RESTful Web Services Project - Report

Miguel Valadares - nº 283781

Miguel Cordeiro - nº 283767

This report consists of a description of our design choices, our difficulties encountered while working on the project and a user manual to

Design Choices

According to the project requirements, we understood that it was necessary to implement the following functionalities:

On the Rental Side:

Based on the provided guidelines, the application needed to handle the following scenarios:

- Bikes are characterized by several attributes, including the owner who offers the bike for rental.
- A user can browse and view the bikes available for rental. However, users are restricted from renting a bike if it is not available.
- After using a bike, a user can leave feedback, including a comment and a rating about the bike's condition.
- If a user is unable to rent a specific bike, they can request to be added to a waiting list. The user is notified when the bike becomes available for rental.

On the Sales Side:

- The sale of a bike is allowed only if it has been rented at least once and is currently available.
- Users can view the price and availability of bikes, add them to a cart, and proceed with purchases.
- It is possible to make purchases in multiple currencies.
- Users can manage the contents of their cart, including viewing the total price.

Model Implementation

To address the requirements, we decided to structure the project as follows:

- **Model:** Contains the business logic (e.g., Bike, User, Rating, Basket, etc.).

- **Controller:** Manages the input and output of the API services.
- **Services:** Responsible for the instantiation and control of the Model's logic, functioning as a middle layer between the Model and Controller.

Implementation and Challenges Encountered

RESTful API Functions to Implement:

1. User

- User Login and Registration: Ensuring secure authentication and new user sign-up.

2. BikeRental

- **GET method:** To retrieve the status and details of available bikes.
- **POST method for Bike Rental:** Allowing users to rent bikes.
- **POST method for Bike Return:** Facilitating the return of rented bikes.
- **PUT method:** To create and add new bikes to the dataset.
- **DELETE method:** To remove bikes from the dataset.

3. GustaveBike

- **GET method:** To retrieve the status and details of available bikes for sale.
- **POST method for Basket:** To add bikes to the user's shopping basket.
- **GET method for Basket:** To view the contents of the basket along with the total price (with an option to convert the total price into another currency).
- **POST method to Buy Basket:** To check the user's balance and make the buyer the owner of the selected bikes.

This structure and approach aim to deliver a robust and scalable system capable of meeting the project's rental and sales requirements efficiently. While implementing, some challenges were encountered, including designing appropriate relationships between entities and ensuring smooth integration between the Controller and Services layers.

Scenarios for System Operations

1. Bike Rental

- **User 1** successfully rents a bike that is available in the system.
- **User 2** attempts to rent the same bike but is notified that it is currently unavailable.
- **User 2** requests to be added to the bike's waiting list.

- Once **User 1** returns the bike, the system automatically notifies **User 2** that the bike is now available for rental.
- **User 2** can then proceed to rent the bike.

