Spring Security

1

1

2

1. Introduction

Spring Security

security framework (started as Acegi Security)
→
authentication and authorization

takes advantage of
dependency injection
aspect-oriented programming (AOP)

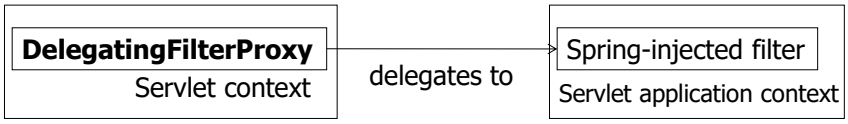
3

3

1.2 Proxying servlet filters

Java web application
starts with an HttpServletRequest

Set up the servlet filters:

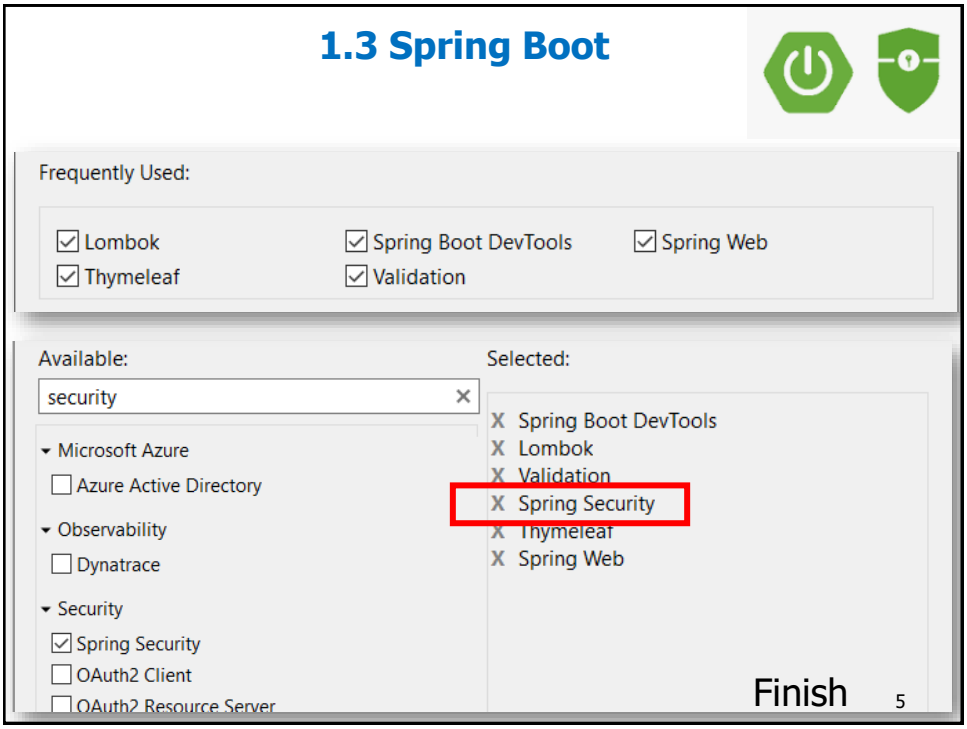


It will intercept request coming into the application.

4

4

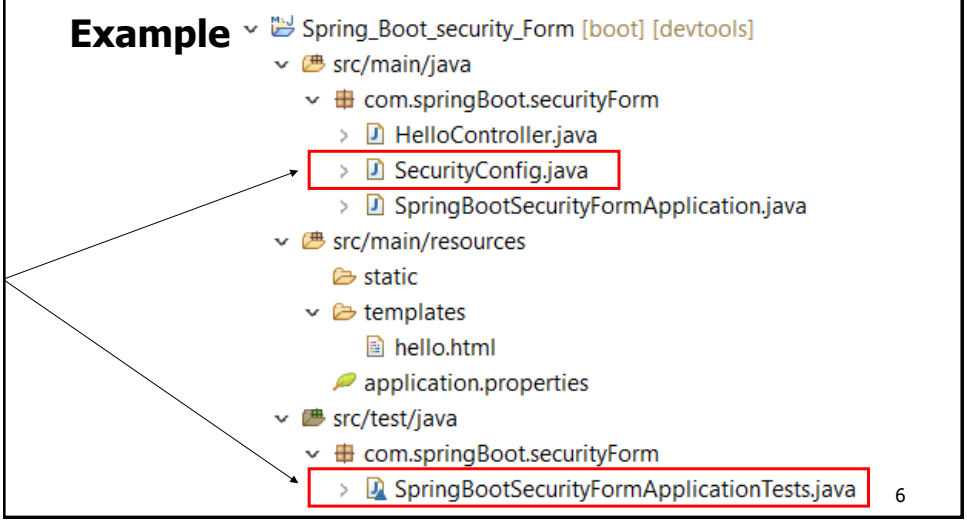
1.3 Spring Boot



5

1.4 Form-based login service (free login page)

Example



6

Example

hello
get → DispatcherServlet

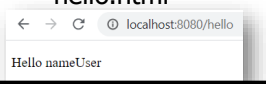
intercept request (Aspect)

