Spring Security - Form-Based Authentication

Last Updated : 23 Jul, 2025

Form-Based Authentication in Spring Security provides a secure way to authenticate users using a custom login form instead of the default security prompt. It allows better control over authentication flow, user experience, and security configurations.

Key Features:

- Customizable login and logout mechanisms.
- Protection against common security threats like session fixation and brute force attacks.
- Seamless integration with Spring Boot and Thymeleaf.
- Easy role-based access control configuration.

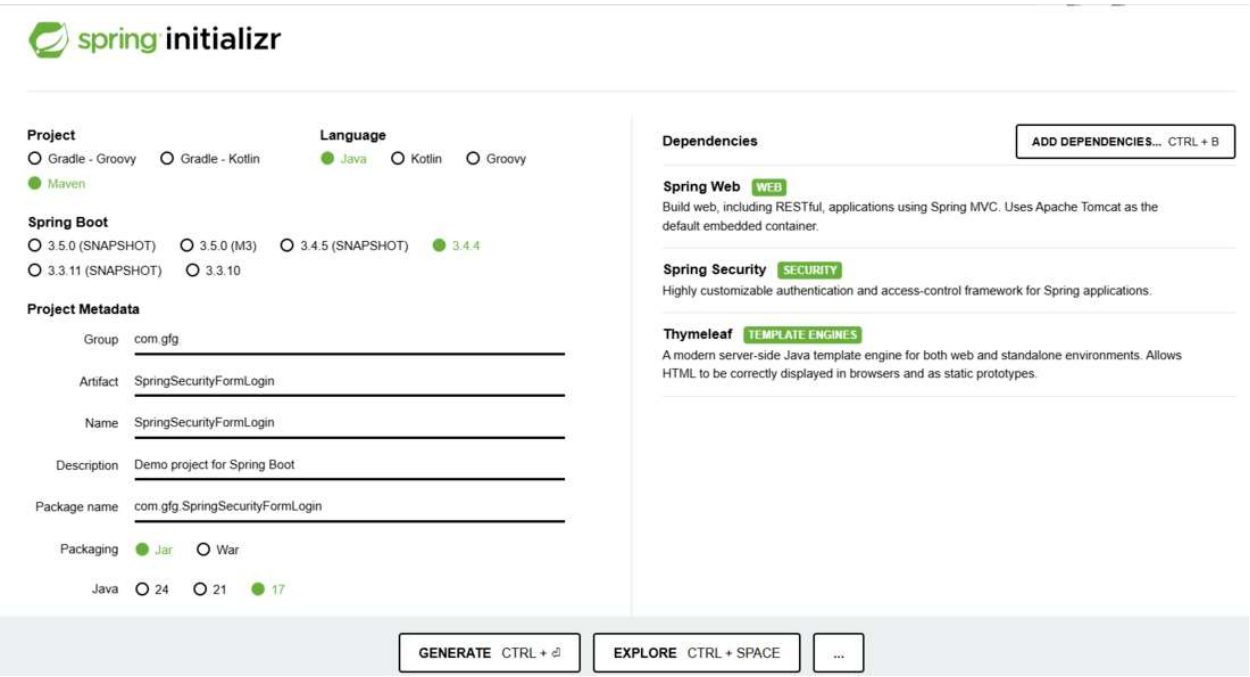
In this article, we will learn how to set up a Spring Boot application with Spring Security to implement a custom login page.

Steps to Create a Custom Login Form with Spring Security

Step 1: Create a Spring Boot Project

Use Spring Initializr to bootstrap your project with the following dependencies:

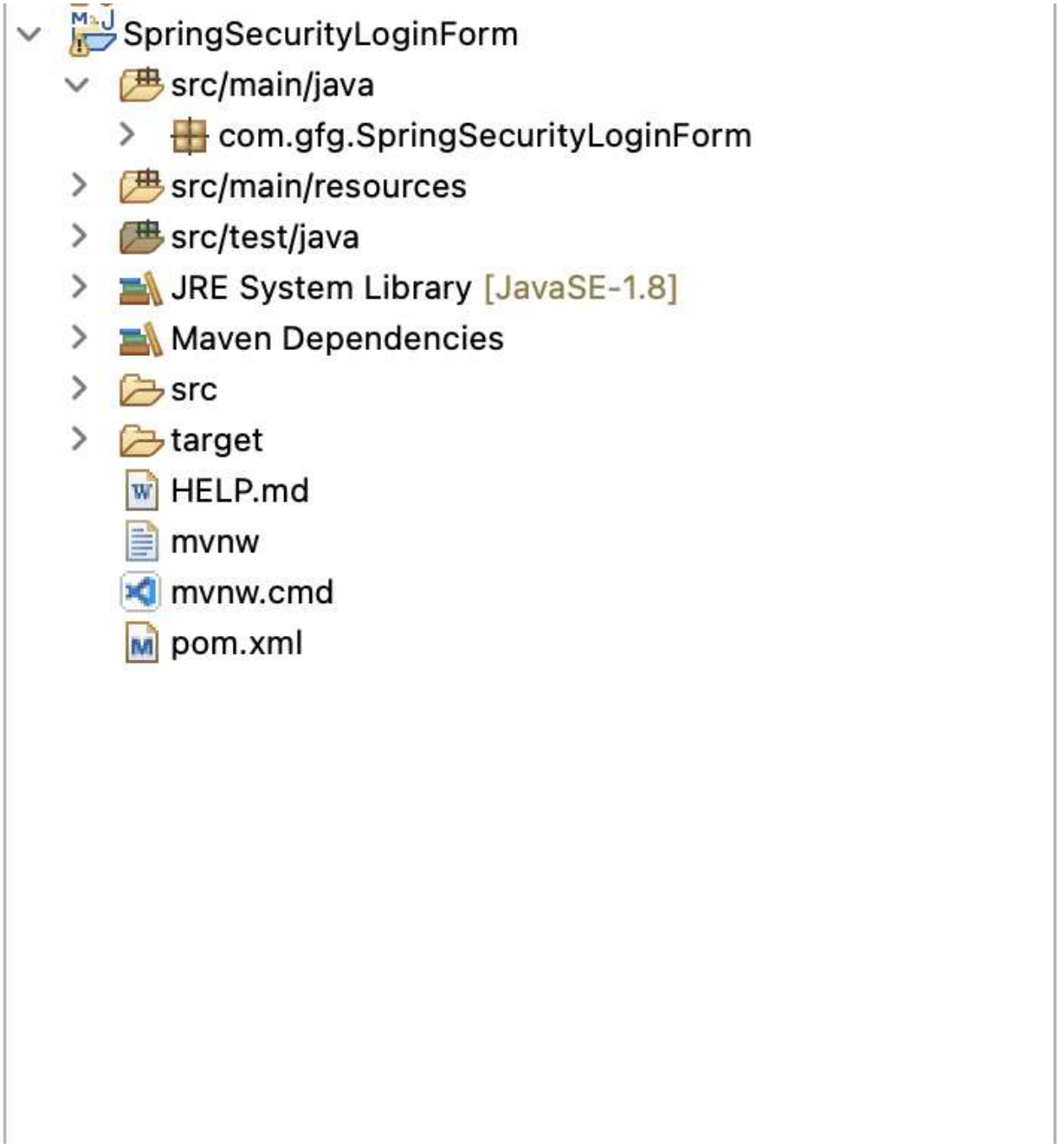
- Spring Web (for building web applications)
- Spring Security (for authentication and authorization)
- Thymeleaf (for rendering HTML templates)



This will create a **Spring boot Starter Project** with a **pom.xml** configuration file.

Project Structure:

The project structure would look something like this:



The pom.xml defines the configuration of the dependencies of the project, we don't need to add other dependencies right now as we are using spring boot and most of the things that we need for this project are auto-configured.

pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="https://maven.apache.org/POM/4.0.0"
         xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://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>3.4.4</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.gfg</groupId>
    <artifactId>SpringSecurityLoginForm</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>SpringSecurityLoginForm</name>
    <description>Demo project for Spring Boot</description>
    <properties>
        <java.version>17</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-thymeleaf</artifactId>
        </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>
```

Step 2: Create a Controller

The controller class handles the business-related logic, it handles requests coming from the client (In this case the browser) and redirects them to the view page. The **LoginController** class in the **com.gfg.SpringSecurityLoginForm** is invoked using the @Controller annotation, it has two GET methods for two requests. The welcome method simply redirects to the welcome page and the login method redirects to the custom login page.

