**Lab 06 - Secure REST Services with Spring Boot using JSON Web Token (JWT)**

# 1. Introduction

A JWT (JSON Web Token) is an open standard (RFC 7519) used for securely transmitting information between two parties (such as a client and a server) in the form of a JSON object. It is widely used in authentication and authorization systems, particularly in modern web applications.

A JWT is typically composed of three parts:

1. **Header**: Contains metadata about the token, such as the signing algorithm.
2. **Payload**: Contains the claims (information) that are being transmitted. These claims can be registered, public, or private.
3. **Signature**: Ensures the integrity of the token and verifies that it was created by the intended issuer.

Imagine you're a new fresher Spring Boot developer of IT Department in a University. Your manager has asked you to develop a RESTful API with registration and login to Orchid Management.

The RESTful API has to support registration, login for an orchid - a standardized usage action verbs better known as Create, Read, Update, Delete (CRUD).

This lab explores creating a RESTful API using **Spring Boot** 3, JWT with Data Source is MS SQL Server. An **SQL Server** **Database** will be created to persist the user information by **Spring Data JPA**

# 2. Lab Objectives

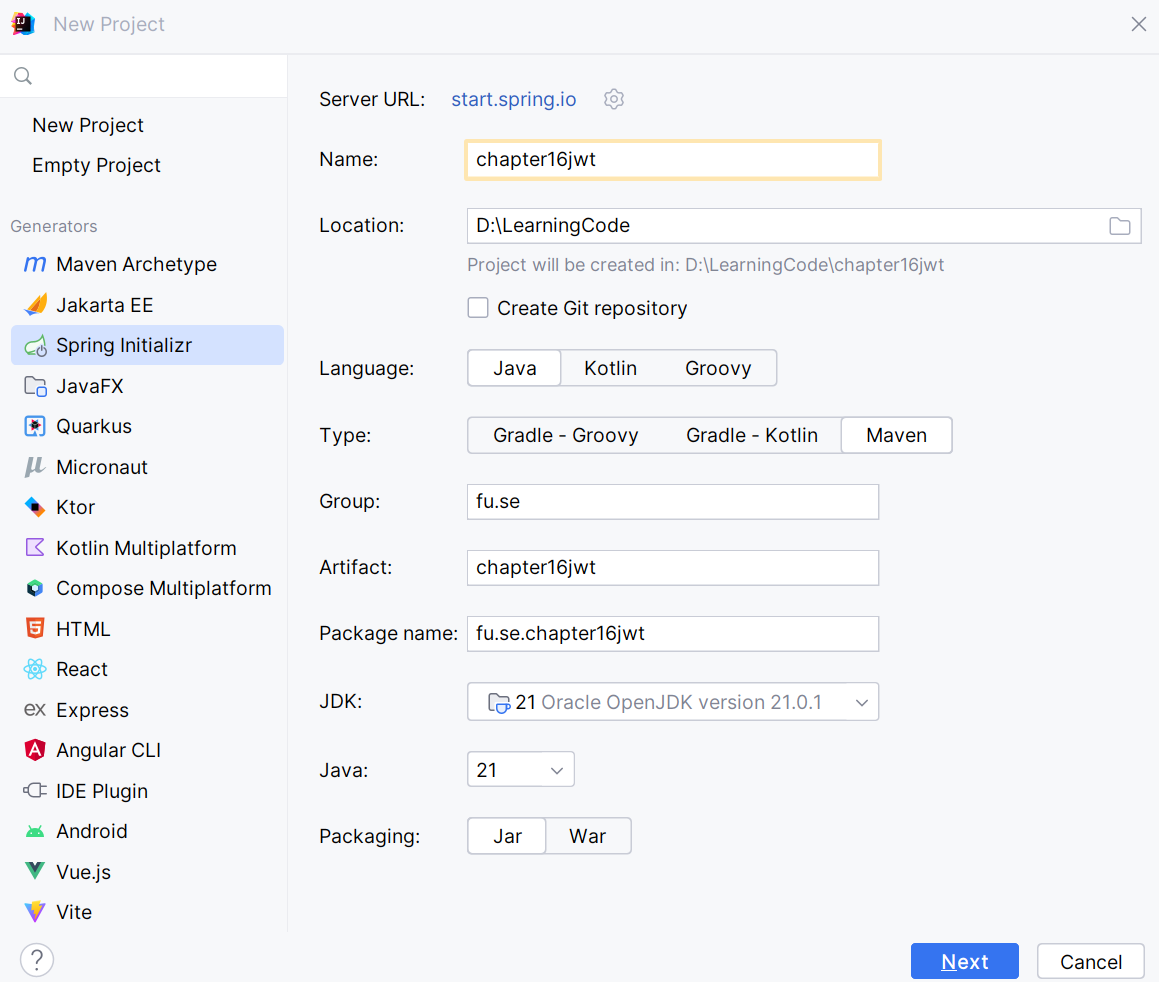
In this lab, you will:

* Use the **Intellij** **IDEA** to create Spring Initializr Project (Maybe you can create the Spring Initializr project in <https://start.spring.io/> then open this project in IntelliJ IDEA).
* Choose **Spring Web** for Building web, including *RESTful*, applications using Spring MVC. Uses Apache Tomcat as the default embedded container. **Spring Data JPA SQL for** persisting data in SQL stores with Java Persistence API using Spring Data and Hibernate. **MS SQL Server Driver SQL -** A JDBC and R2DBC driver that provides access to Microsoft SQL Server and Azure SQL Database from any Java application.
* Use in SQL Server Database as a Data Source
* Develop 3-Layer with Repository Architecture to perform registration and login actions using Spring RESTful API with JWT.
* Run the project and test the application actions.

