© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2024

M. Nardone, C. Scarioni, *Pro Spring Security*
https://doi-org.ezproxy.sfpl.org/10.1007/979-8-8688-0035-1_6

6. Configuring Alternative Authentication Providers

Massimo Nardone[1] and Carlo Scarioni[2]
(1) HELSINKI, Finland
(2) Surbiton, UK

One of Spring Security's strongest points is that you can plug different authentication mechanisms into the framework. Spring Security was built to create, as much as possible, a pluggable architecture model where different things can be plugged into the framework easily and unobtrusively. In the authentication layer, an abstraction exists that takes care of this part of the security process. This abstraction comes mainly in the form of the `AuthenticationProvider` interface, but specific security servlet filters and user details services also support it.

Spring Security 6 supports many different authentication mechanisms, including the following.

- Databases
- LDAP
- X.509
- OAuth 2/OpenID Connect 1.0
- WebSocket
- JSON Web Token (JWT)
- JAAS
- CAS

Most of this chapter explains how these authentication systems work independently of Spring Security. Although it gives you certain key details, it won't be an in-depth explanation. Of course, you see how Spring Security implements each of these authentication mechanisms, and you see that they have many things in common when it comes to the parts of Spring Security they use. This chapter focuses on how to add an H2 database to Spring Boot with Spring Security and JDBC authentication.

Let's look at how to create a new Spring Boot project with Spring Security, Spring Data JDBC, and H2.

Let's go to start.spring.io and create a new project, shown in Figure 6-1, with the following settings.

- Build tool: Maven
- Language: Java
- Packaging: JAR
- Java version: 20

Next, add the following dependencies.

- Web
- Spring Security
- Spring Data JDBC
- H2 Database

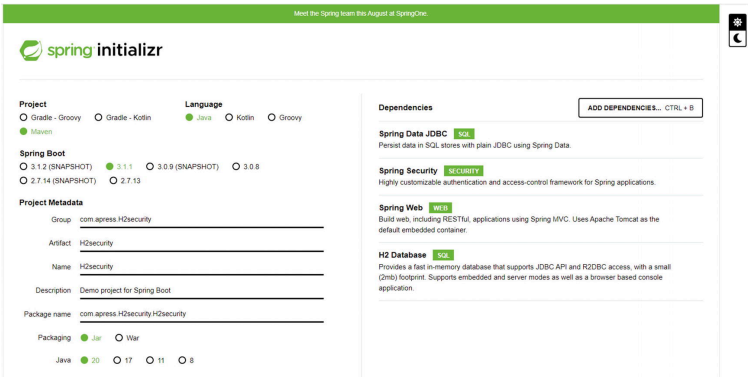

Figure 6-1 Creating a new JDBS and H2 DB project

Generate the project and unzip it on your machine.

You must enable and configure the H2 in-memory database console in the application.properties file as follows.

```
spring.h2.console.enabled=true

spring.datasource.name=securitydb

spring.datasource.url=jdbc:h2:mem:securitydb

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.datasource.driverClassName=org.h2.Driver
```

These lines tell the web application to enable the console, the name of the DB you wish to use, the datasource URL and driver class, and the Spring JOA DB platform.

Let's use the same Java classes and HTML files used in the Chapter 5, as shown in Figure 6-2.

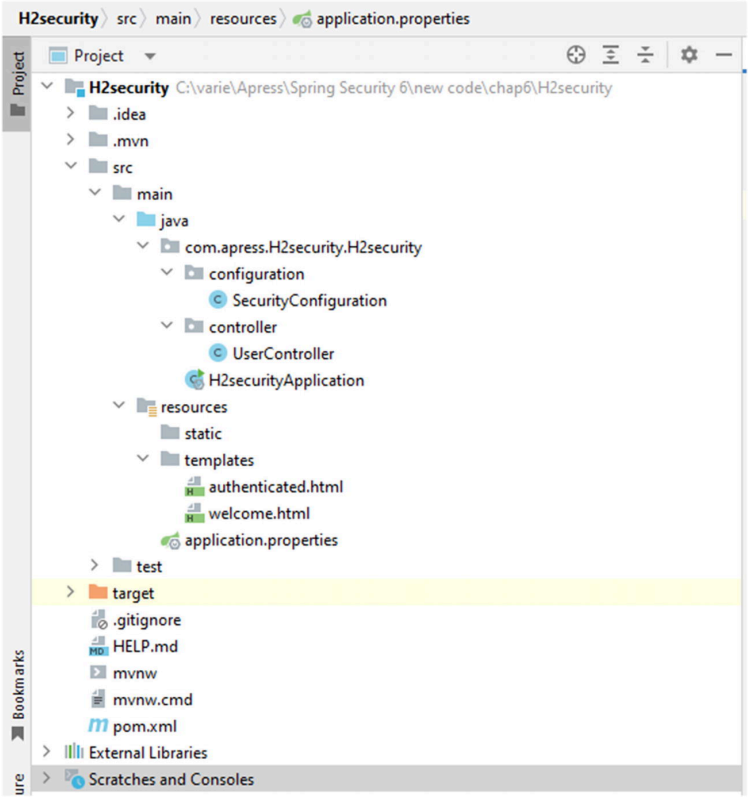

Figure 6-2 New JDBS and H2 DB project

Listing 6-1 shows the new pom.xml file after generating this new project.

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns:="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instanc

    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"

    <modelVersion>4.0.0</modelVersion>

    <parent>

       <groupId>org.springframework.boot</groupId>

       <artifactId>spring-boot-starter-parent</artifactId>

       <version>3.1.0</version>

       <relativePath/> <!-- lookup parent from repository -->

    </parent>
```

```
<groupId>com.apress.H2security</groupId>

<artifactId>H2security</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>H2security</name>

<description>Demo project for Spring Boot</description>

<properties>

   <java.version>20</java.version>

</properties>

<dependencies>

   <dependency>

      <groupId>org.springframework.boot</groupId>

      <artifactId>spring-boot-starter-data-jdbc</artifactId>

   </dependency>

   <dependency>

      <groupId>org.springframework.boot</groupId>

      <artifactId>spring-boot-starter-security</artifactId>

   </dependency>

   <dependency>

      <groupId>org.springframework.boot</groupId>

      <artifactId>spring-boot-starter-web</artifactId>
```

```
        </dependency>

        <dependency>

           <groupId>com.h2database</groupId>

           <artifactId>h2</artifactId>

           <scope>runtime</scope>

        </dependency>

        <dependency>

           <groupId>org.springframework.boot</groupId>

           <artifactId>spring-boot-starter-test</artifactId>

           <scope>test</scope>

        </dependency>

        <dependency>

           <groupId>org.springframework.security</groupId>

           <artifactId>spring-security-test</artifactId>

           <scope>test</scope>

        </dependency>

        <dependency>

           <groupId>org.thymeleaf.extras</groupId>

           <artifactId>thymeleaf-extras-springsecurity6</artifactId>

           <version>3.1.1.RELEASE</version>
```

```
        </dependency>

    </dependencies>



    <build>

        <plugins>

            <plugin>

                <groupId>org.springframework.boot</groupId>

                <artifactId>spring-boot-maven-plugin</artifactId>

            </plugin>

        </plugins>

    </build>



</project>
```

Listing 6-1
Updated pom.xml File

The new Maven dependencies are JDBC and H2.

