**SPRING SECURITY:**

-> Once the dependency is added, Spring Boot's auto-configuration feature will automatically apply security to the application.

(That means to access every endpoint we will need the username and password).

-> By default, Spring Security uses HTTP Basic Authentication.

(The client sends an Authorization header Authorization: Basic<encoded-string>

The server decodes the string, extracts the username and password, and verifies them.

If they're correct, access is granted. Otherwise, an "Unauthorized" response is sent back!)

**ENCODING:**

Credentials are combined into a string like:

Username:

Password:

Which is then encoded using Base64

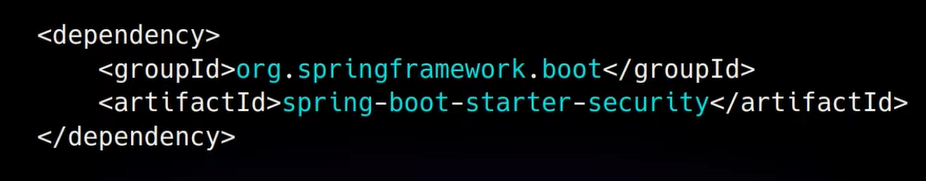
-> By default, all endpoints will be secured. Spring Security will generate a default user with a random password that is printed in the console logs on startup.

-> You can also configure username & password in your application.properties

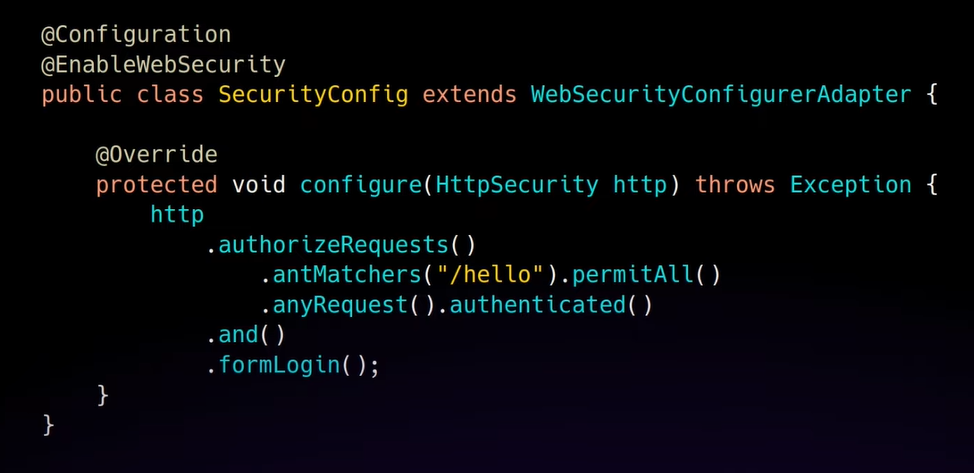
-> spring.security.user.name = user

-> spring.security.user.password = password

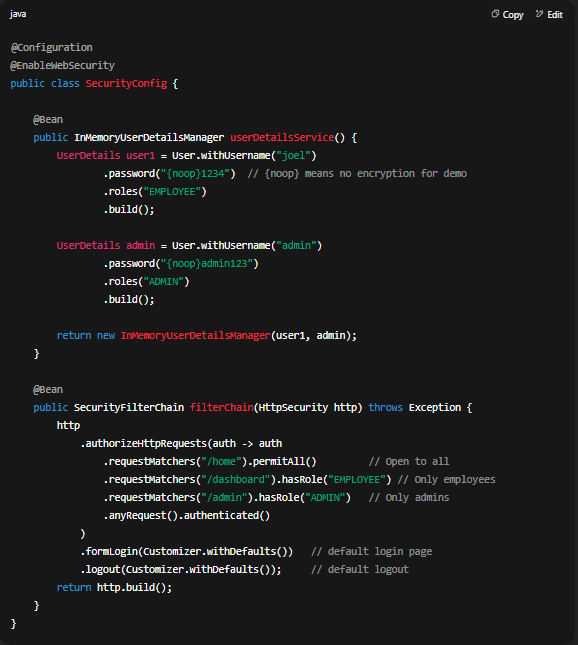
**DEPENDENCY:**



**CONFIG CLASS :**



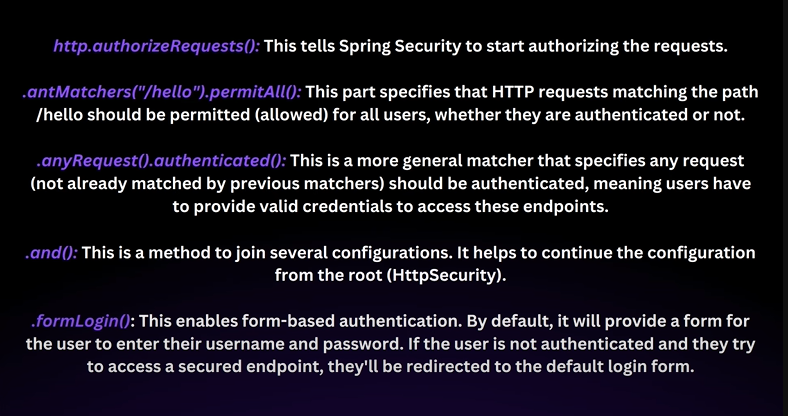
Example from chatGPT:



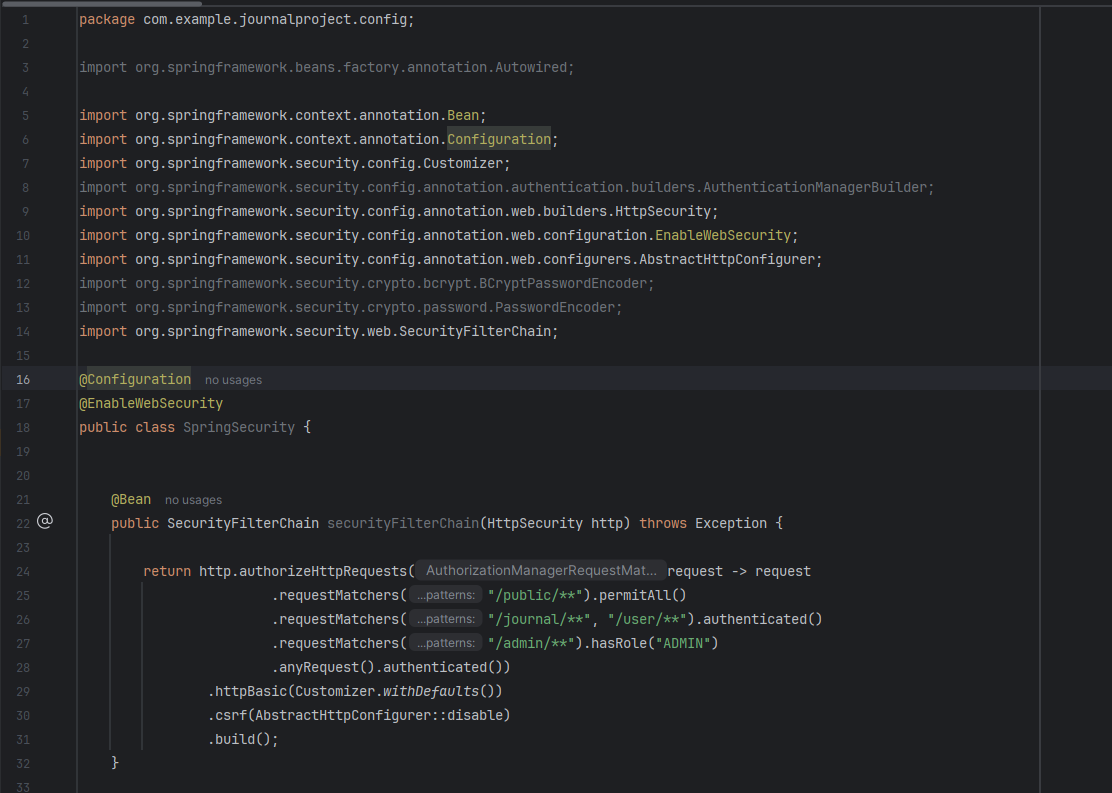
@EnableWebSecurity: This annotation signals Spring to enable its we security support. This is what makes your application secured. Its’s used in conjunction with @Configuration.

WebSecurityConfigureAdapter is a utility class in the Spring Security framework that provides default configurations and allows customization of certain features. By extending it, you can configure and customize Spring Security for your application needs.

Configure method provides a way to configure how requests are secured. It defines how request matching should be done and what security actions should be applied.



**THAT WE HAVE USED:**



.authenticated() : means any other requests need to be authenticated and a form-based login will be used.

You can access /hello without any authentication. However, if you try to access another endpoint, you’ll be redirected to a login form.

When we use the .formLogin() method in our security configuration without specifying .loginPage(“/custom-path”), the default login page becomes active.

Spring Security provides an in-built controller that handles the /login path. This controller is responsible for rendering the default login form when a GET request is made to /login.

.permitAll() : Permits all requests to /hello without authentication.

By default, Spring Security also provides logout functionality. When .logout() is configured, a POST request to /logout will log the user out and invalidate their session.

**BASIC AUTHENTICATION, BY ITS DESIGN, IS STATELESS.**

Some applications do mix Basic Authentication with session management for various reasons. This isn’t standard behavior and management for various reasons. This isn’t standard behavior and requires additional setup and logic. In such scenarios, once the user’s credentials are verified via Basic Authentication, a session might be established, and the client is provided a session cookie. This way, the client won’t need to send the Authorization header with every request, and the server can rely on the session cookie to identify the authenticated user.

**WHEN YOU LOG IN WITH SPRING SECURITY, IT MANAGES YOUR AUTHENTICATION ACROSS MULTIPLE REQUESTS, DESPITE HTTP BEING STATELESS.**

**1. Session Creation:** After successful authentication, an HTTP session is formed. Your authentication details are stored in this session.

**2**. **Session Cookie:** A JESSIONID cookie is sent to your browser, which gets sent back with subsequent requests, helping the server recognize your session.

**3. Security Context:** Using the JSESSIONID, Spring Security fetches your authentication details for each request.

**4. Session Timeout:** Session have a limited life. If you’re inactive past this limit, you’re logged out.

**5. Logout:** When logging out, your session ends, and the related cookie is removed.

**6. Remember-Me:** Spring Security can remember you even after the session end using a different persistence cookie (typically have a longer lifespan).

In essence, Spring Security leverages sessions and cookies, mainly JSESSIONID, to ensure you remain authenticated across requests.

We want our Spring Boot application to authenticate users based on their credentials stored in a MongoDB database.

This means that our users and their passwords (hashed) will be stored in MongoDB, and when a user tries to log in, the system should check the provided credentials against what’s stored in the database.

We want out Spring Boot application to authenticate users based on their credentials stored in a MongoDB database.

* A user entity to represent the user data model.
* A repository “UserEntryRepository” to interact with MongoDB.
* **“UserDetailServiceImpl” implementation to fetch user details.**
* A configuration SecurityConfig to integrate everything with Spring Security.