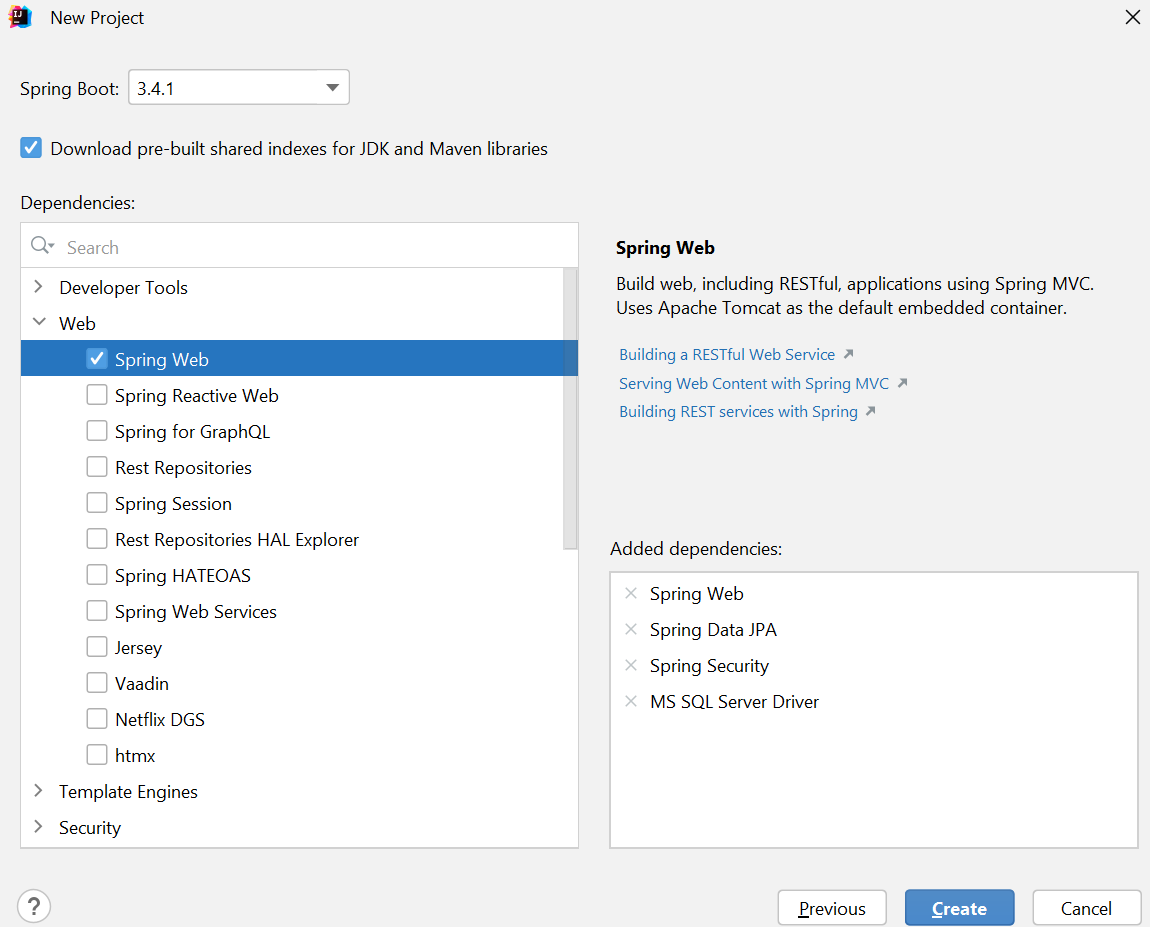
# Activity 01: Design the Registration and Login with JWT

**Step 01**. Open IntelliJ IDEA, File | New | Initializr Project

In the left pane of the New Project wizard, select Spring Initializr Project.

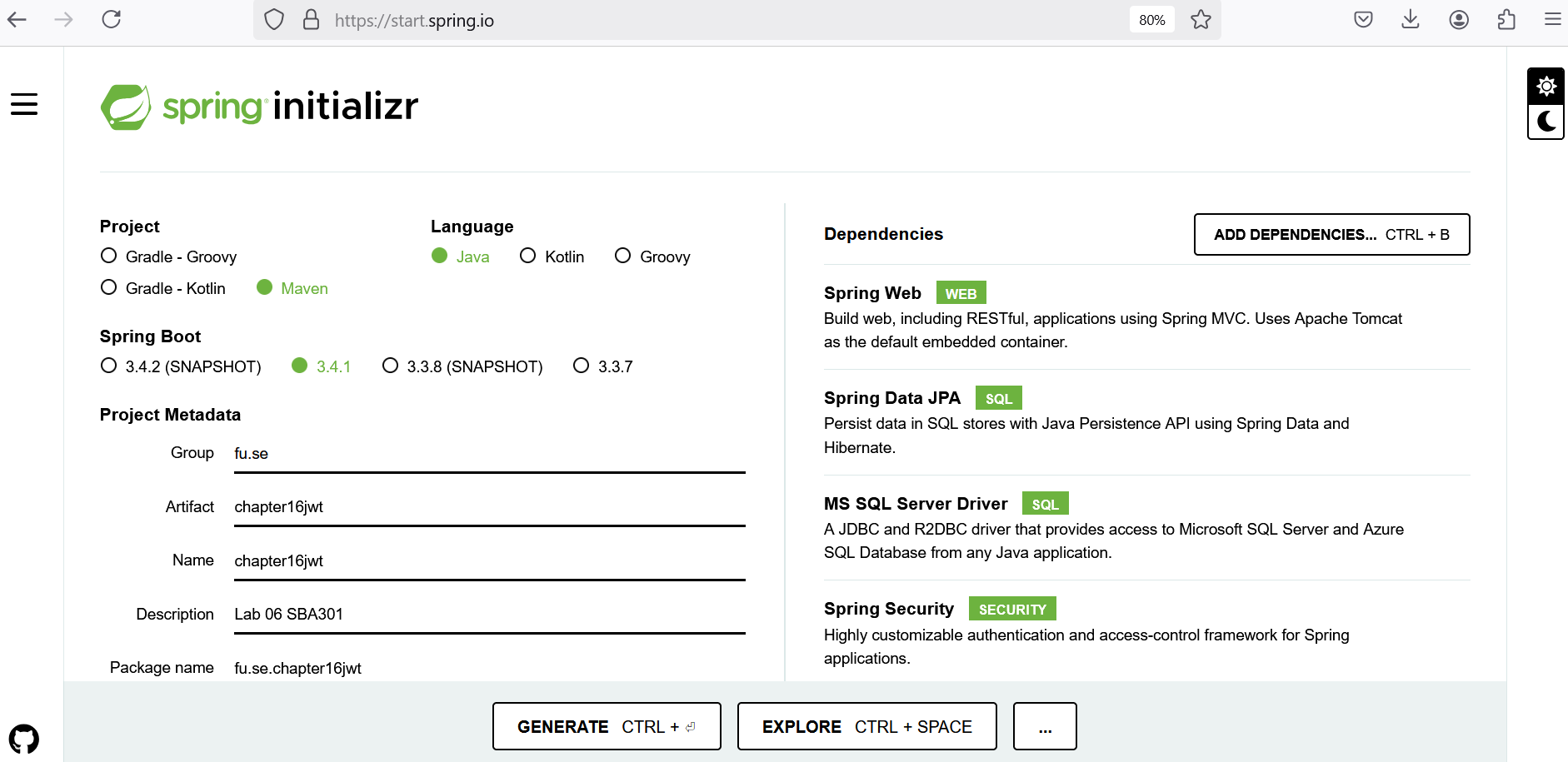


**Step 02.** Fill the information → Next



**Step 03**. Choose the dependency/dependencies → Click Create

Or you can using Spring Initializr project in <https://start.spring.io/> then open this project in IntelliJ IDEA



**Step 04**. Edit pom.xml, add the dependencies for

* MS SQL Server Driver **mssql-jdbc**
* Spring Web - RESTful API: **spring-boot-starter-web**
* Spring Data JPA for Paging and Sort data: **spring-data-jpa**
* Testing **spring-boot-starter-test**