```
        <dependency>

            <groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-starter-data-jdbc</artifactId>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
```

Let's update our files now.

The welcome.html file remains as it is in Chapter 5.

Let's update the authenticated.html file by adding the following lines.

```
<form th:action="@{/h2-console}" method="post">
    <input type="submit" value="check the h2-console"/>
</form>
```

Once the user is authenticated, this creates a new button to open the H2 console and check the updated databases using our example.

The Java class named `UserController` remains the same as in Chapter 5.

Let's update the `SecurityConfiguration` Java class to use the H2 embedded database, as shown in Listing 6-2.

```
package com.apress.H2security.H2security.configuration;




import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.jdbc.datasource.embedded.EmbeddedDatabase;

import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;

import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

import org.springframework.security.config.Customizer;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.core.userdetails.User;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.security.provisioning.JdbcUserDetailsManager;

import org.springframework.security.web.SecurityFilterChain;

import org.springframework.security.web.util.matcher.AntPathRequestMatcher;
```

```
import javax.sql.DataSource;




@Configuration

@EnableWebSecurity




public class SecurityConfiguration {




    @Bean

    public SecurityFilterChain filterChain1(HttpSecurity http) throws Exception {

        http




              .csrf(csrf -> csrf.ignoringRequestMatchers("/h2-console/**"))




                .authorizeHttpRequests((authorize) -> authorize

                        .requestMatchers("/", "/welcome").permitAll()

                        //.requestMatchers("/authenticated").hasRole("ADMIN")

                        .requestMatchers("/authenticated").hasAnyRole("USER", "ADMIN")

                        .requestMatchers(AntPathRequestMatcher.antMatcher("/h2-console/**")).permitAll(

                )
```

```
                .csrf(csrf -> csrf

                        .ignoringRequestMatchers(AntPathRequestMatcher.antMatcher("/h2-console/**")))



                .formLogin(Customizer.withDefaults())



                .headers(headers -> headers.disable())



                .logout((logout) -> logout

                        .logoutSuccessUrl("/welcome")

                        .deleteCookies("JSESSIONID")

                        .invalidateHttpSession(true)

                        .permitAll()

                );



        return http.build();

    }
```

```
@Bean
EmbeddedDatabase datasource() {
    return new EmbeddedDatabaseBuilder()
            .setName("securitydb")
            .setType(EmbeddedDatabaseType.H2)
            .addScript(JdbcDaoImpl.DEFAULT_USER_SCHEMA_DDL_LOCATION)
            .build();
}


@Bean
JdbcUserDetailsManager users(DataSource dataSource, PasswordEncoder encoder) {
    UserDetails user = User.builder()
            .username("user")
            .password(encoder.encode("userpassw"))
            .roles("USER")
            .build();
    UserDetails admin = User.builder()
            .username("admin")
            .password(encoder.encode("adminpassw"))
            .roles("ADMIN")
```

```
                .build();

        JdbcUserDetailsManager jdbcUserDetailsManager = new JdbcUserDetailsManager(dataSource);

        jdbcUserDetailsManager.createUser(user);

        jdbcUserDetailsManager.createUser(admin);

        return jdbcUserDetailsManager;

    }



    @Bean

    public static PasswordEncoder passwordEncoder(){

        return new BCryptPasswordEncoder();

    }



}
```

Listing 6-2
Updated SecurityConfiguration Java Class

Let's analyze this new Java class.

The `PasswordEncoder` bean stays the same as in previous examples.

Spring Security's JdbcDaoImpl implements UserDetailsService to support username-and-password-based authentication that is retrieved using JDBC. JdbcUserDetailsManager extends JdbcDaoImpl to provide management of UserDetails through the UserDetailsManager interface.

UserDetails-based authentication is used by Spring Security when configuring to accept a username/password for authentication.

Spring Security provides default queries for JDBC-based authentication, for which you can adjust the schema to match any customizations to the queries and the database dialect you use.

JdbcDaoImpl requires tables to load the password, account status (enabled or disabled), and a list of authorities (roles) for the user. The default schema is also exposed as a classpath resource named org/springframework/security/core/userdetails/jdbc/users.ddl, which is provided in the following listing.

```
create table users(
      username varchar_ignorecase(50) not null primary key,
      password varchar_ignorecase(500) not null,
      enabled boolean not null
);

create table authorities (
      username varchar_ignorecase(50) not null,
      authority varchar_ignorecase(50) not null,
      constraint fk_authorities_users foreign key(username) references users(username)
);
```

Before configuring JdbcUserDetailsManager, you must create a DataSource. In this example, we set up an embedded DataSource initialized with the default user schema via the `EmbeddedDatabase datasource` bean created to build a new H2 database (in our case named securitydb) using the preconfigured

JdbcDaoImpl default user DDL via `DJdbcDaoImpl.DEFAULT_USER_SCHEMA_DDL_LOCATION`.

```
@Bean
EmbeddedDatabase datasource() {
    return new EmbeddedDatabaseBuilder()
            .setName("securitydb")
            .setType(EmbeddedDatabaseType.H2)
            .addScript(JdbcDaoImpl.DEFAULT_USER_SCHEMA_DDL_LOCATION)
            .build();
```

The next step is to create the `JdbcUserDetailsManager` bean, as described in Listing 6-3.

```
@Bean
JdbcUserDetailsManager users(DataSource dataSource, PasswordEncoder encoder) {
    UserDetails user = User.builder()
            .username("user")
            .password(encoder.encode("userpassw"))
            .roles("USER")
            .build();
    UserDetails admin = User.builder()
```

```
            .username("admin")

            .password(encoder.encode("adminpassw"))

            .roles("ADMIN")

            .build();

    JdbcUserDetailsManager jdbcUserDetailsManager = new JdbcUserDetailsManager(dataSource);

    jdbcUserDetailsManager.createUser(user);

    jdbcUserDetailsManager.createUser(admin);

    return jdbcUserDetailsManager;

}
```

Listing 6-3
JdbcUserDetailsManager Java Bean

In this example, let's create two users to access the authenticated resource: user/userpassw and admin/adminpassw.

The last bean is `SecurityFilterChain`, as shown in Listing 6-4.

