#### Add Spring Security Configuration Class

The first and foremost step to add spring security in our application is to create **Spring Security Java Configuration**. This configuration creates a Servlet Filter known as the springSecurityFilterChain which is responsible for all the security (protecting the application URLs, validating submitted username and passwords, redirecting to the log in form, etc) within our application.

@Configuration

@EnableWebSecurity

**public** **class** SecurityConfiguration **extends** WebSecurityConfigurerAdapter {

@Autowired

**public** **void** configureGlobalSecurity(AuthenticationManagerBuilder auth)

**throws** Exception {

auth.inMemoryAuthentication()

.withUser("user").password("password")

.roles("USER");

auth.inMemoryAuthentication()

.withUser("admin").password("password")

.roles("ADMIN");

auth.inMemoryAuthentication()

.withUser("dba").password("password")

.roles("ADMIN", "DBA");

}

@Override

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

http.authorizeRequests()

.antMatchers("/", "/home").permitAll()

.antMatchers("/admin/\*\*").access("hasRole('ADMIN')")

.antMatchers("/db/\*\*").access("hasRole('ADMIN') and hasRole('DBA')")

.and()

.formLogin()

.and()

.exceptionHandling().accessDeniedPage("/Access\_Denied");

}

}

Method configureGlobalSecurity in above class configures AuthenticationManagerBuilder with user credentials and allowed roles. This AuthenticationManagerBuilder creates AuthenticationManager which is responsible for processing any authentication request. Notice that in above example, we have used in-memory authentication while you are free to choose from JDBC, LDAP and other authentications.

The overridden Method Configure configures HttpSecurity which allows configuring web based security for specific http requests. By default it will be applied to all requests, but can be restricted using requestMatcher(RequestMatcher)/antMathchers or other similar methods.

In above configuration, we say that URL’s ‘/’ & ‘/home’ are not secured, anyone can access them. URL ‘/admin/\*\*’ can only be accessed by someone who have ADMIN role. URL ‘/db/\*\*’ can only be accessed by someone who have both ADMIN and DBA roles.

Method formLogin provides support for form based authentication and will generate a default form asking for user credentials. You are allowed to configure your own login form.We will see examples for the same in subsequent posts.

We have also used exceptionHandling().accessDeniedPage() which in this case will catch all 403 [http access denied] exceptions and display our user defined page instead of showing default HTTP 403 page [ which is not so helpful anyway].

**Above security configuration in XML configuration format would be:**

<http auto-config=*"true"*>

<intercept-url pattern=*"/"* access=*"permitAll"* />

<intercept-url pattern=*"/home"* access=*"permitAll"* />

<intercept-url pattern=*"/admin\*\*"* access=*"hasRole('ADMIN')"* />

<intercept-url pattern=*"/dba\*\*"*

access=*"hasRole('ADMIN') and hasRole('DBA')"* />

<form-login authentication-failure-url=*"/Access\_Denied"* />

</http>

<authentication-manager>

<authentication-provider>

<user-service>

<user name=*"user"* password=*"password"* authorities=*"ROLE\_USER"* />

<user name=*"admin"* password=*"password"* authorities=*"ROLE\_ADMIN"* />

<user name=*"dba"* password=*"password"* authorities=*"ROLE\_ADMIN,ROLE\_DBA"* />

</user-service>

</authentication-provider>

</authentication-manager>

Below specified initializer class registers the springSecurityFilter with application war

**public** **class** SecurityInitializer **extends** AbstractSecurityWebApplicationInitializer {

}

**Above setup in XML configuration format would be:**

<filter>

<filter-name>springSecurityFilterChain</filter-name>

<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>

</filter>

<filter-mapping>

<filter-name>springSecurityFilterChain</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

@Controller

**public** **class** HelloWorldController {

@RequestMapping(value = { "/", "/home" }, method = RequestMethod.***GET***)

**public** String homePage(ModelMap model) {

model.addAttribute("greeting", "Hi, Welcome to my application. ");

**return** "welcome";

}

@RequestMapping(value = "/admin", method = RequestMethod.***GET***)

**public** String adminPage(ModelMap model) {

model.addAttribute("user", getPrincipal());

