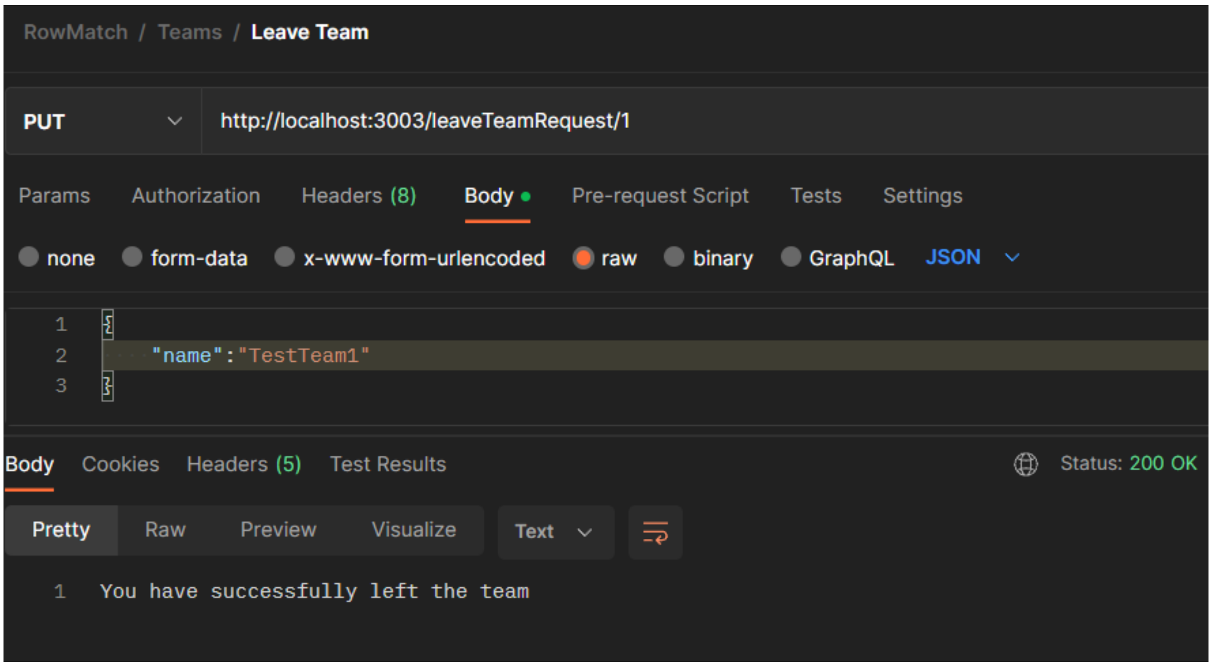
PUT http://localhost:3003/leaveTeamRequest/1

Body: raw JSON

```
{
    "name":"TestTeamNone"
}
```

Status: 400 Bad Request

```
Team doesn't exist!!
```



PUT http://localhost:3003/leaveTeamRequest/1

Body: raw JSON

```
{
    "name":"TestTeam1"
}
```

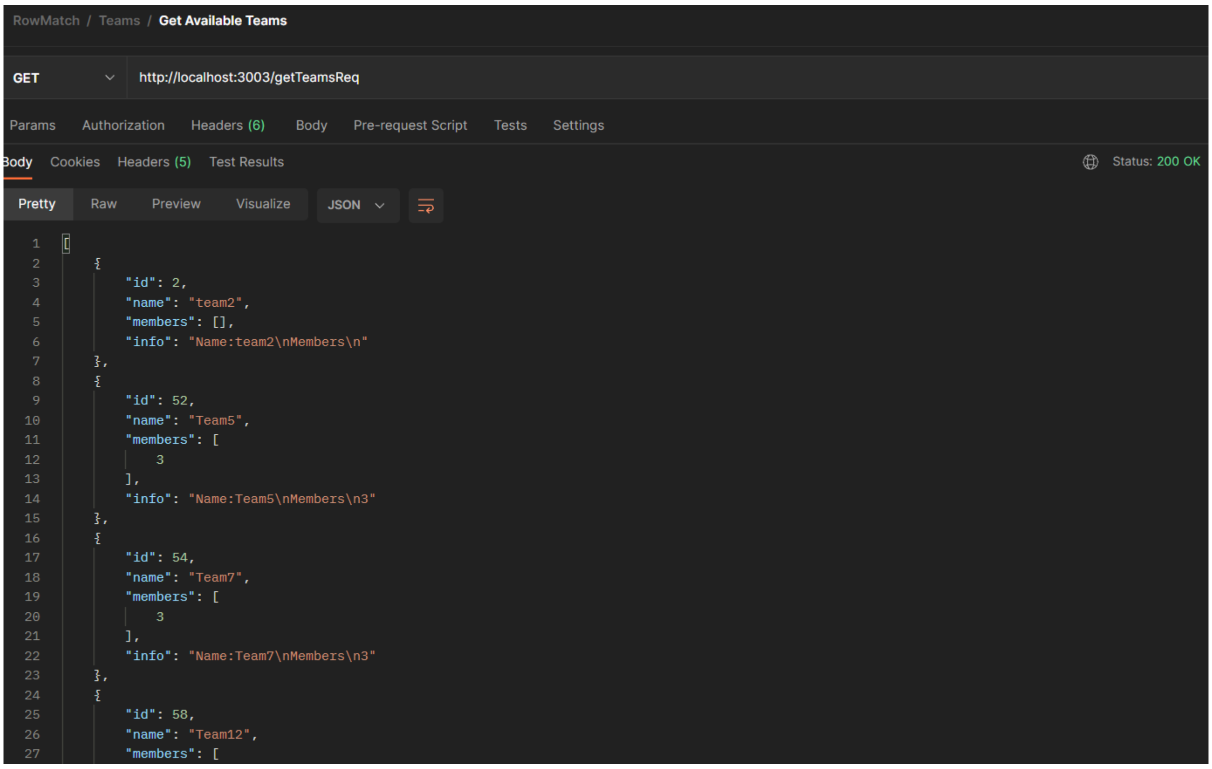
Status: 200 OK

```
You have successfully left the team
```

