Java

Learn Spring framework:

- Understand the core of Spring
- Understand Spring MVC
- Understand Spring AOP
- Understand Spring Data JPA
- Spring Dependency Injection (XML)
- Spring Dependency Injection (Annotations)

Spring Web MVC Security Basic Example Part 2 with Java-based Configuration

Written by Nam Ha Minh
Last Updated on 24 June 2019 | Print   Email

Welcome to part 2 of Spring Web MVC Security tutorial. In the first part, we showed you how to secure a Spring Web MVC application using XML configuration. In this second part, we are going to show you develop the same application as part 1, but use Java configuration instead. No XML at all, even there is no `web.xml` file.

To understand the basics of Spring Security framework and how the sample application works, please refer to part 1:

Spring Web MVC Security Basic Example Part 1 with XML Configuration

Now, let's discover how to use annotations and Java configuration to secure a Spring Web MVC application.

1. Project Setup

In this tutorial, we use Eclipse IDE to create a dynamic web project, and then convert it to Maven project.

The Spring MVC Security Java Config project is developed using the following pieces of technologies (of course you can use newer versions):

- Java 8.
- Tomcat 8 with Servlet 3.1.
- Spring framework 4.2.4.RELEASE.
- Spring Security 4.0.3.RELEASE.
- JSTL 1.2
- Eclipse IDE, Mars Release (4.5.0).

Maven Note: Configuration for Maven dependencies is as same as the first part.

Remember the dependencies for Spring Security as below:

```
   .    <!-- Spring Security Web -->
   2    <dependency>
   3        <groupId>org.springframework.security</groupId>
   4        <artifactId>spring-security-web</artifactId>
   5        <version>${spring.security.version}</version>
   6    </dependency>
   7
   8
   9    <!-- Spring Security Config -->
  10    <dependency>
  11        <groupId>org.springframework.security</groupId>
  12        <artifactId>spring-security-config</artifactId>
  13        <version>${spring.security.version}</version>
  14    </dependency>
```

To see the full Maven dependencies, please refer to part 1: Spring Web MVC Security Basic Example Part 1 with XML Configuration.

2. Coding Index Page

Make a directory called `views` under the `/WEB-INF` directory, then create an `index.jsp` file with the following HTML code:

```
 1    <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
 2        pageEncoding="ISO-8859-1"%>
 3    <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 4        "http://www.w3.org/TR/html4/loose.dtd">
 5    <html>
 6    <head>
 7    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
 8    <title>Spring Security Basic Demo (Java Config)</title>
 9    </head>
10    <body>
11        <div align="center">
12            <h1>Spring Security Basic Demo (Java Config)</h1>
13            <a href="/admin">Go to Administrator Page</a>
14        </div>
15    </body>
16    </html>
```

As you can see, this is very simple page with a heading "*Spring Security Basic Demo (Java Config)*" and a hyperlink to the administrator page which requires authentication to access.

3. Coding Admin Page

Next, create an `admin.jsp` file under the `/WEB-INF/views` directory with the following code:

```jsp
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ page language="java" session="true" contentType="text/html; charset=ISO-8
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Spring Security Basic Demo (Java Config)</title>
</head>
<body>
    <div align="center">
        <h1>${title}</h1>
        <h2>${message}</h2>
        <c:if test="${pageContext.request.userPrincipal.name != null}">
            <h2>Welcome : ${pageContext.request.userPrincipal.name} |
                <a href="/<c:url value="/logout" />" > Logout</a></h2>
        </c:if>
    </div>
</body>
</html>
```

This is the administrator page which requires authentication and authorization to access. We use JSTL expressions to display the title and message in the model. If the user is logged in, display his username along with a logout link.