```
package com.gfg.SpringSecurityLoginForm.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class LoginController {

    @GetMapping("/welcome")
    public String welcome() {
        return "welcome.html";

    }

    @GetMapping("/login")
    public String login() {
        return "login.html";

    }
}
```

Step 3: Create the Login Page

The **login.html** page in the templates folder (in the src/main/resources/templates folder) defines a custom login page with fields as username and password. The form sends a post method after submitting which sends the user input data to the spring configuration. The Theymleaf dependency helps in rendering the login page.

```
<!DOCTYPE html>
```

```
<!DOCTYPE html>
<html xmlns:th="https://www.thymeleaf.org/">
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
    <h1>Login page</h1>
    <form th:action="@{/login}" method="post">
        <div><label>Username: </label><input type="text" name="username">
</div>
        <div><label>Password: </label><input type="password" name="password">
</div>
        <div><button name="submit" type="submit" >Login</button></div>
    </form>
</body>
</html>
```

Step 4: Create the Welcome Page

The welcome.html page in the templates folder is a simple HTML file that shows a successful login message and provides a link to logout out of the application.

```
<!DOCTYPE html>
<html xmlns:th="https://www.thymeleaf.org/">
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
    <h1>LoggedIn Successful</h1>
    <h2>Welcome Back! Click <a th:href="@{/logout}">here</a> to logout.</h2>
</body>
</html>
```

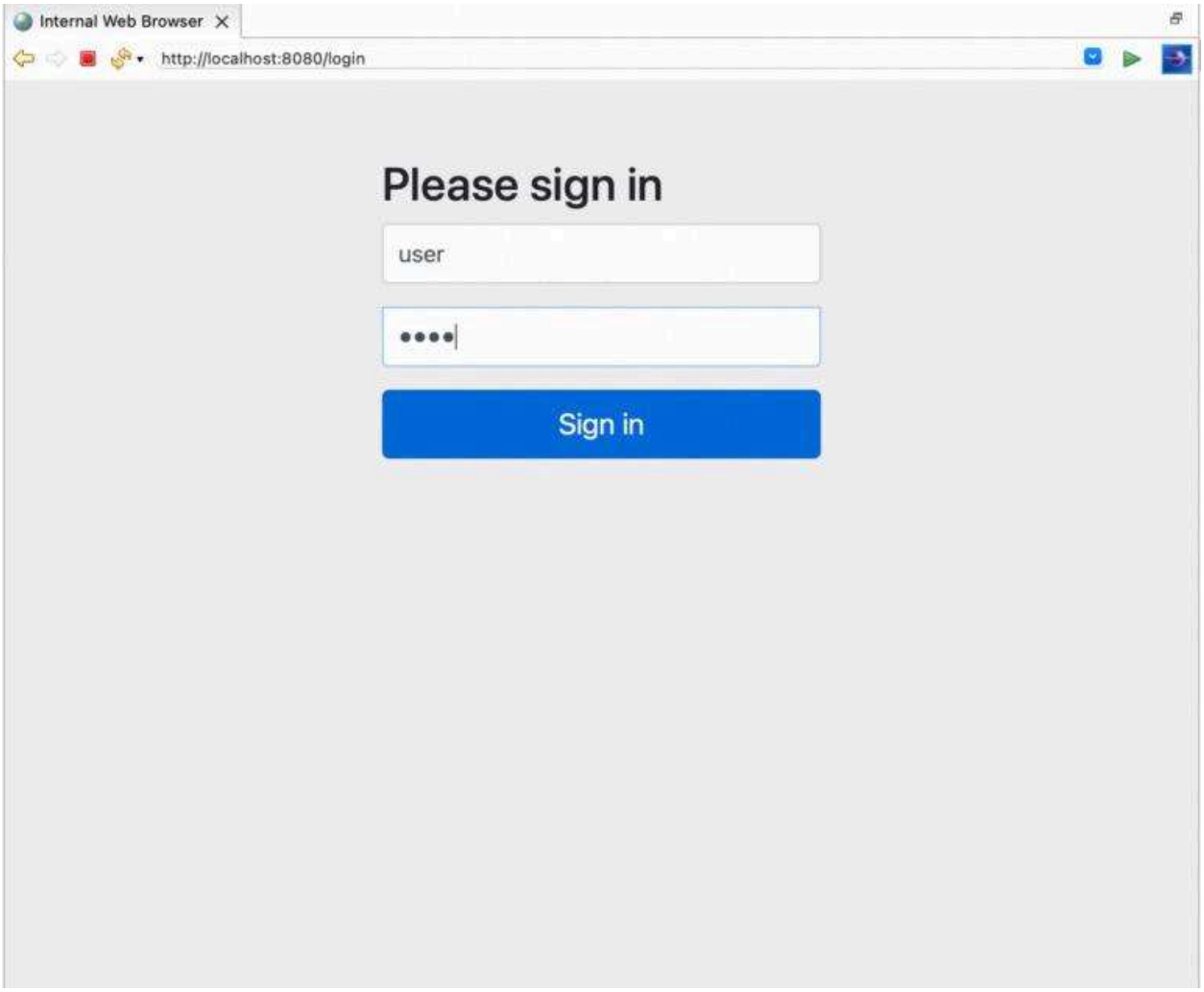
When you run your application for the first time without any custom configuration spring security provides a system-generated password in the console that looks something like this:

```
2022-02-23 01:11:31.862  INFO 30316 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]
2022-02-23 01:11:31.863  INFO 30316 --- [           main] w.s.c.ServletWebServerApplicationContext
2022-02-23 01:11:32.269  INFO 30316 --- [           main] .s.s.UserDetailsServiceAutoConfiguration

Using generated security password: c87e3708-4393-4350-b661-ea0f631ae44a

2022-02-23 01:11:32.339  INFO 30316 --- [           main] o.s.s.web.DefaultSecurityFilterChain
2022-02-23 01:11:32.887  INFO 30316 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
2022-02-23 01:11:32.899  INFO 30316 --- [           main] c.g.S.SpringSecurityLoginFormApplication
```

This form is de-facto for spring security, the formLogin() in the HttpSecurity class is responsible to render the login form and validate user credentials. Spring Security uses a servlet filter that intercepts all the incoming requests and redirects them to this login page. The server validates the credentials passed by the user and provides a Token for that particular session. The user id is "user" by default and a user-generated password is provided in the console.



Step 5: Configure Spring Security

In Spring Security 6.x, the WebSecurityConfigurerAdapter class has been deprecated. Instead, we now use the SecurityFilterChain bean to configure security. This modern approach is more flexible and aligns with current Spring practices.

- **InMemoryUserDetailsManager**: Defines users, their passwords, and roles.
- **SecurityFilterChain:** Configures how incoming requests are handled, including authentication and authorization rules.

How It Works:

- **Step 1:** Define users and roles using InMemoryUserDetailsManager.
- **Step 2:** Configure request handling (e.g., which pages are public, which require authentication) using SecurityFilterChain.

This approach replaces the old configure() methods from WebSecurityConfigurerAdapter and is the recommended way to set up Spring Security in modern applications.

```
package com.gfg.SpringSecurityLoginForm.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSec
urity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SpringSecurityConfig {

    @Bean
    public InMemoryUserDetailsManager userDetailsService() {
        UserDetails user = User.withUsername("user")
                .password("{noop}pass") // {noop} indicates plain text
password (for demo purposes)
                .roles("USER")
                .build();
        return new InMemoryUserDetailsManager(user);
```

```
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/login").permitAll() // Allow access to the
login page
                .anyRequest().authenticated() // Secure all other endpoints
            )
            .formLogin(form -> form
                .loginPage("/login") // Use the custom login page
                .defaultSuccessUrl("/welcome", true) // Redirect to welcome
page after login
            )
            .logout(logout -> logout
                .logoutSuccessUrl("/login") // Redirect to login page after
logout
                .permitAll()
            );

