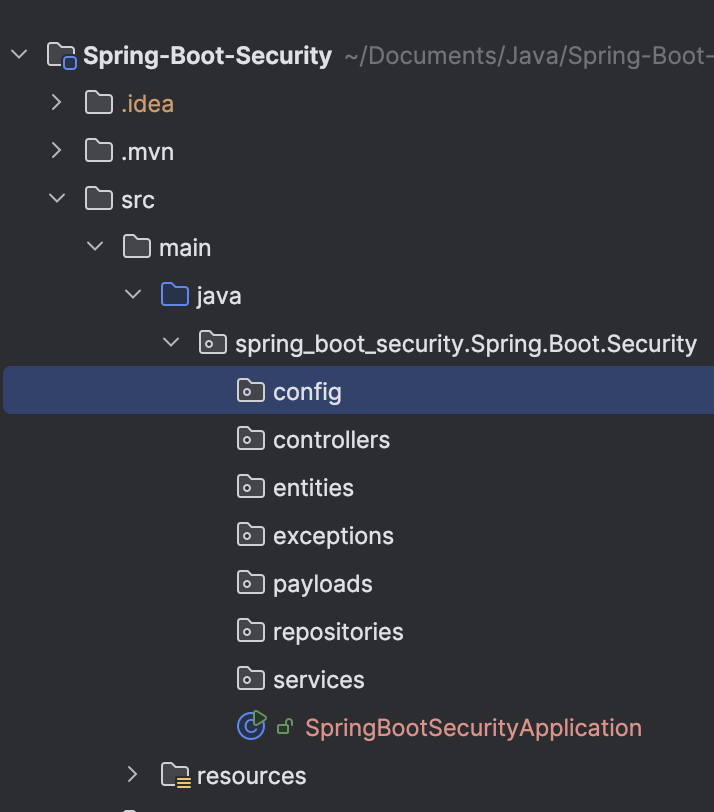
## **Introduction**

1. spring.io dependencies
   * Spring web
   * Spring security
   * Spring boot dev tools
   * Lombok
   * Postgres

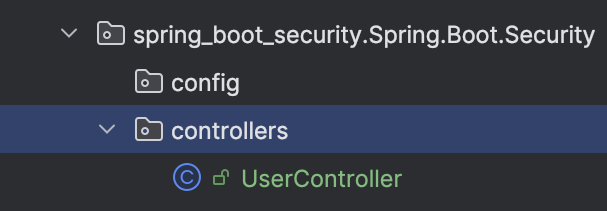
## **Create project structure**

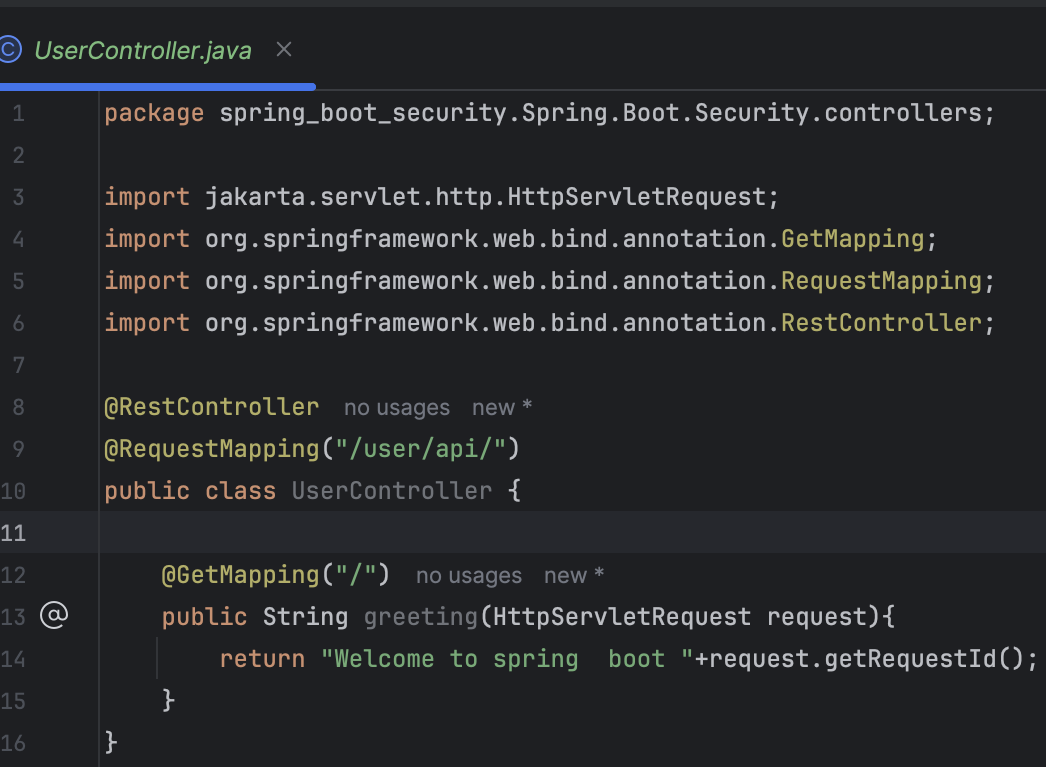
1. Basic project structure



## **Test spring boot project**

1. Create dummy controller
   * Use annotations RestController for now
   * Use GetMapping
   * And return word for now “Welcome to Spring Boot”