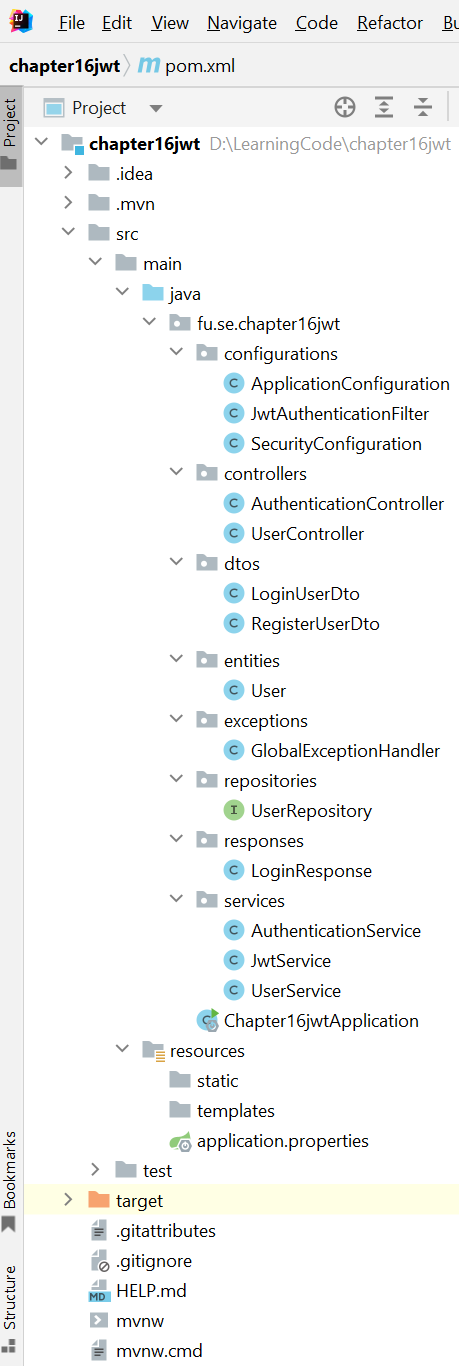
<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>**spring-boot-starter-data-jpa**</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>**spring-boot-starter-web<**/artifactId>  
 </dependency>  
 <dependency>  
 <groupId>com.microsoft.sqlserver</groupId>  
 <artifactId>**mssql-jdbc<**/artifactId>  
 <scope>runtime</scope>  
 </dependency>

<dependency>  
 <groupId>org.springframework.security</groupId>  
 <artifactId>**spring-security-test**</artifactId>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>**spring-boot-starter-test**</artifactId>  
 <scope>test</scope>  
 </dependency>  
</dependencies>

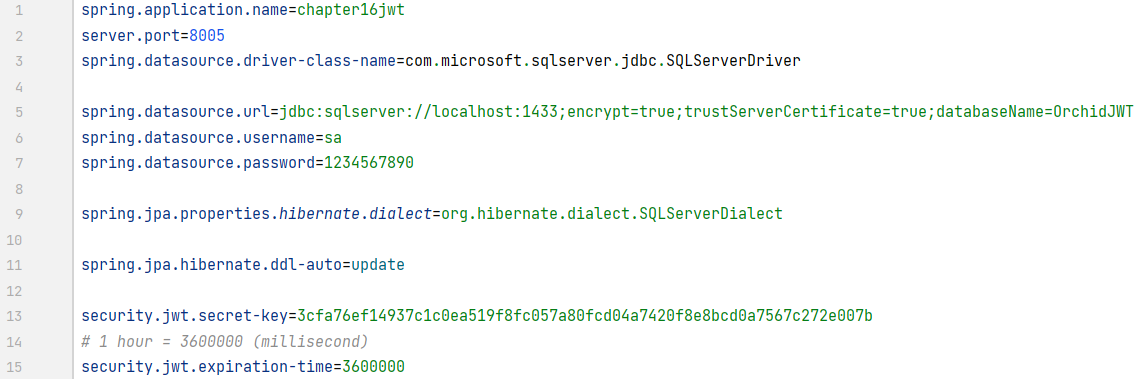
*Add JWT dependecies*

****

**Step 05**. The structure of the Lab project

****

**Step 06**. The application.properties file



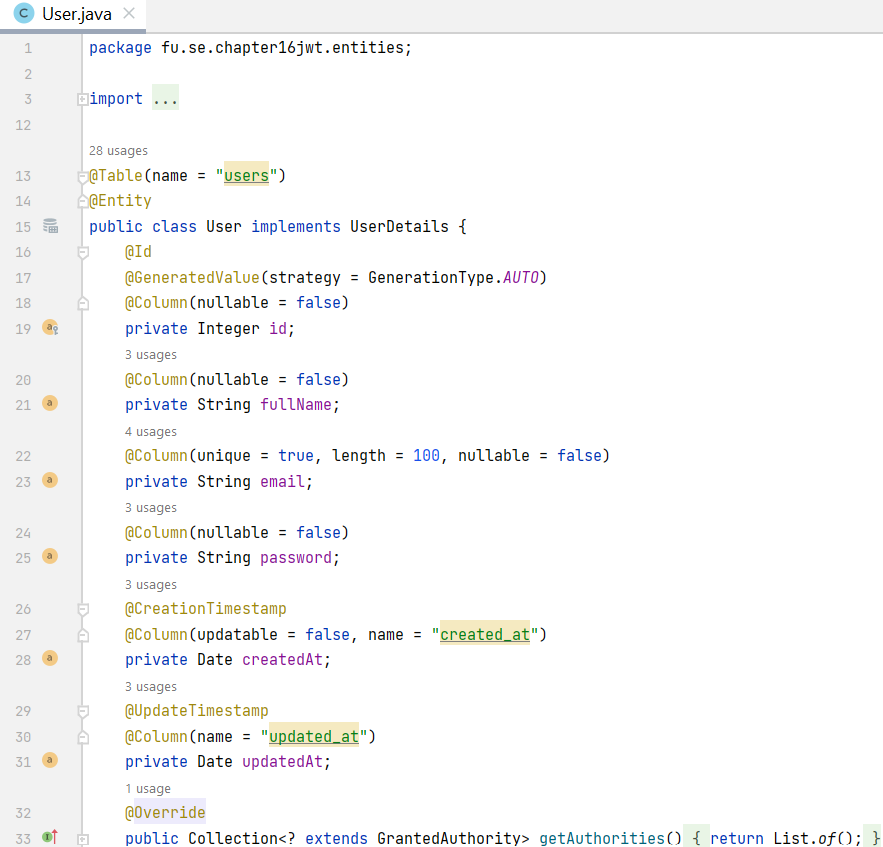
# Activity 02: Implement RESTful API with JWT for registration and login functions

**Step 01**. Create **entities** package, then create User class

* Create the user entity. *Then, extend the User Entity with authentication details.*
* To manage user details related to authentication, Spring Security provides an interface named **“UserDetails”** with properties and methods that the User entity must override the implementation.
* The method “**getAuthorities()**” returns the user’s roles list; it is helpful to manage permissions.
* The method “**getUsername()**” returns the email address because it is unique information about the user.

