# II. Implementation and DevOps Practices

## 4. Backend Development

The backend is the single source of truth in the form of a REST API, which the frontend can interact with. There are two main areas which are handled by the backend.

1. Business logic: As with any logic handling money or other sensitive data, these operations must be done in an environment where bad actors or clueless users cannot alter the result. Therefore, since we are developing a gambling site, the actual game logic and results are calculated, handled and stored only in the backend. Our backend has API endpoints for playing games for users such as the stateless coin-flip endpoint.

```json
{
    "actions": [
        {
            "name": "play",
            "method": "POST",
            "href": "/game/coin-flip",
            "type": "application/json",
            "fields": [
                {
                    "name": "choice",
                    "type": "TEXT",
                    "options": [
                        { "value": "HEADS", },
                        { "value": "TAILS", }
                    ]
                },
                {
                    "name": "betAmount",
                    "type": "NUMBER",
                }
            ]
        },
        {
            "name": "result",
            "method": "GET",
            "href": "/game/coin-flip/{id}",
            "type": "application/x-www-form-urlencoded"
        }
    ],
    "links": [ { "rel": [ "self" ], "href": "/game/coin-flip" } ]
}
```

*Simplified response listing the actions and links available on `/game/coin-flip`.*

If the above JSON did not give it away, the API exposes hypermedia to alter and interact with the state of the backend. Exactly how will be discussed later, for now the most important aspect of the request is that we have two endpoints to interact with the coin-flip game. We can either

```
1. Play the game via the POST method
2. Get the result of a specific coin-flip game.
```

As mentioned before, this game is stateless, meaning once a client POST's or rather plays the coin-flip game, the result of the game is immediately calculated, saved and an ID linking to the result is returned for the client to consume via the second action 'GET /game/coin-flip/{id}. An example finished game could be the following json

```json
{
    "properties": {
        "betAmount": 100,
        "gameResult": "USER_LOSE",
        "userChoice": "HEADS",
        "id": 1,
        "result": "TAILS"
    },
    "links": [ { "rel": [ "self" ], "href": "/game/coin-flip/1" } ]
}
```

*Simplified response for a specific coin-flip game.*

Here after sending a `POST` request with the body `{ "choice": "HEADS", "betAmount": 100 }`, the user lost as the result was "TAILS".

2. Users: The second main area of our API is handling users and authentication. We decided, to challenge ourselves and roll our own authentication and JWT tokens, the exact specifics of the JWT tokens and how they are created and managed will be examined later. In short, if a user wants to play a game they must first be registered and authenticate with a username and password to retrieve a short lived *access-token* (JWT) and a long lived *refresh-token* (UUID). The *access-token* grants the user access to locked endpoints based on the user's assigned roles like playing the coin-flip game and the *refresh-token* allows a user to refresh the *access-token* before it expires to create a new *access-token*. This is to reduce the number of times a user must send their credentials to the backend for increased security. If the *access-token* expires the user must login again with credentials and the same is true for the *refresh-token*.

**Quarkus Framework Implementation** Our choice of programming language and then framework was not up for much debate. We all have some experience with Java and therefore it was an obvious choice. However, instead of selecting Spring as our web framework we chose Quarkus for variation. Quarkus alike Spring supports the Jakarta EE standards meaning many of the core principles of developing a web server is the same between them. We chose Quarkus over Spring for a mild challenge in something new but familiar.