**return** "admin";

}

@RequestMapping(value = "/db", method = RequestMethod.***GET***)

**public** String dbaPage(ModelMap model) {

model.addAttribute("user", getPrincipal());

**return** "dba";

}

@RequestMapping(value="/logout", method = RequestMethod.***GET***)

**public** String logoutPage (HttpServletRequest request, HttpServletResponse response) {

Authentication auth = SecurityContextHolder.*getContext*().getAuthentication();

**if** (auth != **null**){

**new** SecurityContextLogoutHandler().logout(request, response, auth);

}

**return** "welcome";

}

@RequestMapping(value = "/Access\_Denied", method = RequestMethod.***GET***)

**public** String accessDeniedPage(ModelMap model) {

model.addAttribute("user", getPrincipal());

**return** "accessDenied";

}

**private** String getPrincipal(){

String userName = **null**;

Object principal = SecurityContextHolder.*getContext*().getAuthentication().getPrincipal();

**if** (principal **instanceof** UserDetails) {

userName = ((UserDetails)principal).getUsername();

} **else** {

userName = principal.toString();

}

**return** userName;

}

}

Methods in controller class are trivial. Method getPrincipal is a generic function which returns the logged in user name from Spring SecurityContext. Method logoutPage handles the logging out with a simple call to**SecurityContextLogoutHandler().logout(request, response, auth);**. It’s handy and saves you from putting cryptic logout logic in your JSP’s which is not really manageable. You might have noticed that ‘/login’ is missing, it is because it will be generated and handled by default by Spring Security.

#### Add SpringMVC Configuration Class

@Configuration

@EnableWebMvc

@ComponentScan(basePackages = "in.spring4buddies.application")

**public** **class** WebApplicationConfiguration {

@Bean(name = "HelloWorld")

**public** ViewResolver viewResolver() {

InternalResourceViewResolver viewResolver = **new** InternalResourceViewResolver();

viewResolver.setViewClass(JstlView.**class**);

viewResolver.setPrefix("/WEB-INF/views/");

viewResolver.setSuffix(".jsp");

**return** viewResolver;

}

}

#### Add Initializer class

**public** **class** WebApplicationInitializer **extends** AbstractAnnotationConfigDispatcherServletInitializer {

@Override

**protected** Class<?>[] getRootConfigClasses() {

**return** **new** Class[] { WebApplicationConfiguration.**class** };

}

@Override

**protected** Class<?>[] getServletConfigClasses() {

**return** **null**;

}

@Override

**protected** String[] getServletMappings() {

**return** **new** String[] { "/" };

}

}

Notice that above initializer class extends AbstractAnnotationConfigDispatcherServletInitializerwhich is the base class for all WebApplicationInitializer implementations. Implementations of WebApplicationInitializer configures ServletContext programatically, for Servlet 3.0 environments. It means we won’t be using web.xml and we will deploy the app on Servlet 3.0 container.

welcome.jsp

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"* pageEncoding=*"ISO-8859-1"*%>

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>

<title>HelloWorld page</title>

</head>

<body>

Greeting : ${greeting}

This is a welcome page.

</body>

</html>

admin.jsp

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"* pageEncoding=*"ISO-8859-1"*%>

<%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"*%>

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>

<title>HelloWorld Admin page</title>

</head>

<body>

Dear <strong>${user}</strong>, Welcome to Admin Page.

<a href=*"*<c:url value=*"/logout"* />*"*>Logout</a>

</body>

</html>

dba.jsp

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"* pageEncoding=*"ISO-8859-1"*%>

<%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"*%>

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>

<title>DBA page</title>

</head>

<body>

Dear <strong>${user}</strong>, Welcome to DBA Page.