2. Bike Purchase

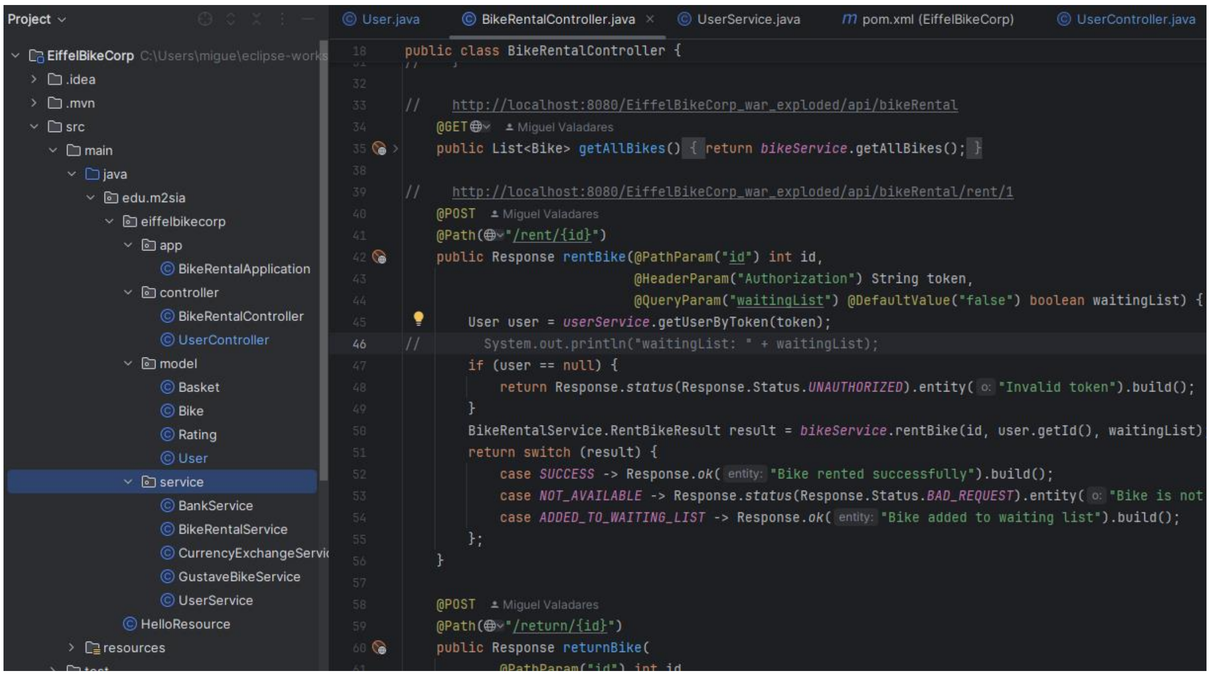
- A user explores the catalog of bikes available for purchase, where only bikes that have been rented at least once are listed for sale.
- The user selects a bike of interest and adds it to their shopping basket.
- The user proceeds to checkout, selecting their preferred currency for payment.
- The system interacts with the **Bank Service** to validate the payment and ensure sufficient funds are available.
- Upon successful payment validation, the system finalizes the transaction, transferring bike ownership to the user and updating the inventory accordingly.

Difficulties Encountered

1. Eclipse IDE

We found working with Eclipse challenging because some of the libraries we intended to use did not integrate as expected. To address this issue, we switched to IntelliJ IDEA with a Maven build system. This choice allowed for easier library management and seamless imports, significantly improving the development process.

Additionally, IntelliJ's **Code With Me** feature enabled collaborative coding in real-time, making it easier for team members to work together without the need to create a whole new server setup to test the project.



```java
public class BikeRentalController {

//    http://localhost:8080/EiffelBikeCorp_war_exploded/api/bikeRental
    @GET    Miguel Valadares
    public List<Bike> getAllBikes() { return bikeService.getAllBikes(); }

//    http://localhost:8080/EiffelBikeCorp_war_exploded/api/bikeRental/rent/1
    @POST   Miguel Valadares
    @Path("/rent/{id}")
    public Response rentBike(@PathParam("id") int id,
                             @HeaderParam("Authorization") String token,
                             @QueryParam("waitingList") @DefaultValue("false") boolean waitingList) {
        User user = userService.getUserByToken(token);
//        System.out.println("waitingList: " + waitingList);
        if (user == null) {
            return Response.status(Response.Status.UNAUTHORIZED).entity( o: "Invalid token").build();
        }
        BikeRentalService.RentBikeResult result = bikeService.rentBike(id, user.getId(), waitingList)
        return switch (result) {
            case SUCCESS -> Response.ok( entity: "Bike rented successfully").build();
            case NOT_AVAILABLE -> Response.status(Response.Status.BAD_REQUEST).entity( o: "Bike is not
            case ADDED_TO_WAITING_LIST -> Response.ok( entity: "Bike added to waiting list").build();
        };
    }

    @POST   Miguel Valadares
    @Path("/return/{id}")
    public Response returnBike(
```

