Authentication

Theory

25 minutes reading

Unskip this topic Start practicing

2 required topics

✓ Spring Security Crypto

✓ Getting started with Spring Security

1 dependent topic

Authorization

Authentication is how we verify the identity of whoever is trying to access our application. A common way to authenticate users is by asking them to enter their login and password. If a user enters the correct data, the system assumes the identity is valid and grants access.

As you probably know, when we add Spring Security starter dependency, Spring Security puts our app behind the authentication process and generates a default user. In most cases, one user is not enough and what we typically need is a lot of users. In this topic, you'll learn how authentication can be configured in Spring Security and we'll create a couple of hardcoded in-memory users. We will start step by step and then review the example in full.

§1. AuthenticationManagerBuilder

To configure what authentication should do in Spring Security, we can use a special builder, `AuthenticationManagerBuilder`. With the help of this builder and method chaining we can create hardcoded in-memory users, connect our database with user info, and set other configurations. There are two steps:

1. Obtain `AuthenticationManagerBuilder`
2. Set the configuration using this builder

One of the ways to get `AuthenticationManagerBuilder` is by extending `WebSecurityConfigurerAdapter` and overriding its method `configure`. Spring Security will pass the builder to this method as an argument. Let's see how we can start:

▼ Java

```
@EnableWebSecurity
public class WebSecurityConfigurerImpl extends
WebSecurityConfigurerAdapter {
    //..
}
```

▼ Kotlin

```
@EnableWebSecurity
class WebSecurityConfigurerImpl : WebSecurityConfigurerAdapter() {
    //..
}
```

Note that the class that extends the adapter is annotated with `@EnableWebSecurity` annotation. As long as this annotation is included, our configurations written in the class will be detected by Spring, otherwise they will be ignored. Now let's override the method `configure`:

▼ Java

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws
```

```
Exception {
    //..
}
```

▼ Kotlin

```

override fun configure(auth: AuthenticationManagerBuilder) {
    //..
}
```

As you can see, the method receives `AuthenticationManagerBuilder`.

The adapter we mentioned allows overriding three methods with the same name: `configure`. The two remaining methods have another purpose and don't receive `AuthenticationManagerBuilder`. When overriding a method pay attention to what it receives.

To create hardcoded users we will use one of the methods of the builder, `inMemoryAuthentication()`, and then, using method chaining, we'll specify the login and password pair for one or more users. Here's an example with one user:

▼ Java

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
    auth.inMemoryAuthentication()
            .withUser("user1")
            .password("pass1")
            .roles();
}
```

▼ Kotlin

```
override fun configure(auth: AuthenticationManagerBuilder) {
    auth.inMemoryAuthentication()
        .withUser("user1")
        .password("pass1")
        .roles()
}
```

Note that apart from specifying the login and password we have one additional method call, `roles()`. This method is used to specify zero or more user roles. In our case, we aren't using any roles. You'll learn about user roles in the upcoming topics.

This approach is useful for testing and providing examples. Usually, user info will be stored in a database. We will show how to store users in a database in a separate topic.

If we want more users, we can separate them with the `and()` method call. Here's an example for 2 users:

▼ Java

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
    auth.inMemoryAuthentication()
            .withUser("user1")
            .password("pass1")
            .roles()
            .and()
```

```
                .withUser("user2")
                .password("pass2")
                .roles()
                // more users
}
```

▼ Kotlin

```
override fun configure(auth: AuthenticationManagerBuilder) {
    auth.inMemoryAuthentication()
        .withUser("user1")
        .password("pass1")
        .roles()
        .and()
        .withUser("user2")
        .password("pass2")
        .roles()
    // more users
}
```

There is one more thing we need to do to have multiple hardcoded users. If we run a program with the implementation mentioned above and then try to access the resource of our app, we will see an error in the console after inputting one of the correct login/password pairs. The console log will contain the following entry:

```
java.lang.IllegalArgumentException: There is no PasswordEncoder mapped
for the id "null"
```

It indicates that the password encoder is not specified. Let's learn what it means and how to fix it.

§2. Password encoders

For security reasons passwords should not be stored in plain text and should be encoded. For example, if they are stored in plain text in a database and someone gets access to that database, be it a hacker or another employee, they will be able to copy unencoded passwords, log in as a user and perform account-related operations (send messages, transfer money, blackmail a real user, and so on). Storing passwords encoded makes it much harder for someone to impersonate a user.

In Spring Security, password encoding is done with the implementation of the `PasswordEncoder` interface. This interface has two abstract methods:

▼ Java

- `String encode(CharSequence rawPassword)` receives a raw password and returns an encoded password. It's used before storing a password.
- `boolean matches(CharSequence rawPassword, String encodedPassword)` receives a raw password and an encoded password. Returns true if passwords match, otherwise returns false. It's used in the authentication process by Spring Security to check if the input raw password matches the encoded password that is stored.

▼ Kotlin

- `encode(rawPassword: CharSequence): String` receives a raw password and returns an encoded password. It's used before storing a password.
- `matches(rawPassword: CharSequence, encodedPassword: String): Boolean` receives a raw password and an encoded password. Returns true if

passwords match, otherwise returns false. It's used in the authentication process by Spring Security to check if the input raw password matches the encoded password that is stored.

Spring Security forces a developer to use a password encoder, otherwise the program won't work properly. In our case, we don't use a database to store user credentials. Passwords are visible, but we still need to use a password encoder.

Spring Security provides a few implementations of this interface that we can use. There is also a possibility to create a custom encoder by implementing this interface if the default implementations are not enough. Let's take a look at two of the encoders provided by Spring Security:

- `BCryptPasswordEncoder` uses a bcrypt strong hashing function to encode a password, usually the best default solution available.
- `NoOpPasswordEncoder` doesn't do any encoding of the password and returns it the way it was. As for matching, it only compares the strings using `equals()`. Should only be used for testing and examples.

To make our program work, we need to encode a password before storing it and tell Spring Security what encoder we used so it can use the encoder in the authentication process. First, we need a password encoder, so let's provide a bean with `BCryptPasswordEncoder`:

▼ Java

```java
@Bean
public PasswordEncoder getEncoder() {
    return new BCryptPasswordEncoder();
}
```

▼ Kotlin

```kotlin
@Bean
fun getEncoder(): PasswordEncoder {
    return BCryptPasswordEncoder()
}
```

Now we can encode a password and specify which encoder should be used in the authentication process. Here's an example:

▼ Java

```java
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
            .withUser("user1")
            .password(getEncoder().encode("pass1")) // encoding a password
            .roles()
             // more users
            .and()
            .passwordEncoder(getEncoder()); // specifying what encoder we used
}
```

▼ Kotlin

```kotlin
override fun configure(auth: AuthenticationManagerBuilder) {
    auth.inMemoryAuthentication()
        .withUser("user1")
        .password(getEncoder().encode("pass1")) // encoding a password
        .roles()
        // more users
        .and()
        .passwordEncoder(getEncoder()); // specifying what encoder we used
}
```

In the case of using `NoOpPasswordEncoder`, the process is the same but this class doesn't have a public constructor. To get an instance of it we need to call the static method `getInstance()`. Here's an example:

```
NoOpPasswordEncoder.getInstance();
```

Also, as already mentioned, `NoOpPasswordEncoder` returns the same password so there is no need to encode the password, so this part can be skipped.

Now let's see the full code example and discuss what happens during authentication.

§3. Putting pieces together

The full implementation looks like this:

▼ Java

```java
import org.springframework.context.annotation.Bean;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

// Extending the adapter and adding the annotation
@EnableWebSecurity
public class WebSecurityConfigurerImpl extends WebSecurityConfigurerAdapter {