Login page

Login ok

DispatcherServlet → HelloController

hello.html



7

7

Spring configuration file

```
package com.springBoot.securityForm;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
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.EnableWebSecurity;
import org.springframework.security.crypto.factory.PasswordEncoderFactories;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;


@Configuration
@EnableWebSecurity
public class SecurityConfig{
```

8

8

Spring configuration file

```
...
@Bean
SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests(requests ->
                requests.requestMatchers("/*")
                        .hasRole("USER")).
        formLogin(form ->
                form.defaultSuccessUrl("/hello", true));

        return http.build();
}
```

Intercept requests for **all URLs**
and
restrict access to only authenticated users who have the **USER** role.

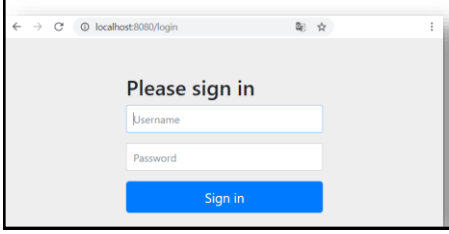
9

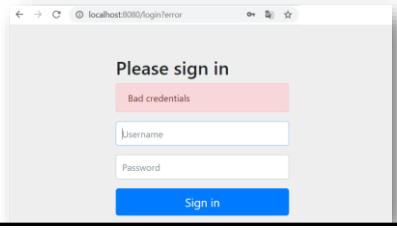
9

Spring configuration file

```
...
@Bean
SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests(requests ->
                requests.requestMatchers("/*")
                        .hasRole("USER")).
        formLogin(form ->
                form.defaultSuccessUrl("/hello", true));

        return http.build();
}
```

Form-based login service (free login page)





10

10

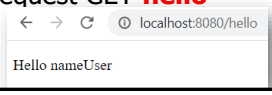
Spring configuration file

```
...
@Bean
SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
                http.authorizeHttpRequests()
    .requestMatchers("/*")
    .hasRole("USER")
    .and()
    .formLogin().defaultSuccessUrl("/hello", true);

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

Login page

Login ok

request GET **hello**



11

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth)
                                                throws Exception {

    PasswordEncoder encoder =
      PasswordEncoderFactories.createDelegatingPasswordEncoder();

    auth.inMemoryAuthentication()
    .withUser("nameUser").password(encoder.encode("12345678")).roles("USER").and()
    .withUser("nameAdmin").password(encoder.encode("admin")).roles("USER","ADMIN");
}
```

The PasswordEncoderFactories uses the **BCryptPasswordEncoder** as the default encoder, but it can also use other encoders like NoOpPasswordEncoder or Pbkdf2PasswordEncoder.

Bcrypt is a widely used algorithm for password hashing, known for its security and strength against brute-force attacks.

Now, with this configuration, we're storing our in-memory password using BCrypt in the following format:

```
nameUser= $2a$10$TNRcZbiM7UQp9CMBE5cGBev8O5NQJRF0qbrhRnld70WylVMnPSre6
nameAdmin= $2a$10$6nh2ry4NFP6jiCYpYt2mXuNAq5L/cT/OtTkSww5oTmE7dfo4IuvbW
```

12

12

If, for any reason, we don't want to encode the configured password, we can make use of the NoOpPasswordEncoder.
To do so, we simply prefix the passphrase we provide to the *password()* method with the *{noop}* identifier:

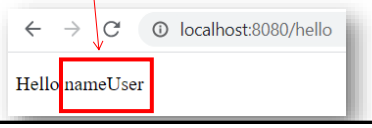
```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth)
                                                    throws Exception {
    auth
        // enable in memory based authentication with a user named
        // "user" and "admin"

        //use NoOpPasswordEncoder for
        //in-memory authentication

        .inMemoryAuthentication()
        .withUser("nameUser").password("{noop}user").roles("USER").and()
        .withUser("nameAdmin").password("{noop}admin").roles("USER", "ADMIN");
}
```

13

13

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
    ...
    .withUser("nameUser").password(encoder.encode("user")).roles("USER").and()...
```