2. Testing API Requests

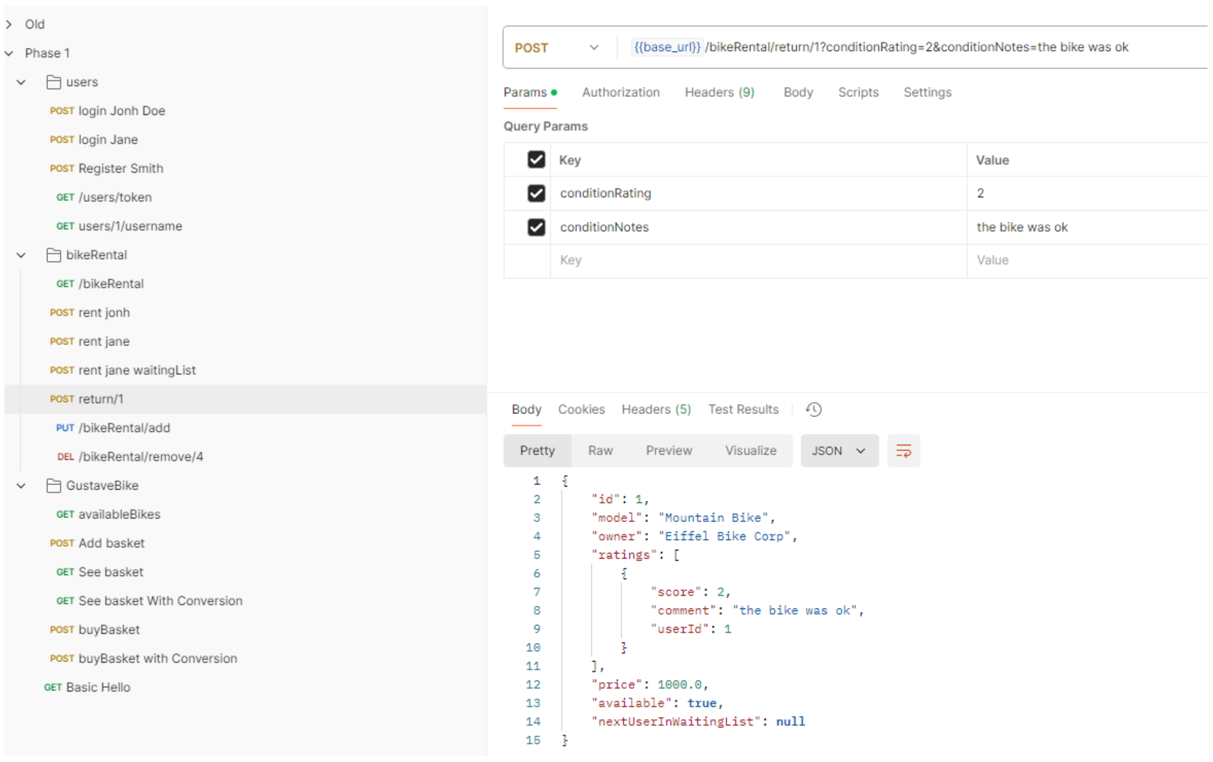
Initially, we used the `curl` tool to test our API requests. While `curl` is a powerful and versatile command-line tool, it presented several limitations:

- It lacked the ability to store and organize multiple API requests in a structured manner.
- Reproducing complex requests with headers, body payloads, or query parameters required significant manual effort.
- It was difficult to visualize and compare the responses of different API calls effectively.

Given these constraints, we opted to use **Postman**, a dedicated tool for API development and testing. Postman provided several key advantages:

- **Organized Collections:** We could group related API endpoints into collections, making it easier to test and retest them systematically.
- **Environment Variables:** Postman allowed the use of environment variables for dynamic testing, such as storing authentication tokens or server URLs, reducing repetitive tasks.
- **Visual Interface:** Its user-friendly interface simplified crafting and debugging complex requests, such as POST or PUT requests with JSON payloads or multipart data.
- **Response History:** Postman automatically saved the history of our requests and responses, allowing us to track changes and debug issues effectively.

