Spring Security in Servlet Web Application using DAO, JDBC, In-Memory authentication

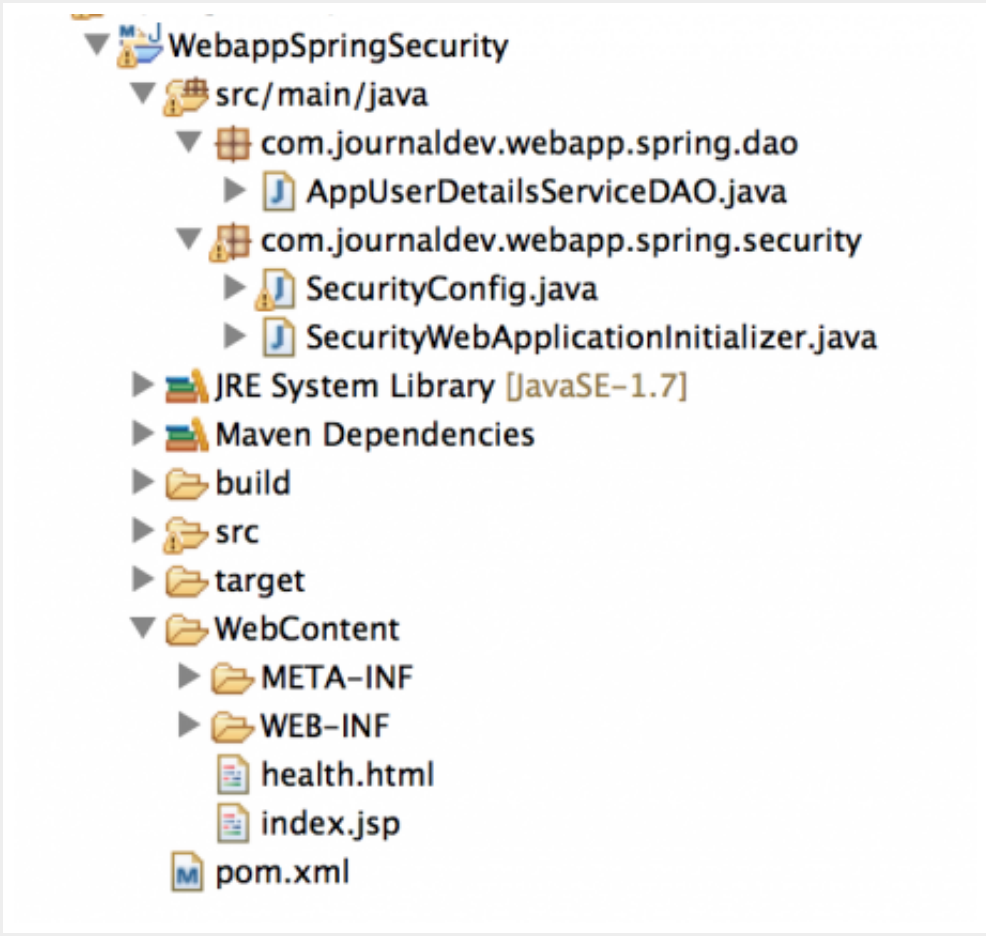
Spring Security provides ways to perform authentication and authorization in a web application. We can use spring security in any servlet based web application.

Some of the benefits of using Spring Security are:

1. Proven technology, it's better to use this than reinvent the wheel. Security is something where we need to take extra care, otherwise our application will be vulnerable for attackers.
2. Prevents some of the common attacks such as CSRF, session fixation attacks.
3. Easy to integrate in any web application. We don't need to modify web application configurations, spring automatically injects security filters to the web application.
4. Provides support for authentication by different ways – in-memory, DAO, JDBC, LDAP and many more.
5. Provides option to ignore specific URL patterns, good for serving static HTML, image files.
6. Support for groups and roles.

Today we will create a web application and integrate it with Spring Security. Create a web application using "**Dynamic Web Project**" option in Eclipse, so that our skeleton web application is ready. Make sure to convert it to maven project because we are using Maven for build and deployment. If you are unfamiliar with these steps, please refer Java Web Application Tutorial.

Once we will have our application secured, final project structure will look like below image.



