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

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