```
@Bean

public SecurityFilterChain filterChain1(HttpSecurity http) throws Exception {

    http



            .authorizeHttpRequests((authorize) -> authorize
```

```
                .requestMatchers("/", "/welcome").permitAll()
                .requestMatchers("/authenticated").hasAnyRole("USER", "ADMIN")
                .requestMatchers(AntPathRequestMatcher.antMatcher("/h2-console/**")).permitAll()
        )


        .csrf(csrf -> csrf
                .ignoringRequestMatchers(AntPathRequestMatcher.antMatcher("/h2-console/**")))


        .formLogin(Customizer.withDefaults())


        .headers(headers -> headers.disable())


        .logout((logout) -> logout
                .logoutSuccessUrl("/welcome")
                .deleteCookies("JSESSIONID")
                .invalidateHttpSession(true)
                .permitAll()
        );
```

```
    return http.build();

}
```

Listing 6-4
SecurityFilterChain Java Bean

First, create `requestMatchers` in this bean .

- `.requestMatchers("/", "/welcome").permitAll()` permits all to access "/" and "welcome " pages.
- `.requestMatchers("/authenticated").hasAnyRole("USER", "ADMIN")` permits the user and admin to access the authenticated page.
- `.requestMatchers(AntPathRequestMatcher.antMatcher("/h2-console/**")).permitAll()` permits access to the H2 console.

Since our Spring Boot project uses Spring Security and the class is annotated with the @EnableWebSecurity annotation, you must disable the HTTP header frame options and add the following to that class's configure() method.

The frame options are necessary to prevent a browser from loading your HTML page in an <iframe> or <frame> tag. To enable the H2 console page to load, you need to disable this option with this line.