<a href=*"*<c:url value=*"/logout"* />*"*>Logout</a>

</body>

</html>

accessDenied.jsp

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"* pageEncoding=*"ISO-8859-1"*%>

<%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"*%>

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>

<title>AccessDenied page</title>

</head>

<body>

Dear <strong>${user}</strong>, You are not authorized to access this page

<a href=*"*<c:url value=*"/logout"* />*"*>Logout</a>

</body>

</html>

Mvn clean install - > mvn jetty:run

* localhost:8080/ sf-security-default-login-form/
* Now try to access admin page on localhost:8080/ sf-security-default-login-form/admin, you will be prompted for login.
  + Provide credentials of a ‘USER’ role. Submit, you will see AccessDenied Page
  + Now logout and try to access admin page again, Provide wrong password,

we have seen the default login form provided by Spring Security in case we don’t specify one. In this post, we will create our own Custom login form. Basically, the idea is, in Security Configuration, attach a call to **loginPage(URL)** function with **formLogin()** like shown below

.and().formLogin().loginPage("/login")

And then, Map this ‘/login’ URL in your Spring MVC Controller which will return the login view defined by you. Now, on login attempt, the specified login view will be displayed.Rest of the login functionality remains same. Below provided is complete example for this scenario.

.and().formLogin().loginPage("/login")

        .usernameParameter("ssoId").passwordParameter("password")

        .and().csrf()

This code creates a custom login page with ‘/login’ url, which will accept ssoId as username and password Http request parameters. We have also shown a call to **csrf()** which is optional as it is by default active in Spring Security 4. This call is, however, required if you want to disable CSRF protection by using **csrf().disable()** although it is not a good idea to disable it.

<http auto-config=*"true"*>

<intercept-url pattern=*"/"* access=*"permitAll"* />

<intercept-url pattern=*"/home"* access=*"permitAll"* />

<intercept-url pattern=*"/admin\*\*"* access=*"hasRole('ADMIN')"* />

<intercept-url pattern=*"/dba\*\*"*

access=*"hasRole('ADMIN') and hasRole('DBA')"* />

  <form-login  login-page="/login" username-parameter="ssoId" password-parameter="password" authentication-failure-url="/Access\_Denied" />

        <csrf/>

</http>

<authentication-manager>

<authentication-provider>

<user-service>

<user name=*"user"* password=*"password"* authorities=*"ROLE\_USER"* />

<user name=*"admin"* password=*"password"* authorities=*"ROLE\_ADMIN"* />

<user name=*"dba"* password=*"password"* authorities=*"ROLE\_ADMIN,ROLE\_DBA"* />

</user-service>

</authentication-provider>

</authentication-manager>

only changes are new loginPage method to handle ‘/login’ requests and adapting logout to redirect to login page on logout, as shown below:

@RequestMapping(value = "/login", method = RequestMethod.***GET***)

**public** String loginPage() {

**return** "login";

}

@RequestMapping(value="/logout", method = RequestMethod.***GET***)

**public** String logoutPage (HttpServletRequest request, HttpServletResponse response) {

Authentication auth = SecurityContextHolder.*getContext*().getAuthentication();

**if** (auth != **null**){

**new** SecurityContextLogoutHandler().logout(request, response, auth);

}

**return** "redirect:/login?logout";

}

WebApplicationConfiguration class extend from **WebMvcConfigurerAdapter** [just a convenience class] and implementing method **addResourceHandlers** which handles static resources(CSS/images/..) to be used in views.

@Override

**public** **void** addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/static/\*\*").addResourceLocations( "/static/");

}

Notice the CSRF related line in above jsp:

<input type="hidden" name="${\_csrf.parameterName}" value="${\_csrf.token}" /></strong>

This is required to protect against CSRF attacks. As you can see, the CSRF parameters are accessed using EL Expressions in your JSP, you may additionally prefer to force EL expressions to be evaluated, by adding following to the top of your JSP:

<%@ page isELIgnored="false"%>

# Secure View Fragments using taglibs

This tutorial shows you how to secure view layer, show/hide parts of jsp/view based on logged-in user’s roles, using Spring Security tags in Spring MVC web application.

First of all, in order to use Spring Security tags, we need to include spring-security-taglibs dependency in pom.xml as shown below:

<dependency>

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

    <artifactId>spring-security-taglibs</artifactId>

    <version>4.0.1.RELEASE</version>

</dependency>

Then the next step would be to include taglib in your views/JSP’s.

|  |
| --- |
| <%@ taglib prefix="sec" uri="<http://www.springframework.org/security/tags>"%>  Finally, we can use Spring Security expresssions like hasRole, hasAnyRole, etc.. in Views as shown below: |