**PHASE3-FOURTHPROJECT-WRITEUP**

**Implement Spring Security with Authentication**

**Project Objective:**

As a developer, build Authentication Provider in Spring Security.

**Background of the problem statement:**

You have been assigned a task by the team to add more flexibility rather than using the standard scenario in building Spring Security.

**Following requirements should be met:**

* You should have a spring MVC web application as part of your submission. (JSP or Thymeleaf are both acceptable front-ends)
* User Password style in-memory storage is acceptable but variations on this (such as storing to database) are also acceptable.
* Users should be redirected to a login.html page if not authenticated before being redirected to the original page once authenticated.
* A few of the source code should be tracked on GitHub repositories. You need to document the tracked files which are ignored during the final push to the GitHub repository.
* The submission of your GitHub repository link is mandatory. In order to track your task, you need to share the link of the repository in the document.
* The step-by-step process involved in completing this task should be documented.

What is an authentication provider?

Authentication providers **define users, groups, and roles used for authentication**. User names, IDs, passwords, regional settings, personal preferences are some examples of information stored in the providers.

**1. Overview**

In this tutorial, we'll learn how to set up an **Authentication Provider in Spring Security,** allowing for additional flexibility compared to the standard scenario using a simple *UserDetailsService*.

**2. The Authentication Provider**

Spring Security provides a variety of options for performing authentication. These options follow a simple contract; **an *Authentication* request is processed by an *AuthenticationProvider,*** and a fully authenticated object with full credentials is returned.

The standard and most common implementation is the *DaoAuthenticationProvider,* which retrieves the user details from a simple, read-only user DAO, the *UserDetailsService*. This User Details Service **only has access to the username** in order to retrieve the full user entity, which is enough for most scenarios.

More custom scenarios will still need to access the full *Authentication* request to be able to perform the authentication process. For example, when authenticating against some external, third party service (such as [Crowd](https://www.atlassian.com/software/crowd)), **both the *username* and *password* from the authentication request will be necessary**.

For these more advanced scenarios, we'll need to **define a custom Authentication Provider**:

**CustomAuthenticationProvider.java**

**package** com.SpringSecurityAuth;

**import** org.springframework.security.authentication.AuthenticationProvider;

**import** org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

**import** org.springframework.security.core.Authentication;

**import** org.springframework.security.core.AuthenticationException;

**import** org.springframework.stereotype.Component;

**import** java.util.ArrayList;

@Component

**public** **class** CustomAuthenticationProvider **implements** AuthenticationProvider {

@Override

**public** Authentication authenticate(Authentication authentication)

**throws** AuthenticationException {

String name = authentication.getName();

String password = authentication.getCredentials().toString();

**if** (shouldAuthenticateAgainstThirdPartySystem()) {

// use the credentials

// and authenticate against the third-party system

**return** **new** UsernamePasswordAuthenticationToken(

name, password, **new** ArrayList<>());

} **else** {

**return** **null**;

}

}

**private** **boolean** shouldAuthenticateAgainstThirdPartySystem() {

// **TODO** Auto-generated method stub

**return** **false**;

}

@Override

**public** **boolean** supports(Class<?> authentication) {

**return** authentication.equals(UsernamePasswordAuthenticationToken.**class**);

}

}

Notice that the granted authorities set on the returned *Authentication* object are empty. This is because authorities are, of course, application specific.

**3. Register the Auth Provider**

Now that we've defined the Authentication Provider, we need to specify it in the XML Security Configuration using the available namespace support:

<**http** use-expressions="true">

<**intercept-url** pattern="/\*\*" access="isAuthenticated()"/>

<**http-basic**/>

</**http**>

<**authentication-manager**>

<**authentication-provider**

ref="customAuthenticationProvider" />

</**authentication-manager**>

**4. Java Configuration**

Next, we'll take a look at the corresponding Java configuration:

**SpringSecurityAuthApplication.java**

**package** com.SpringSecurityAuth;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

**public** **class** SpringSecurityAuthApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringSecurityAuthApplication.**class**, args);

}

}

**ServletInitializer.java**

**package** com.SpringSecurityAuth;

**import** org.springframework.boot.builder.SpringApplicationBuilder;

**import** org.springframework.boot.web.servlet.support.SpringBootServletInitializer;

**public** **class** ServletInitializer **extends** SpringBootServletInitializer {

@Override

**protected** SpringApplicationBuilder configure(SpringApplicationBuilder application) {

**return** application.sources(SpringSecurityAuthApplication.**class**);

}

}

**DemoController.java**

**package** com.SpringSecurityAuth;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.ResponseBody;

//Authorization

@Controller

**public** **class** DemoController {

@ResponseBody

@RequestMapping("/")