4. Coding Spring MVC Controller

Next, we write code for a Spring controller in order to handle requests coming to the application. Create a Java class named `MainController` under the package `net.codejava.spring` with the following code:

```java
package net.codejava.spring;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

/**
 * Spring Web MVC Security Java Config Demo Project
 * Primary controller of the application.
 *
 * @author www.codejava.net
 *
 */
@Controller
public class MainController {

    @RequestMapping(value="/", method = RequestMethod.GET)
    public ModelAndView visitHome() {
        return new ModelAndView("index");
    }

    @RequestMapping(value="/admin", method = RequestMethod.GET)
    public ModelAndView visitAdmin() {
        ModelAndView model = new ModelAndView("admin");
        model.addObject("title", "Admministrator Control Panel");
        model.addObject("message", "This page demonstrates how to use Spring

        return model;
    }
}
```

As you can see, this controller is designed to handle 2 requests:

- **"/"**: the request to the application's context root, i.e. the home page.
- **"/admin"**: the request to the administrator page, which will be secured by Spring security.

The annotations **@Controller** and **@RequestMapping** are used to declare this is a controller which has two HTTP GET handle methods. These annotations will be scanned by Spring as we will configure in the Spring's application context file.

5. Configuring Spring Dispatcher Servlet

Instead of using XML configuration in the **web.xml** file as usual, here we write Java code.

Create a Java file called **SpringWebAppInitializer** with the following content:

```java
package net.codejava.spring;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.support.AnnotationConfigWebApplication
import org.springframework.web.servlet.DispatcherServlet;

/**
 * Spring Web MVC Security Java Config Demo Project
 * Bootstraps Spring Dispacher Servlet in Servlet 3.0+ environment.
 *
 * @author www.codejava.net
 *
 */
public class SpringWebAppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletExcept
        AnnotationConfigWebApplicationContext appContext = new AnnotationCont
        appContext.register(MvcConfig.class);

        ServletRegistration.Dynamic dispatcher = servletContext.addServlet(
                "SpringDispatcher", new DispatcherServlet(appContext));
        dispatcher.setLoadOnStartup(1);
        dispatcher.addMapping("/");

    }

}
```

Here, the **SpringWebAppInitializer** is a type of `WebApplicationInitializer` which is bootstrapped automatically by any Servlet 3.0+ container (See more: Bootstrapping a Spring Web MVC application programmatically).

The above code is equivalent to the following XML snippet in the web.xml:

```xml
<servlet>
    <servlet-name>SpringController</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring-mvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>SpringController</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

6. Configuring Spring Security Filter

Spring Security Filter is a servlet filter that intercepts requests for securing the application, so we need to load it upon application's start up.

ad of using XML, here's Java code to load the Spring Security Filter:

```java
1    package net.codejava.spring;
2
3    import org.springframework.security.web.context.AbstractSecurityWebApplicati
4
5    /**
6     * Spring Web MVC Security Java Config Demo Project
7     * Bootstraps Spring Security Filter.
8     *
9     * @author www.codejava.net
10    *
11    */
12   public class SecurityWebApplicationInitializer extends AbstractSecurityWebApp
13
14       public SecurityWebApplicationInitializer() {
15           super(SecurityConfig.class);
16       }
17
18   }
```

It's equivalent to the following XML snippet in the **web.xml** file:

```xml
1    <filter>
2        <filter-name>springSecurityFilterChain</filter-name>
3        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filte
4    </filter>
5
6    <filter-mapping>
7        <filter-name>springSecurityFilterChain</filter-name>
8        <url-pattern>/*</url-pattern>
9    </filter-mapping>
```

Note that the **SecurityWebApplicationInitializer** class passes **SecurityConfig.class** to its super's constructor, which tells Spring Security looks for configuration in the class named **SecurityConfig** - which is described below.

7. Configuring Authentication and Authorization for the Spring MVC application

Create a class named **SecurityConfig** that extends the **WebSecurityConfigurerAdapter** with the following code:

```java
package net.codejava.spring;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.builders
import org.springframework.security.config.annotation.web.builders.HttpSecur
import org.springframework.security.config.annotation.web.configuration.Enab
import org.springframework.security.config.annotation.web.configuration.WebSe