getTeamReq:

```
public ResponseEntity<List<Team>> getTeamsReq() {
    List<Team> availableTeams = repository.findAll();
    Random rand = new Random();
    availableTeams.removeIf(availableTeam -> availableTeam.getMembers().size() >=
10);
    while (availableTeams.size() > 10) {
        int randomIndex = rand.nextInt(availableTeams.size());
        availableTeams.remove(randomIndex);
    }
    if (availableTeams.size() > 0) {
        return ResponseEntity.status(200).body(availableTeams);
    }
    return ResponseEntity.status(200).body(null);
}
```

- getTeamsReq(): This function will give the user currently available 10 teams randomly. Firstly, it will take every team and sort the available ones with .removeIf(), then randomly choose 10 team between the available ones to show the user. If there are no available teams the function will return null.



GET http://localhost:3003/getTeamsReq

```
[
    {
        "id": 2,
        "name": "team2",
        "members": [],
        "info": "Name:team2\nMembers\n"
    },
    {
        "id": 52,
        "name": "Team5",
        "members": [
            3
        ],
        "info": "Name:Team5\nMembers\n3"
    },
    {
        "id": 54,
        "name": "Team7",
        "members": [
            3
        ],
        "info": "Name:Team7\nMembers\n3"
    },
    {
        "id": 58,
        "name": "Team12",
        "members": [
```

- **Admin Services**

These service functions are meant for developers which can use, see or edit information directly.

saveTeam:

```
public Team saveTeam(Team team) {
   boolean existsTeam = repository
           .selectExistsName(team.getName());
   if (existsTeam) {
       throw new RuntimeException(
               "Team name " + team.getName() + " taken"
       );
   }
   return repository.save(team);
}
```

- String saveTeam(): This function saves a team to the teamRepository with the information given by the admin. If the team name already exists it will throw a RuntimException and give the relevant message
-

getTeams:

```
public List<Team> getTeams() {
   return repository.findAll();
}
```

- String getTeam(): This function will return all the teams with their information from the teamRepository.
-

getTeamById:

```
public Team getTeamById(int id) {
   return repository.findById(id).orElseThrow(() -> new
EntityNotFoundException(String.format("Team does not exist for id = %s!", id)));
}
```

- String getTeamById(): This function will return the team of the given ID. If the given id cant be found in the teamRepository, it will throw a EntityNotFoundException with the appropriate message.

deleteTeam:

```
public String deleteTeam(int id) {
   repository.deleteById(id);
   return "Team with id: " + id + " is removed!";
}
```

- String delete(): This function will delete the team of the given ID.

updateTeam:

```
public Team updateTeam(Team team) {
   Team existingTeam = repository.findById(team.getId()).orElseThrow(() -> new
EntityNotFoundException(String.format("Team doesnt exist for id = %s!",
team.getId())));
   existingTeam.setName(team.getName());
   existingTeam.setMembers(team.getMembers());

   return repository.save(existingTeam);
}
```

- String delete(): This function will update the given team with using the information from the request body. If the function cant find the team, It will throw an EntityNotFoundException with the appropriate message.

Controllers:

User Controllers

- **User Specific Controls**

updateTeam:

```
@PostMapping("/createUserRequest")
public String createUserRequest(){
   return service.createUserRequest();
}
```

- String createUserRequest(): This function maps the createUserRequest service to the given post URL then returns the services' result to the user.

updateTeam:

```
@PutMapping ("/updateUserRequest/{id}")
public String updateUserRequest(@PathVariable int id){
   return service.updateLevelRequest(id);
}
```

- String updateUserRequest(int id): This function maps the updateUserRequest service to the given put URL then returns the services' result to the user. It will use the id given by the user to go to the correct route and update relevant user.