**public** String all() {

**return** "all simplilearn can access";

}

@ResponseBody

@RequestMapping("/admin")

**public** String admin() {

**return** "hi admins";

}

@ResponseBody

@RequestMapping("/user")

**public** String user() {

**return** "hi simplilearn users";

}

@ResponseBody

@RequestMapping("/user1")

**public** String user1() {

**return** "hi simplilearn users";

}

}

**AppSecurityConfig.java**

**package** com.SpringSecurityAuth;

**import** org.springframework.beans.factory.annotation.Autowired;

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

**import** org.springframework.context.annotation.ComponentScan;

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

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

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

**import** org.springframework.security.config.annotation.web.configuration.~~WebSecurityConfigurerAdapter~~;

**import** org.springframework.security.crypto.password.~~NoOpPasswordEncoder~~;

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

@Configuration

@EnableWebSecurity

@ComponentScan("com.SpringSecurityAuth")

**public** **class** AppSecurityConfig **extends** ~~WebSecurityConfigurerAdapter~~ {

@Autowired

**private** CustomAuthenticationProvider authProvider;

**protected** **void** configure1(AuthenticationManagerBuilder auth) **throws** Exception {

auth.authenticationProvider(authProvider);

}

**protected** **void** configure1(HttpSecurity http) **throws** Exception {

http.authorizeRequests().anyRequest().authenticated()

.and().httpBasic();

}

//authentication

@Override

**protected** **void** configure(AuthenticationManagerBuilder auth) **throws** Exception {

// chain of config

auth.inMemoryAuthentication()

.withUser("admin")

.password("admin")

.roles("ADMIN")

.and()

.withUser("user")

.password("user")

.roles("USER")

.and()

.withUser("user1")

.password("user1")

.roles("USER");

}

@Bean

**public** PasswordEncoder getpassword() {

//dont change the password

**return** ~~NoOpPasswordEncoder~~.~~getInstance~~();

}

//authorization

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http.authorizeRequests()

.antMatchers("/admin").hasRole("ADMIN")

.antMatchers("/user").hasRole("USER")

.antMatchers("/user").hasRole("ADMIN")

.antMatchers("/user1").hasRole("USER")

.antMatchers("/user1").hasRole("ADMIN")

.antMatchers("/").permitAll().and().formLogin().and().httpBasic();

}

}

**SpringSecurityAuthApplicationTests**

**package** com.SpringSecurityAuth;

**import** org.junit.jupiter.api.Test;

**import** org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest

**class** SpringSecurityAuthApplicationTests {

@Test

**void** contextLoads() {

}

}

**SpringSecurityAuth/ *POM.xml***

<?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-instance"*

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>2.7.1</version>

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

</parent>

<groupId>com.SpringSecurityAuth</groupId>

<artifactId>SpringSecurityAuth</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>war</packaging>

<name>SpringSecurityAuth</name>

<description>Demo project for Authentication Provider in Spring Security</description>

<properties>

<java.version>1.8</java.version>

</properties>

<dependencies>

<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>org.springframework.boot</groupId>

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

<scope>provided</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>

</dependencies>

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

</build>

</project>

**5. Performing Authentication**

Requesting Authentication from the Client is basically the same with or without this custom authentication provider on the back end.

We'll use a simple *curl* command to send an authenticated request:

curl --header "Accept:application/json" -i --user user1:user1Pass

http://localhost:8080/ SpringSecurityAuth /api/foo/1

For the purposes of this example, we secured the REST API with Basic Authentication.

And we get back the expected 200 OK from the server:

HTTP/1.1 200 OK

Server: Apache-Tomcat/1.1

Set-Cookie: JSESSIONID=638BBEA8EE81B5257DA285B5B03EF962; Path=/SpringSecurityAuth/; HttpOnly

Content-Type: application/json;charset=UTF-8

Transfer-Encoding: chunked

Date: Tue, 29 Jun 2022 06:18:35 GMT

**6. Conclusion**

In this article, we explored an example of a custom authentication provider for Spring Security.

**7. Pushing the code to your GitHub repositories:**

* Open your command prompt and navigate to the folder where you have created your files.

**cd <folder path>**

* Initialize your repository using the following command:

**git init**

* Add all the files to your git repository using the following command:

**git add .**

* Commit the changes using the following command:

**git commit . -m “Changes have been committed.”**

* Push the files to the folder you initially created using the following command:

**git push -u origin master**