```
.headers(headers -> headers.disable())
```

The line `.csrf(csrf -> csrf`

```
                    .ignoringRequestMatchers(AntPathRequestMatcher.antMatcher("/h2-console/**")))
```

allows ignoring RequestMatchers for the H2 "/h2-console/**" console path.

Build and run the Spring Boot application and open http://localhost:8080/welcome in the browser window. Authenticate with "user/userpassw" or "admin/adminpassw", as shown in Figures 6-3 and 6-4.

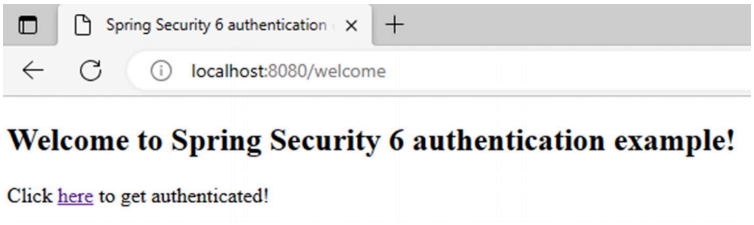

Figure 6-3 welcome.html page

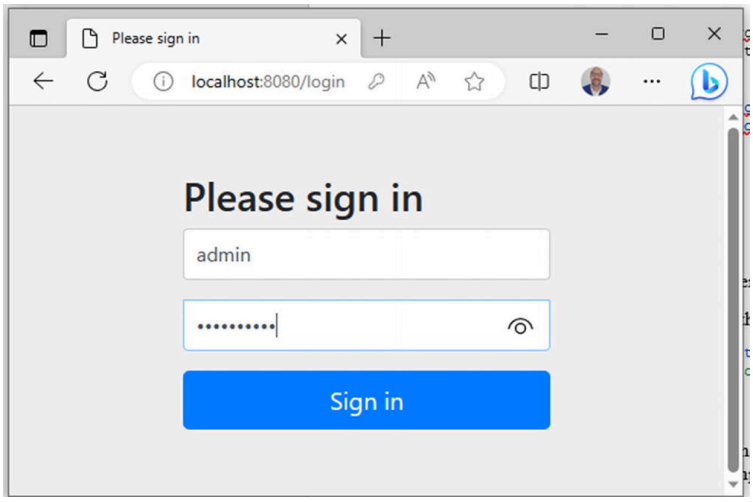

Figure 6-4 Login page

If authenticated, you can access the autnenticated.html page, which informs that the admin is an authenticated user and provides a "check the h2-console" button, as shown in Figure 6-5.

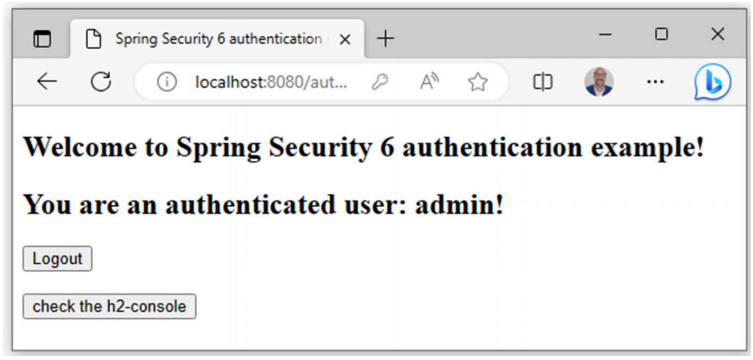

Figure 6-5 authenticated.html page

Click the "check the h2-console" button to log in to the H2 console, as shown in Figure 6-6.

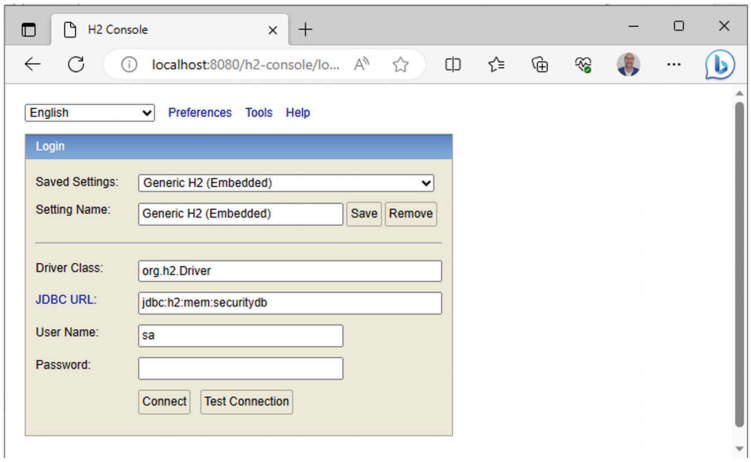

Figure 6-6 H2 login console page

Let's connect now to the H2 and discover the content, as shown in Figure 6-7.

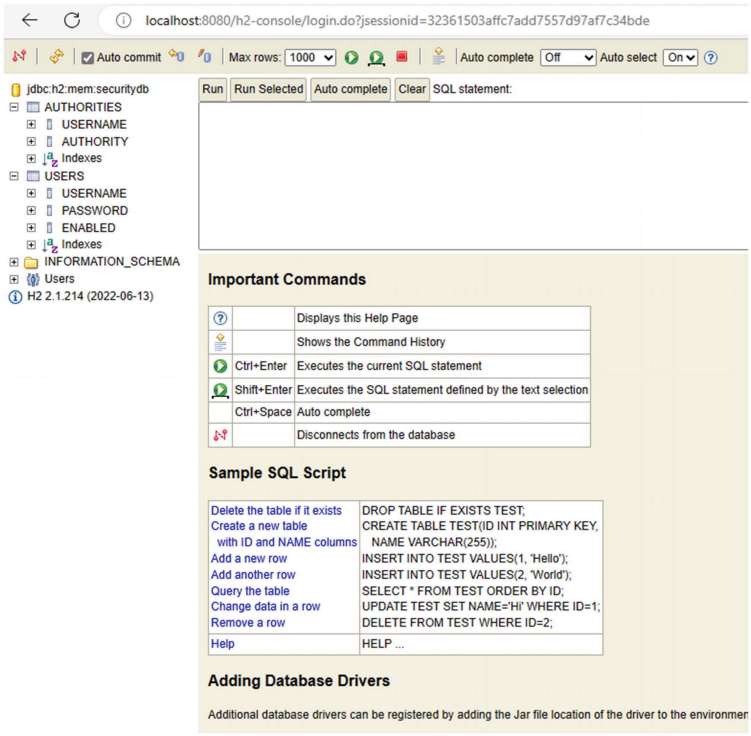

Figure 6-7 H2 console page with securitydb tables

The H2 console includes the two tables used for this example via JDBC authentication, such as Authorities and Users.

Let's run the following SQL scripts against the securitydb database to see the content of the tables.

The result of running the SELECT * FROM AUTHORITIES SQL script is shown in Figure 6-8.

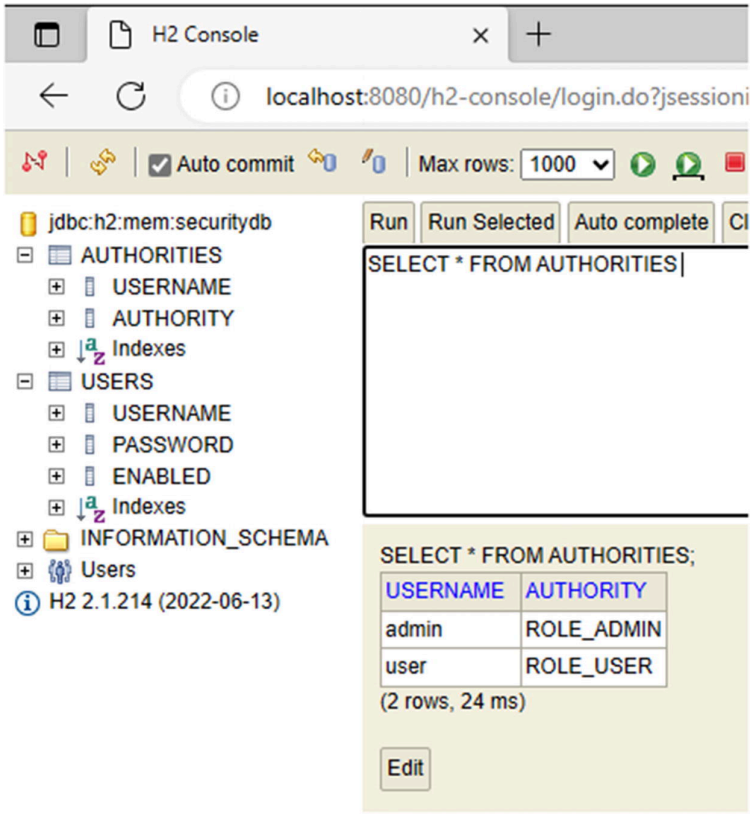

Figure 6-8 H2 authorities table script outcome

There are two new authorities in the database: admin/ROLE_ADMIN and user/ROLE_USER.

The result of running the SELECT * FROM USERS SQL script is shown in Figure 6-9.

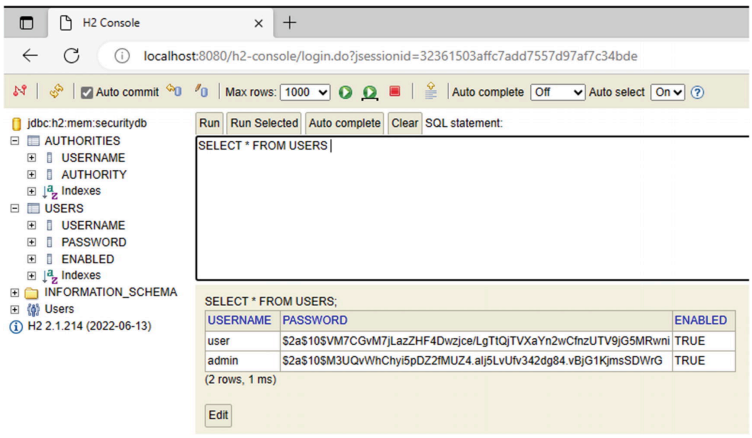

Figure 6-9 H2 users table script outcome

There are two new enabled users: “user/userpassw” and “admin/adminpassw”.

These are the users who authenticate the example.

LDAP Authentication

The Lightweight Directory Access Protocol (LDAP) is an application-level, message-oriented protocol for storing and accessing information through an accessible tree-like directory. A directory, in general, is simply an orga-

nized data store that allows for easy queries in its particular domain. For example, a TV Guide is a directory that allows you to find TV shows easily, and a phone book is a directory that provides easy access to phone numbers.

LDAP allows the storage of very different kinds of information in a directory. Probably the most widely known use of LDAP-like structures is the Microsoft Windows Active Directory system. Other LDAP systems are widely used to store the corporate user databases of many companies that serve as the centralized user store.

LDAP is not easy to understand, and we try to explain it using the example in this section. Let's use the same code as in the previous section but modify it to work with LDAP instead of database authentication. Remember that the previous section offered a bootstrap application to start working on all the examples in this chapter, including this one.

The first thing to do is configure your users in the LDAP directory. To do this, you need to understand the LDAP information model, which defines the type of data you can store in your directory.

Entries, attributes, and values define the data in LDAP. An *entry* is the basic unit of information in the directory and commonly represents an entity from the real world, like a user. Entries are normally defined by a particular object class. Each entry in the directory has an identification known as a *distinguished name* (or, more commonly, DN). Each entry in the directory also has a set of *attributes* that describe different things about the entry. Each attribute has a type and one or more *values*.

You must define the data you need. You use users, groups, and credentials, as you have done so far. In LDAP, the user entry definition is commonly known as *people*, so use that name to define the user entries. Your user also uses the standard LDAP object class `person` to define its attributes.

Using an Embedded LDAP

For simplicity, an embedded LDAP server is used in this example since Spring Security uses ApacheDS 1.x, which is no longer maintained, and unfortunately, ApacheDS 2.x has only released milestone versions with no stable release. Consider updating once a stable release of ApacheDS 2.x is available.

If you wish to use Apache DS, specify the following Maven dependencies.