    // Acquiring the builder
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {

        // storing users in memory
        auth.inMemoryAuthentication()
                .withUser("user1")
                .password(getEncoder().encode("pass1")) // encoding a password
                .roles()
                .and() // separating sections
                .withUser("user2")
                .password(getEncoder().encode("pass2"))
                .roles()
                .and()
                .passwordEncoder(getEncoder()); // specifying what encoder we used
    }

    // creating a PasswordEncoder that is needed in two places
    @Bean
    public PasswordEncoder getEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

▼ Kotlin

```
import org.springframework.context.annotation.Bean
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder
import org.springframework.security.crypto.password.PasswordEncoder


// Extending the adapter and adding the annotation
@EnableWebSecurity
class WebSecurityConfigurerImpl : WebSecurityConfigurerAdapter() {

    // Acquiring the builder
    override fun configure(auth: AuthenticationManagerBuilder) {

        // storing users in memory
        auth.inMemoryAuthentication()
            .withUser("user1")
            .password(getEncoder().encode("pass1")) // encoding a password
            .roles()
            .and() // separating sections
            .withUser("user2")
            .password(getEncoder().encode("pass2"))
            .roles()
            .and()
            .passwordEncoder(getEncoder()) // specifying what encoder we used
    }

    // creating a PasswordEncoder that is needed in two places
    @Bean
    fun getEncoder() = BCryptPasswordEncoder()
}
```

Note that since we override the default configuration, the default user won't be created.

If we create and run a program with this implementation included, we'll be able to access it using one of the valid login/password pairs, and form-based or HTTP basic auth, which, as you already know, are enabled by default.

When a user tries to pass authentication, Spring Security will search for a user with a specified login. If the user is found, the password encoder and its method `matches` will be used to check if the inputted raw password matches the stored encoded one. If everything is correct, the user is allowed to access the app, and authentication is completed.

It doesn't matter where user credentials are loaded from, the process is similar. In the example above, we stored the users in memory. Usually, there is a database with user info and an endpoint responsible for user registration. This endpoint will populate the database with user info.

The default config related to form-based and HTTP basic auth can be configured too.

§4. HttpSecurity

To specify which authentication methods are allowed (form-based, HTTP basic) and how they are configured, we can override another `configure` method of `WebSecurityConfigurerAdapter` that receives the `HttpSecurity` object.

The example below shows the configuration equal to the default one.

▼ Java

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().anyRequest().authenticated() // (1)
        .and()
        .formLogin() // (2)
        .and()
        .httpBasic(); // (3)
}
```

▼ Kotlin

Table of contents:

```
override fun configure(http: HttpSecurity) {
    http.authorizeRequests().anyRequest().authenticated() // (1)
        .and()
        .formLogin() // (2)
        .and()
        .httpBasic() // (3)
}
```

1. To access any URI (`anyRequest()`) on our app, a user needs to authenticate (`authenticated()`).
2. Enables form-based auth with default settings.
3. Enables HTTP Basic auth.

This configuration is why your application is on lockdown as soon as you add the Spring Security starter dependency.

Removing `.formLogin()` in the code above will disable this type of authentication. Also, by placing some additional method calls after `.formLogin()` we can change the look of the login page and some other things. With HTTP basic auth the situation is similar: if we omit `.httpBasic()`, we will disable this type of auth.

When we override the `configure` method, we override the default config. For example, if after overriding we don't enable HTTP basic auth explicitly, it will not be enabled. This also applies to the form-based auth.

Also, authentication is not only about logins and passwords. We may want to implement fingerprint authentication or authentication via SMS where the user has to enter a code that has been sent to their phone in an SMS as proof of their identity. Spring Security allows configuring this functionality too, but we will not consider it in this topic.

§5. Conclusion

In this topic, you've learned how to override default configuration related to authentication and create one or more hardcoded in-memory users. You've also seen how to enable form-based and HTTP basic authentication explicitly or disable them. The way we chose to add hardcoded users is not the only possible one. Spring Security is more flexible, and there are often other ways to add the same functionality.

Report a typo

118 users liked this piece of theory. 12 didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing   Unskip this topic

Comments (11)   Useful links (2)   Show discussion