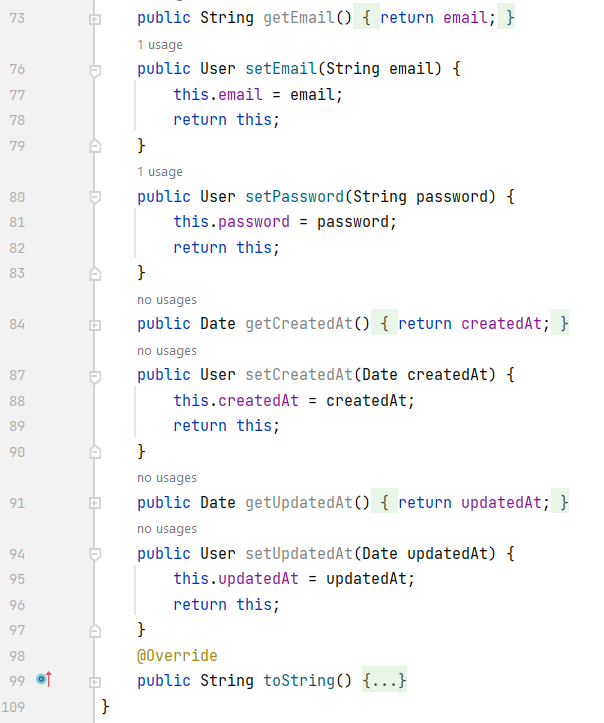
Annotations:

* @Entity: Marks the class as a JPA entity
* @Id: Specifies the primary key field
* @GeneratedValue: Strategy for generating primary key values
* @Column: Maps a field to a database column



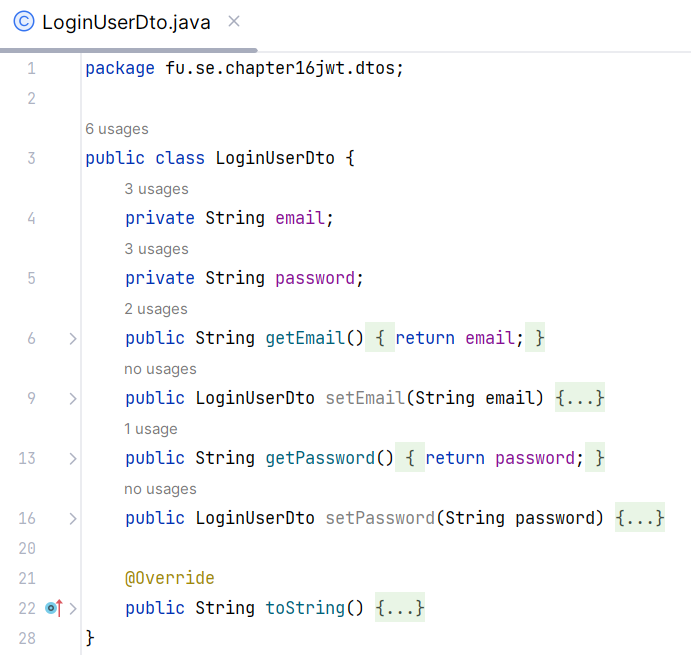


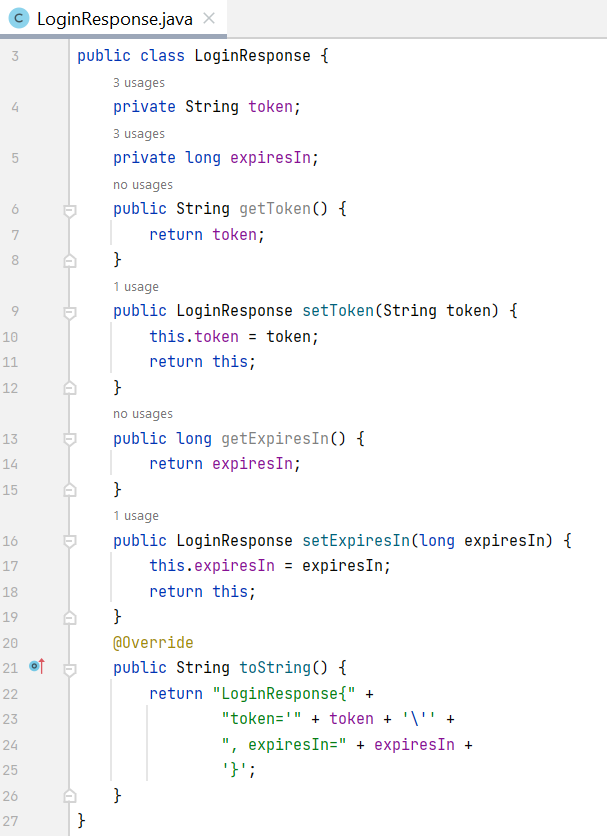




**Step 02**. Create a DTO and response data classes

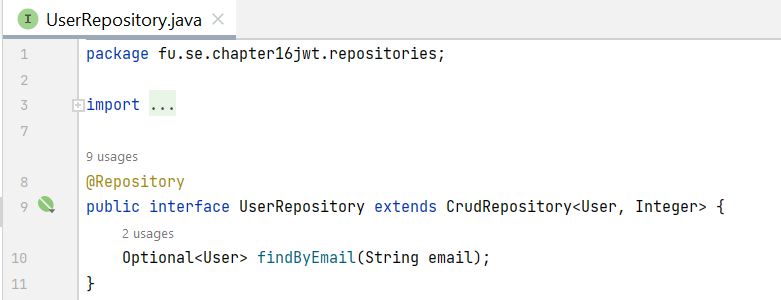






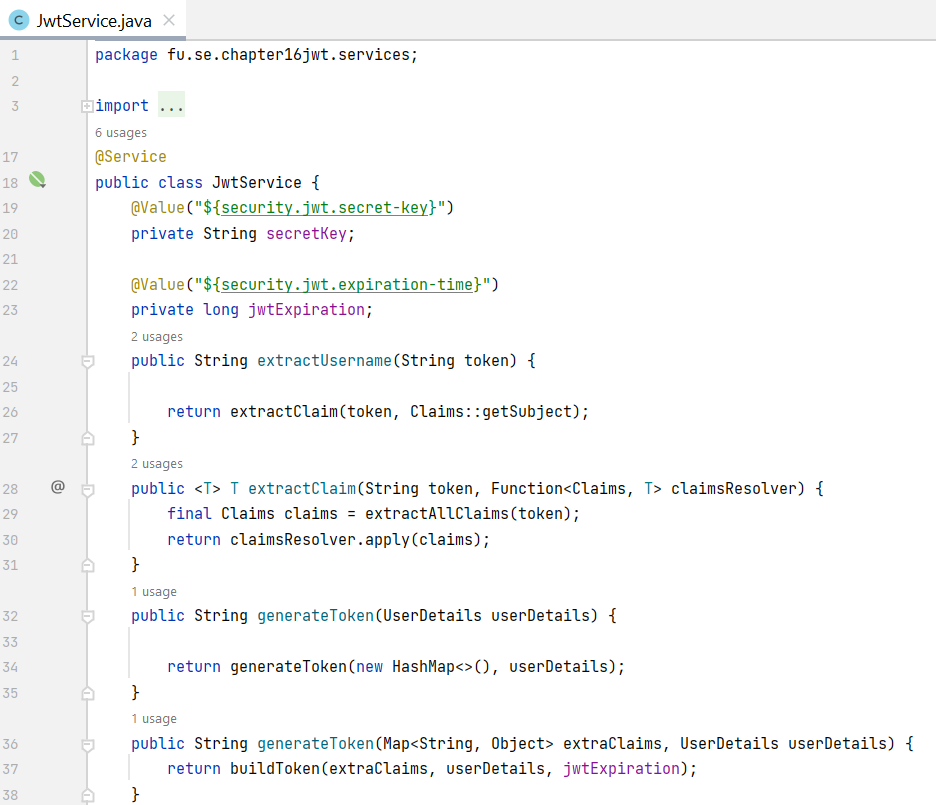
**Step 03.** Implement Repository by creating **repositories** package, UserRepository interface