1. Now run the project and test it
2. By default, it will show a login UI provided by Spring Boot

But if you comment out the Spring Boot security dependency, it will show the default page, which means our controller is not being guarded

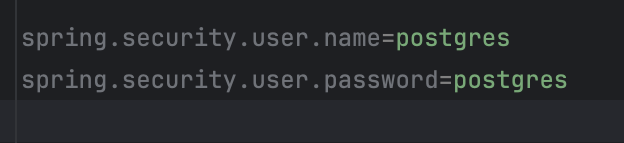
1. **Password** will appear on console, **username** = user
2. Now run the project and test it



Note: we will not see this page if we disable security dependency on **pom.xml**

## **Change username and password**

1. Go to src>main>java>resources>application.properties



## **How are we seeing the login page when we request for the default page ?**

**Request Processing Flow:**

* When you access a controller in a web application, your request typically goes through:
  + **Filter Chain:** Filters intercept the request before it reaches the controller. These filters can perform tasks like authentication, logging, or request modification.
  + **Front Controller:** In frameworks like Spring MVC, the **Front Controller** is typically a DispatcherServlet. It serves as the central entry point for all incoming requests, delegating them to the appropriate controller.
* After the request is processed by the controller, the response follows the reverse path:
  + The response is processed by any filters (in reverse order) before being sent back to the client.

**Authentication and Validation:**

* If authentication or validation (e.g., checking the email and password) fails, the application (or filter) sends an appropriate response, such as an error message or a redirection to a login page. This can happen:
  + **In Filters:** If the filters are responsible for authentication, they might stop the request before it even reaches the controller.
  + **In the Controller:** If authentication or validation is done at the controller level, the controller will send the appropriate error message.

**Servlet and Tomcat:**

* These processes (filters, front controllers, and controllers) run within the **Servlet Container**, which is part of the **Tomcat Server**.
* Tomcat provides the runtime environment for Servlets and JSPs, handling tasks like managing threads, handling HTTP requests, and sending responses.

## **How do we change the way security filter chain works**

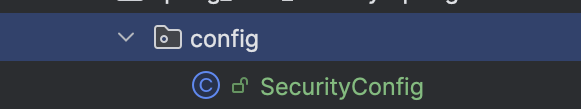
1. By default spring security provide us a lot of chain
2. But how to configure in our own way

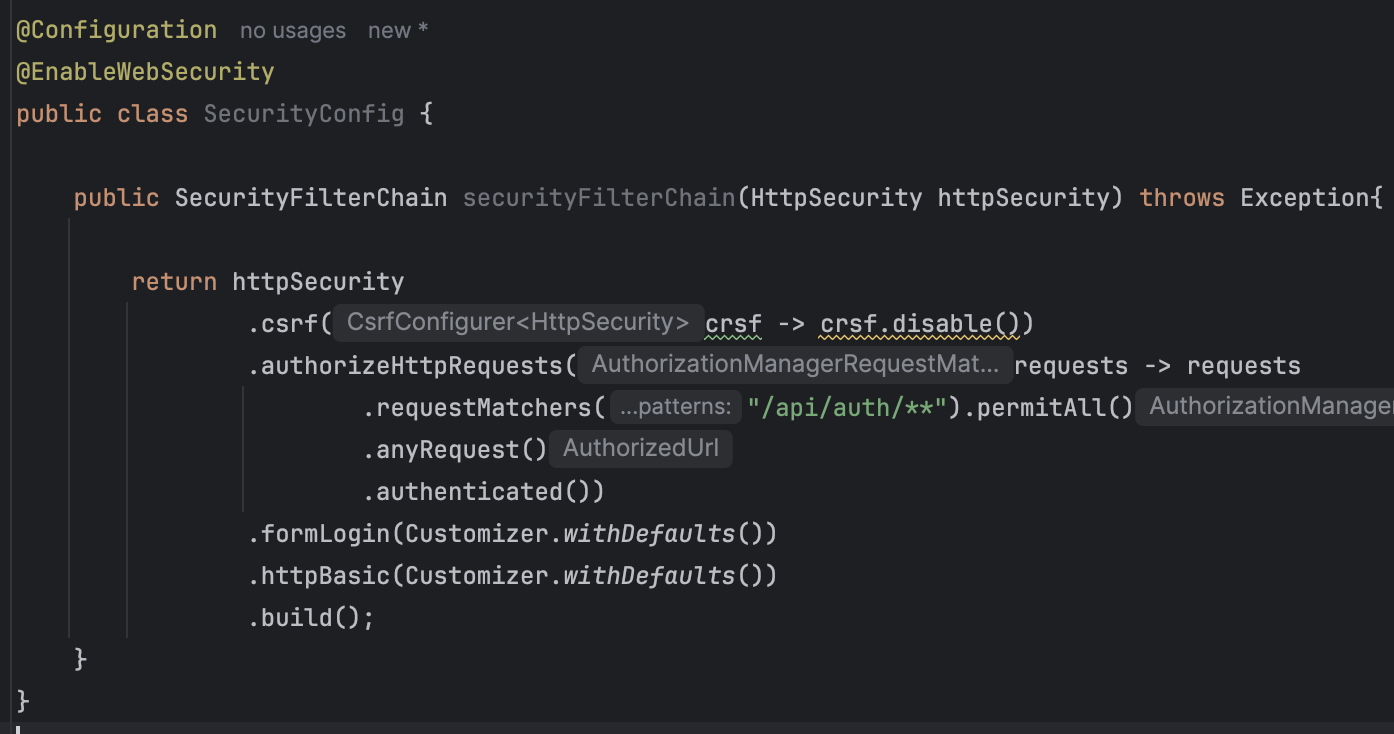
We can do that creating **config class**