We will look into three authentication methods – **in-memory**, **DAO** and **JDBC** based. For JDBC, I am using MySQL database and have following script executed to create the user details tables.

```
CREATE TABLE `Employees` (
  `username` varchar(20) NOT NULL DEFAULT '',
  `password` varchar(20) NOT NULL DEFAULT '',
  `enabled` tinyint(1) NOT NULL DEFAULT '1',
  PRIMARY KEY (`username`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `Roles` (
  `username` varchar(20) NOT NULL DEFAULT '',
  `role` varchar(20) NOT NULL DEFAULT '',
  PRIMARY KEY (`username`,`role`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO `Employees` (`username`, `password`, `enabled`)
VALUES
    ('pankaj', 'pankaj123', 1);

INSERT INTO `Roles` (`username`, `role`)
VALUES
    ('pankaj', 'Admin'),
    ('pankaj', 'CEO');

commit;
```

We would also need to configure JDBC DataSource as JNDI in our servlet container, to learn about this please read Tomcat JNDI DataSource Example.

Spring Security Dependencies

Here is our final pom.xml file.

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.or
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache
    <modelVersion>4.0.0</modelVersion>
    <groupId>WebappSpringSecurity</groupId>
    <artifactId>WebappSpringSecurity</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>war</packaging>
    <dependencies>
        <!-- Spring Security Artifacts - START -->
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-web</artifactId>
            <version>3.2.3.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-config</artifactId>
            <version>3.2.3.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-taglibs</artifactId>
            <version>3.0.5.RELEASE</version>
        </dependency>
        <!-- Spring Security Artifacts - END -->

        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>jstl</artifactId>
            <version>1.2</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>jsp-api</artifactId>
            <version>2.1</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.0.1</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>1.1.1</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>4.0.2.RELEASE</version>
        </dependency>
    </dependencies>
    <build>
        <sourceDirectory>src</sourceDirectory>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
```

```
                <version>3.1</version>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                </configuration>
            </plugin>
            <plugin>
                <artifactId>maven-war-plugin</artifactId>
                <version>2.3</version>
                <configuration>
                    <warSourceDirectory>WebContent</warSourceDirectory>
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

We have following dependencies related to Spring Framework:

1. **spring-jdbc**: This is used for JDBC operations by JDBC authentication method. It requires DataSource setup as JNDI. For complete example of it's usage, please refer Spring DataSource JNDI Example
2. **spring-security-taglibs**: Spring Security tag library, I have used it to display user roles in the JSP page. Most of the times, you won't need it though.
3. **spring-security-config**: It is used for configuring the authentication providers, whether to use JDBC, DAO, LDAP etc.
4. **spring-security-web**: This component integrates the Spring Security to the Servlet API. We need it to plugin our security configuration in web application.

Also note that we will be using Servlet API 3.0 feature to add listener and filters through programmatically, that's why servlet api version in dependencies should be 3.0 or higher.

View Pages

We have JSP and HTML pages in our application. We want to apply authentication in all the pages other than HTML pages.

health.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Health Check</title>
</head>
<body>
    <h3>Service is up and running!!</h3>
</body>
</html>
```

index.jsp

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

```
    pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="sec" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Home Page</title>
</head>
<body>
<h3>Home Page</h3>

    <p>
      Hello <b><c:out value="${pageContext.request.remoteUser}"/></b><br>
      Roles: <b><sec:authentication property="principal.authorities" /></b>
    </p>

    <form action="logout" method="post">
      <input type="submit" value="Logout" />
      <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}
    </form>
</body>
</html>
```

I have included index.jsp as welcome-file in the application deployment descriptor. Spring Security takes care of CSRF attack, so when we are submitting form for logout, we are sending the CSRF token back to server to delete it. The CSRF object set by Spring Security component is **_csrf** and we are using it's property name and token value to pass along in the logout request.

Let's look at the Spring Security configurations now.

UserDetailsService DAO Implementation

Since we will be using DAO based authentication also, we need to implement `UserDetailsService` interface and provide the implementation for `loadUserByUsername()` method.

Ideally we should be using some resource to validate the user, but for simplicity I am just doing basic validation.

AppUserDetailsServiceDAO.java