        return http.build();
    }
}
```

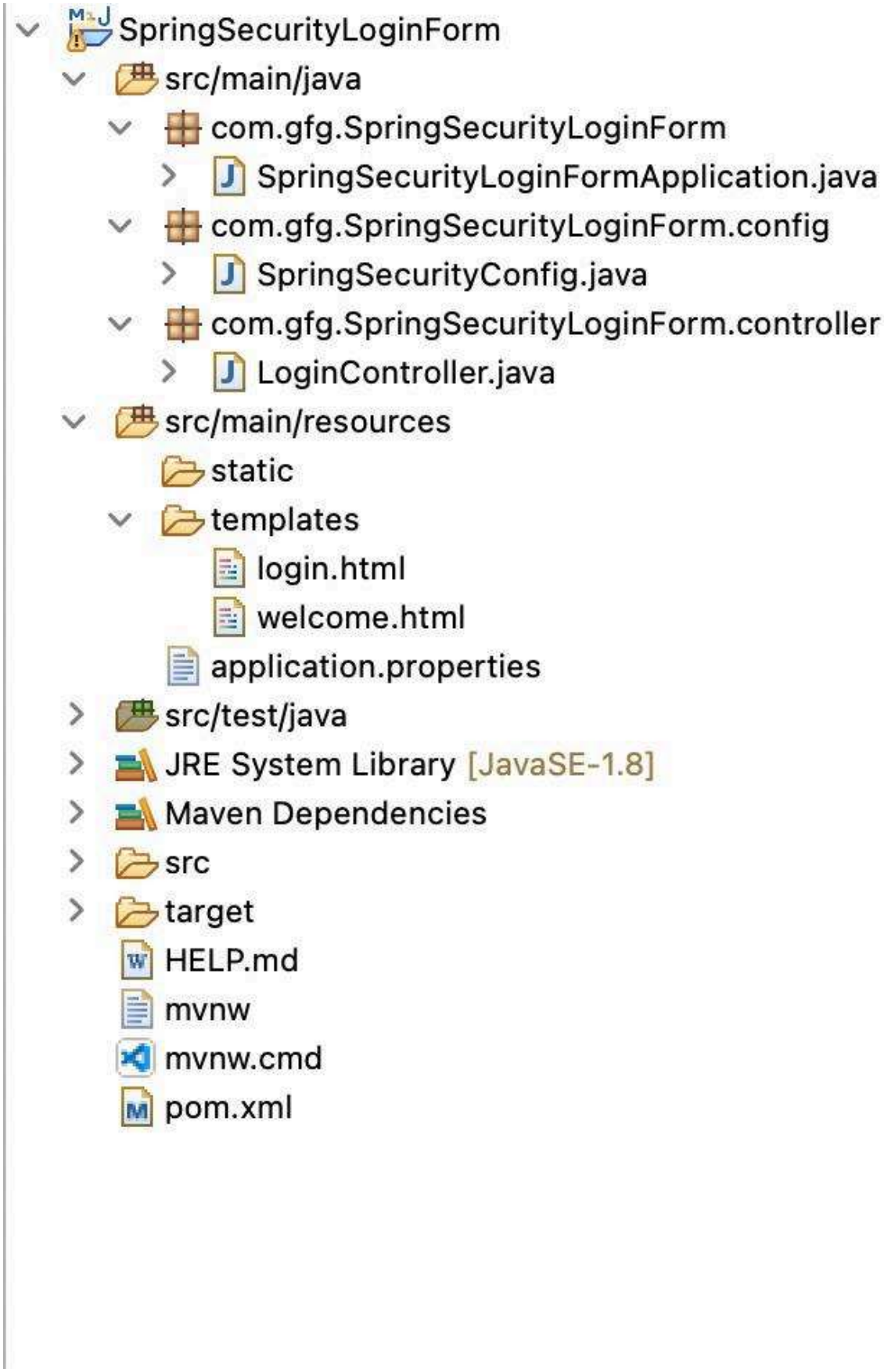
We can also define our custom login and password instead of the system-generated password. You can define the customer id and password by method a spring security property in the application.properties files in the resource folder. Mention the username and password in place of user and pass respectively.

spring.security.user.name=user

spring.security.user.password=pass

Updated Project Structure:

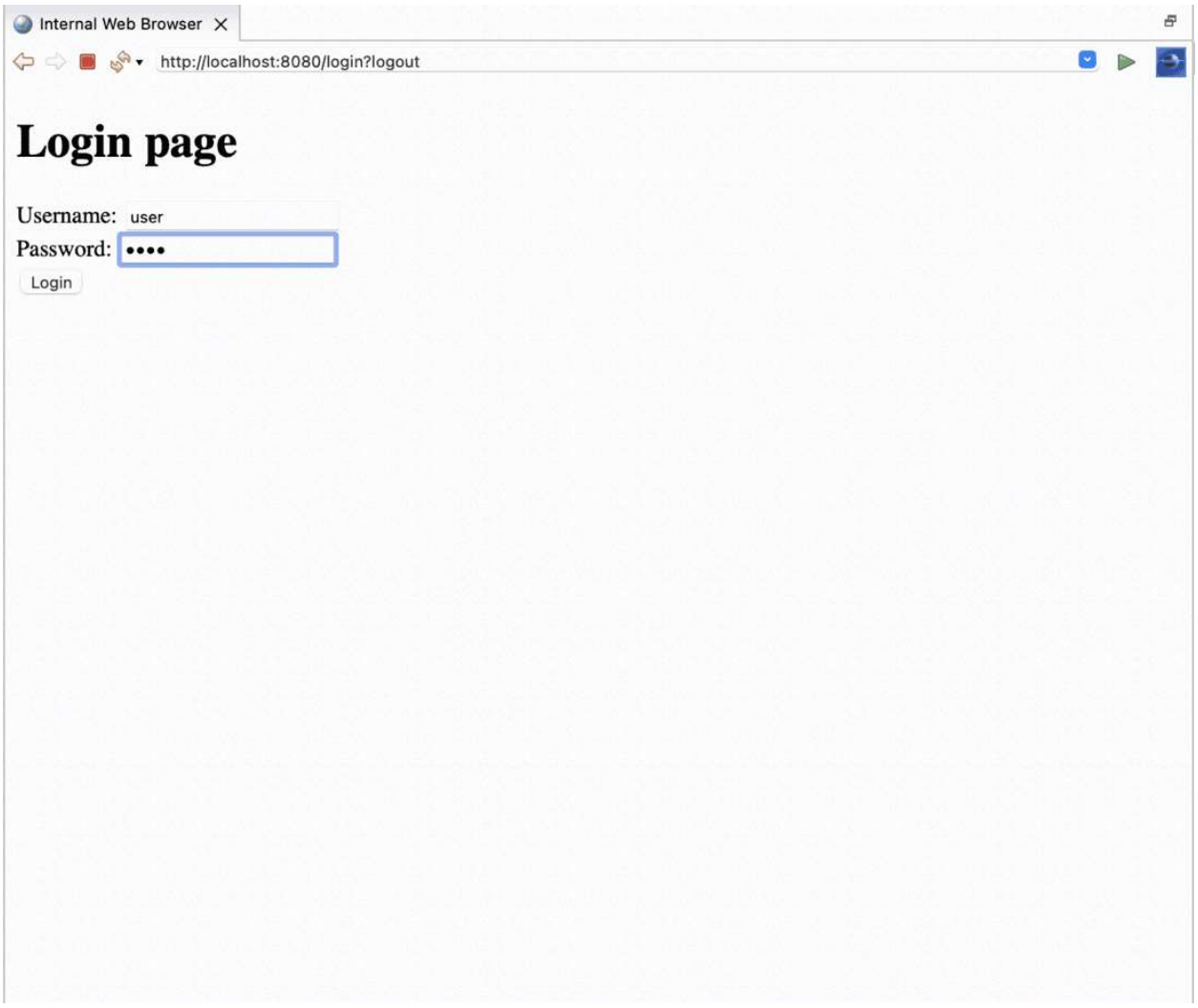