- **Admin Controls**

addUser:

```
@PostMapping("/addUser")
public User addUser(@RequestBody User user){
   return service.saveUser(user);
}
```

- addUser(User user): This function maps the saveUser service to the given post URL then returns the services' result to the user.

findAllUsers:

```
@GetMapping("/users")
public List<User> findAllUsers(){
   return service.getUsers();
}
```

- findAllUsers(): This function maps the findAllUser service to the given get URL then returns the services' result to the user.

findUser:

```
@GetMapping("/user/{id}")
public User findUser(@PathVariable int id){
   return service.getUserbyId(id);
}
```

- findUser(User user): This function maps the findUser service to the given post URL then returns the services' result to the user. It will use the id given by the user to go to the correct route and find the appropriate user.

updateUser:

```
@PutMapping("/updateUser")
public User updateUser(@RequestBody User user){
   return service.updateUser(user);
}
```

- updateUser(User user): This function maps the updateUser service to the given put URL then returns the services' result to the user. It will use the id given by the user to go to the correct route and update the appropriate user.

deleteUser:

```
@DeleteMapping("/deleteUser/{id}")
public String deleteUser(@PathVariable int id){
   return service.deleteUser(id);
}
```

- deleteUser(User user): This function maps the deleteUser service to the given delete URL then returns the services' result to the user. It will use the id given by the user to go to the correct route and delete the appropriate user.

Team Controllers

- ## User Specific Controls

createTeamRequest:

```
@PostMapping("/createTeamRequest/{userId}")
public ResponseEntity<String> createTeamRequest(@PathVariable int
userId,@RequestBody Team team) {
   return service.createTeamRequest(team,userId);
}
```

- createTeamRequest(int userId, Team team): This function maps the createTeamRequest service to the given post URL then returns the services' result to the user.It will use the team information given by the user to go to the correct route and create a team.

JoinTeamRequest:

```
@GetMapping("/joinTeamRequest/{userId}")
public ResponseEntity<String> joinTeamRequest(@RequestBody Team team,
@PathVariable int userId){
   return service.joinTeamRequest(team,userId);
}
```

- JoinTeamRequest(Team team, int userId): This function maps the JoinTeamRequest service to the given get URL then returns the services' result to the user. It will use the team information given by the user to go to the correct route and update relevant team.

leaveTeamRequest:

```
@PutMapping("/leaveTeamRequest/{userId}")
```

```
public ResponseEntity<String> leaveTeamRequest(@RequestBody Team team,
@PathVariable int userId){
    return service.leaveTeamRequest(team,userId);
}
```

- leaveTeamRequest(User user): This function maps the leaveTeamRequestservice to the given put URL then returns the services' result to the user. It will use the team information given by the user to go to the correct route and leave relevant team.

getTeamsReq:

```
@GetMapping("/getTeamsReq")
public ResponseEntity<List<Team>> getTeamsReq(){
    return service.getTeamsReq();
}
```

- getTeamsReq(): This function maps the leaveTeamRequestservice to the given get URL then returns the services' result to the user.

- ## Admin Controls

addTeam:

```
@PostMapping("/addTeam")
public Team addTeam(@RequestBody Team team){
    return service.saveTeam(team);
}
```

- addTeam(Team team): This function maps the saveTeam service to the given post URL then returns the services' result to the user. It will use the team information given by the user to go to the correct route and save the team into the teamRepository.

findAllTeams:

```
@GetMapping("/teams")
public List<Team> findAllTeams(){
    return service.getTeams();
}
```

- findAllTeams(User user): This function maps the get Teams service to the given get URL then returns the services' result to the user.

findTeam:

```
@GetMapping("/team/{id}")
public Team findTeam(@PathVariable int id){
   return service.getTeamById(id);
}
```

- findTeam(int id): This function maps the getTeambyId service to the given get URL then returns the services’ result to the user. It will use the id given by the user to go to the correct route and get the relevan team.

updateTeam:

```
@PutMapping("/updateTeam")
public Team updateTeam(@RequestBody Team team){
   return service.updateTeam(team);
}
```

- updateTeam(Team team): This function maps the updateTeam service to the given put URL then returns the services’ result to the user. It will use the team information given by the user to go to the correct route and update the relevant team.

deleteTeam:

```
@DeleteMapping("/deleteTeam/{id}")
public String deleteTeam(@PathVariable int id){
   return service.deleteTeam(id);
}
```

- deleteTeam(int id): This function maps the deleteTeam service to the given delete URL then returns the services’ result to the user. It will use the team information given by the user to go to the correct route and delete the relevant team.

Chapter Three

CURRENT BUGS AND PROBLEMS

Currently there is only seems to be a minor bug in the joinTeamRequest function wich only occurs when entering bad request bodies.