* Create JPA repository interface, this interface extending CrudRepository
* Provides CRUD operations and more
* Just implement the findByEmail(String email) function



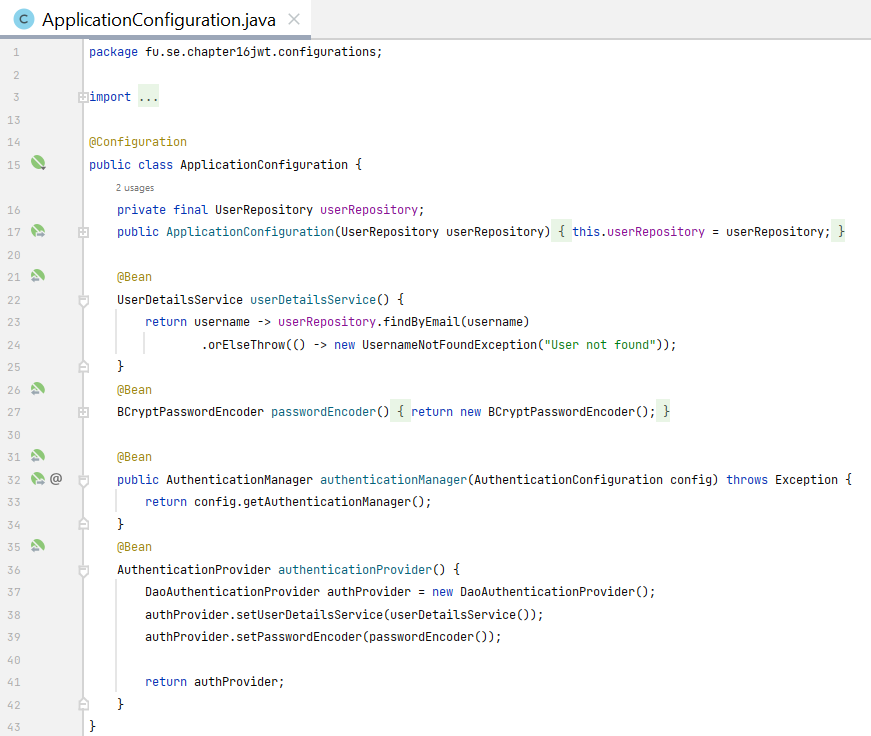
**Step 04**. Create related JWT classes

* Generate JWTs: Use io.jsonwebtoken.Jwts to create tokens with claims, expiration, and signing.
* Validate JWTs: Verify signature and extract claims.

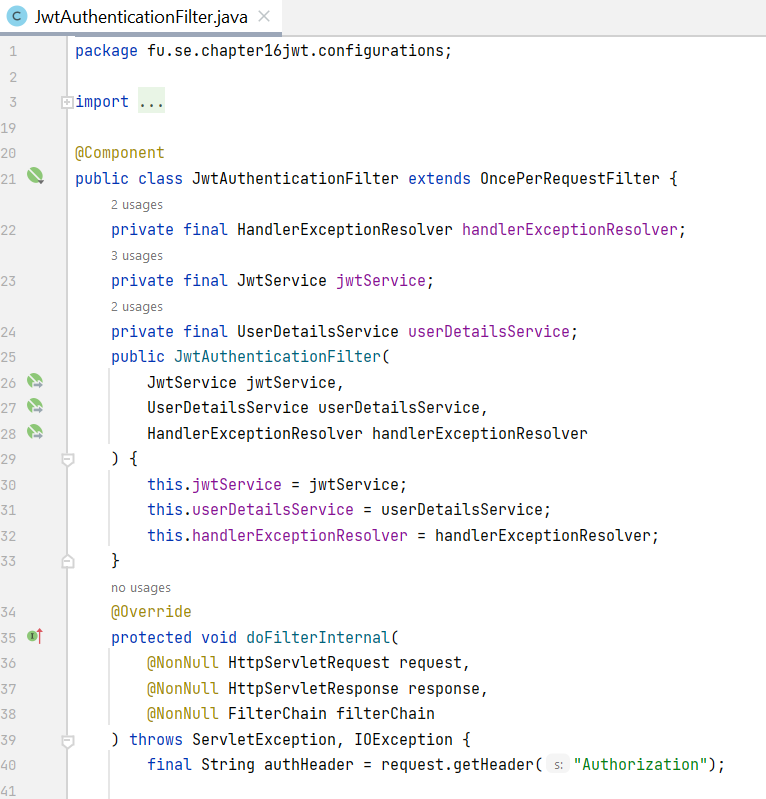


Override the security configuration

* *userDetailsService()* defines how to retrieve the user using the UserRepository that is injected.
* *passwordEncoder()* uses to encode the plain user password.
* *authenticationProvider()* sets the new strategy to perform the authentication.