```
@Controller
public class HelloController {

    @GetMapping(value = "/hello")
    public String printWelcome(Model model,
                                        Principal principal) {

        model.addAttribute("username", principal.getName());
        return "hello";    }
}
```



14

14

```
package com.springBoot.securityForm;
import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestBuilders.formLogin;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.model;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.redirectedUrl;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.view;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.context.annotation.Import;
import org.springframework.security.test.context.support.WithMockUser;
import org.springframework.test.web.servlet.MockMvc;

@WebMvcTest
@Import(SecurityConfig.class)
class SpringBootSecurityFormApplicationTests {

        @Autowired
         private MockMvc mockMvc;
```

The **@Import** annotation is used to import the security configuration class SecurityConfig.

15

15

```
 @WithMockUser /*When you use @WithMockUser without parameters, Spring Security Test: sets the username to "user" and assigns the role "USER" by default*/
 @Test
 void testAccessWithUserRole() throws Exception {
      mockMvc.perform(get("/hello"))
                .andExpect(status().isOk())
                .andExpect(view().name("hello"))
                 .andExpect(model().attributeExists("username"))
                 .andExpect(model().attribute("username", "user"));
 }

@WithMockUser(username = "user", roles = { "NOT_USER" })
 @Test
 void testNoAccessWithWrongUserRole() throws Exception
 {
      mockMvc.perform(get("/hello"))
                .andExpect(status().isForbidden());
 }
```

We are using the **@WithMockUser** annotation **to simulate** authentication with different roles.

16

16

```
@Test
void testWrongPassword() throws Exception {
    mockMvc.perform(formLogin("/login")
        .user("username", "nameUser")
        .password("password", "wrongPassword"))
        . andExpect(status().isFound())
//OR . andExpect(status().is(302))
        .andExpect(redirectedUrl("/login?error"));
}
```

isFound() method checks if the response status code is 302, indicating **a redirection**. It is the same as "andExpect(status().is(302))".

mockMvc.perform(formLogin("/login")…
andExpect(redirectedUrl("/login?error"));

The assertion is checking whether the login request with a wrong password is redirected **to the login page with an error message**.

17

17

```
@Test
void testCorrectPassword() throws Exception {
    mockMvc.perform(formLogin("/login")
        .user("username", "nameUser")
        .password("password", "12345678"))

