Complex Systems and DevOps: Deliverable 2

Date: November 25, 2024

Course Details

Course Name: Complex Systems and DevOps **Course Code:** 62582 **Semester:** Fall 2024

Team Members

Name	Student Number
Christoffer Fink	*s205449*
Kasper Falch Skov	*s205429*
Johan Søgaard Jørgensen	*s224324*
Henrik Lynggaard Skindhøj	*s205464*
Kevin Wang Højgaard	*s195166*
Sebastian Halfdan Lauridsen	*s215769*

Link to GitHub Project

Table of Contents

4. Backend Development
    - Quarkus Framework Implementation
    - REST API Design with Siren Hypermedia
    - Database Integration
    - Business Logic Implementation
    - Security Implementation
        - JWT Authentication
    - Testing Strategy
        - JUnit Implementation
        - REST-assured Testing
    - OpenAPI Documentation
5. Frontend Development
    - React Application Structure
    - TypeScript Integration
    - Vite Build Tool Implementation
    - State Management
    - Component Architecture
    - Security Features
        - Token Security Implementation
    - Package Management
6. DevOps Implementation
    - Version Control Practices
        - Git Workflow
        - GitHub Integration
    - Continuous Integration/Continuous Deployment
        - GitHub Actions Configuration
        - Build Server Setup
        - Testing Pipeline
    - Containerization
        - Docker Implementation
        - Container Registry
    - Cloud Deployment
        - Google Cloud Setup
        - Netlify Frontend Deployment
    - Monitoring and Maintenance
7. Conclusion
    - Project Outcomes
    - Future Improvements
    - Lessons Learned

+– 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 "u

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 domai

Another feature of this model is our separation of DTOs from implementation classes. None of

A further improvement to this structure would be, in the same vein as our DTOs, shielding th

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

To secure our API we use Quarkus Security Jakarta Persistance. Our User entity's variables h

```java
@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:

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

Response:

- `200 OK`: User is validated successfully.

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

- `401 Unauthorized`: Invalid credentials provided.

  ```
  {
    "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:

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

   Response:

   - `201 Created`: User successfully added.

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

   - `409 Conflict`: User already exists.

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

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

```
{
  "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.

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

   - `404 Not Found`: User not found.

     ```
     {
       "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.

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

   - `404 Not Found`: User not found.

```
{
  "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**:

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

   Response:

   - `200 OK`: Balance updated successfully.

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

   - `404 Not Found`: User not found.

     ```
     {
       "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**:

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

Response:

- `200 OK`: Balance updated successfully.

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

- `404 Not Found`: User not found.

```
{
  "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). ### 5. Frontend Development

React Application Structure

TypeScript Integration

Vite Build Tool Implementation

State Management

Component Architecture

Security Features

Token Security Implementation

Package Management