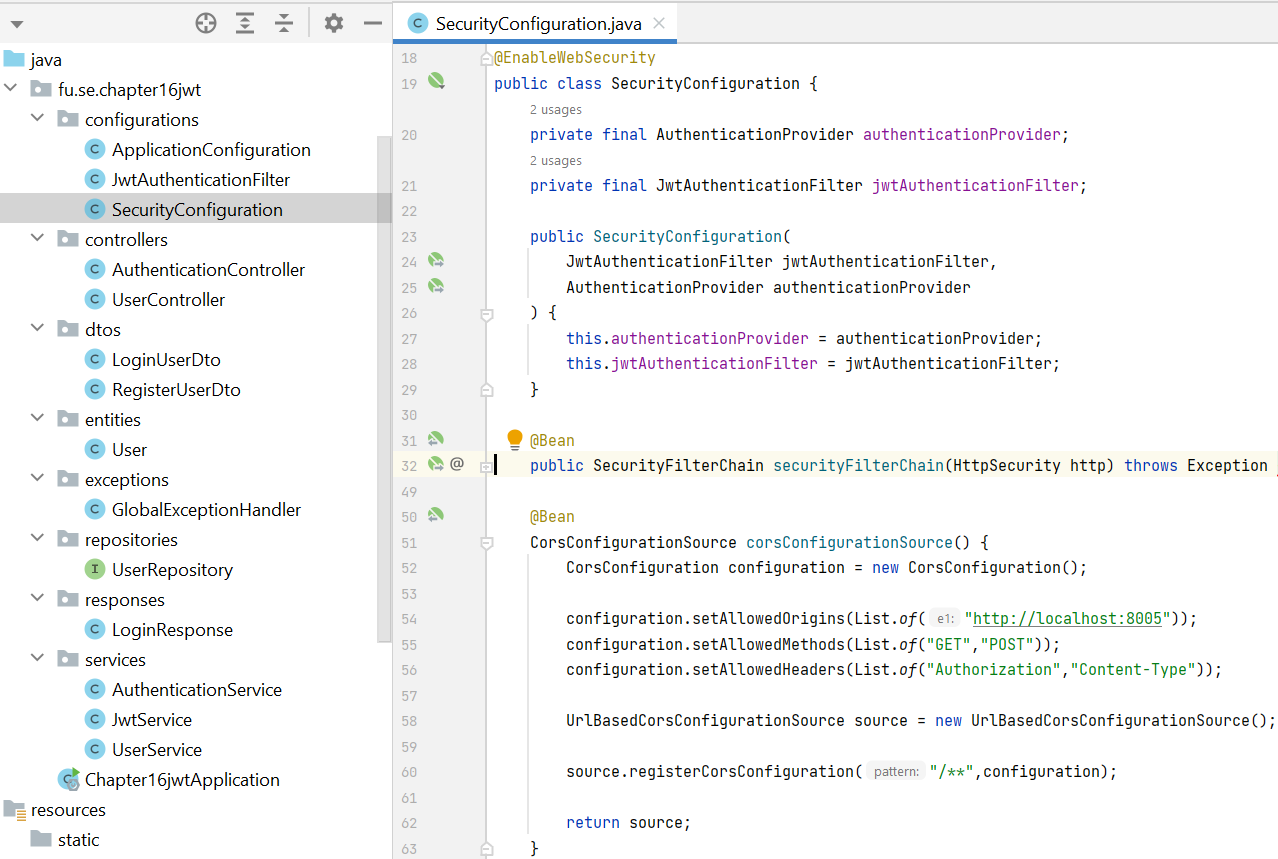


**Step 05**. Configure Spring Security

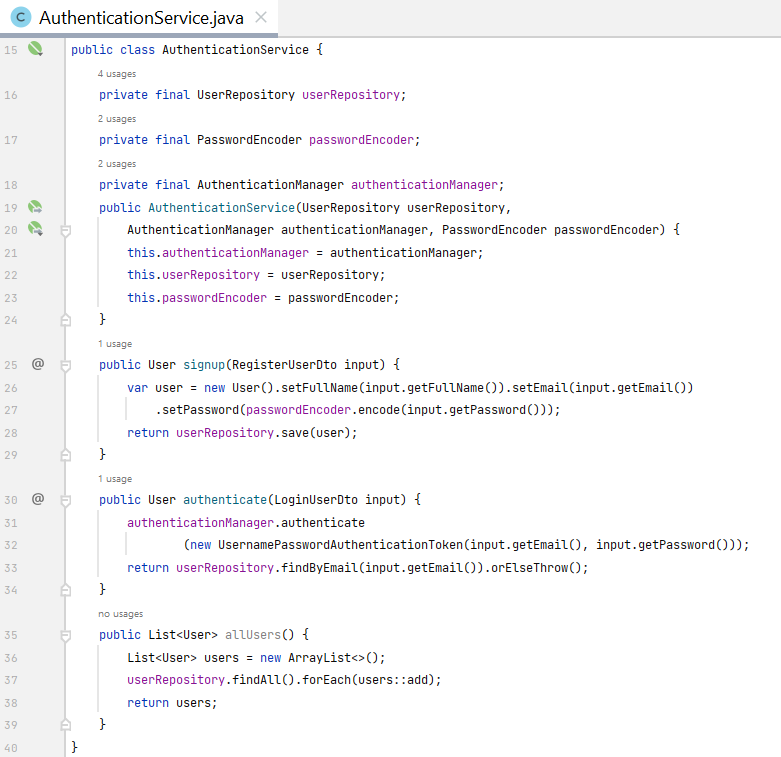




*Configure the requester filter*

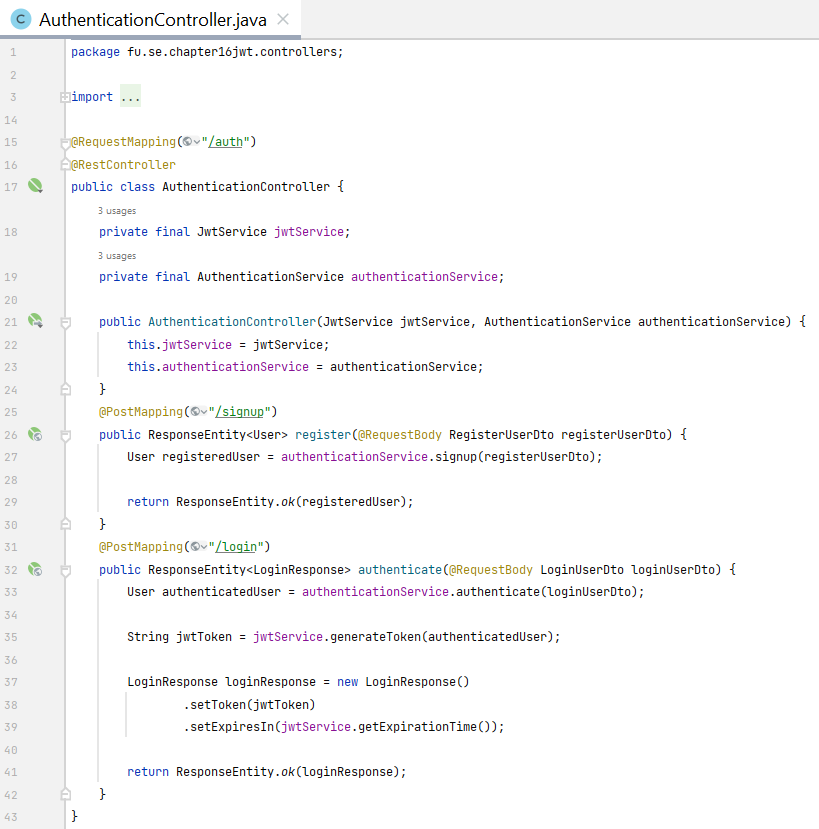


*Create the authentication service*

**

**Step 06**. Create an authentication controller

* Handle login requests.
* Authenticate the user.
* Generate JWT and send it in the response.



@RequestMapping - Maps HTTP requests to specific controller methods based on the request path, HTTP method, and other criteria. It can be used at both the class and method level. Attributes of @RequestMapping:

* value: Specifies the request path.
* method: Specifies the HTTP method (e.g., RequestMethod.GET, RequestMethod.POST).
* params: Specifies request parameters that must be present.
* headers: Specifies request headers that must be present.
* consumes: Specifies the content type of the request body.
* produces: Specifies the content type of the response body.

The annotations are shortcuts for @RequestMapping with specific HTTP methods. They simplify the code and improve readability.

* @GetMapping
* @PostMapping
* @PutMapping
* @DeleteMapping
* @PatchMapping

@PathVariable: Extracts values from URI path variables and binds them to method parameters.

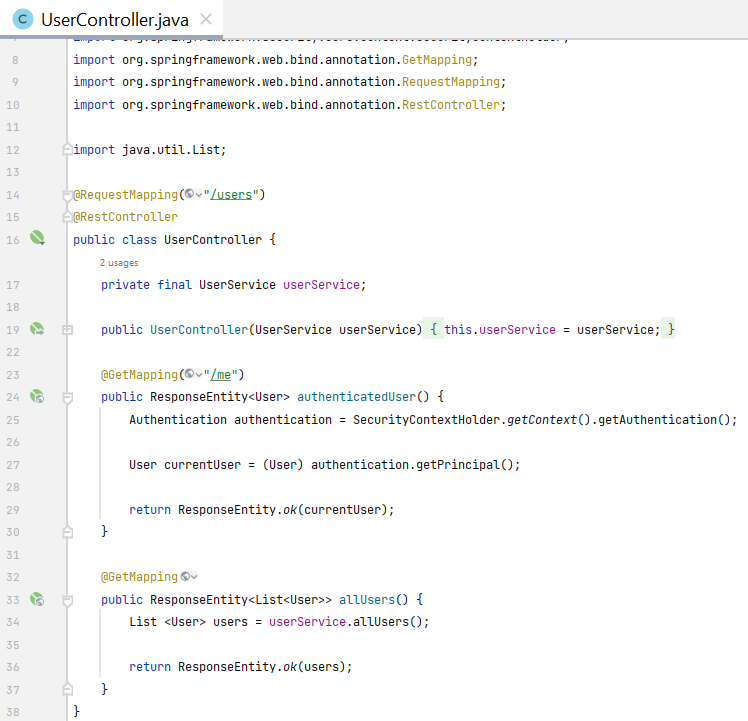
@RequestParam - Extracts values from query parameters and binds them to method parameters.

