Spring is considered a trusted framework in the Java ecosystem and is widely used. It's no longer valid to refer to Spring as a framework, as it's more of an umbrella term that covers various frameworks.

One of these frameworks is Spring Security, which is a powerful and customizable authentication and authorization framework.

It is considered the de facto standard for securing Spring-based applications, so if you're looking to implement a Spring JWT token solution, it makes sense to base it on Spring Security.

Despite its popularity, I must admit that when it comes to single-page applications, Spring's not simple and straightforward to configure.

I suspect the reason is that it started more as an MVC application-oriented framework, where webpage rendering happens on the server-side and communication is session-based.

If the back end is based on Java and Spring, it makes sense to use Spring Security with JWT for authentication/authorization and configure it for stateless communication.

Defining Terminology

Before diving into the technical details, I want to explicitly define the terminology used in the Spring Security context just to be sure that we all speak the same language.

These are the terms we need to address:

- **Authentication** refers to the process of verifying the identity of a user, based on provided credentials. A common example is entering a username and a password when you log in to a website. You can think of it as an answer to the question *Who are you?*.
- **Authorization** refers to the process of determining if a user has proper permission to perform a particular action or read particular data, assuming that the user is successfully authenticated. You can think of it as an answer to the question *Can a user do/read this?*.
- **Principle** refers to the currently authenticated user.
- **Granted authority** refers to the permission of the authenticated user.
- **Role** refers to a group of permissions of the authenticated user.

Spring Security Filters Chain

When you add the Spring Security framework to your application, it automatically registers a filters chain that intercepts all incoming requests. This chain consists of various filters, and each of them handles a particular use case.

For example:

- Check if the requested URL is publicly accessible, based on configuration.
- In case of session-based authentication, check if the user is already authenticated in the current session.
- Check if the user is authorized to perform the requested action, and so on.

One important detail I want to mention is that Spring Security filters are registered with the lowest order and are the first filters invoked. For some use cases, if you want to put your custom filter in front of them, you will need to add padding to their order. This can be done with the following configuration:

Once we add this configuration to our `application.properties` file, we will have space for 10 custom filters in front of the Spring Security filters.

AuthenticationManager

You can think of `AuthenticationManager` as a coordinator where you can register multiple providers, and based on the request type, it will deliver an authentication request to the correct provider.

AuthenticationProvider

`AuthenticationProvider` processes specific types of authentication. Its interface exposes only two functions:

- `authenticate` performs authentication with the request.
- `supports` checks if this provider supports the indicated authentication type.

One important implementation of the interface that we are using in our sample project is `DaoAuthenticationProvider`, which retrieves user details from a `UserDetailsService`.

UserDetailsService

`UserDetailsService` is described as a core interface that loads user-specific data in the Spring documentation.
In most use cases, authentication providers extract user identity information based on credentials from a database and then perform validation. Because this use case is so common, Spring developers decided to extract it as a separate interface, which exposes the single function:

- `loadUserByUsername` accepts username as a parameter and returns the user identity object.

Authentication Using JWT with Spring Security

To customize Spring Security for JWT use, we need a configuration class annotated with `@EnableWebSecurity` annotation in our classpath. Also, to simplify the customization process, the framework exposes a `WebSecurityConfigurerAdapter` class. We will extend this adapter and override both of its functions so as to:

1. Configure the authentication manager with the correct provider
2. Configure web security (public URLs, private URLs, authorization, etc.)