Switching to Postman significantly enhanced our ability to test and validate API endpoints, ensuring a smoother development workflow and reducing debugging time.



```
> Old
v Phase 1
  v users
      POST login Jonh Doe
      POST login Jane
      POST Register Smith
      GET /users/token
      GET users/1/username
  v bikeRental
      GET /bikeRental
      POST rent jonh
      POST rent jane
      POST rent jane waitingList
      POST return/1
      PUT /bikeRental/add
      DEL /bikeRental/remove/4
  v GustaveBike
      GET availableBikes
      POST Add basket
      GET See basket
      GET See basket With Conversion
      POST buyBasket
      POST buyBasket with Conversion
    GET Basic Hello
```

POST {{base_url}} /bikeRental/return/1?conditionRating=2&conditionNotes=the bike was ok

Params ● Authorization Headers (9) Body Scripts Settings

Query Params

✓	Key	Value
✓	conditionRating	2
✓	conditionNotes	the bike was ok
	Key	Value

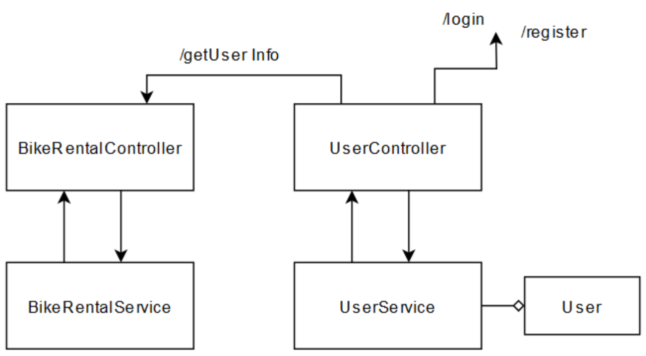
Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
{
    "id": 1,
