Ceng 453 Group1
Catan Game: Backend Design Document

Mustafa İlbey Deniz - 2448314
Muhammet Fatih Berberoğlu - 2380202

Table of Contents

1. List of Figures

2. List of Tables

3. Scope

This backend design document covers the project's structure, how data is stored in the database and their relations, the API endpoints for communication, and the authentication structure. Also the tests that are conducted are included, so are their coverage and pass conditions.

4. Backend Design

4.1. Project Structure

Our architecture follows the general Layered Architecture of Spring Boot, it can be seen in Figure 1.

Our project's package structure can be seen in Figure 2. We tried to consider the existing Spring Boot projects' package structures and concluded that there are common recurring patterns in folder structure.

We separated controllers, services, entities and repository interfaces into packages. We also have a DTO package storing each request DTO, config package for classes annotated with @Configuration and enum package.

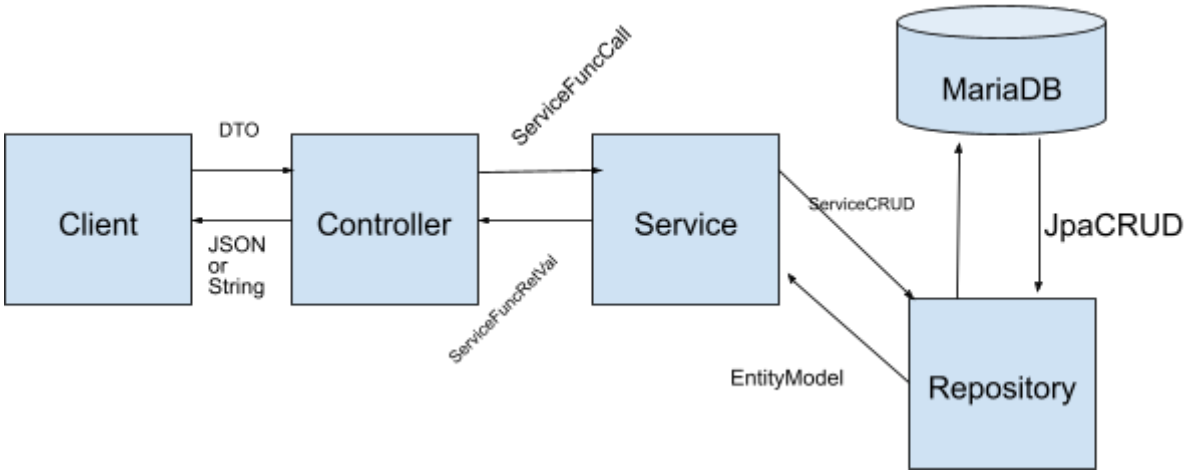

Figure 1: Project Layered Architecture

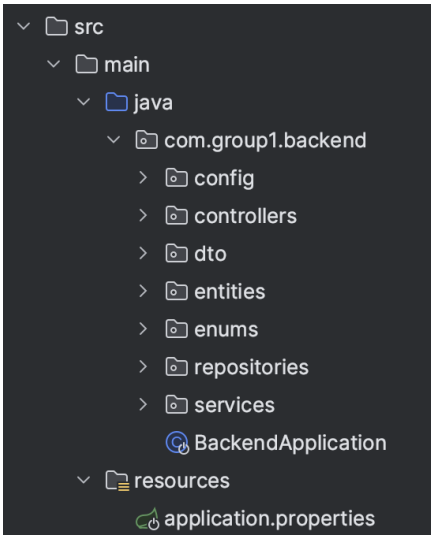

Figure 2: Project Folder Structure

4.2. Database Structure

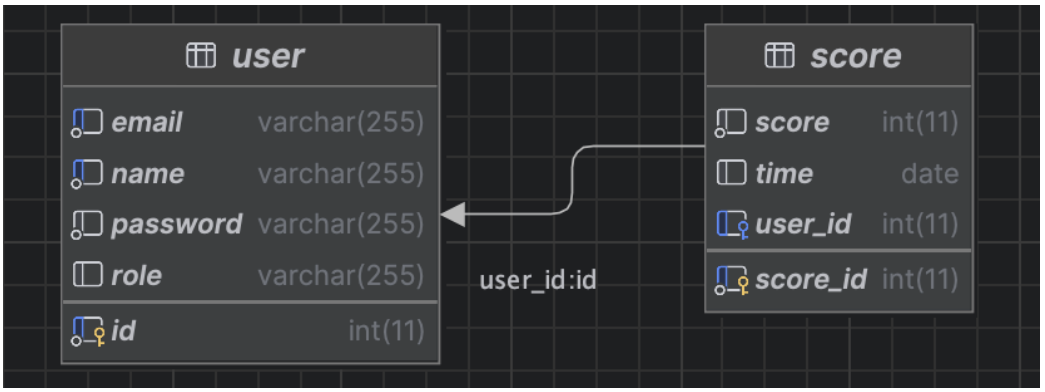

Figure 3: Database Diagram

Currently we have two tables: user and score. User has an auto generated id, unique name and unique email attributes and password attribute that stores hashed and salted passwords. Lastly, a role attribute which we currently don't utilize entirely.

The score table is related to the user in a many to one relation. It has an auto generated score_id as primary key. It takes the id attribute of the user as foreign key. score and time attributes are intuitive.

4.3. Authorization

We implemented the Bearer Token authorization using JWTs(see Figure 4).

