Spring Security 4 - Hello World example

Spring Security framework is a powerful and highly customizable authentication and authorization (access-control) framework for Java EE based applications.

Authentication is the process of verifying the user identity. Whereas authorization is the process of verifying that user has access to resources.

In this post, I will show you how to integrate a spring MVC application with Spring Security framework for securing the URLs access with authentication.

Tools and technologies used for this application are-

* Spring Security 4.2.3.RELEASE
* Spring MVC 4.3.10.RELEASE
* Java SE 1.8
* Maven 3.3.9
* Eclipse Neon.3
* Apache Tomcat 7.0.47

Let’s begin step by step and see how to secure Spring web application with Spring Security.

### Step 1 - Create maven project

Create a maven project for our Spring MVC + Spring Security application in Eclipse IDE.

### Step 2 - Add jar dependencies

To integrate the Spring security in Spring MVC application, you need to add the spring-security-web, spring-security-config and spring-security-taglibs jar dependencies in your pom.xml file.

Open pom.xml file and write the following code in it.

**pom.xml**

<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 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.boraji.tutorial.spring</groupId>

<artifactId>spring-security-hello-world-example</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>war</packaging>

<properties>

<failOnMissingWebXml>false</failOnMissingWebXml>

</properties>

<dependencies>

*<!-- Spring MVC Dependency -->*

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.3.10.RELEASE</version>

</dependency>

*<!-- Spring Security Dependency -->*

<dependency>

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

<artifactId>spring-security-web</artifactId>

<version>4.2.3.RELEASE</version>

</dependency>

<dependency>

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

<artifactId>spring-security-config</artifactId>

<version>4.2.3.RELEASE</version>

</dependency>

<dependency>

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

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

<version>4.2.3.RELEASE</version>

</dependency>

*<!-- Servlet Dependency -->*

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.1.0</version>

<scope>provided</scope>

</dependency>

*<!-- JSP Dependency -->*

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>javax.servlet.jsp-api</artifactId>

<version>2.3.1</version>

<scope>provided</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.5.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

*<!-- Embedded Apache Tomcat required for testing web application -->*

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<version>2.2</version>

<configuration>

<path>/</path>

</configuration>

</plugin>

</plugins>

</build>

</project>

### Step 3 - Create controller class

Create a simple @Controller class inside com.boraji.tutorial.spring.controller package as follows.

**UserController.java**

**package** com.boraji.tutorial.spring.controller;

**import** java.security.Principal;

**import** org.springframework.security.core.context.SecurityContext;

**import** org.springframework.security.core.context.SecurityContextHolder;

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

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

@Controller

**public** **class** **UserController** {

@GetMapping("/")

**public** String **index**() {

**return** "index";

}

@GetMapping("/user")

**public** String **user**(Principal principal) {

*// Get authenticated user name from Principal*

System.out.println(principal.getName());

**return** "user";

}

@GetMapping("/admin")

**public** String **admin**() {

*// Get authenticated user name from SecurityContext*

SecurityContext context = SecurityContextHolder.getContext();

System.out.println(context.getAuthentication().getName());

**return** "admin";

}

}

### Step 4 - Create JSP views

Create index.jsp, user.jsp and admin.jsp files under src\main\webapp\WEB-INF\views folder.

**index.jsp**

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

pageEncoding="ISO-8859-1"%>

<%@ taglib uri="http://www.springframework.org/security/tags"

prefix="security"%>

**<!DOCTYPE html >**

<html>

<head>

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

<title>BORAJI.COM</title>

</head>

<body>

<h2>Spring Security 4 - Hello World Example</h2>

<hr />

<h3>

Welcome !

<security:authorize access="isAnonymous()">

Guest

</security:authorize>

*<!-- Print the logged in user name -->*

<security:authorize access="isAuthenticated()">

<security:authentication property="principal.username" />

</security:authorize>

</h3>

<security:authorize access="isAnonymous()">

Login as <a href="user">User</a> or <a href="/admin">Admin</a>

</security:authorize>

<security:authorize access="isAuthenticated()">

<security:authorize access="hasRole('USER')">

<a href="user">My Profile</a>

</security:authorize>

<security:authorize access="hasRole('ADMIN')">

<a href="admin">My Profile</a>

</security:authorize>

<a href="logout">Logout</a>

</security:authorize>

</body>

</html>

**user.jsp**

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

<%@ taglib uri="http://www.springframework.org/security/tags" prefix="security" %>