```
package com.journaldev.webapp.spring.dao;

import java.util.Collection;
import java.util.List;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundExceptio

public class AppUserDetailsServiceDAO implements UserDetailsService {

```

```
    protected final Log logger = LogFactory.getLog(getClass());

    @Override
    public UserDetails loadUserByUsername(final String username)
            throws UsernameNotFoundException {

        logger.info("loadUserByUsername username="+username);

        if(!username.equals("pankaj")){
            throw new UsernameNotFoundException(username + " not found");
        }

        //creating dummy user details, should do JDBC operations
        return new UserDetails() {

            private static final long serialVersionUID = 2059202961588104658L;

            @Override
            public boolean isEnabled() {
                return true;
            }

            @Override
            public boolean isCredentialsNonExpired() {
                return true;
            }

            @Override
            public boolean isAccountNonLocked() {
                return true;
            }

            @Override
            public boolean isAccountNonExpired() {
                return true;
            }

            @Override
            public String getUsername() {
                return username;
            }

            @Override
            public String getPassword() {
                return "pankaj123";
            }

            @Override
            public Collection<? extends GrantedAuthority> getAuthorities() {
                List<SimpleGrantedAuthority> auths = new java.util.ArrayList<S
                auths.add(new SimpleGrantedAuthority("admin"));
                return auths;
            }
        };
    }

}
```

Notice that I am creating anonymous inner class of `UserDetails` and returning it. You can create an implementation class for it and then instantiate and return it. Usually that is the way to go in actual applications.

Spring Security WebSecurityConfigurer implementation

We can implement `WebSecurityConfigurer` interface or we can extend the base implementation class `WebSecurityConfigurerAdapter` and override the methods.

SecurityConfig.java

```java
package com.journaldev.webapp.spring.security;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;

import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.builders
import org.springframework.security.config.annotation.web.builders.WebSecurity
import org.springframework.security.config.annotation.web.configuration.Enable
import org.springframework.security.config.annotation.web.configuration.WebSe

import com.journaldev.webapp.spring.dao.AppUserDetailsServiceDAO;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    public void configure(AuthenticationManagerBuilder auth)
            throws Exception {

        // in-memory authentication
        // auth.inMemoryAuthentication().withUser("pankaj").password("pankaj1

        // using custom UserDetailsService DAO
        // auth.userDetailsService(new AppUserDetailsServiceDAO());

        // using JDBC
        Context ctx = new InitialContext();
        DataSource ds = (DataSource) ctx
                .lookup("java:/comp/env/jdbc/MyLocalDB");

        final String findUserQuery = "select username,password,enabled "
                + "from Employees " + "where username = ?";
        final String findRoles = "select username,role " + "from Roles "
                + "where username = ?";

        auth.jdbcAuthentication().dataSource(ds)
                .usersByUsernameQuery(findUserQuery)
                .authoritiesByUsernameQuery(findRoles);
    }

    @Override
    public void configure(WebSecurity web) throws Exception {
        web
            .ignoring()
                // Spring Security should completely ignore URLs ending with
                .antMatchers("/*.html");
    }

}
```

Notice that we are ignoring all HTML files by overriding `configure(WebSecurity web)` method.

The code shows how to plugin JDBC authentication. We need to configure it by providing DataSource. Since we are using custom tables, we are also required to provide the select queries to get the user details and it's roles.

Configuring in-memory and DAO based authentication is easy, they are commented in above code. You can uncomment them to use them, make sure to have only one configuration at a time.

`@Configuration` and `@EnableWebSecurity` annotations are required, so that spring framework know that this class will be used for spring security configuration.

Spring Security Configuration is using Builder Pattern and based on the authenticate method, some of the methods won't be available later on. For example, `auth.userDetailsService()` returns the instance of `UserDetailsService` and then we can't have any other options, such as we can't set DataSource after it.

Integrating Spring Web Security with Servlet API

The last part is to integrate our Security configuration class to the Servlet API. This can be done easily by extending `AbstractSecurityWebApplicationInitializer` class and passing the Security configuration class in the super class constructor.

SecurityWebApplicationInitializer.java