Quarkus is a performant container first web application and microservice framework. For our needs we utilized the web server components for constructing an API to accomodate our gambling website's business logic and user management. The main benefits we have utilized from a web application framework such as Quarkus is the ease of adding native but tailored components. As an example, we will delve into JWT tokens and how we use them later. The other main benefit of such a framework is Jakarta bean validation and dependency injection. We use this liberally with the `@Inject` annotation for *injecting* a class into another without having to manually construct it or the like. Instead Quarkus takes care of the scope of the class via `@RequestScoped` or `@ApplicationScoped` among a few others to figure out when and how a new instance of a class will be created and injected.

```
@ApplicationScoped
public class AuthService {

    @Inject
    TokenService tokenService;

    @Inject
    UserService userService;

    //...
}
```

*Simplified implementation of the `AuthService` class.*

Another key aspect of using Quarkus is the ease of creating a web application, specifically selecting which endpoints or resources should be exposed with what method.

```
@Path("/auth")
@Produces(Siren4J.JSON_MEDIATYPE)
public class AuthResource {

    //...

    @GET
    @Path("/")
    public Entity getAuth() throws Siren4JException {
        //...
    }

    @POST
```

```java
    @Path("/token")
    @Consumes(MediaType.APPLICATION_JSON)
    public Entity createToken(CreateTokenRequest credentials) {
        //...
    }

    @POST
    @Path("/token/refresh")
    @Consumes(MediaType.APPLICATION_JSON)
    public Entity refreshAccessToken(RefreshAccessTokenRequest request) {
        //...
    }
}
```

*Simplified implementation of the `AuthResource` class.*

Simple annotations define

- the top level path for the class or relative function level path via the `@Path` annotation,
- the common return type for all the endpoints in the class via the `@Produces` annotation,
- the HTTP method of the function via the `@GET`, `@PUT`, `@POST`, etc. annotations,
- the accepted Content-Type for a method via the `@Consumes` annotation and
- many other handy and useful features.

**REST API Design with Siren Hypermedia** The backend architecture evolved from a basic Level 3 REST implementation utilizing standard HTTP methods (GET, POST, DELETE) to incorporate HATEOAS (Hypermedia as the Engine of Application State) capabilities. This enhancement enables dynamic API navigation through hypermedia controls, where clients discover available resources and operations by traversing links from the root endpoint rather than relying on predefined URL patterns. The hypermedia approach mirrors web browsing behavior, where navigation occurs through semantic controls rather than explicit URL construction. For example, rather than hardcoding paths to submit orders, clients can programmatically follow named links to execute operations. This design pattern eliminates the need for clients to maintain URL knowledge or external API documentation, as the server communicates available state transitions through hypermedia responses. The implementation effectively transforms HTTP interactions from direct URL manipulation to a state machine driven by discoverable, context-aware controls.

We chose Siren as our hypermedia specification, although we also considered HAL and HAL-FORMS, but found HAL to be too primitive and HAL-FORMS to have insufficient support in Quarkus. Siren is an extension of the JSON format, `application/vnd.siren+json`. It has 5 main components.

1. Properties

   The `properties` keyword is an optional reserved keyword which alike in a traditional JSON response describes the state of an entity.

   ```
   {
       "properties": {
           "id": 1,
           "username": "admin"
       }
   }
   ```

   *Example properties in Siren.*

2. Links

   The `links` keyword is a required keyword. A staple of hypermedia is that it enables the discoverability and relational nature of APIs. A given resource is required to contain the `self` relation with an `href` to the very URL or resource the client is viewing. Other common relations are, for pagination, "prev" or "next", as well as other closely related resources.

   ```
   {
       "links": [
           { "rel": [ "self" ], "href": "/users/1" }
       ]
   }
   ```

   *Example self link in Siren.*

3. Entities

   The `entities` keyword is optional but usually used for collections. For example the endpoint `/users` probably returns a collection of all the users in the system. An API architect can decide to embed the full resource or a partial representation of it. A client may then see the full resource representation by navigating via the entity's "self" `rel`.