/**
 * Spring Web MVC Security Java Config Demo Project
 * Configures authentication and authorization for the application.
 *
 * @author www.codejava.net
 *
 */
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exc
        auth
            .inMemoryAuthentication()
                .withUser("admin").password("nimda").roles("ADMIN");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        http.authorizeRequests()
            .antMatchers("/").permitAll()
            .antMatchers("/admin**").access("hasRole('ADMIN')")
            .and().formLogin();

        http.csrf().disable();
    }
}
```

This is equivalent to the following configuration in XML:

```xml
<http auto-config="true">
    <intercept-url pattern="/admin**" access="hasRole('ADMIN')" />
    <csrf disabled="true" />
</http>

<authentication-manager>
    <authentication-provider>
        <user-service>
            <user name="admin" password="nimda" authorities="ADMIN" />
        </user-service>
    </authentication-provider>
</authentication-manager>
```

Note that we disable CSRF for the sake of simplicity in this tutorial:

```java
http.csrf().disable();
```

That tells Spring Security to intercept the *logout* link as an HTTP GET request. In production, you should enable CSRF for best security practice.

Instead of using XML, we configure the view resolvers in Spring MVC using Java code like this:

```
package net.codejava.spring;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

/**
 * Spring Web MVC Security Java Config Demo Project
 * Configures Spring MVC stuffs.
 *
 * @author www.codejava.net
 *
 */
@Configuration
@ComponentScan("net.codejava.spring")
public class MvcConfig {

    @Bean(name = "viewResolver")
    public InternalResourceViewResolver getViewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewF
        viewResolver.setPrefix("/WEB-INF/views/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
}
```

As you can see, we configure this `InternalResourceViewResolver`bean to resolve logical view names to physical JSP pages. It's equivalent to the following XML:

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
</bean>
```

9. The Whole Project Structure

Lastly, the project's structure would look like this in Eclipse IDE:
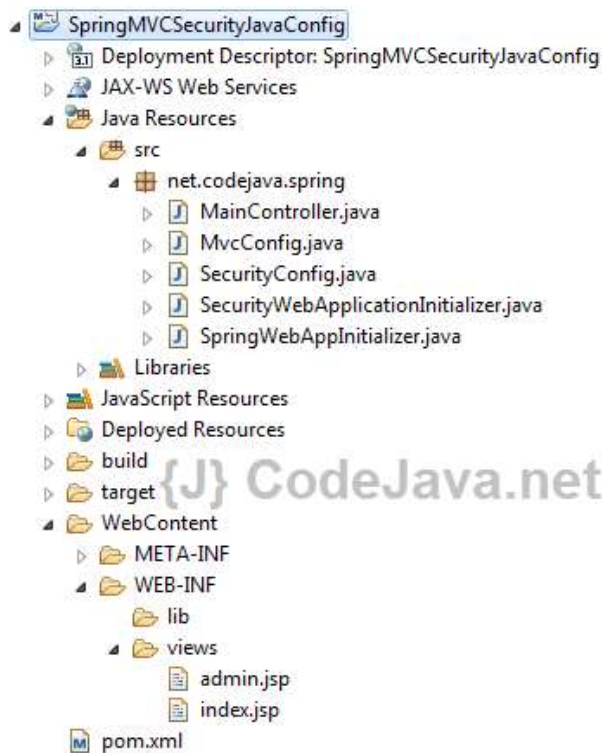
NOTE: For you convenience, we provide full source code of this Spring MVC Security Java Config Project (Eclipse-Maven) in the Download Attachments section.