@RequestParam attributes:

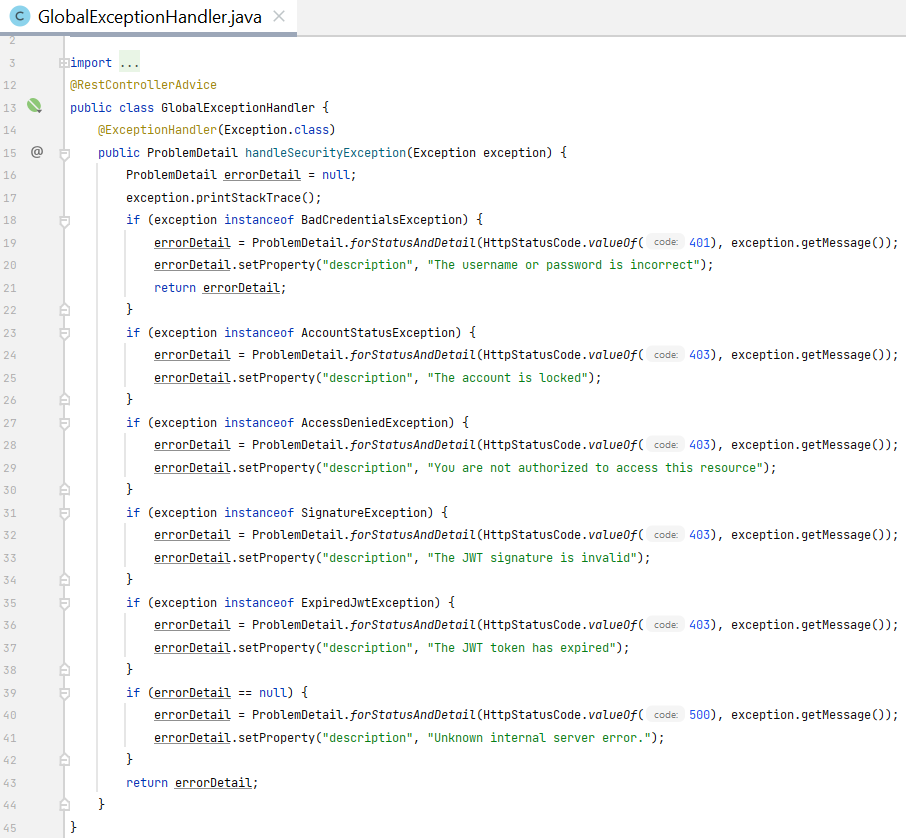
* value: Specifies the name of the query parameter.
* required: Indicates whether the parameter is required.
* defaultValue: Specifies a default value if the parameter is not present.