   ```
   {
       "entities": [
           {
               "class": [ "user" ],
               "rel": [ "item" ],
               "properties": {
                   "id": 1,
                   "username": "admin"
               },
               "links": [ { "rel": [ "self" ], "href": "/users/1" } ]
           },
   ```

```
        {
            "class": [ "user" ],
            "rel": [ "item" ],
            "properties": {
                "id": 2,
                "username": "user"
            },
            "links": [ { "rel": [ "self" ], "href": "/users/2" } ]
        }
    ]
}
```

*Example users entities list in Siren.*

4. Actions

   The `actions` keyword is an optional list of all related actions to the retrieved resource. Usually reserved for more advanced request like path templated GET requests `/users/{id}` or POST and PUT methods which may require specific contents in their request bodies. Actions were the missing piece for us when contemplating the incorporation of HAL. Without them a hypermedia API in our opinion is inadequate.

```
{
    "actions": [
        {
            "name": "create-user",
            "method": "POST",
            "href": "/users",
            "title": "Create user",
            "type": "application/json",
            "fields": [
                {
                    "name": "username",
                    "title": "Username",
                    "type": "TEXT",
                    "required": true
                },
                {
                    "name": "password",
                    "title": "Plain-text password",
                    "type": "TEXT",
                    "required": true
                }
            ]
        }
    ]
}
```

*Example "create-user" action in Siren.*

The resulting generic request for an action would be

```
curl -X '{action method}' \
    -H "Content-Type: {action type}" \
    -d '{
        # for each field in the action:
        "{field name}": "<value>"
    }' \
    {action href}
```

*Generic `curl` command when parsing Siren actions.*

For the specific "create-user" action above the request would be

```
curl -X POST \
    -H "Content-Type: application/json" \
    -d '{
      "username": "JohnDoe",
      "password": "password1234"
    }' \
    /users
```

*Example "create-user" action as a `curl` command.*

5. Class

   The final main component of the Siren specification is the optional `class` keyword. It describes the nature of an entity's content, usually it is used as a descriptor for the returned resource.

   See the previous `entities` component for an example.

We chose a third party library for our Siren implementation. Unlike Spring with its format agnostic `WebMvcLinkBuilder`, Quarkus lacks a mature and cohesive hypermedia builder, hence we selected what we believe to be the most intuitive and feature rich Java Siren library, Siren4J. This library contains both a builder like the Spring WebMvcLinkBuilder for more custom runtime responses but so far we have only used the resource API through annotations.

A simple resource like our root `/` resource contains discoverable links to other areas and resources of our API.

```java
@Siren4JEntity(entityClass = "root", uri = "/", links = {
    @Siren4JLink(rel = "users", href = "/users"),
    @Siren4JLink(rel = "auth", href = "/auth"),
    @Siren4JLink(rel = "game", href = "/game")
})
public class GetRootResponse {
}
```

*The `GetRootResponse` class implementation.*

Notice we link to three resources: “users”, “auth” and “game”, each with their own path “/users” “/auth” and “/game”. To properly serialize this response we use Siren4Js - admittedly slow - `ReflectingConverter` to convert the Java object into a Siren `Entity`.

```java
@ApplicationScoped
public class RootController {

    public Entity getRoot() throws Siren4JException {
        GetRootResponse rootResponse = new GetRootResponse();
        return ReflectingConverter.newInstance().toEntity(rootResponse);
    }
}
```

*The `RootController` class implementation‘.*

And finally, our `RootResource`, which defines our endpoint, simply calls the controller

```java
@Path("/")
@Produces(Siren4J.JSON_MEDIATYPE)
public class RootResource {

    @Inject
    RootController rootController;

    @GET
    public Entity getRoot() throws Siren4JException {
        return this.rootController.getRoot();
    }
}
```

*The `RootResource` java class implementation.*

Performing a GET request on the `http://localhost:8080/` results in the following Siren JSON

```json
{
    "class": [ "root" ],
    "properties": {
        "$siren4j.class": "org.csdg8.root.dto.GetRootResponse"
    },
    "links": [
        { "rel": [ "self" ], "href": "/" },
        { "rel": [ "users" ], "href": "/users" },
        { "rel": [ "auth" ], "href": "/auth" },
        { "rel": [ "game" ], "href": "/game" }
```

```
    ]
}
```

*The Siren response from `GET /`.*

Notice the `class` keyword defined is the `entityClass` value "root" and the links are created one to one as they are defined in the annotation of the `GetRootResponse` class.