        . andExpect(status().isFound())
        .andExpect(redirectedUrl("/welcome"));
}
```

The assertion is checking whether the login request with a correct password is redirected to **welcome**.

18

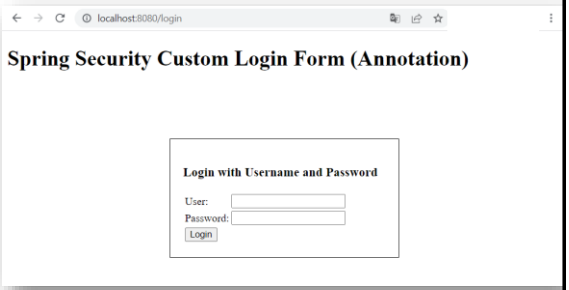
18

1.5 Own login page

Form-based login service (own login page)

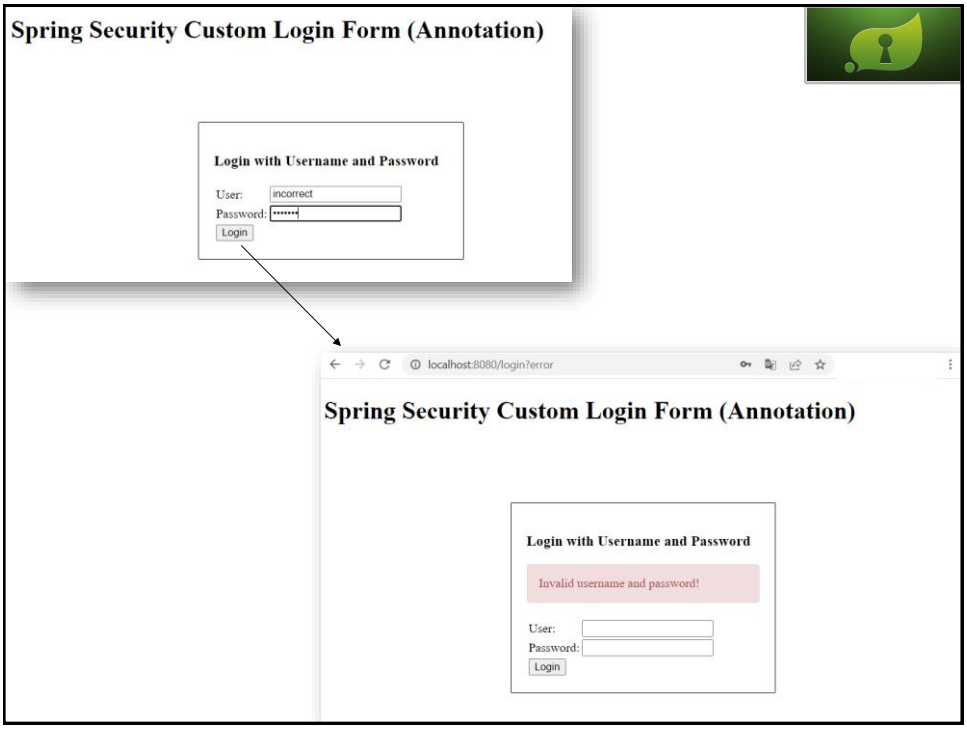
http://www.mkyong.com/spring-security/spring-security-custom-login-form-annotation-example/

- Spring_Boot_security [boot] [devtools]
  - src/main/java
    - com.springBoot.security
      - LoginController.java
      - SecurityConfig.java
      - SpringBootSecurityApplication.java
      - WelcomeController.java
  - src/main/resources
    - static
      - css
    - templates
      - hello.html
      - login.html
    - application.properties
  - src/test/java
    - com.springBoot.security
      - SpringBootSecurityApplicationTests.java



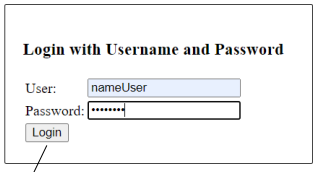
19

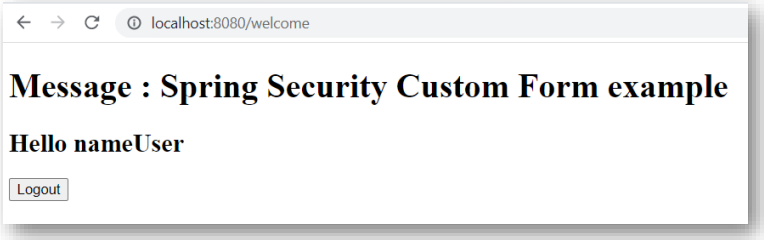
19



20

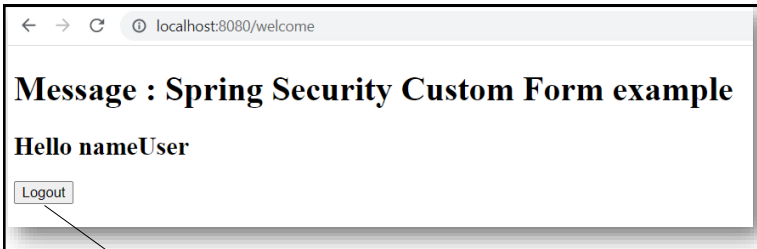
Spring Security Custom Login Form (Annotation)

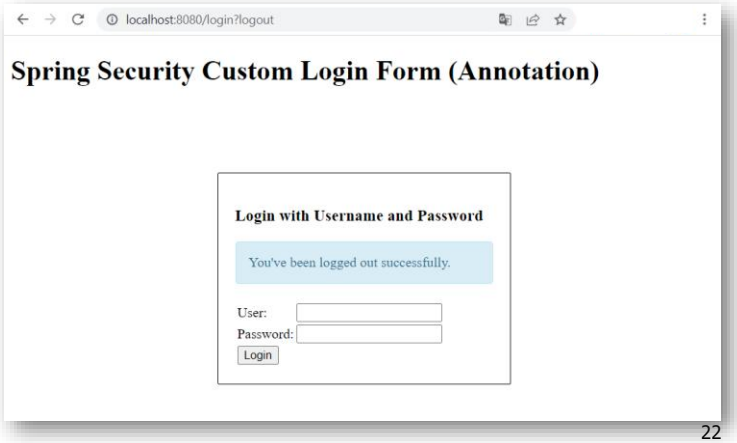




21

21





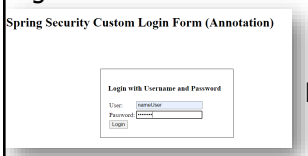
22

22

Example

welcome
get → DispatcherServlet

intercept welcome request (Aspect)



Login page

Login ok
DispatcherServlet → *request welcome GET* → LoginController
← hello.html

23

Spring configuration file