10. Test the Spring Security Sample Application

It's time to test the Spring MVC Security application we've built so far. After deploying the project on Tomcat, type the following URL in your browser:

> ***http://localhost:8080/SpringMVCSecurityJavaConfig***

The home page appears:

Click on the hyperlink **Go to Administrator Page** (it's equivalent to type the URL: ***http://localhost:8080/SpringMVCSecurityJavaConfig/admin***), Spring automatically generates a login page looks like this:

First, let's type an invalid username and password, Spring security automatically generates an error message in red at the top of the form, which is looking like this:

Next, let's type the correct username and password (**admin** and **nimda**, as per configured in this application), Spring security authenticates the login and returns the administrator page as follows:

Awesome! Now, let's test the logout functionality. Click on the **Logout** link, Spring security filter automatically intercepts the*logout* URL, invalidates the session and take you to the login page again:
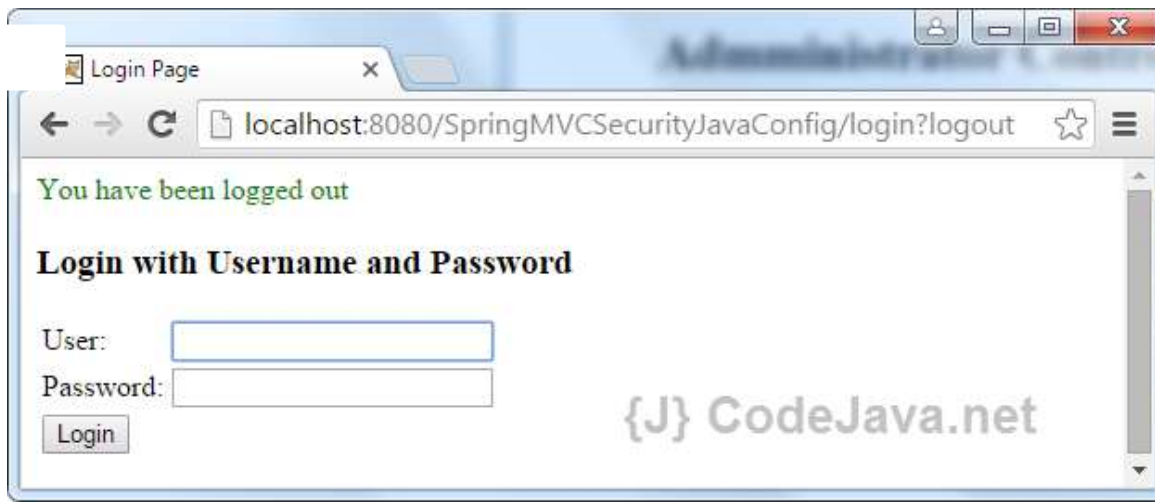
Notice the message 'You have been logged out' is displayed in green at the top of the form, and the URL got changed to **http://localhost:8080/SpringMVCSecurityJavaConfig/login?logout**

To make sure Spring security already invalidated the session, let's try to access the page **http://localhost:8080/SpringMVCSecurityJavaConfig/admin** again, we will see the login page. It's working seamlessly!

Perfect! We've done a basic Spring Web MVC Security project with Java configuration.

References:

- Spring Security Reference (4.0.3.RELEASE)
- Spring Security Project

Related Spring Security Tutorials:

- Spring Web MVC Security Basic Example Part 1 with XML Configuration

Other Spring Tutorials:

- Understand the core of Spring framework
- Understand Spring MVC
- Understand Spring AOP
- Spring MVC beginner tutorial with Spring Tool Suite IDE
- Spring MVC Form Handling Tutorial
- Spring MVC Form Validation Tutorial
- 14 Tips for Writing Spring MVC Controller
- Spring Web MVC Security Basic Example (XML Configuration)
- Understand Spring Data JPA with Simple Example

About the Author:

Nam Ha Minh is certified Java programmer (SCJP and SCWCD). He started programming with Java in the time of Java 1.4 and has been falling in love with Java since then. Make

friend with him on Facebook.

Attachments:

📦 **SpringMVCSecurityJavaConfig.zip**	**[Spring Web MVC Security Java Config Eclipse-Maven Project]**	**22 kB**

Add comment

Name	E-mail

```
comment



```
500 symbols left

☐ Notify me of follow-up comments

☐ I'm not a robot

reCAPTCHA
Privacy - Terms

Send

Comments

#1 molly   2017-04-13 05:15

Hi,

I download this project. When I attempted to access admin page and login as admin. It redirects to home page not admin page. How to solve it?

Quote

Refresh comments list