Siren4J automatically creates the `$siren4j.class` keyword in the response's `properties` and populates it with the full class path.

This same strategy of construction annotations on a DTO are used throughout our API on all endpoints. A limitation of this is that we must be careful when changing links as we have to hard-code the value in the annotations since Java annotations can only contain compile-time values. Although, as we will see later, tests help us in properly verifying that links match. If we needed a response with a runtime link or another runtime value we can utilize Siren4J's `LinkBuilder`. The following is an example from the library's GitHub page:

EXAMPLE BUILDER:

```java
// Create a new self Link
Link selfLink = LinkBuilder.newInstance()
        .setRelationship(Link.RELATIONSHIP_SELF)
        .setHref("/self/link")
        .build();

// Create a new Entity
Entity result = EntityBuilder.newInstance()
        .setEntityClass("test")
        .addProperty("foo", "hello")
        .addProperty("number", 1)
        .addLink(selfLink)
        .build();
```

*Example of the Siren4J `LinkBuilder` to construct hypermedia during runtime.*

**Database Integration** To save the state of our application, users, and user's games we utilize a PostgreSQL database. We chose PostgreSQL as it is fast and easy to use and some of us have previous experience with this database type. However, the exact database type is redundant as we use Hibernate to interact with the database. Hibernate is an object to relational mapping (ORM) framework which allows us to define Java classes (instantiated as objects) and convert them to database entities.

A simple example is the following `CoinFlipGame` entity class

```java
@Entity
```

```java
@Table(name = "coin-flip-game")
public class CoinFlipGame extends PanacheEntity {
    private CoinFlipState choice;
    private CoinFlipState result;
    private Long betAmount;
    private CoinFlipGameResult gameResult;

    //...
}
```

*Simplified implementation of the `CoinFlipGame` class.*

The class is annotated with the `@Entity` jakarta persistence annotation to signal that this class is an entity and must be stored in our database. We also specify the table name and optionally in the `@Table` annotation one could also specify the schema if desired. The `CoinFlipGame` entity class extends the Hibernate PanacheEntity. This gives many quality of life methods for manipulating a single object/entity or fetching multiple from the database. Functions we have utilized include `findById(...)`, `persist()` and `delete()`. The general database table constructed from the `CoinFlipGame` entity will look like the following:

| ID | choice | result | betAmount | gameResult |
|---|---|---|---|---|
| 1 | TAILS | HEADS | 100 | USER_LOSE |
| 2 | HEADS | HEADS | 150 | USER_WIN |
| ... | ... | ... | ... | ... |

*Example database table for the `CoinFlipGame` entity.*

By using the Hibernate ORM framework we avoid having to manually handle serialization and deserialization with SQL to the PostgreSQL database. Hibernate is almost entirely database agnostic meaning we can easily switch our database for another type without sweeping changes to our implementation.

**Business Logic Implementation** *business logic separation, service classes.*

The general structure of our backend application is by domain. We organize Java classes by their domain instead of the classic "Model View Controller" (MVC) file structure. We believe this allows for better maintainability for future changes. Instead of having too look through a "service" folder with 20 different service classes to find the `TokenService`, instead since this class is related to the "auth" domain it resides in the "auth" folder.