**<!DOCTYPE html >**

<html>

<head>

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

<title>BORAJI.COM</title>

</head>

<body>

<h2>Spring Security 4 - Hello World Example</h2>

<hr />

<h3>User dashboard </h3>

<security:authorize access="isAuthenticated()">

<b>Welcome! <security:authentication property="principal.username" /></b>

</security:authorize>

<br />

<security:authorize access="isAuthenticated()">

<a href="/">Home</a> | <a href="logout">Logout</a>

</security:authorize>

</body>

</html>

**admin.jsp**

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

pageEncoding="ISO-8859-1"%>

<%@ taglib uri="http://www.springframework.org/security/tags"

prefix="security"%>

**<!DOCTYPE html >**

<html>

<head>

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

<title>BORAJI.COM</title>

</head>

<body>

<h2>Spring Security 4 - Hello World Example</h2>

<hr />

<h3>Admin dashboard</h3>

<security:authorize access="isAuthenticated()">

<b>Welcome! <security:authentication property="principal.username" /></b>

</security:authorize>

<br />

<security:authorize access="isAuthenticated()">

<a href="/">Home</a> | <a href="logout">Logout</a>

</security:authorize>

</body>

</html>

**<security:authorize/>** tag evaluates the access expression, specified in the access attribute, to true for authenticate user. View [Common Built-In Expressions](https://docs.spring.io/spring-security/site/docs/current/reference/html/el-access.html#el-common-built-in) which can be used in access attribute of  the <security:authorize/> tag.

**<security:authentication>** tag is use to access the current Authentication object stored in the security context.

### Step 5 - Create spring security configuration class

The first step is to create a @Configuration class by extending the WebSecurityConfigurerAdapter class as follows.

**WebSecurityConfig.java**

**package** com.boraji.tutorial.security.config;

**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.web.util.matcher.AntPathRequestMatcher;

@EnableWebSecurity

**public** **class** **WebSecurityConfig** **extends** **WebSecurityConfigurerAdapter** {

@Override

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

auth.inMemoryAuthentication()

.withUser("sunil").password("pass123").roles("USER")

.and()

.withUser("admin").password("pass123").roles("ADMIN");

}

@Override

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

http.authorizeRequests().antMatchers("/").permitAll()

.and()

.authorizeRequests().antMatchers("/user\*\*").hasRole("USER")

.and()

.authorizeRequests().antMatchers("/admin\*\*").hasRole("ADMIN")

.and()

.formLogin()

.and()

.logout().logoutRequestMatcher(**new** AntPathRequestMatcher("/logout"));

}

}

This Java configuration class creates a Servlet Filter known as the springSecurityFilterChain, which is responsible for all security (protecting the application URLs, validating submitted username and passwords, redirecting to the log in form, etc) within your application.

The overridden method configure(AuthenticationManagerBuilder auth) configure the in memory authentication with user credentials and roles. You can configure the other authentications too such as JDBC, LDAP etc.

The overridden method configure(HttpSecurity http) configure the web based security for all HTTP request. By default it will be applied to all requests, but can be restricted using the requestMatcher() or other similar methods.

From the above configuration class, it is clear that –

* URL '/'  is not secured and  accessible by everyone.
* Any URLs that starts with '/user' are secured and only accessible by users who have the role 'USER'.
* Any URLs that starts with '/admin' are secured and only accessible by users who have the role 'ADMIN'.

### Step 6 - Register springSecurityFilterChain Filter

In Java configuration, you can register the spring springSecurityFilterChain using the base class AbstractSecurityWebApplicationInitializer as follows.

**SecurityWebApplicationInitializer.java**

**package** com.boraji.tutorial.security.config;

**import** org.springframework.security.web.context.AbstractSecurityWebApplicationInitializer;

**public** **class** **SecurityWebApplicationInitializer**

**extends** **AbstractSecurityWebApplicationInitializer** {

}

This configuration only registers the springSecurityFilterChain Filter for every URL in your application.

Here is the equivalent XML configuration for registering the spring springSecurityFilterChain -

<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>

### Step 7 - Create spring web configuration class

Create a web @Configuration class annotated with @EnableWebMvc and @ComponentScan as follows.