* + Create SecurityConfig class

src > main > java >package > new package (name it as SecurityConfig)

Then create config class on config package





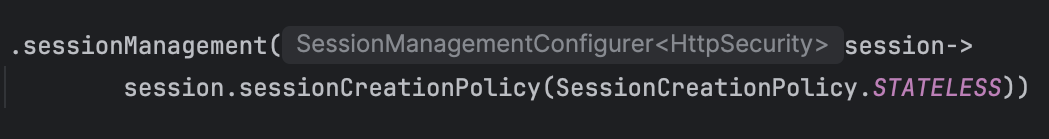
* + Annocations

Using this with config class mean

**@Configuration** => Tells Spring that this is a configuration class.

**@EnableWebSecurity** => Disables the default security configuration, allowing you to define custom security rules

* + Session less (it will create session on every request)



## **Who is working behind the scene for validating username and password (NOT USED IN REAL FOR TESTING PURPOSE ONLY)**

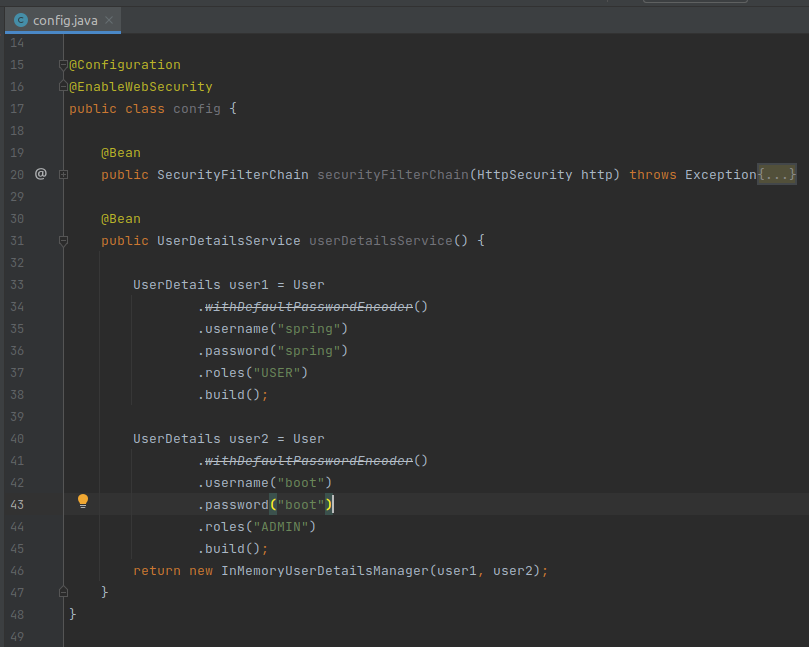
1. UsernamePasswordAuthenticationFilter

Takes the credentials and starts the authentication process

1. UserDetailsService

Helper to load user details for example fetching from database

1. In config file
   * Use Bean, because it will be in the spring container, and your spring security will pick it from there.
   * Create bean of UserDetailService



1. For demo we have create two user detail (but in real we get usually don't do that)
   * Now we can test it from POSTMAN