The following is a snippet of our file structure utilizing the domain organization

```
...
+-- csdg8
    +-- auth
```

```
|   +-- AuthController.java
|   +-- AuthResource.java
|   +-- AuthService.java
|   +-- dto
|   |   +-- CreateTokenRequest.java
|   |   +-- CreateTokenResponse.java
|   |   +-- GetAuthResponse.java
|   |   +-- RefreshAccessTokenRequest.java
|   |   +-- RefreshAccessTokenResponse.java
|   +-- TokenService.java
+-- user
|   +-- dto
|   |   +-- CreateUserRequest.java
|   |   +-- GetCollectionUserResponse.java
|   |   +-- GetUserResponse.java
|   +-- UserController.java
|   +-- User.java
|   +-- UserResource.java
|   +-- UserService.java
|
...
```

*Snippet of the backend file structure showing its organization by the domains "auth" and "user".*

To further simplify new features or changes our general class hierarchy is as follows

1. Resource
2. Controller
3. Service
4. Business Logic

The resource classes only have direct interaction with a single controller in the same domain or subdomain. A controller only has direct interaction with resources and service classes. The service classes may interact with multiple controllers and services and finally the business logic, in our case the game logic, may only interact with other business logic classes or preferrably a single service class.

Another feature of this model is our separation of DTOs from implementation classes. None of the API responses return a class that is directly used in a service or in business logic. Instead, they are mapped to a DTO to both shield the actual business logic from the web application framework and vice versa but also to limit accidentally exposing sensitive information, like passwords for returned user objects.

A further improvement to this structure would be, in the same vein as our DTOs, shielding the service and business logic from our database. This would allow a, not-so, “hot-swap” of frameworks or database fully isolating the API from the business logic and service, and the database.

**Security Implementation** *quarkus security with rolesallowed, 401 vs 403.*

To secure our API we use Quarkus Security Jakarta Persistance. Our User entity’s variables have security annotations provided by the library to easily configure security attributes. The most important annotation is the `@Roles` attribute. We secure our endpoints with `@RolesAllowed` annotations and if a user does not have a matching role defined in their `@Roles` anntotated `Set` they are denied access.

```
@Entity
@Table(name = "app-user")
@UserDefinition
public class User extends PanacheEntity {

    @Username
    public String username;

    @Password
    public String password;

    @Roles
    public Set<String> role;

    //...
}
```

The `@UserDefinition` tells our application that this entity is a source of identity information, whilst the accompanying `@Username` indicates that this field is a username, the `@Password` indicates that this field is a hashed password and finally the `@Roles` indicates that this field is a collection (a `Set` in our case to force unique roles) of roles.

To secure a specific endpoint we use the `@RolesAllowed` annotation with a single or multiple specified roles. In the following example users accessing the `/auth/token/refresh` endpoint must have either the “user” or “admin” role.

```
@POST
@RolesAllowed({ "user", "admin" })
@Path("/token/refresh")
@Consumes(MediaType.APPLICATION_JSON)
public Entity refreshAccessToken(RefreshAccessTokenRequest request) throws Siren4JException
    //...
}
```

*Simplified endpoint which uses the `@RolesAllowed` annotation to limit access.*

The annotation can also be used for more fine-grained control on service methods if a part of the application must be extra secure, however we have not utilized this functionality yet.

**JWT Authentication** For authentication we decided to roll our own JWT tokens via the MicroProfile JWT RBAC specification. Quarkus conveniently provides a library for this named `quarkus-smallrye-jwt`. In short, a JWT token is a server provided signed token which anyone can verify was signed by the server, meaning the contents and access which it grants are valid. To accomplish this JWT tokens are signed with a *private key* by the server and can be verified with the linked *public key*. Contrary to a regular encrypted transaction where the *public key* signs some data which can then only be unlocked with the *private key*. In our self-rolled JWT implementation, when a registered user sends their username and password in a POST request to the `/auth/token` endpoint, the credentials are validated and a JWT token is generated via the *private key*.

```
public String generateAccessToken(User user) {
    return Jwt.issuer(this.issuer)
        .upn(user.getUsername())
        .subject(user.id.toString())
        .groups(user.getRole())
        .expiresIn(Duration.ofMinutes(5))
        .sign();
}
```

*Our self-rolled JWT token generation.*

Of note here is the `upn` which is our main unique ID in our JWT tokens, but we also have a `subject` claim which we use to identify a user in our backend.