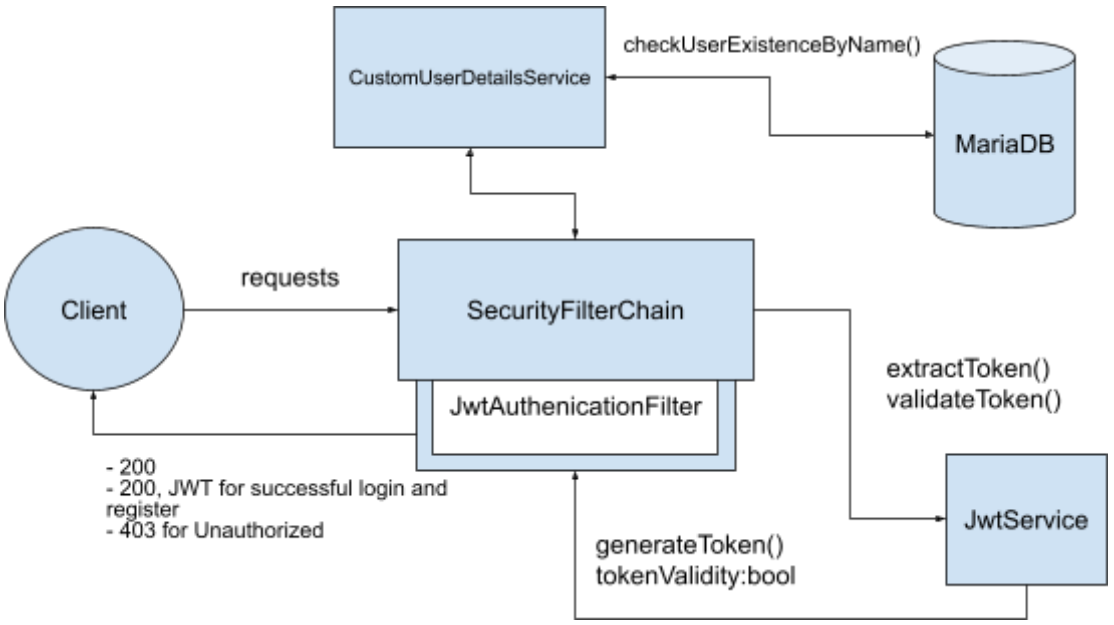

Figure 4: JWT Authorization Implementation Diagram

We used Spring Boot's security package, which provides a filter structure for each HTTP request that is made to our server. We require authorization for any endpoint that is not being whitelisted.

When a client makes a request, our SecurityFilterChain checks whether it's whitelisted or not. If it is, then returns the content and status code 200. If not, it applies the JwtAuthenticationFilter and checks for the Authorization Bearer Token header; if there is no

token, the token is not valid(expired or signed with a wrong signature) or the user with provided credentials does not exist, returns 403 status code.

For /login and /register requests, a token is generated and returned in the response after a successful attempt.

We are using a symmetric approach for signing our token since our backend service also works as an Authentication Provider, therefore we have no doubt about our token's integrity and signer authority.

4.4. API Endpoints

Endpoint	**token Required**	**Request Body**	**Response Type**	**Behaviour**
GET /api/get/users	YES	{}	JSON	Lists all users and their attributes
GET /api/get/top Scorers/weekly	YES	{}	JSON	Lists top scored users' names and scores for the past week
GET /api/get/topScorers/monthly	YES	{}	JSON	Lists top scored users' names and scores for the past month
GET /api/get/topScorers/all	YES	{}	JSON	Lists top scored users' names and scores of all time
POST /api/testemail	NO	{}	String	Sends a test email to a predetermined address

POST /api/saveScore	YES	{
		"name":String,
		"score":Int
		}
			String	Sets a score value for the specified user

POST /api/login	NO	{
		"name": String,
		"password": String
		}
			String (JWT)	Sends a login request

POST /api/register	NO	{
		"name":String,
		"email": String,
		"password": String
		}
			String (JWT)	Sends a register request

POST	NO	{	String	Validates user's

/api/forgotPassword		"name": String, "email":String }		name and email, and sends a random password to the user's email

POST /api/changePassword	YES	{
		"name": String,
		"oldPassword": String,
		"newPassword": String,
		}
			String	Validates user's name and current password and changes it with the provided one.

Table 1: API Endpoints Table

4.5. Tests

4.6. Unit Tests

All the test users are created in the following format:

- name: TestUser#
- email: TestUser#@mail.com
- password: HardPassword#
- role: ROLE_USER

Test Method	**Description**
testCreateUser()	Creates one predefined user, and saves it to the database. Checks the name, mail and role attributes. Deletes the user eventually.
testCreateMultipleUser()	It checks whether the test user exists in database, if it exists, it creates 10 new users, naming incrementally starting from the latest TestUser# that exists in the database.
testDeleteUser()	Deletes the latest TestUser#. Checks user table row count decrement.
testSaveUserScore()	Saves a random score to the latest TestUser#. Checks whether its fetched total score increased by the saved score or not.
testAddMultipleScoreToOneUser()	Saves 10 random score values to last TestUser#. Checks whether its fetched total score increased by the saved score or not.
testAddMultipleScoreToMultipleUser()	Saves 10 random score values to the first 10 of TestUser#. Checks whether its fetched total score increased by the saved score or not.