**WebConfig.java**

**package** com.boraji.tutorial.security.config;

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

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

**import** org.springframework.web.servlet.config.annotation.EnableWebMvc;

**import** org.springframework.web.servlet.config.annotation.ViewResolverRegistry;

**import** org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@Configuration

@EnableWebMvc

@ComponentScan(basePackages = { "com.boraji.tutorial.spring.controller" })

**public** **class** **WebConfig** **extends** **WebMvcConfigurerAdapter** {

@Override

**public** **void** **configureViewResolvers**(ViewResolverRegistry registry) {

registry.jsp().prefix("/WEB-INF/views/").suffix(".jsp");

}

}

### Step 8 - Create application initializer class

Create a MvcWebApplicationInitializer class, which will replace our traditional web.xml, to initialize the Servlet container.

Load the WebSecurityConfig and WebConfig classes using the getRootConfigClasses() and getServletConfigClasses() methods as follows.

**MvcWebApplicationInitializer.java**

**package** com.boraji.tutorial.security.config;

**import** org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

**public** **class** **MvcWebApplicationInitializer**

**extends** **AbstractAnnotationConfigDispatcherServletInitializer** {

@Override

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

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

}

@Override

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

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

}

@Override

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

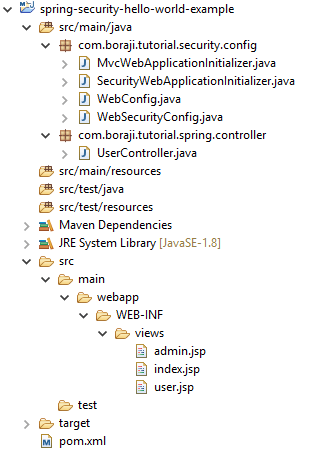
**return** **new** String[] { "/" };

}

}

### Step 9 - Review the final project structure

Here is the final project structure of our Spring MVC + Spring Security application.



### Build + Deploy + Run application

Use the following maven commands to build, deploy and run Tomcat server.

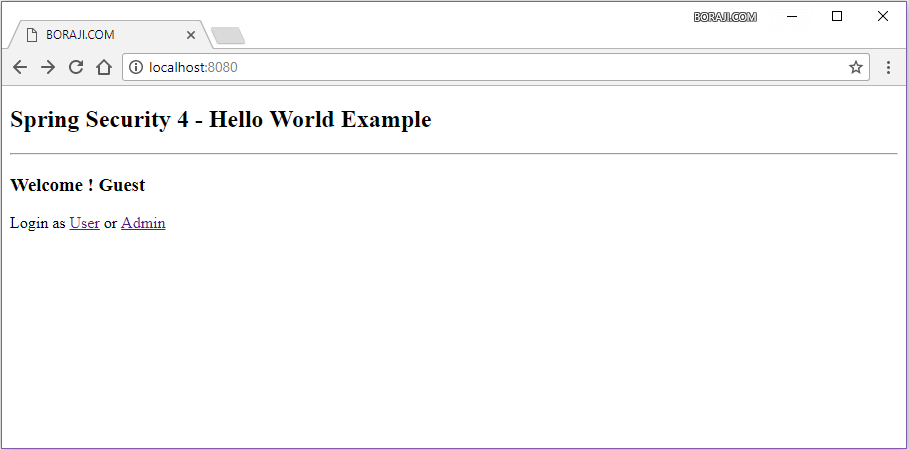
mvn clean install  (This command triggers war packaging)

mvn tomcat7:run (This command run embedded tomcat and deploy war file automatically)

You can refer this [link](https://www.boraji.com/how-to-create-a-web-project-using-maven-in-eclipse#maven-run-tomcat)to learn how to run the above commands in Eclipse IDE.

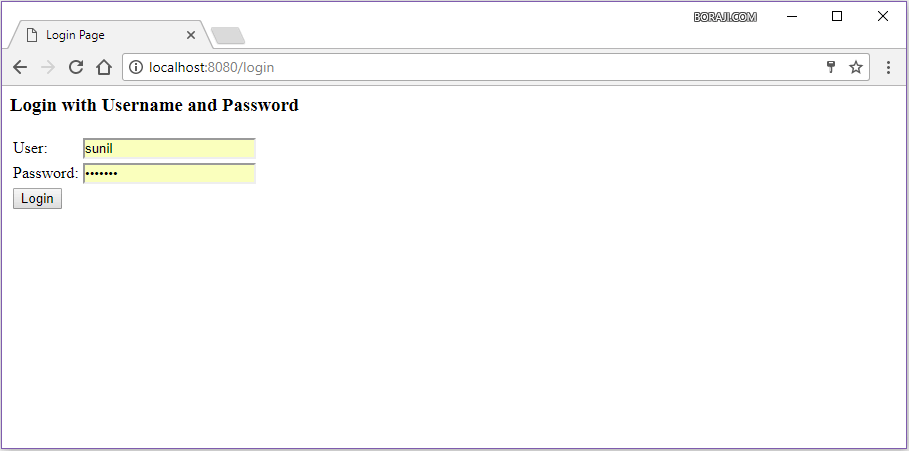
Type the following URL in browser's address bar to test our application.