    "model": "Mountain Bike",
    "owner": "Eiffel Bike Corp",
    "ratings": [
        {
            "score": 2,
            "comment": "the bike was ok",
            "userId": 1
        }
    ],
    "price": 1000.0,
    "available": true,
    "nextUserInWaitingList": null
}
```

3. Dependency between Services and controllers

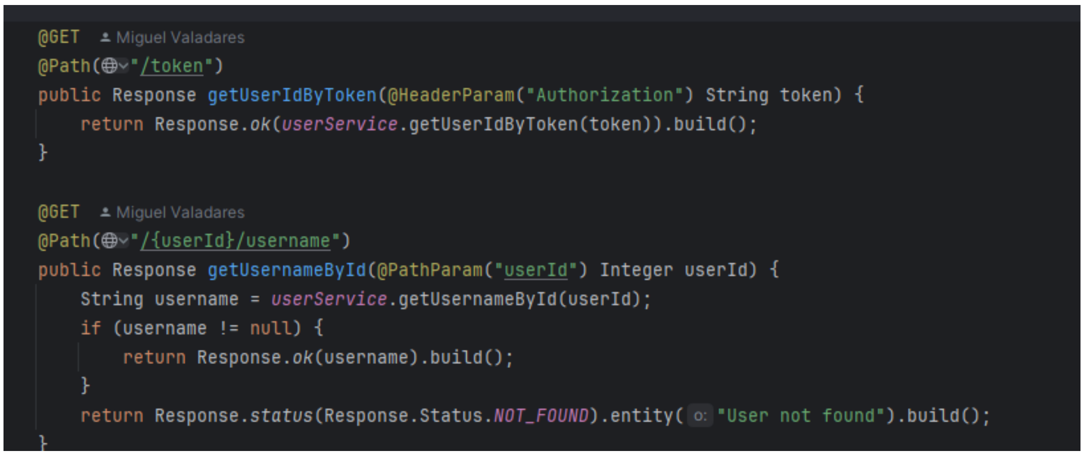
In an ideal architecture, each Service should correspond to its own Controller. This design ensures clear separation of concerns and minimizes dependencies, making the system modular and easier to maintain.



However, during the implementation, we encountered several practical challenges:

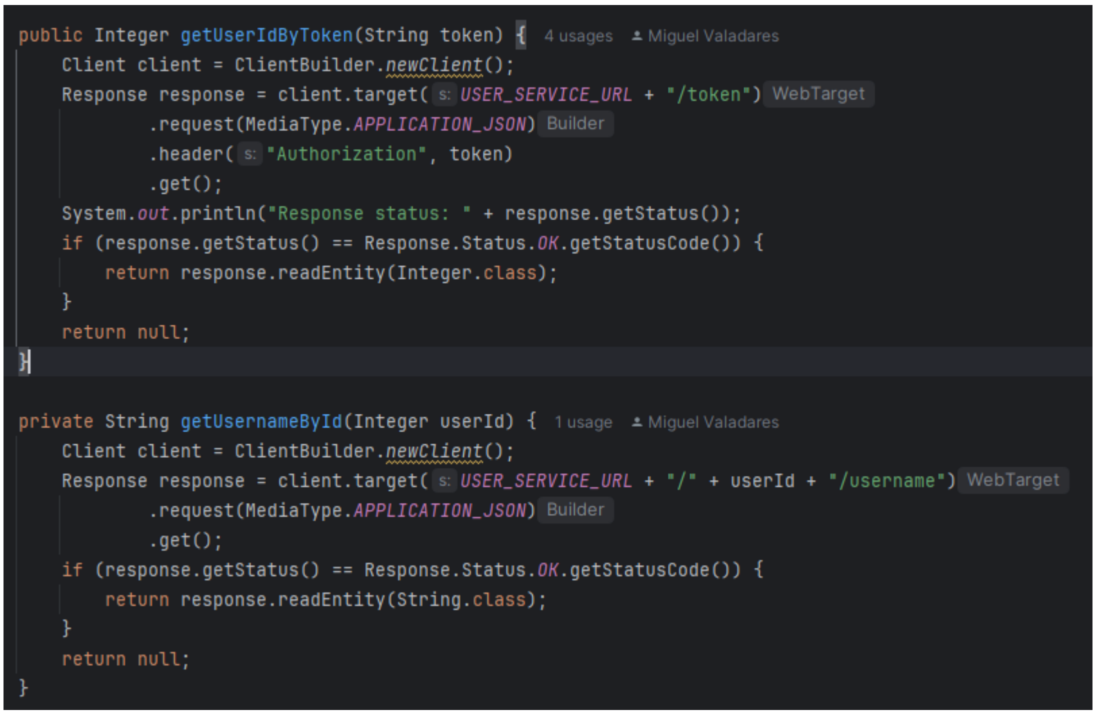
- **Overlapping Functionalities:** Some features required collaboration between multiple services, such as synchronizing bike rental and sale data. This made it difficult to create fully independent controllers without introducing redundant code.

- **Increased Complexity:** Strict adherence to a one-to-one mapping between Services and Controllers led to unnecessary complexity