```
1  package com.journaldev.webapp.spring.security;
2
3  import org.springframework.security.web.context.AbstractSecurityWebApplicatio
4
5  public class SecurityWebApplicationInitializer extends
6          AbstractSecurityWebApplicationInitializer {
7
8      public SecurityWebApplicationInitializer() {
9          super(SecurityConfig.class);
10     }
11 }
```

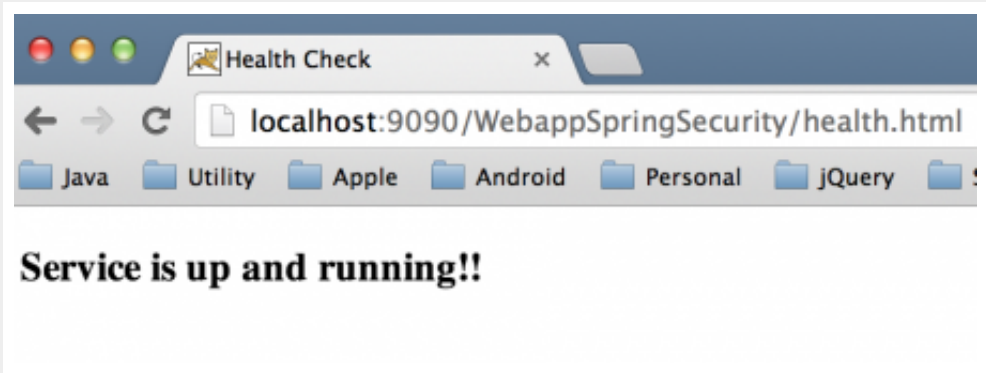
When our context startup, it uses ServletContext to add ContextLoaderListener listener and register our configuration class as Servlet Filter.