## **Refactor and connect with database**

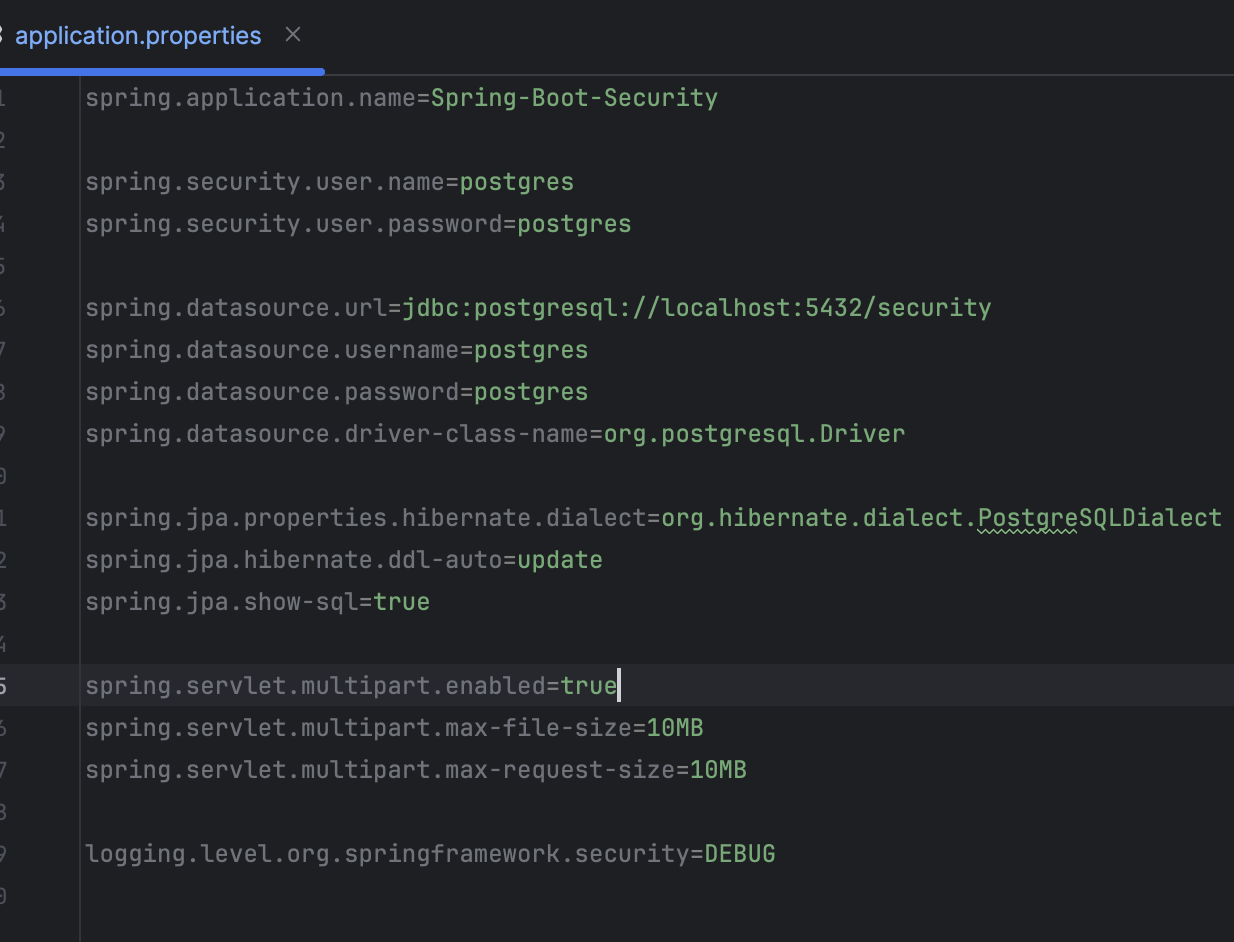
1. When we pass a credential object (**Authentication Object,** still not authenticated) to the **Authentication Provider** (who will provide the service of checking it and validating it), it will provide an **Authentication Objec**t (Authenticated object).

**NOTE :-** Controllers use services to handle business logic. Services interact with repositories to fetch or manipulate data, and repositories communicate with the database to perform these operations.

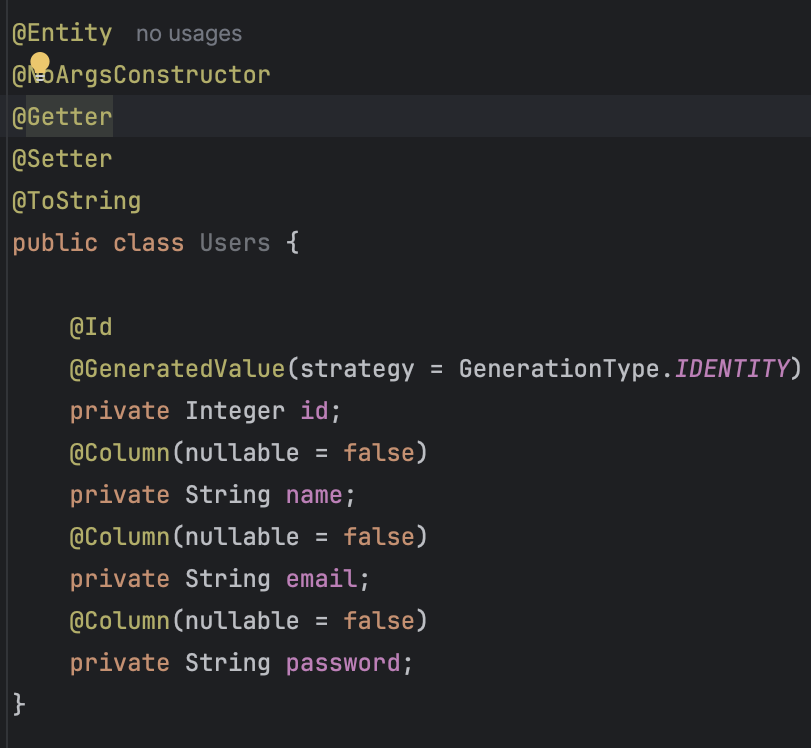
1. Add two dependencies
   * JPA
   * Postgresql
   * ModelMapper
   * Jakarta
   * Validation