Once the JWT token is generated and signed it is returned to the user. When the user then sends a request to a locked endpoint (recall the `@RolesAllowed` annotation) the token is automatically verified by the Quarkus framework with the *public key* and verified if it was actually signed by the server. If it is valid and if the user has the required roles the request may proceed.

As mentioned earlier we utilize a access-token (JWT) and refresh-token pair. With the access-token being short-lived and the refresh-token being long lived. The refresh-token is a simple generated UUID and the backend keeps track of which UUIDs it generates. A future improvement will be storing the list of these generated UUIDs in persistent storage instead of in-memory as it is now. If the server terminates for whatever reason the list of valid refresh-tokens is lost and all users must re-login to get a new refresh-token. The access-token should not be persisted as it is assumed, because it is cryptographically signed, that it is always valid if it is not expired. To re-emphasize, a JWT token can only come from the server as it is the only one with the *private key*.

**Testing Strategy for UserService**

The **testing strategy** for the `UserService` class ensures that core functionalities related to user management—such as user validation, adding users, fetching users, and handling user roles and balances—work as expected. It also addresses edge cases and error scenarios, ensuring the service remains reliable and resilient under various conditions.

The strategy includes both unit tests and integration tests to confirm that the system's behavior aligns with business requirements.

---

### 1. JUnit Implementation for User Validation

- **Valid User Credentials**: Ensures users with correct credentials are validated successfully.
  - **Test**: `shouldValidateUserWithCorrectCredentials`
  - **Strategy**: Verify that the system returns the correct user when provided with valid credentials using JUnit.
- **Invalid User Credentials**: Ensures the system handles invalid credentials gracefully.
  - **Test**: `shouldNotValidateUserWithIncorrectCredentials`
  - **Strategy**: Verify that incorrect credentials do not authenticate the user using JUnit.

---

### 2. JUnit Implementation for User Role Management

- **Fetching Roles for Existing Users**: Confirms that user roles can be retrieved for valid users.
  - **Test**: `shouldGetUserRoleForExistingUser`
  - **Strategy**: Verify that the correct roles are returned for valid users using JUnit.
- **Fetching Roles for Non-Existent Users**: Ensures the service does not return roles for non-existent users.
  - **Test**: `shouldNotGetUserRoleForNonExistentUser`
  - **Strategy**: Verify that a non-existent user returns an empty set or a proper error using JUnit.

---

### 3. JUnit Implementation for User Management (CRUD Operations)

- **Adding New Users**: Ensures the system correctly handles user creation.
  - **Test**: `shouldAddUserSuccessfully`
  - **Strategy**: Verify that a new user is added successfully with the correct username and role using JUnit.

- **Handling Existing Users**: Ensures the system throws an appropriate exception when adding an existing user.
  - **Test**: `shouldThrowUserAlreadyExistsExceptionWhenAddingExistingUser`
  - **Strategy**: Verify that the service throws a `UserAlreadyExistsException` for duplicate users using JUnit.
- **Invalid Usernames or Passwords**: Tests edge cases with invalid credentials.
  - **Tests**: `shouldThrowInvalidCredentialsExceptionForInvalidUsername`, `shouldThrowInvalidCredentialsExceptionForInvalidPassword`
  - **Strategy**: Verify that invalid credentials trigger appropriate exceptions using JUnit.
- **Fetching User by Username**: Ensures users can be retrieved based on their username.
  - **Test**: `shouldGetUserByUsername`
  - **Strategy**: Verify that a user can be found and retrieved correctly by username using JUnit.
- **Handling Non-Existent User Fetching**: Ensures the system handles fetching non-existent users appropriately.
  - **Test**: `shouldThrowUserNotFoundExceptionForNonExistentUser`
  - **Strategy**: Verify that fetching a non-existent user results in a `UserNotFoundException` using JUnit.
- **Fetching All Users**: Validates that the system returns all users correctly.
  - **Test**: `shouldGetAllUsers`
  - **Strategy**: Verify that all users in the system are returned in a list using JUnit.