Note that this will work only in Servlet-3 complaint servlet containers. So if you are using Apache Tomcat, make sure it's version is 7.0 or higher.

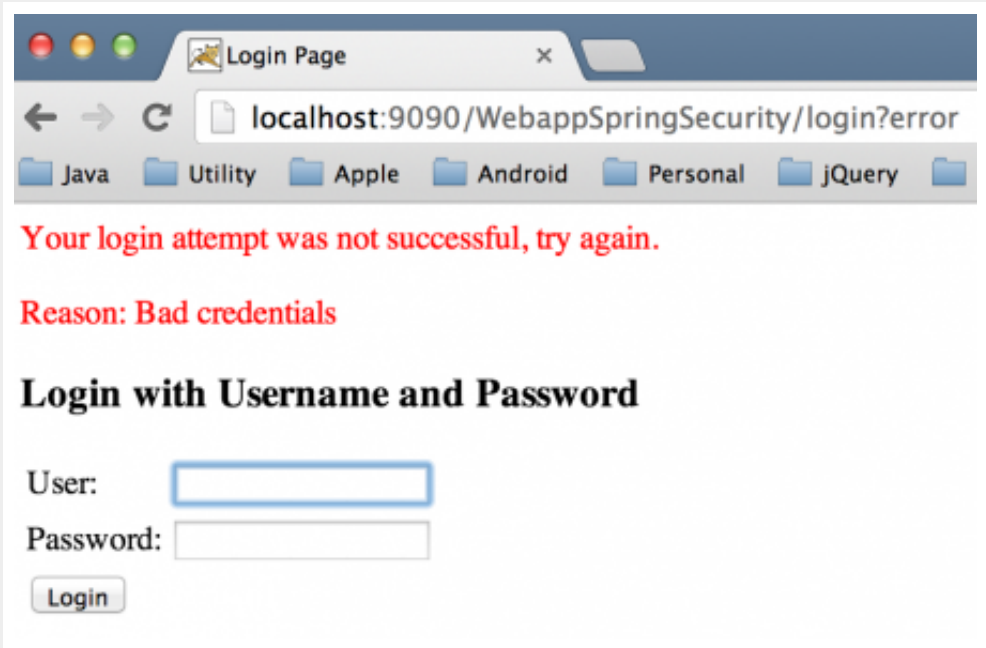
Our project is ready, just deploy it in your favorite servlet container. I am using Apache Tomcat-7 for running this application.

Below images show the response in various scenarios.

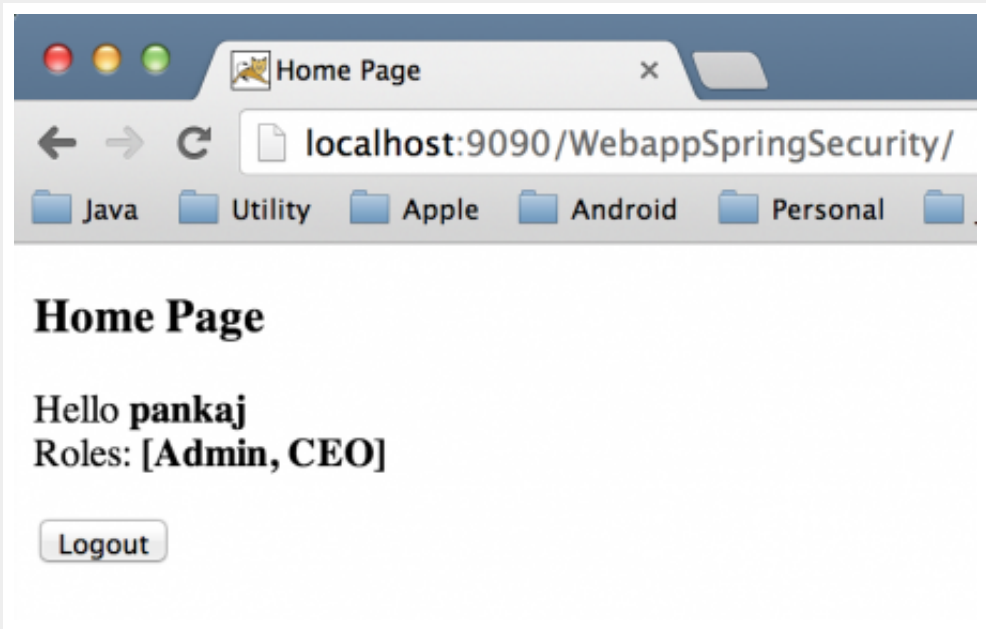
Accessing HTML Page without Security



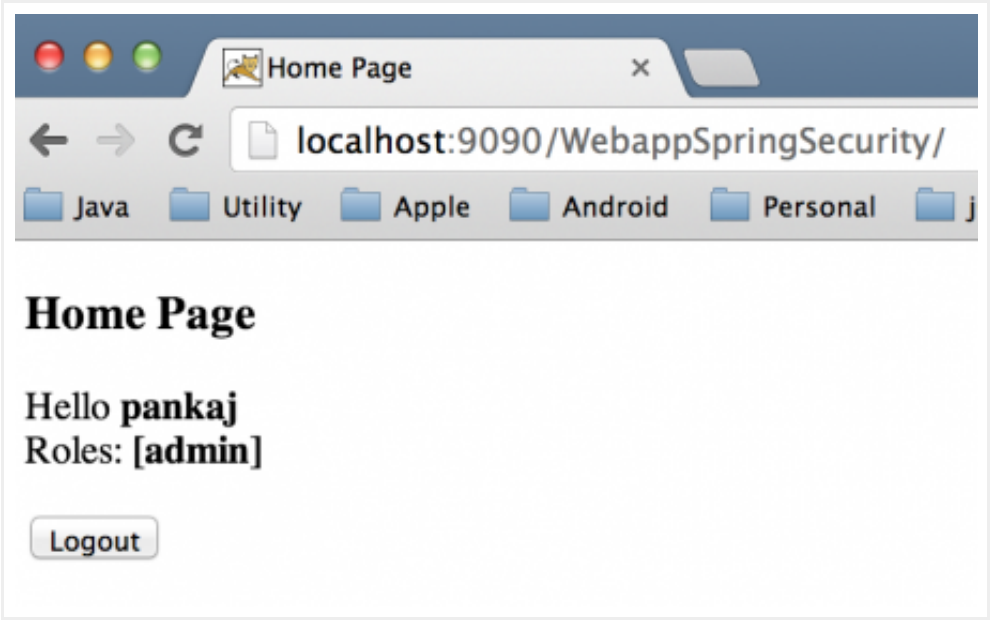
Authentication Failed for Bad Credentials



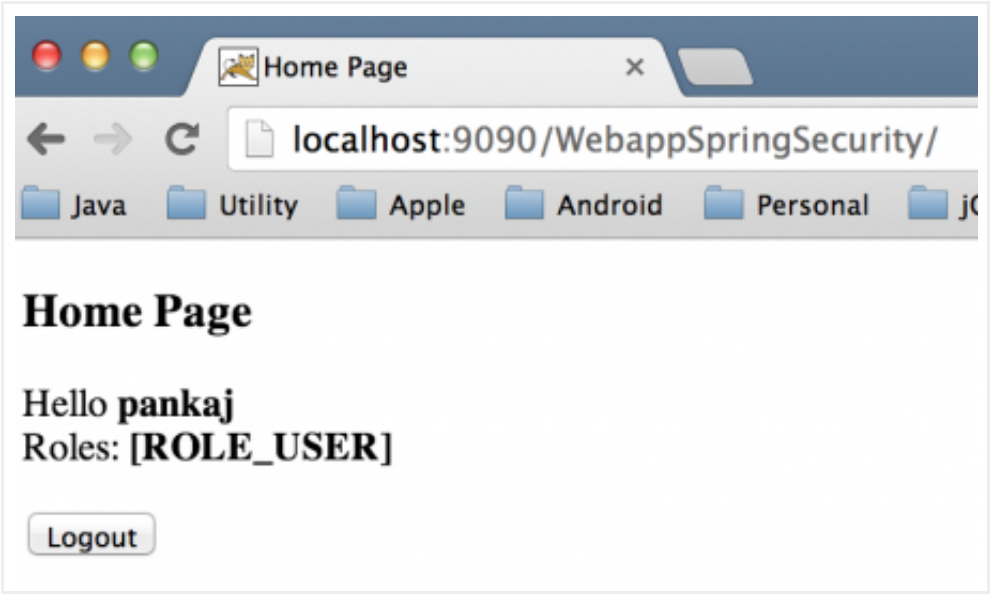
Home Page with Spring Security JDBC Authentication