```
<dependency>

      <groupId>org.apache.directory.server</groupId>
```

```
        <artifactId>apacheds-core</artifactId>

        <version>1.5.5</version>

        <scope>runtime</scope>

    </dependency>

    <dependency>

        <groupId>org.apache.directory.server</groupId>

        <artifactId>apacheds-server-jndi</artifactId>

        <version>1.5.5</version>

        <scope>runtime</scope>

    </dependency>
```

Let's create a new Spring Boot project with Spring Security, Spring Web, and LDAP.

Go to start.spring.io to create a new project (as shown in Figure **6-10**) with the following settings.

- Build Tool: Maven
- Language: Java
- Packaging: Jar
- Java Version: 17

Next, add the following dependencies.

- Web
- Spring Security
- Spring LDAP

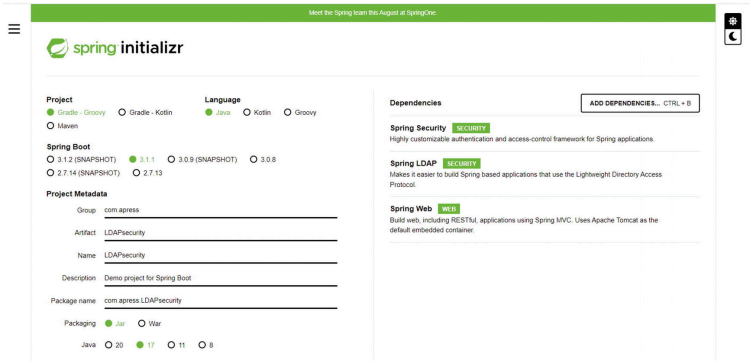

Figure 6-10 Creating a new LDAP project

Generate the project and unzip it on your machine.

You must enable and configure the H2 in-memory database console in our application.properties file as follows.

```
spring.thymeleaf.check-template-location=false

spring.ldap.embedded.port=8389

spring.ldap.embedded.ldif=classpath:*.ldif
```

Let's use the same Java classes from the previous example, as shown in Figure 6-11.

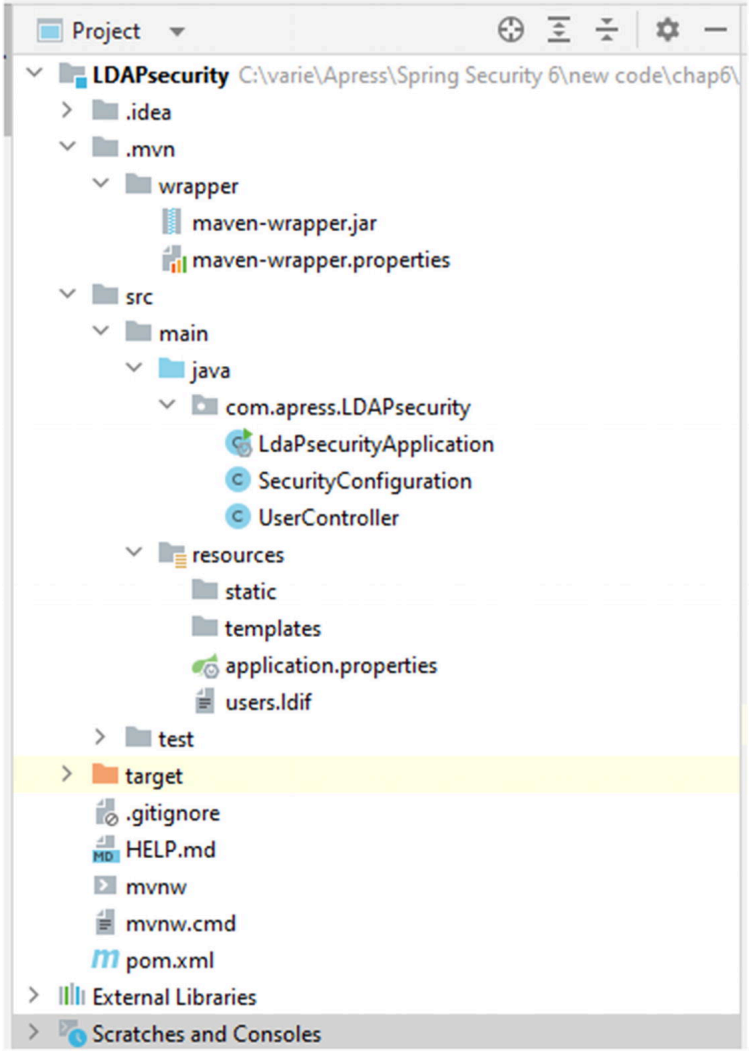

Figure 6-11 LDAPSecurity project structure

Listing 6-5 shows the new pom.xml file after generating this new project.

```
<?xml version=”1.0” encoding=”UTF-8”?>

<project xmlns:=”http://maven.apache.org/POM/4.0.0” xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instanc

    xsi:schemaLocation=”http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd”

    <modelVersion>4.0.0</modelVersion>

    <parent>

       <groupId>org.springframework.boot</groupId>

       <artifactId>spring-boot-starter-parent</artifactId>
```

```
    <version>3.1.0</version>
    <relativePath/> <!—lookup parent from repository —>
</parent>
<groupId>com.apress.LDAPsecurity</groupId>
<artifactId>LDAPsecurity</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>LDAPsecurity</name>
<description>Demo project for Spring Boot</description>



<properties>
    <java.version>17</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!—tag::security[]—>
    <dependency>
        <groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.ldap</groupId>
            <artifactId>spring-ldap-core</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-ldap</artifactId>
        </dependency>
        <dependency>
            <groupId>com.unboundid</groupId>
            <artifactId>nbounded-ldapsdk</artifactId>
            <version>6.0.9</version>
            <scope>runtime</scope>
        </dependency>        <!—end::security[]—>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
```

```
        </dependency>

        <dependency>

            <groupId>org.springframework.security</groupId>

            <artifactId>spring-security-test</artifactId>

            <scope>test</scope>

        </dependency>

    </dependencies>



    <build>

        <plugins>

            <plugin>

                <groupId>org.springframework.boot</groupId>

                <artifactId>spring-boot-maven-plugin</artifactId>

            </plugin>

        </plugins>

    </build>



</project>
```

Listing 6-5

Updated pom.xml

The following are the specific Maven dependencies needed in the LDAP Spring Security 6 example.