---

### 4. JUnit Implementation for User Balance Management

- **Adding Balance to User**: Ensures user balances can be incremented correctly.
  - **Test**: `shouldAddBalanceToUser`
  - **Strategy**: Verify that adding balance to a user's account works correctly using JUnit.
- **Subtracting Balance from User**: Ensures balances can be deducted correctly.
  - **Test**: `shouldSubtractBalanceFromUser`
  - **Strategy**: Verify that balance subtraction functions correctly using JUnit.

---

### 5. JUnit Implementation for Edge Cases and Error Scenarios

- **User Already Exists Exception**: Ensures the service prevents duplicate user creation.

- **Test**: `shouldThrowUserAlreadyExistsExceptionWhenAddingExistingUser`
  - **Strategy**: Verify that the system throws `UserAlreadyExistsException` for duplicate users using JUnit.
- **Invalid Credentials**: Confirms that invalid credentials trigger appropriate error handling.
  - **Tests**: `shouldThrowInvalidCredentialsExceptionForInvalidUsername`, `shouldThrowInvalidCredentialsExceptionForInvalidPassword`
  - **Strategy**: Verify that invalid credentials are handled properly using JUnit.
- **Non-Existent User Fetching**: Ensures requests for non-existent users are handled appropriately.
  - **Test**: `shouldThrowUserNotFoundExceptionForNonExistentUser`
  - **Strategy**: Verify that the service handles invalid user requests with proper exception handling using JUnit.

---

## 6. Transactional and Database Integrity

- **Database Transactions**: Ensures changes made during tests are rolled back after each test.
  - **Strategy**: Verify that database changes during tests are rolled back to maintain isolation between tests using JUnit.

---

## 7. REST-assured Testing for API Endpoints

- **User Validation Endpoint**: Ensures the API correctly validates users.
  - **Test**: `shouldValidateUserEndpoint`
  - **Strategy**: Use REST-assured to verify that the user validation endpoint returns the correct response for valid and invalid credentials.
- **User Role Management Endpoint**: Ensures the API correctly handles user roles.
  - **Test**: `shouldManageUserRolesEndpoint`
  - **Strategy**: Use REST-assured to verify that the user role management endpoint returns the correct roles for valid users and handles errors for non-existent users.
- **User Management Endpoints**: Ensures the API correctly handles CRUD operations for users.
  - **Tests**: `shouldAddUserEndpoint`, `shouldFetchUserEndpoint`, `shouldUpdateUserEndpoint`, `shouldDeleteUserEndpoint`
  - **Strategy**: Use REST-assured to verify that the user management endpoints correctly handle adding, fetching, updating, and deleting users.
- **User Balance Management Endpoint**: Ensures the API correctly handles user balances.

   - **Tests**: `shouldAddBalanceEndpoint`, `shouldSubtractBalanceEndpoint`
   - **Strategy**: Use REST-assured to verify that the user balance management endpoints correctly handle adding and subtracting balances.

---

### 8. Test Automation and Continuous Integration

- **Automated Test Suites**: Ensures the user service works as expected across different scenarios using JUnit and REST-assured.
  - **Strategy**: Integrate automated tests into a CI pipeline to catch issues early.
- **Regression Testing**: Ensures new changes or features do not break existing functionality.
  - **Strategy**: Rerun tests regularly to maintain application stability.

### OpenAPI Documentation

The **OpenAPI Documentation** provides a comprehensive specification for the API endpoints that interact with the `UserService`. It outlines the available operations for managing users, validating credentials, handling user roles, and updating user balances. This section ensures that the API is clearly defined, making it easy for developers to integrate with the service, understand the request/response formats, and know the expected behavior.