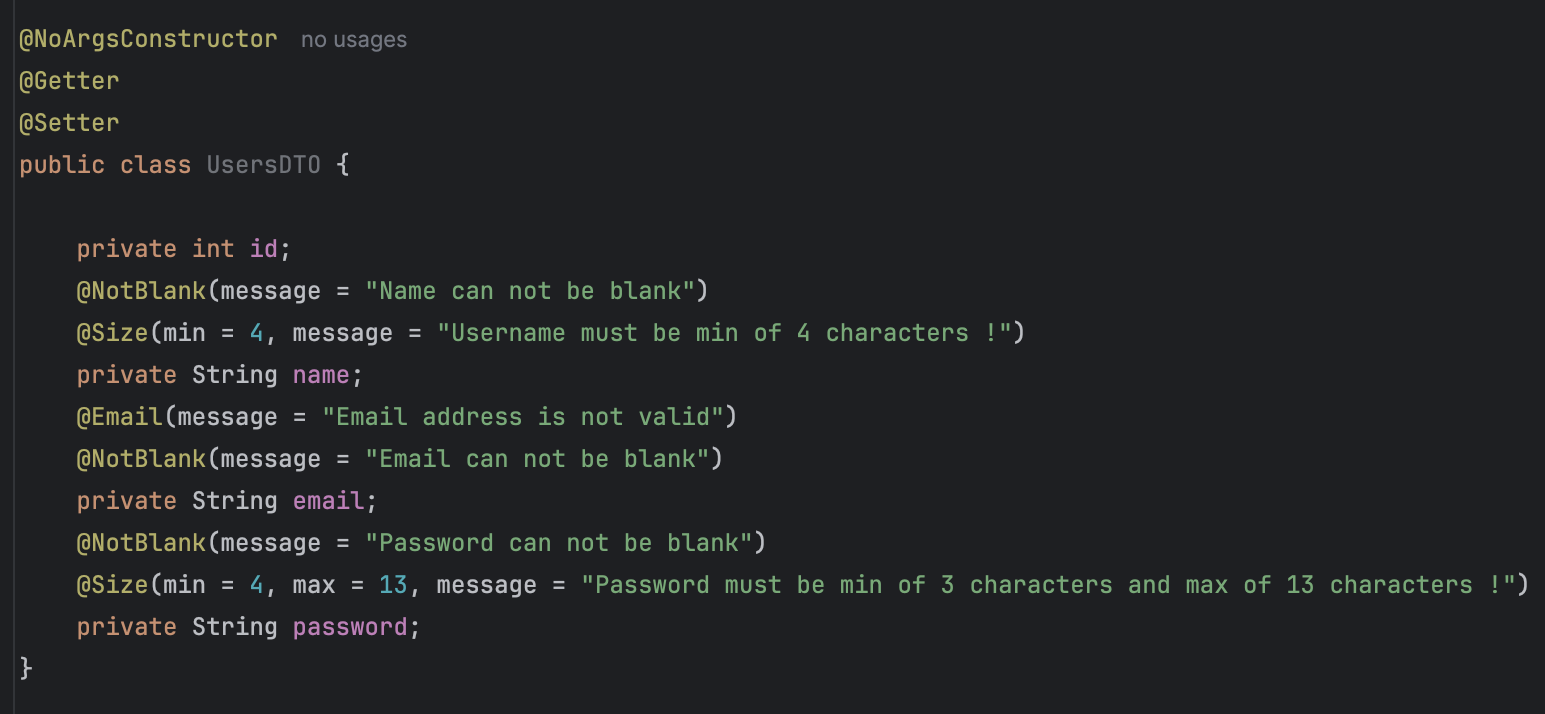
1. Create database
2. Update application.properties



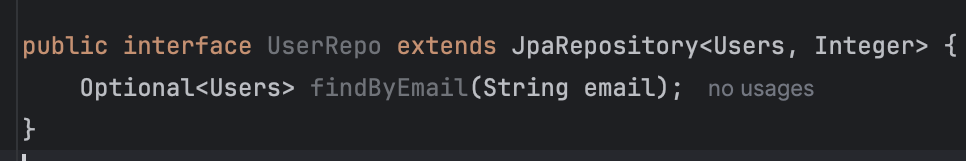
1. Create Users entity inside the entities package (**NOTE** : set email as unique = true)