```
<dependency>
    <groupId>com.unboundid</groupId>
    <artifactId>unboundid-ldapsdk</artifactId>
    <version>6.0.9</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.ldap</groupId>
    <artifactId>spring-ldap-core</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
```

```
    <artifactId>spring-security-ldap</artifactId>

</dependency>
```

The next step is to create the LDIF file used as an embedded LDAP server.

An LDIF file is a text file that uses LDIF formatting, a standard format for describing directory entries in LDAP. It allows you to import and export your directory data into or from another LDAP directory in a standard way or just to create new data or modify existing data. You use it here to import the data with your users. Listing 6-6 shows the LDIF file you import. You can name this file whatever you like, but it is `users.ldif` in this example.

```
dn: dc=example,dc=com

objectclass: top

objectclass: domain

dc: example

dn: ou=groups,dc=example,dc=com

objectclass: organizationalUnit

objectclass: top

ou: groups

dn: cn=administrators,ou=groups,dc=example,dc=com

objectclass: groupOfUniqueNames

objectclass: top

cn: administrators

uniqueMember: uid=mnardone,ou=people,dc=example,dc=com
```

```
ou: admin

dn: cn=users,ou=groups,dc=example,dc=com

objectclass: groupOfNames

objectclass: top

cn: users

member: uid=lnardone,ou=people,dc=example,dc=com

ou: user

dn: ou=people,dc=example,dc=com

objectclass: organizationalUnit

objectclass: top

ou: people

dn: uid=lnardone,ou=people,dc=example,dc=com

objectclass: inetOrgPerson

objectclass: organizationalPerson

objectclass: person

objectclass: top

cn: Leo Nardone

sn: Leo

uid: lnardone

userPassword: {SHA}F1OkcxtiioPmVX3tJlIHzZsXkDQ=

dn: uid=mnardone,ou=people,dc=example,dc=com
```

```
objectclass: inetOrgPerson

objectclass: organizationalPerson

objectclass: person

objectclass: top

cn: Massimo Nardone

sn: Nardone

uid: mnardone

userPassword: {SHA}xcS5y9T0kjBXDpYijejbhmILFwY=
```

Listing 6-6

LDIF File with the Two Users You Want to Import into the LDAP Directory

As you can see, this code generated two users, mnardone, part of the administrators group, and lnardone, who is not included in that group.

The SHA password was generated via **http://aspirine.org/htpasswd_en.html**. The passwords are

- *nardone01 = {SHA}xcS5y9T0kjBXDpYijejbhmILFwY=*
- *nardone02 = {SHA}F1OkcxtiioPmVX3tJlIHzZsXkDQ=*

You can see in Listing **6-6** the hierarchical nature of the directory and how everything inherits the DN `dc=example,dc=com`. You can also see how the different entries use different standard object classes. You have created two groups: administrators and users. You also established that *lnardone* is a member of the users group, *mnardone* is a member of the administrators group, and the two SHA passwords.

```
"""""""""""uccessfuccessfu—>""""""""uccessfully"""\'""""""""""
```

The main class is `LdaPsecurityApplication`, as shown in Listing **6-7**.

```
package com.apress.LDAPsecurity.LDAPsecurity;
```

```
import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;



@SpringBootApplication

public class LdaPsecurityApplication {



    public static void main(String[] args) {

       SpringApplication.run(LdaPsecurityApplication.class, args);

    }



}
```

Listing 6-7

LdaPsecurityApplication Java Class

Next, create some needed Java classes. Start with a simple `UserController`, shown in Listing 6-8, which is similar to one used in other projects in this book.

```
import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;
```

```
@RestController

public class UserController {



    @GetMapping("/")

    public String getLoginPage() {

        return "You have successfully logged in Using Spring Security 6 LDAP Authentication!";

    }



}
```

Listing 6-8
UserController Java Class

Listing 6-8 works so that when a login is successful, the user receives a simple message: “You have successfully logged in Using Spring Security 6 LDAP Authentication!”.

Next, update the Java class named `SecurityConfiguration`, shown in Listing 6-9, to take care of the LDAP authentication.

```
package com.apress.LDAPsecurity;
```

```
import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.ldap.core.support.BaseLdapPathContextSource;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.ldap.LdapPasswordComparisonAuthenticationManagerFactory;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.ldap.userdetails.DefaultLdapAuthoritiesPopulator;

import org.springframework.security.ldap.userdetails.LdapAuthoritiesPopulator;

import org.springframework.security.web.SecurityFilterChain;



import static org.springframework.security.config.Customizer.withDefaults;



@Configuration

@EnableWebSecurity



public class SecurityConfiguration {
```

```
@Bean
public SecurityFilterChain filterChain1(HttpSecurity http) throws Exception {
    http
            .authorizeHttpRequests((authorize) -> authorize
                    .anyRequest().fullyAuthenticated()
            )
            .formLogin(withDefaults());
    return http.build();
}



@Bean
AuthenticationManager authenticationManager(BaseLdapPathContextSource contextSource) {
    LdapPasswordComparisonAuthenticationManagerFactory factory = new LdapPasswordComparisonAuthenti
            contextSource, new BCryptPasswordEncoder());
    factory.setUserDnPatterns("uid={0},ou=people");
    factory.setPasswordAttribute("userPassword");
    factory.setUserSearchBase("ou=people");
    factory.setPasswordEncoder(new BCryptPasswordEncoder());
    return factory.createAuthenticationManager();
}
```

```
    @Bean
    LdapAuthoritiesPopulator authorities(BaseLdapPathContextSource contextSource) {
        String groupSearchBase = "ou=groups";
        DefaultLdapAuthoritiesPopulator authorities = new DefaultLdapAuthoritiesPopulator
                (contextSource, groupSearchBase);
        authorities.setGroupSearchFilter("(member={0})");
        return authorities;
    }


}
```

Listing 6-9
SecurityConfiguration Java Class

Let's discuss the Java class.

The following lines force you to request full authentication for any URL via formLogin.