**Step 07**. Secure endpoints



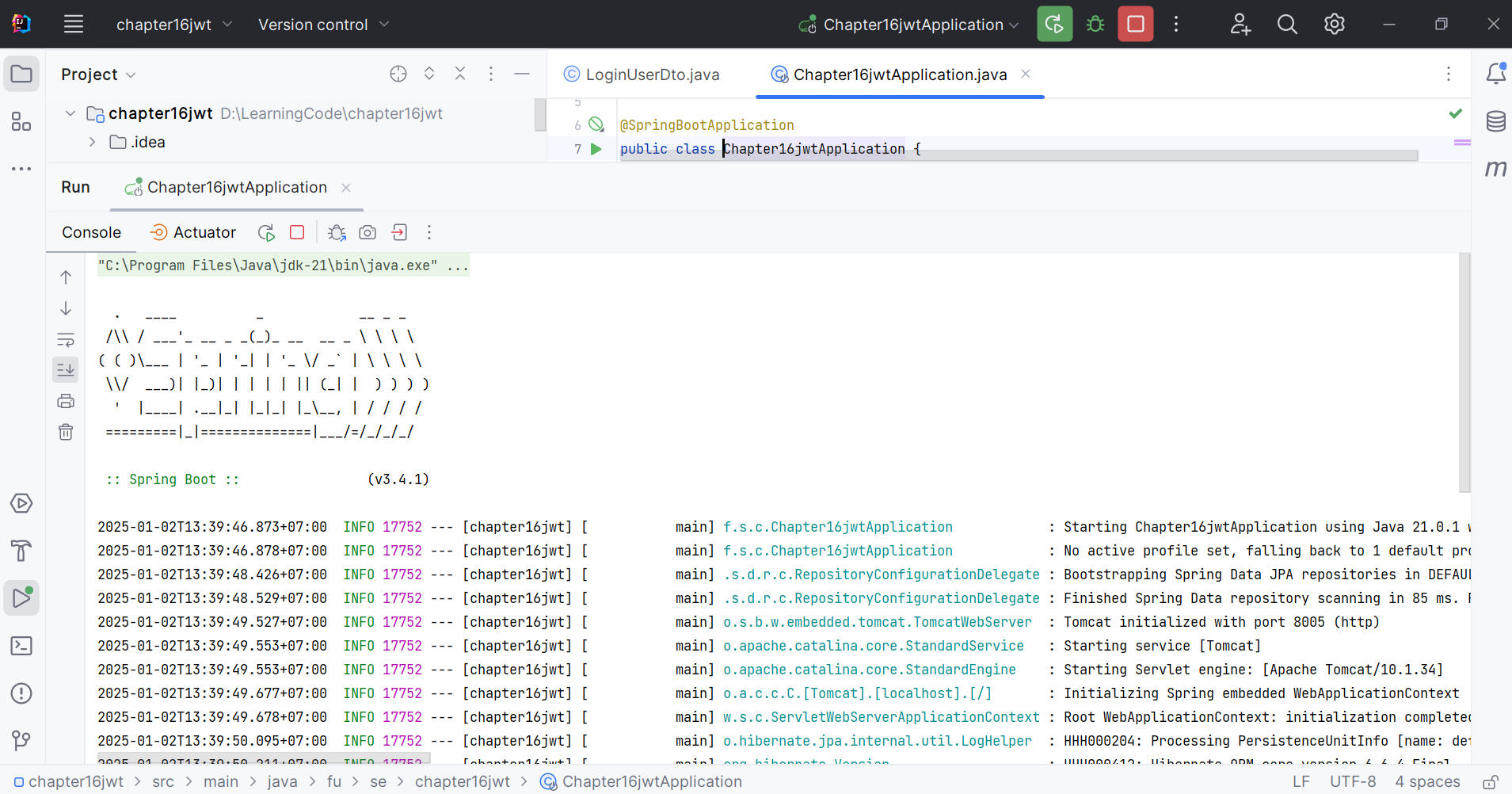


**Step 08**. Customize authentication error messages



# Activity 03: Run and test the RESTful API with JWT

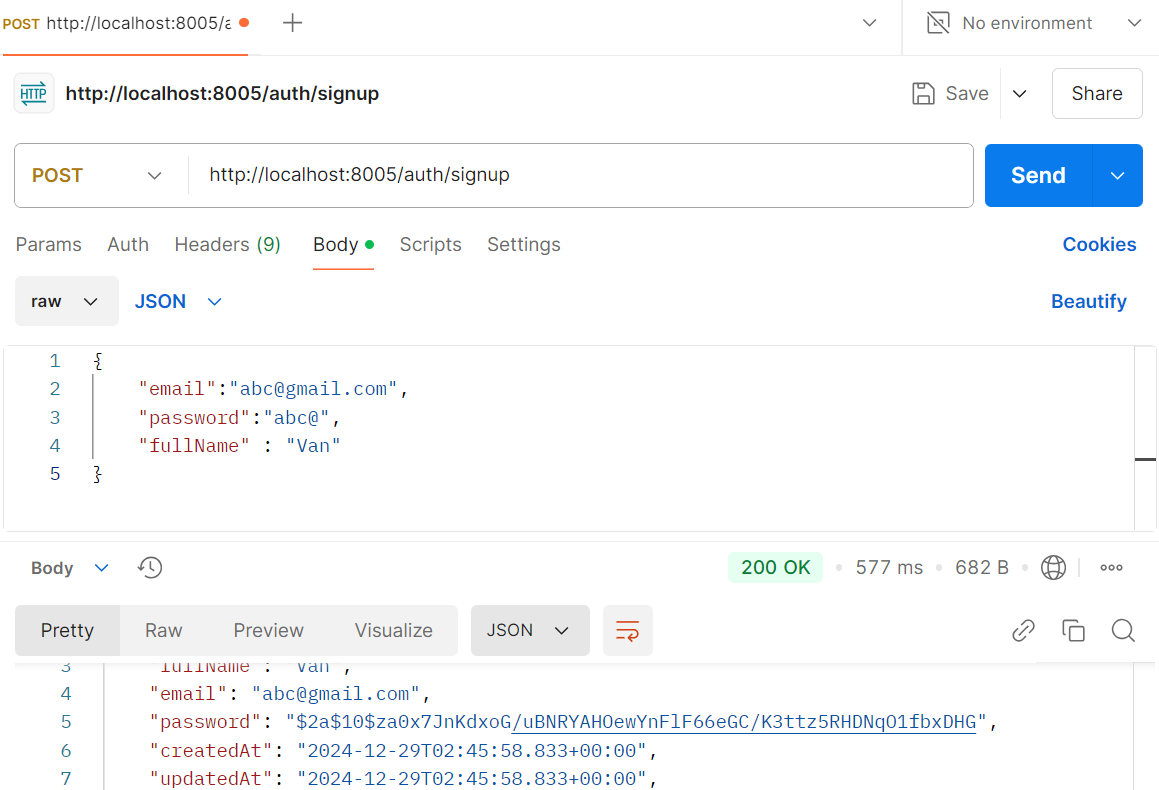
**Step 01.** Run the RESTful API Project



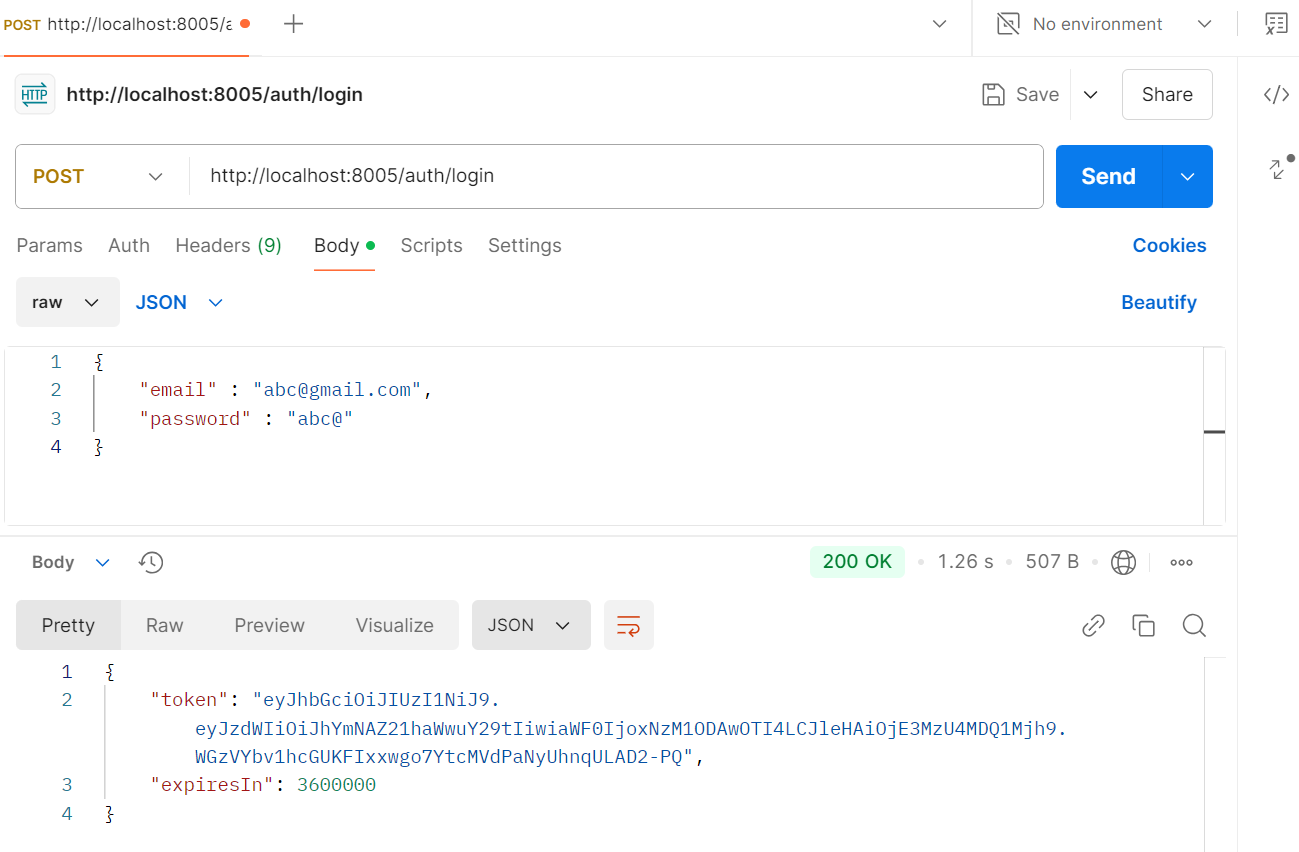
**Step 02.** Test the RESTful API using Postman

|  |  |  |
| --- | --- | --- |
|  | HTTP Method | Path |
| 1 | POST | *http://localhost:8005/*auth/signup |
| 2 | POST | *http://localhost:8005/*auth/login |
| 3 | GET | *http://localhost:8005/*users/me |
| 4 | GET | *http://localhost:8005/*users |

**Step 03.** Test registration function using the POST method



**Step 03.** Test login function using the POST method



**Step 04.** Test user detail function using the GET method