**http://localhost:8080/**

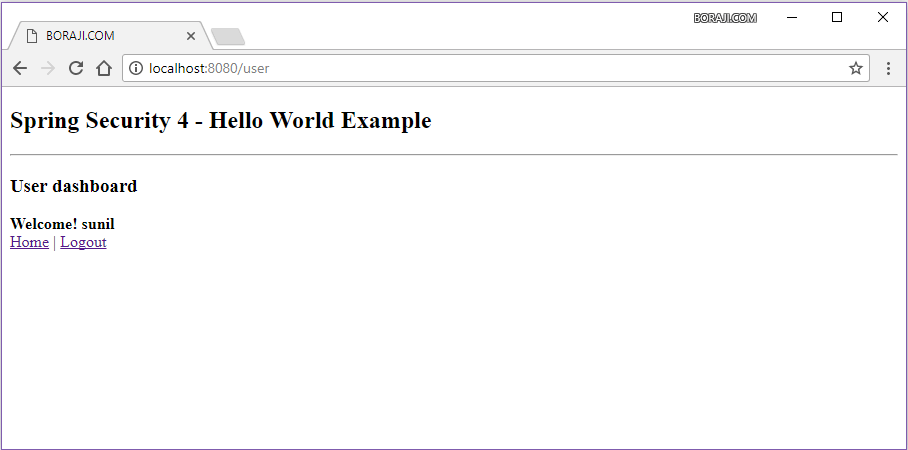


Now click on the **User**link or enter URL**http://localhost:8080/user**  in browser's address bar.

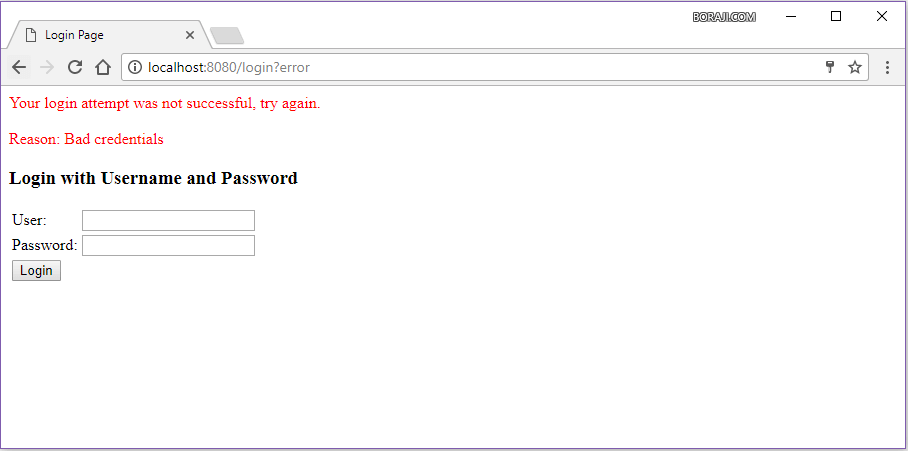
Since, any URL that start with **'/user'** are restricted to users who have the role **'USER'**. So spring security will generate a login page asking for credentials.



On successful login, you will see the user dashboard page as follows.