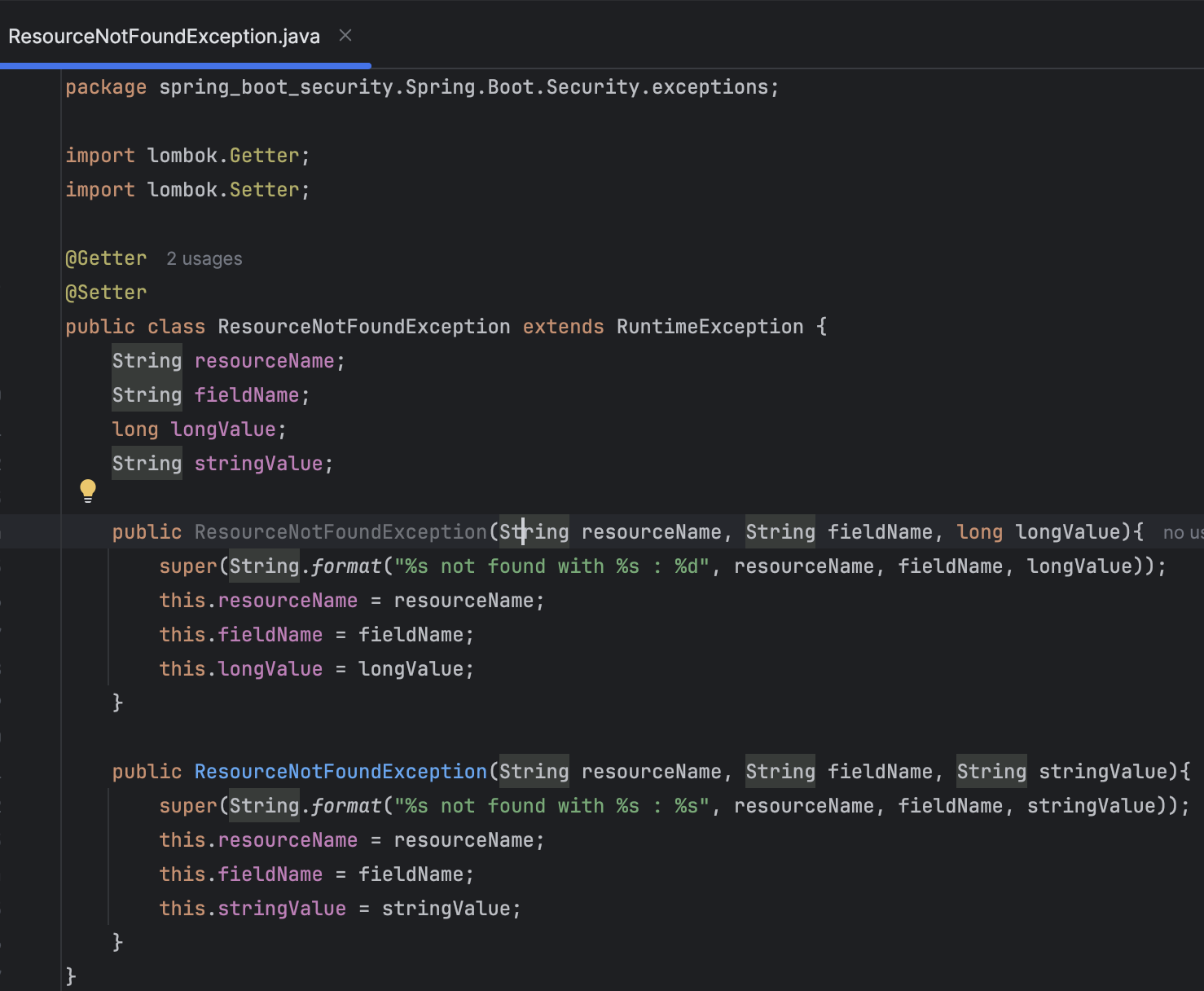
1. Create Users DTO inside the dtos package