```
        http
                .authorizeHttpRequests((authorize) -> authorize
                        .anyRequest().fullyAuthenticated()
```

```
                )

                .formLogin(withDefaults());

        return http.build();
```

This example uses LDAP password authentication, where password comparison is when the password supplied by the user is compared with the one stored in the repository. This can be done by retrieving the value of the password attribute and checking it locally or by performing an LDAP "compare" operation, where the supplied password is passed to the server for comparison, and the real password value is never retrieved. An LDAP comparison cannot be done when the password is properly hashed with a random salt.

The following lines define the LDAP password authentication.

```
authenticationManager(BaseLdapPathContextSource contextSource) {

        LdapPasswordComparisonAuthenticationManagerFactory factory = new LdapPasswordComparisonAuthenti

                contextSource, new BCryptPasswordEncoder());

        factory.setUserDnPatterns("uid={0},ou=people");

        factory.setPasswordAttribute("userPassword");

        factory.setUserSearchBase("ou=people");

        factory.setPasswordEncoder(new BCryptPasswordEncoder());

        return factory.createAuthenticationManager();

    }
```

Finally, to determine which authorities are returned for the user, we used the following LdapAuthoritiesPopulator code.

```
@Bean
LdapAuthoritiesPopulator authorities(BaseLdapPathContextSource contextSource) {
    String groupSearchBase = "ou=groups";
    DefaultLdapAuthoritiesPopulator authorities = new DefaultLdapAuthoritiesPopulator
            (contextSource, groupSearchBase);
    authorities.setGroupSearchFilter("(member={0})");
    return authorities;
}
```

Start the application now with this configuration. You should be able to see the login page via `http://localhost:8080/`.

Log in and access the Spring Security 6 login page, as shown in Figure 6-12.

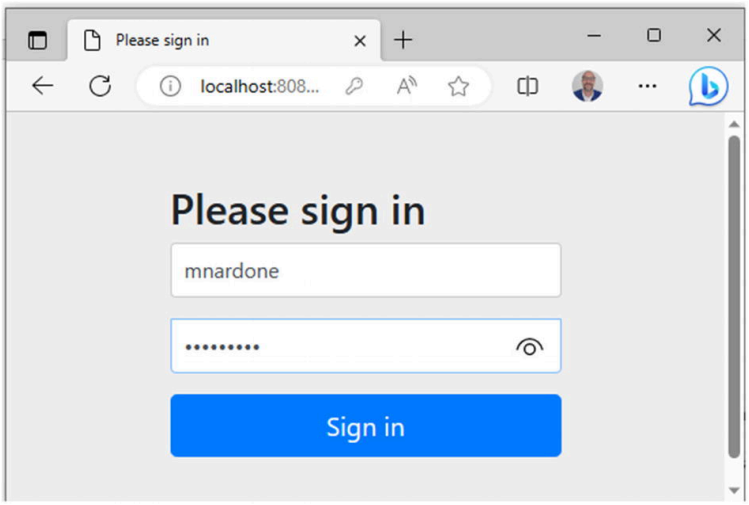

Figure 6-12 Application login page

In the LDAP, there are two users: mnardone and lnardone. The difference is that mnardone is part of the administrators group, so they can access the successful page.

Log in with the **mnardone** username and the **nardone02** password to access the authorized page, as shown in Figure 6-13.

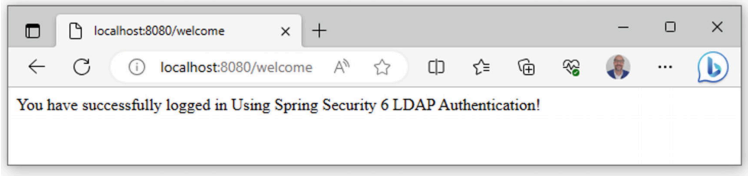

Figure 6-13 LDAP login page

The configuration is not that difficult to understand. The login configuration is within the `SpringSecurityConfiguration` Java class.

The `SecurityFilterChain filterChain1(HttpSecurity http)` configures the HTTP for features such as login and logout so that users can only access the URL in the LDAP as part of the administrators group.

```
dn: cn=administrators,ou=groups,dc=example,dc=com

objectclass: groupOfUniqueNames

objectclass: top

cn: administrators

uniqueMember: uid=mnardone,ou=people,dc=example,dc=com

ou: admin
```

In this case, mnardone is part of the administrators LDAP group, so they can access the `secure` page.

Remember that all your entries are relative to the domain name `dc=example,dc=com`.

As you can see from the example, configuring LDAP's basic support as the authentication solution for your application with Spring Security is not that complex. Thanks to the modular architecture and well-thought-out XML and Java namespace, it is very straightforward. The complexity of

LDAP is LDAP itself. Although it is a simple hierarchical system (very much like the file system in your standard Unix box), some of the nomenclature and functionality seem a bit complex and very different from the database-based solution you explored in the previous section.

Using LDAP as your authentication solution makes great sense in the context of corporate intranets, where the company user base is already stored in LDAP-like directories in a centralized manner. Plugins into this already existing user-management infrastructure are a good way to reuse the user information within the company instead of writing a parallel authentication datastore that needs to be kept in sync with the main repository.

X.509 Authentication

X.509 authentication is an authentication scheme that uses client-side certificates instead of username-password combinations to identify the user. Using this approach, a scheme known as *mutual authentication* takes place between the client and the server. In practice, mutual authentication means that, as part of the Secure Sockets Layer (SSL) handshake, the server requests that the client identify himself by providing a certificate. In a production-ready server, a proper certificate-signing authority must issue and sign the incoming client certificate.

To work with client certificates, the application must be configured to use SSL channels in the sections expected to deal with the authenticated user because the X.509 authentication protocol is part of the SSL protocol.

To enable X.509 client authentication in Spring Security 6, you must add the <x509/> element to your HTTP security namespace configuration, as follows.

```
<http>

...