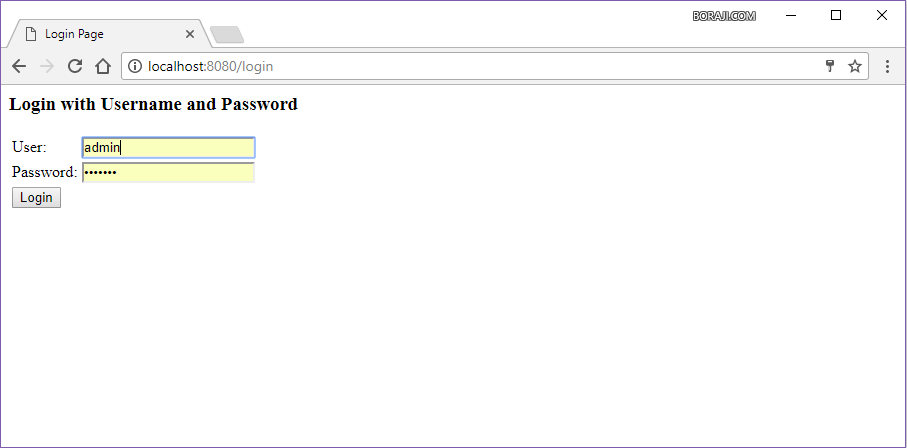


On providing invalid credentials, you see the failure message on login page as follows.

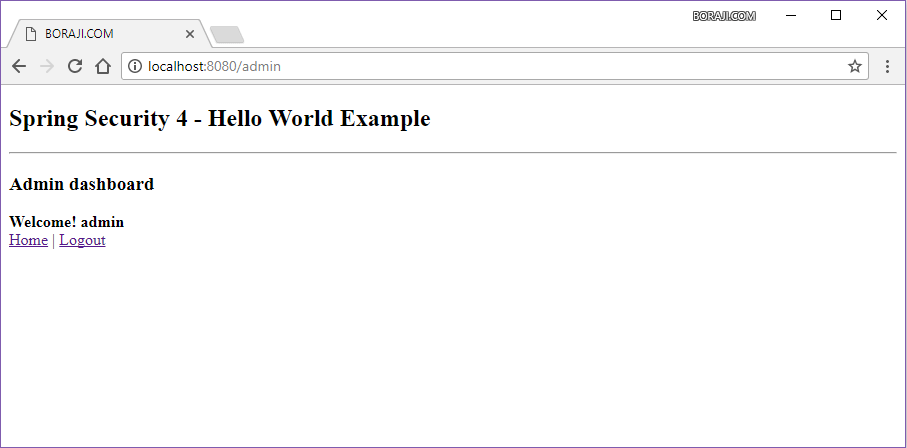


Similarly, click on the ***Admin***link in home page or enter URL**http://localhost:8080/admin**in browser's address bar.

Since, any URL that start with **'/admin'** are restricted to users who have the role**'ADMIN'**. So spring security will generate a login page asking for credentials.



On successful login, you will see the admin dashboard page as follows.



[**Spring Security**](https://www.boraji.com/index.php/category/spring-security)

[**Spring MVC**](https://www.boraji.com/index.php/category/spring-mvc)

**Spring Security 4: JDBC Authentication and Authorization in MySQL**

### I am going to explain how to use Spring Security in a Spring MVC Application to authenticate and authorize users against user details stored in a MySQL Database.

 Run the following statements in mysql server. This sets up the user table and the user\_roles table for us with some initial data to start with.4. Let us get back to our application now. First thing to do is to modify pom.xml file to include spring-jdbc and mysql-connector

CREATE TABLE users (

username VARCHAR(45) NOT NULL ,

password VARCHAR(45) NOT NULL ,

enabled TINYINT NOT NULL DEFAULT 1 ,

PRIMARY KEY (username));

CREATE TABLE user\_roles (

user\_role\_id int(11) NOT NULL AUTO\_INCREMENT,

username varchar(45) NOT NULL,

role varchar(45) NOT NULL,

PRIMARY KEY (user\_role\_id),

UNIQUE KEY uni\_username\_role (role,username),

KEY fk\_username\_idx (username),

CONSTRAINT fk\_username FOREIGN KEY (username) REFERENCES users (username));

INSERT INTO users(username,password,enabled)

VALUES ('priya','priya', true);

INSERT INTO users(username,password,enabled)

VALUES ('naveen','naveen', true);

INSERT INTO user\_roles (username, role)

VALUES ('priya', 'ROLE\_USER');

INSERT INTO user\_roles (username, role)

VALUES ('priya', 'ROLE\_ADMIN');

INSERT INTO user\_roles (username, role)

VALUES ('naveen', 'ROLE\_USER');

Do not ever use plain text for passwords. The right way to do this is to use password encryption.You can find the link to an updated and detailed tutorial below,