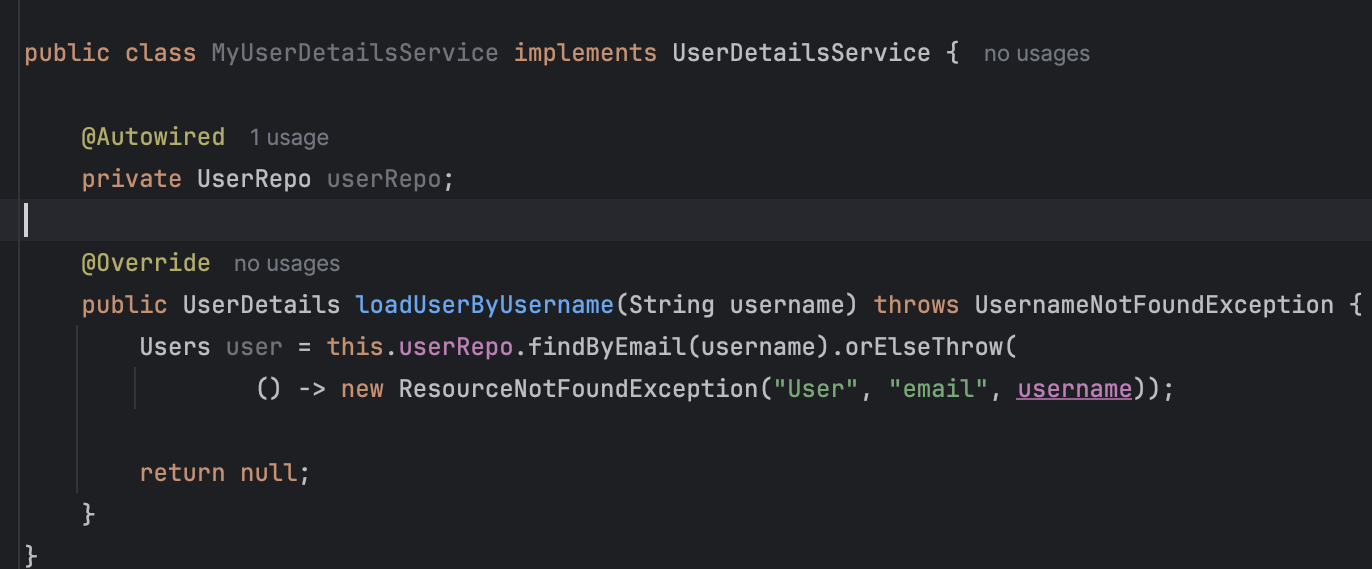
1. Create “useRepo” file (interface) inside the repo package



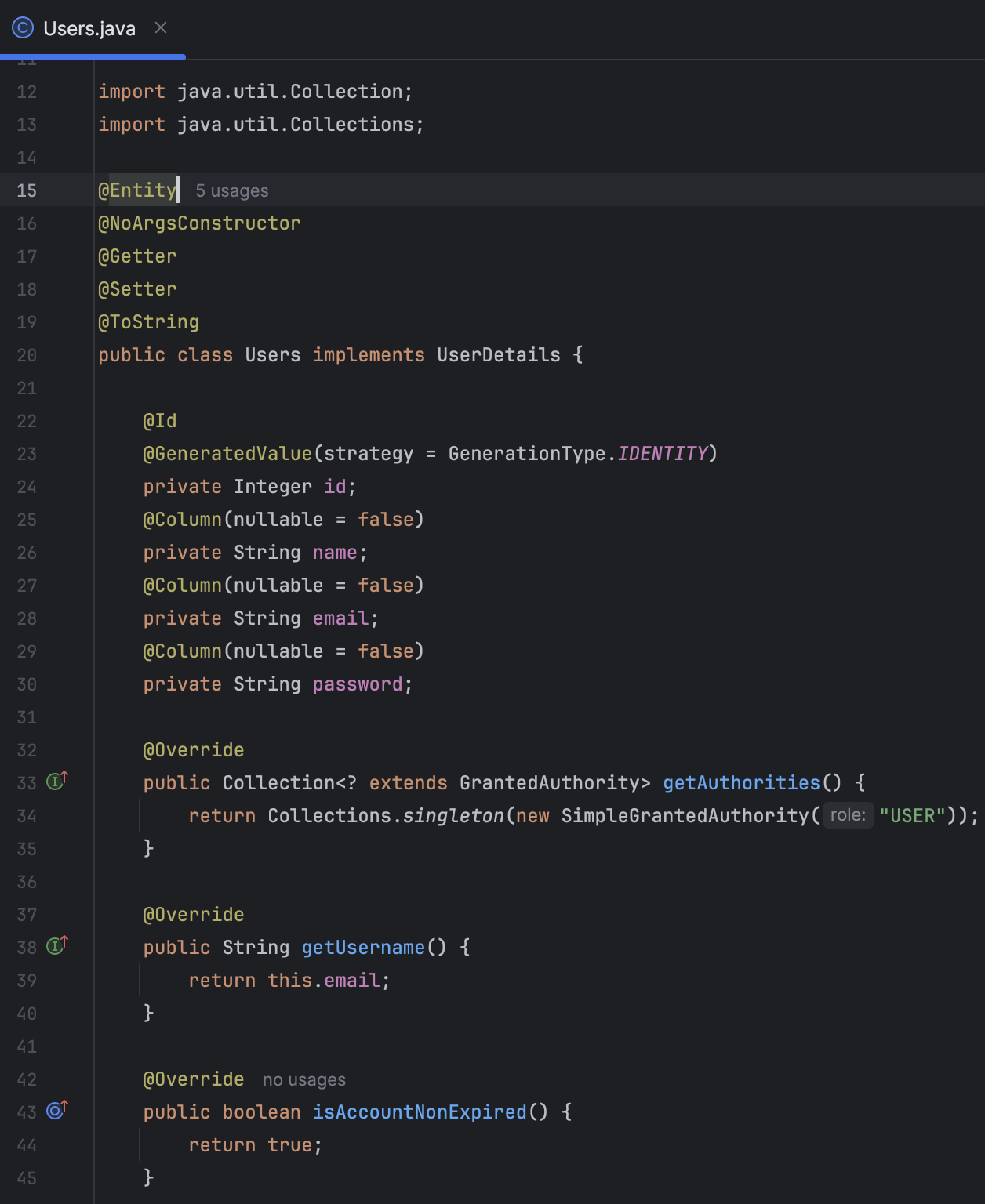
1. Inside the exceptions package create ResourceNotFound exception package to handle global exception



1. Inside the service package create one service that implements UserDetailsService (**NOTE** : use @Service annotation).