The following is the structure of the OpenAPI documentation for the endpoints related to the `UserService`:

1. **Validate User**

   - **Endpoint**: `/api/users/validate`
   - **Method**: `POST`
   - **Description**: Validates the credentials of a user by checking the provided username and password.

   **Request**:

   ```json
   {
     "username": "string",
     "password": "string"
   }
   ```

   **Response**:

   - `200 OK`: User is validated successfully.

     ```json
     {
       "username": "string",
       "roles": ["string"]
     }
     ```

- `401 Unauthorized`: Invalid credentials provided.

  ```json
  {
    "error": "Invalid credentials"
  }
  ```

2. **Add User**

   - **Endpoint**: `/api/users`
   - **Method**: `POST`
   - **Description**: Creates a new user in the system with the provided username, password, and roles.

   **Request**:

   ```json
   {
     "username": "string",
     "password": "string",
     "roles": ["string"]
   }
   ```

   **Response**:

   - `201 Created`: User successfully added.

     ```json
     {
       "username": "string",
       "roles": ["string"]
     }
     ```

   - `409 Conflict`: User already exists.

     ```json
     {
       "error": "User already exists"
     }
     ```

   - `400 Bad Request`: Invalid credentials or malformed request.

     ```json
     {
       "error": "Invalid credentials"
     }
     ```

3. **Get User Roles**

   - **Endpoint**: `/api/users/{username}/roles`
   - **Method**: `GET`
   - **Description**: Retrieves the roles associated with a specific user.

   **Request**:

   - **Path Parameter**:
     - `username` (string): The username of the user whose roles are to be retrieved.

**Response**:

- `200 OK`: Returns the roles associated with the user.

```json
{
  "roles": ["string"]
}
```

- `404 Not Found`: User not found.

```json
{
  "error": "User not found"
}
```

4. **Get User by ID**

- **Endpoint**: `/api/users/{userId}`
- **Method**: `GET`
- **Description**: Fetches the details of a user based on their unique user ID.

**Request**:

- **Path Parameter**:
  - `userId` (integer): The unique ID of the user to retrieve.

**Response**:

- `200 OK`: Successfully retrieves the user information.

```json
{
  "id": "integer",
  "username": "string",
  "roles": ["string"],
  "balance": "integer"
}
```

- `404 Not Found`: User not found.

```json
{
  "error": "User not found"
}
```

5. **Update User Balance**

- **Endpoint**: `/api/users/{userId}/balance`
- **Method**: `PATCH`
- **Description**: Updates the balance of a specific user by adding or subtracting a specified amount.

**Request**:

- **Path Parameter**:

- `userId` (integer): The unique ID of the user whose balance is to be updated.

- **Body**:

```json
{
  "amount": "integer"
}
```

**Response**:

- `200 OK`: Balance updated successfully.

```json
{
  "id": "integer",
  "username": "string",
  "balance": "integer"
}
```

- `404 Not Found`: User not found.

```json
{
  "error": "User not found"
}
```

6. **Remove User Balance**

- **Endpoint**: `/api/users/{userId}/balance`
- **Method**: `DELETE`
- **Description**: Removes a specified amount from the user's balance.

**Request**:

- **Path Parameter**:
  - `userId` (integer): The unique ID of the user whose balance is to be updated.

- **Body**:

```json
{
  "amount": "integer"
}
```

**Response**:

- `200 OK`: Balance updated successfully.

```json
{
  "id": "integer",
  "username": "string",
  "balance": "integer"
}
```

- `404 Not Found`: User not found.

```json
{
  "error": "User not found"
}
```

### Response Codes Summary

- `200 OK`: Request successfully completed.
- `201 Created`: Resource successfully created.
- `400 Bad Request`: Malformed request or invalid input.
- `401 Unauthorized`: Invalid credentials provided.
- `404 Not Found`: Resource (user) not found.
- `409 Conflict`: Resource already exists (e.g., attempting to create a user that already exists).