[Spring Security JDBC Authentication with Password Encryption](http://www.programming-free.com/2015/09/spring-security-password-encryption.html)

4. Let us get back to our application now. First thing to do is to modify pom.xml file to include spring-jdbc and mysql-connector

### 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 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>org.programmingfree</groupId>

<artifactId>pf-securing-web-jdbc</artifactId>

<version>0.1.0</version>

<packaging>war</packaging>

<parent>

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

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

<version>1.2.2.RELEASE</version>

</parent>

<dependencies>

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-jasper</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

</dependency>

<!-- tag::web[] -->

<dependency>

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

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

</dependency>

<!-- end::web[] -->

<!-- tag::security[] -->

<dependency>

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

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

</dependency>

<!-- end::security[] -->

<!-- JDBC -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

</dependency>

<!-- MySQL -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

</build>

</project>

5. Now we have to provide a definition to the mysql datasource in our MvcConfig class which has all necessary information to connect to the database we created before.

### MvcConfig.java

package hello;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.jdbc.datasource.DriverManagerDataSource;

import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration

public class MvcConfig extends WebMvcConfigurerAdapter {

@Override

public void addViewControllers(ViewControllerRegistry registry) {

registry.addViewController("/home").setViewName("home");

registry.addViewController("/").setViewName("home");

registry.addViewController("/hello").setViewName("hello");

registry.addViewController("/login").setViewName("login");

registry.addViewController("/403").setViewName("403");

}

@Bean(name = "dataSource")

public DriverManagerDataSource dataSource() {

DriverManagerDataSource driverManagerDataSource = new DriverManagerDataSource();

driverManagerDataSource.setDriverClassName("com.mysql.jdbc.Driver");

driverManagerDataSource.setUrl("jdbc:mysql://localhost:3306/userbase");

driverManagerDataSource.setUsername("root");

driverManagerDataSource.setPassword("root");

return driverManagerDataSource;

}

@Bean

public InternalResourceViewResolver viewResolver() {

InternalResourceViewResolver resolver = new InternalResourceViewResolver();

resolver.setPrefix("/WEB-INF/jsp/");

resolver.setSuffix(".jsp");

return resolver;

}

}

Note that I have added one more line to addViewControllers method to register a view for "403" (access denied) page.  This page will be displayed whenever an user tries to access a page he/she is not authorized to.

6. Next we have to modify the security configuration class to use the jdbc datasource we have defined for authenticating and authorize users.

### WebSecurityConfig.java

package hello;

import javax.sql.DataSource;

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

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.WebSecurityConfigurerAdapter;

import org.springframework.security.config.annotation.web.servlet.configuration.EnableWebMvcSecurity;

@Configuration

@EnableWebMvcSecurity

public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

@Autowired

DataSource dataSource;

@Autowired

public void configAuthentication(AuthenticationManagerBuilder auth) throws Exception {

auth.jdbcAuthentication().dataSource(dataSource)

.usersByUsernameQuery(

"select username,password, enabled from users where username=?")

.authoritiesByUsernameQuery(

"select username, role from user\_roles where username=?");

}

@Override

protected void configure(HttpSecurity http) throws Exception {

http.authorizeRequests()

.antMatchers("/hello").access("hasRole('ROLE\_ADMIN')")

.anyRequest().permitAll()

.and()

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

.usernameParameter("username").passwordParameter("password")

.and()

.logout().logoutSuccessUrl("/login?logout")

.and()

.exceptionHandling().accessDeniedPage("/403")

.and()

.csrf();

}

}

## In Summary

1. First we declare a datasource object annotated with @Autowired. This will look for Datasource definition in all classes under the same package. In this example we have it defined in MvcConfig.java
2. Next we set up two queries for AuthenticationManagerBuilder. One for authentication in usersByUsernameQuery and the other for authorization in authoritiesByUserNameQuery.
3. Finally we configure HttpSecurity to define what pages must be secured, authorized, not authorized, not secured, login page, logout page, access denied page, etc. One important thing to notice here is the order of configuration. Configuration that is specific to certain pages or urls must be placed first than configurations that are common among most urls.

### 403.jsp

7. Finally, write a jsp page to be displayed whenever access is denied to an user.

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<title>Access Denied - ProgrammingFree</title>

</head>

<body>

<h1>You do not have permission to access this page!