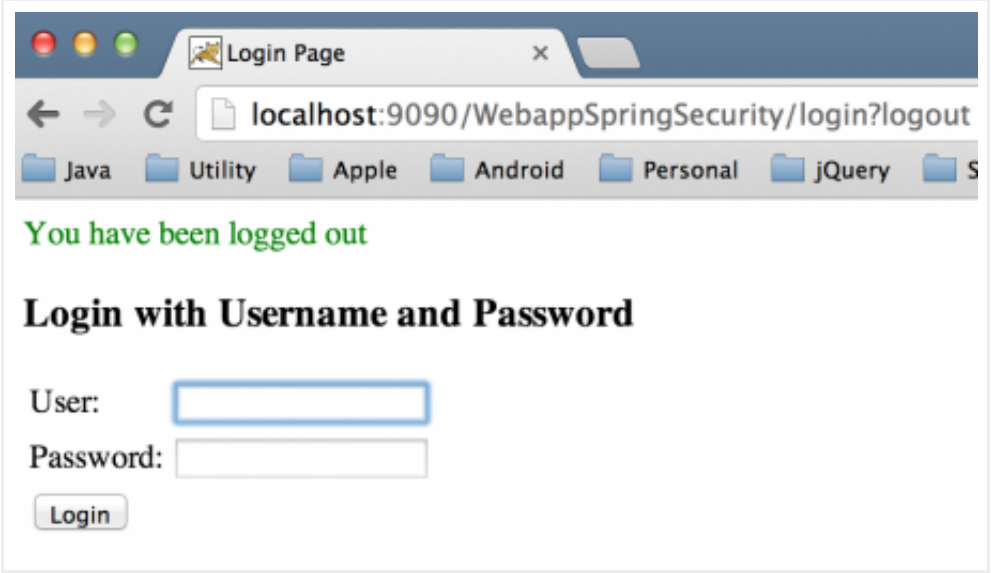
Home Page with Spring Security UserDetailsService DAO Authentication



Home Page with Spring Security In-Memory Authentication



Logout Page



If you want to use Servlet Container that doesn't support Servlet Specs 3, then you would need to

register `DispatcherServlet` through deployment descriptor. See JavaDoc of `WebApplicationInitializer` for more details.

That's all for Spring Security integration in Servlet Based Web Application. Please download the sample project from below link and play around with it to learn more.