      <x509 subject-principal-regex="CN=(.*?)," user-service-ref="userService"/>;

</http>
```

The element has two optional attributes.

- subject-principal-regex. The regular expression used to extract a username from the certificate's subject name. The default value is

shown in the preceding listing. This username is passed to the UserDetailsService to load the authorities for the user.

- user-service-ref. This is the bean ID of the UserDetailsService to be used with X.509. It is unnecessary if only one is defined in your application context.

The subject-principal-regex should contain a single group. For example, the default expression (CN=(.*?)) matches the common name field. So, if the subject name in the certificate is "CN=Jimi Hendrix, OU=…", this gives a user name of "Jimi Hendrix". The matches are case-insensitive. So "emailAddress=(.*?)," matches "EMAILADDRESS=jimi@hendrix.org,CN=…", giving a user name "jimi@hendrix.org". If the client presents a certificate and a valid username is successfully extracted, there should be a valid Authentication object in the security context. The security context remains empty if no certificate or corresponding user can be found. You can use X.509 authentication with other options, such as a form-based login.

To set up SSL in Tomcat, you could use the pre-generated certificates in the Spring Security samples repository to enable SSL for testing if you do not want to generate your own. The server.jks file contains the server certificate, the private key, and the issuing authority certificate. Some client certificate files are also for the users from the sample applications. You can install these in your browser to enable SSL client authentication.

To run Tomcat with SSL support, drop the server.jks file into the tomcat conf directory and add the following connector to the server.xml file.

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true" scheme="https" secure="true"
                   clientAuth="true" sslProtocol="TLS"
                   keystoreFile="${catalina.home}/conf/server.jks"
                   keystoreType="JKS" keystorePass="password"
                   truststoreFile="${catalina.home}/conf/server.jks"
                   truststoreType="JKS" truststorePass="password"
/>
```

clientAuth can also be set to want if you still want SSL connections to succeed, even if the client does not provide a certificate. Clients who do not present a certificate cannot access any objects secured by Spring Security unless they use a non-X.509 authentication mechanism, such as form authentication.

OAuth 2.0

OAuth 2.0 is the industry-standard protocol for authorization. You can get more information at **https://oauth.net/2/**.

Since Spring Security 5.0 supports the OAuth 2.0 authorization framework and OpenID Connect 1.0. with Spring Security 5.1. It introduced new Resource Server support as well as additional client support.

The OAuth 2.0 implementation for authentication, which conforms to the OpenID Connect specification and is OpenID Certified, can be used for authentication and authorization via Google's OAuth 2.0 APIs. For more information, please go to **https://developers.google.com/identity/protocols/OAuth2**.

OAuth 2.0 supports

- Client
- Resource server
- Authorization server

The OAuth 2.0 client features support the client role defined in the OAuth 2.0 authorization framework. You can get more information at **https://auth0.com/**.

When developing an OAuth 2.0 Client, the following main features are available.

- Authorization code grant
- Client credentials grant
- The WebClient extension for servlet environments (for making protected resource requests)

Spring Security 5 introduced a new `OAuth2LoginConfigurer` class that you can use to configure an external authorization server.

More information is at **https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/jwt.html#oauth2resourceserver-jwt-architecture**,

JSON Web Token

Spring Security 5 supports JSON Web Token (JWT) authentication.

You need to do the following.

- Configure Spring Security for JWT.
- Expose the REST POST API with mapping/authenticate.

- Configure a valid JSON Web Token.

Specifically, to configure Spring Security and JWT, you need to perform two operations.

1.Generate JWT by
   a. Exposing a POST API with mapping/authenticating
   b. Passing the correct username and password
2.Validate JWT by
   a. When trying to access the GET API with a certain mapping like `/Testing`
   b. Allowing access only if a request is valid

Spring Security and JWT dependencies include the following.

```
<dependency>
<groupId>io.jsonwebtoken</groupId>
<artifactId>jjwt</artifactId>
<version>0.9.1</version>
</dependency>
```

Spring WebSocket

Since Spring Security 5 MVC supports Spring WebSocket. For WebSocket implementation, you want to add the following Maven dependencies.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-websocket</artifactId>
    <version>6.0.10</version>
</dependency>
```

```
<dependency>

    <groupId>org.springframework</groupId>

    <artifactId>spring-messaging</artifactId>

    <version>6.0.10</version>

</dependency>

<dependency>

    <groupId>org.springframework.security</groupId>

    <artifactId>spring-security-messaging</artifactId>

    <version>6.1.1</version>

</dependency>
```

Then, define the configuration of WebSocket-specific security as follows:

```
@Configuration

@EnableWebSocketSecurity

public class WebSocketSecurityConfig {



    @Bean

    AuthorizationManager<Message<?>> messageAuthorizationManager(MessageMatcherDelegatingAuthorizationM

        messages
```

```
                .simpDestMatchers("/user/**").hasRole("USER")




        return messages.build();

    }

}
```

For more information on WebSockets, check out the Spring Security 6 reference documentation at **https://docs.spring.io/spring-security/reference/servlet/integrations/websocket.html#page-title**.

Java Authentication and Authorization Service

The Java Authentication and Authorization Service (JAAS) is the standard Java support for managing authentication and authorization. Its functionality overlaps with that of Spring Security.

JAAS is a relatively large standard involving much more than the small amount of information covered here. However, the main concepts are the ones we showed you, and the goal of the section is to show the building blocks for integrating them with Spring Security.

For more information on JAAS, check out the Spring Security 5 reference documentation at **https://docs.spring.io/spring-security/reference/servlet/authentication/jaas.html**.

Central Authentication Service

The Central Authentication Service (CAS) is an enterprise single sign-on solution built in Java and open source. It has a great support community and integrates into many Java projects. CAS provides a centralized place for authentication and access control in an enterprise.

The JA-SIG (**www.ja-sig.org**) CAS a is a simple, open source, independent platform that supports proxy capabilities. Spring Security fully supports CAS for single applications and multiple-application deployments secured by an enterprise-wide CAS server. You can learn more about CAS at **www.apereo.org/projects/cas**.

One important characteristic of CAS is that it is designed to serve as a proxy for different authentication storage solutions. It can be used with

LDAP, JDBC, or other user stores containing real user data. This looks a lot like the way Spring Security leverages these same user data stores.

For more information on CAS, check out the Spring Security 6 reference documentation at **https://docs.spring.io/spring-security/reference/servlet/authentication/cas.html**.

Summary

This chapter illustrated how to use Spring Security's modular architecture to integrate different authentication mechanisms relatively easily. We explained some of the authentication mechanisms that come with the framework. We demonstrated how to authenticate your users in a database, an LDAP server, and by using client X.509 certificates. JAAS, OAuth 2.0/OpenID Connect 1.0, WebSocket, JWT, and CAS were also introduced.

This chapter focused on showing how all these different authentication providers relate to each other when used inside the framework. The goal was to show you that integrating new providers into the framework is simple enough for you to try. Of course, how easy it is depends on the authentication scheme that you want to plug in.

Other authentication providers weren't covered in this chapter, but the main ideas remain the same: create a connector into Spring Security that deals with the particulars of the integrating protocol and adapt it to use the Spring Security model of authentication and authorization.