</h1>

<form action="/logout" method="post">

<input type="submit" value="Sign in as different user" />

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

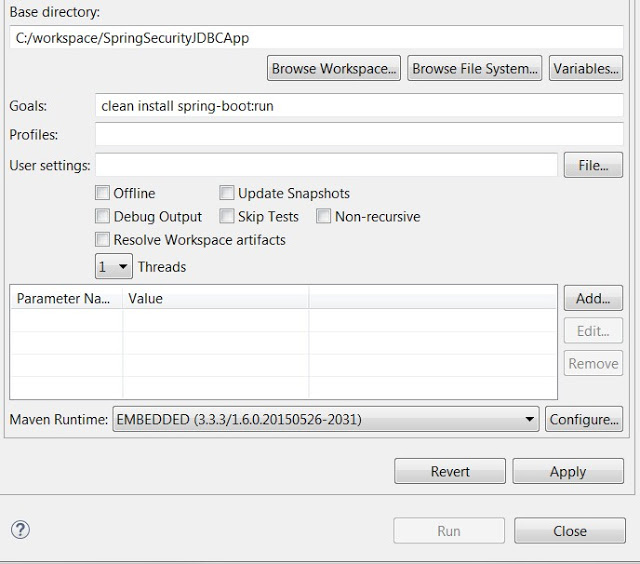
</form>

</body>

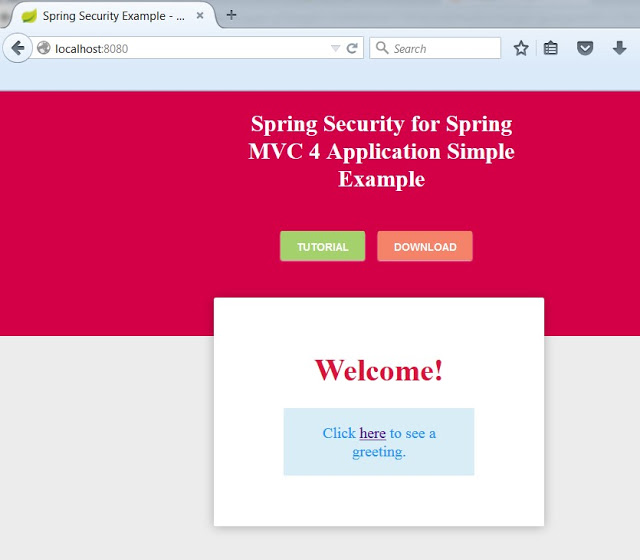
</html>

## Running the Application

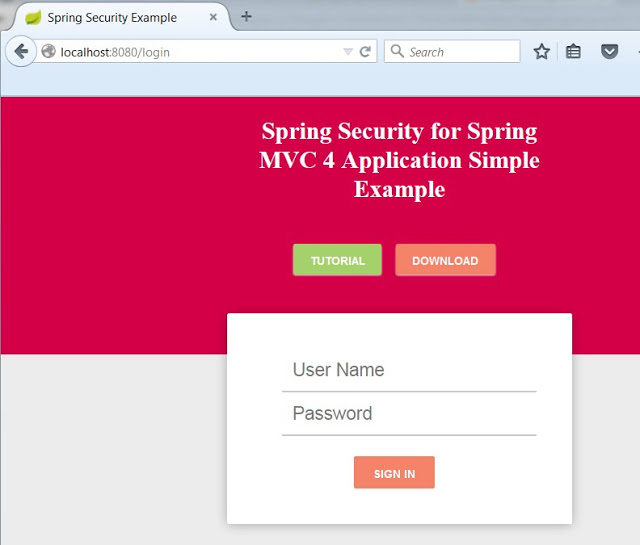
To run the application, run as Maven Build,

[](http://1.bp.blogspot.com/-w4EGN5ogbV4/VfnaDW-ABYI/AAAAAAAADZM/3GgZQ7a7bJY/s1600/spring_security_jdbc_authentication_1.jpg)