After adding all the configuration files and classes your project structure should look something like this:


SpringSecurityLoginForm
src/main/java
com.gfg.SpringSecurityLoginForm
SpringSecurityLoginFormApplication.java
com.gfg.SpringSecurityLoginForm.config
SpringSecurityConfig.java
com.gfg.SpringSecurityLoginForm.controller
LoginController.java
src/main/resources
static
templates
login.html
welcome.html
application.properties
src/test/java
JRE System Library [JavaSE-1.8]
Maven Dependencies
src
target
HELP.md
mvnw
mvnw.cmd
pom.xml

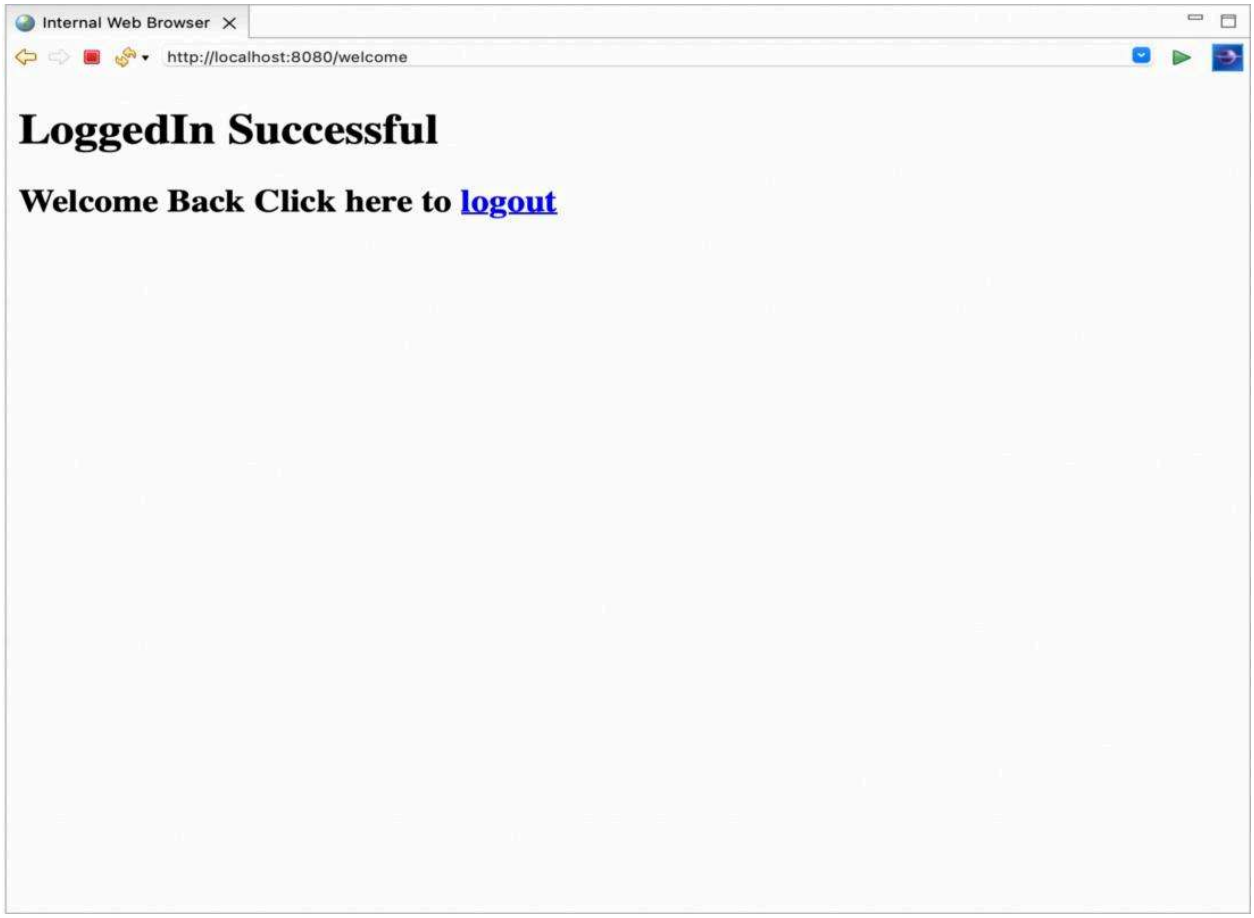
Step 6: Run the Application

Now it's time to run your created project, run your program as a Java application,

- Open your browser and navigate to http://localhost:8080/welcome.
- You will be redirected to the custom login page (http://localhost:8080/login).
- Enter the username (user) and password (pass).
- Upon successful login, you will be redirected to the welcome page.



After successful authentication spring will automatically redirect to the welcome page.



So, we have created a very basic custom Form-Based Authentication using spring security and tested it locally.

Comment More info Advertise with us