```java
@GET
@Path("/token")
public Response getUserIdByToken(@HeaderParam("Authorization") String token) {
    return Response.ok(userService.getUserIdByToken(token)).build();
}

@GET
@Path("/{userId}/username")
public Response getUsernameById(@PathParam("userId") Integer userId) {
    String username = userService.getUsernameById(userId);
    if (username != null) {
        return Response.ok(username).build();
    }
    return Response.status(Response.Status.NOT_FOUND).entity("User not found").build();
}
```



```java
public Integer getUserIdByToken(String token) {
    Client client = ClientBuilder.newClient();
    Response response = client.target(USER_SERVICE_URL + "/token")
            .request(MediaType.APPLICATION_JSON)
            .header("Authorization", token)
            .get();
    System.out.println("Response status: " + response.getStatus());
    if (response.getStatus() == Response.Status.OK.getStatusCode()) {
        return response.readEntity(Integer.class);
    }
    return null;
}

private String getUsernameById(Integer userId) {
    Client client = ClientBuilder.newClient();
    Response response = client.target(USER_SERVICE_URL + "/" + userId + "/username")
            .request(MediaType.APPLICATION_JSON)
            .get();
    if (response.getStatus() == Response.Status.OK.getStatusCode()) {
        return response.readEntity(String.class);
    }
    return null;
}
```

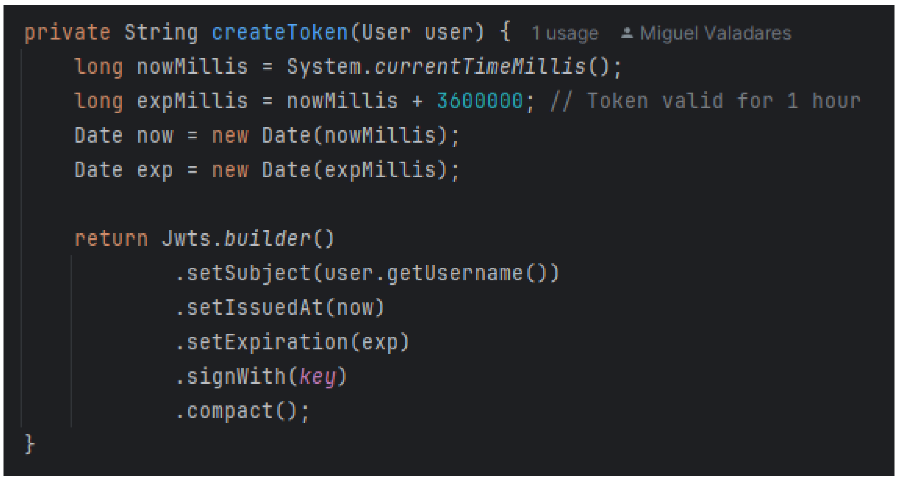
Example of Implematation of

4. Methods RestFUL with authentication

Implementing secure RESTful methods requires careful consideration of authentication mechanisms to protect endpoints from unauthorized access. For this we use Token-Based Authentication.

We used **JWT (JSON Web Token)** for user authentication. This approach allowed us to securely verify user identity while keeping the system stateless, which aligns with REST principles.

- **Login:** Upon successful login, the server generates a JWT for the user, which is sent back to the client.
- **Request Authentication:** Clients include this token in the headers (e.g., `Authorization: <token>`) for subsequent API calls.



```java
private String createToken(User user) {  1 usage  Miguel Valadares
    long nowMillis = System.currentTimeMillis();
    long expMillis = nowMillis + 3600000; // Token valid for 1 hour
    Date now = new Date(nowMillis);
    Date exp = new Date(expMillis);

    return Jwts.builder()
            .setSubject(user.getUsername())
            .setIssuedAt(now)
            .setExpiration(exp)
            .signWith(key)
            .compact();
}
```

RESTful endpoints were secured based on user roles. Tokens were configured to expire after a certain time, requiring users to refresh them periodically, adding an extra layer of security.

5. Notifying users

Due to time and resource constraints, we couldn't fully implement the user notification system. However, we planned the following approach for future iterations:

- **Notification Triggers:**
  Notifications would be sent based on specific events, like:
  - A bike on the waiting list becomes available.
  - Successful rental or purchase transactions.
  - Account changes, such as password updates.

- **Methods for Notification Delivery:**
  - **Via Email Notification:** Create an STMP server that would allow us to send those notifications via email.
  - **In-App Notifications:** Users would see alerts in their dashboard. However, this feature would depend on the implementation of the GUI, as it requires a user interface to display notifications effectively.

6.Implememantion of GUI

We planned to implement a user-friendly graphical user interface (GUI) for the system but faced time constraints that prevented its completion.

Initially, we explored two approaches for the GUI:

1. **JavaFX:**
   - JavaFX was considered for its ability to create desktop-based applications with a rich user interface.
2. **Web-Based Interface (HTML, CSS, JavaScript):**
   - We also try working on a web-based GUI using basic HTML for structure, with some basic CSS for styling, and JavaScript for interactivity.

Unfortunately, due to limited time, we couldn’t fully implement either approach. Our efforts were directed primarily towards building and validating the backend functionalities. Possible future iterations of the project will focus on completing the GUI to deliver a seamless user experience.