Once embedded tomcat in the application starts, Open localhost:8080

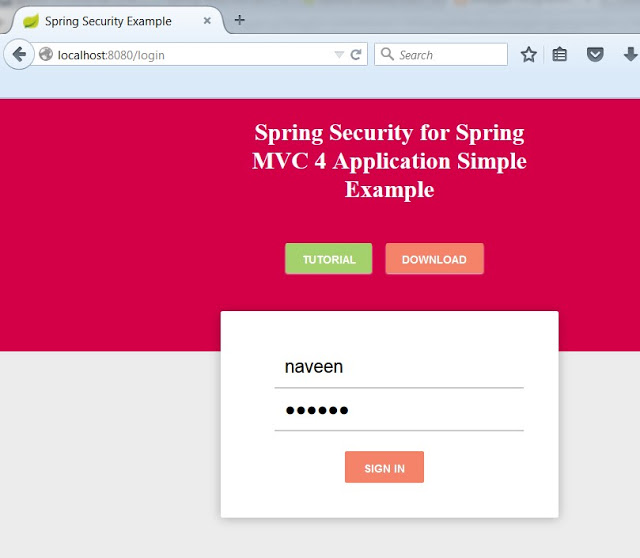
Welcome Page

Click on the link to see greeting page, you will be redirected to login page,

[](http://4.bp.blogspot.com/-dPg_GIEL6ww/Vfncyc2a5QI/AAAAAAAADZo/vx6stti1ulE/s1600/spring_security_jdbc_authentication_3.jpg)

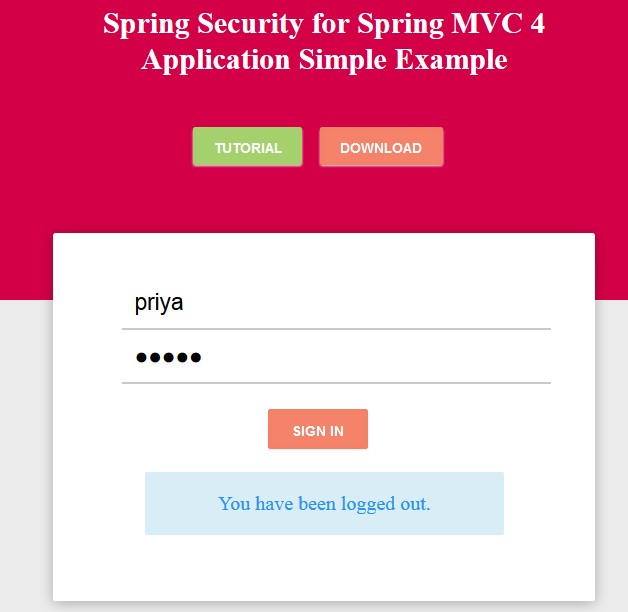
Login Page

Only admin users are authorized to see the greeting. Login as a Non- Admin user and try to access the greeting page:

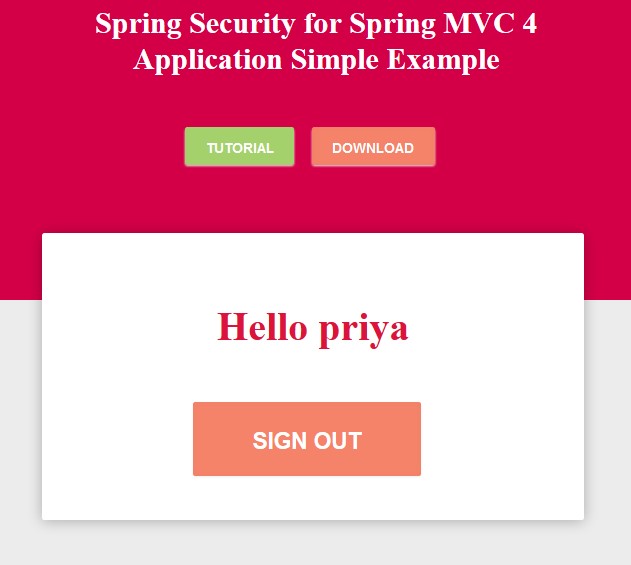
[](http://4.bp.blogspot.com/-vfkN6JO9U14/VfndSJVZ_II/AAAAAAAADZw/C1aRljqkkJo/s1600/spring_security_jdbc_authentication_4.jpg)

Access Denied Page

Logout and sign in as an ADMIN User, then you must be able to access the greeting page,

[](http://3.bp.blogspot.com/-gia3VP5uduI/VfneU-ObWLI/AAAAAAAADaM/TqdvVseTGgk/s1600/spring_security_jdbc_authentication_6.jpg)

Greeting Page

[](http://4.bp.blogspot.com/-hqk-4LXs-l8/VfnelrPeUkI/AAAAAAAADaU/jRnek8mKnQM/s1600/spring_security_jdbc_authentication_7.jpg)