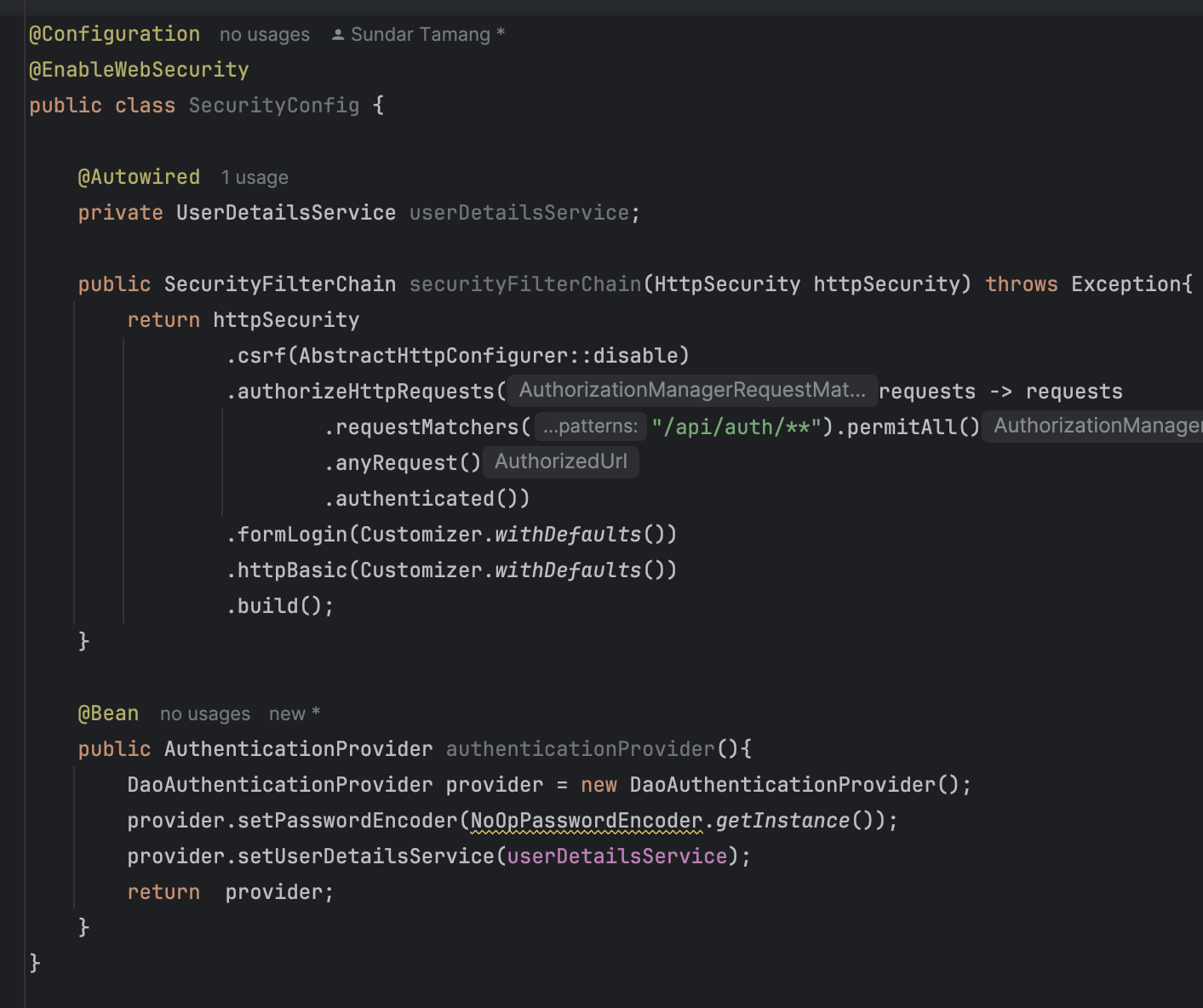


1. Now go back to user entities and implement UserDetails on Users entity





1. Go to SecurityConfig then update code



## **What have we done so far ?**

1. We have added two dependencies JPA, postgres, validation, jakarta, model mapper
2. Then updated application properties
3. Then configured security config (**config package > SecurityConfig**)
   * By default it was using a authentication provider,
   * But we have used out DaoAuthenticationProvider

Here we have to pass one password encoder and UserDetailsService

1. Since we want our own we have to create out own class (**service package > MyUserDetailsService**)
   * To work with we have to implement one method which is **loadUserByUsername**
   * To work with we have to work with repo (repo package > UserRepo)

Repo will do the database connectivity

* + **loadUserByUsername** return user object so created UserPrinciple (which will give us current user details)