```
package com.springBoot.security;
import…
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

        http.csrf(csrf -> csrf.csrfTokenRepository(
                                new HttpSessionCsrfTokenRepository())).
        authorizeHttpRequests(requests ->
                requests.requestMatchers("/login**").permitAll()
                                .requestMatchers("/css/**").permitAll()
                                .requestMatchers("/*").hasRole("USER")).
        formLogin(form ->
                form
                 .defaultSuccessUrl("/welcome", true)
                .loginPage("/login"));

        return http.build();
```

Spring Security has added protection against Cross Site Request Forgery (CSRF) attacks.
https://docs.spring.io/spring-security/reference/features/exploits/csrf.html

24

24

Cross Site Request Forgery (CSRF)

When a user visits the application, Spring Security generates a unique CSRF token and stores it in the **user's session**. The token is also included in the form as a **hidden field**.

When the form is submitted via HTTP POST, the server-side code reads the **CSRF token** from **the hidden field** and compares it with the token stored in the user's session.

If the tokens match, the request is considered legitimate and is allowed to proceed.
If the tokens do not match, the request is rejected as a potential CSRF attack.

```
<form th:action="@{/logout}" method="post">
    <input type="submit" value="Logout" />

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

</form>
```

25

25

Cross Site Request Forgery (CSRF)

Spring Security provides several options for storing CSRF tokens
The CSRF token is stored in

the user's session:

```
http.csrf(csrf -> csrf.csrfTokenRepository(new HttpSessionCsrfTokenRepository())).
```

This method is secure and suitable for traditional web applications where session management is handled by the server.

a cookie:

```
http.csrf(csrf ->
  csrf.csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())).
```

Making it accessible from client-side JavaScript: scenarios where client-side code needs to interact with the token.

the header:
This is useful for RESTful APIs: the CSRF token needs to be sent explicitly with each request.
a custom Repository:
A custom repository offers flexibility to store the CSRF token according to your specific needs. If the built-in methods don't suffice, you can create a custom repository to handle the token as required by your application.

26

26

```
…        http.csrf(csrf -> csrf.csrfTokenRepository(new HttpSessionCsrfTokenRepository())).
         authorizeHttpRequests(requests ->
                  requests.requestMatchers("/login**").permitAll()
                           .requestMatchers("/css/**").permitAll()
                           .requestMatchers("/*").hasRole("USER")…
```

The **permitAll()** method is used in Spring Security to grant access to specific URLs for all users, regardless of their role.
The **login page** and **css files** are accessible.

…requestMatchers("/login").permitAll()…**
requestMatchers("/*").hasRole("USER"))

This means that any url that matches the pattern **/*** (i.e., all urls), except **/login**** will only be accessible to users who have the USER role.

If you **do not write login permitAll** then you are in your **infinite loop**: you send a request -> not logged in -> login request is sent -> all urls not accessible, not logged in -> login request is sent, etc.

ERR_TOO_MANY_REDIRECTS

27

27

```
…formLogin(form ->
          form.defaultSuccessUrl("/welcome", true)
               .loginPage("/login"));
```

is used to specify that the user should be redirected to the **/welcome URL** after a successful login.

loginPage() is used to specify the URL of the custom login page that the user should be redirected to if they are not authenticated or if they attempt to access a protected resource without being authorized.

28

28

Spring configuration file

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth)
                                                    throws Exception {
    PasswordEncoder encoder =
            PasswordEncoderFactories.createDelegatingPasswordEncoder();

    auth.inMemoryAuthentication()

    .withUser("nameUser").password(encoder.encode("12345678"))
    .roles("USER");
}
```

29

29

Example → part 1

welcome
get

→ DispatcherServlet

intercept welcome request (Aspect)

Login page

30

30

Spring configuration file

```
http.csrf(…).authorizeHttpRequests(requests ->
        requests.requestMatchers("/login**").permitAll()
                .requestMatchers("/css/**").permitAll()
                .requestMatchers("/*").hasRole("USER")).
    formLogin(form -> form.defaultSuccessUrl("/welcome", true)
        .loginPage("/login"));
```

localhost:8080 localhost:8080/welcome

intercept request (Aspect)

↓

request **login GET**

```
@Controller
@RequestMapping("/login")
public class LoginController {

@GetMapping
public String login(String error, String logout, Model model) {
```

31

31

LoginController

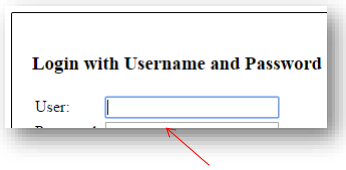
```
@Controller
@RequestMapping("/login")
public class LoginController {

    @GetMapping
    public String login(String error, String logout, Model model) {

        if (error != null) {
            model.addAttribute("error", "Invalid username and password!");
        }
        if (logout != null) {
            model.addAttribute("msg", "You've been logged out successfully.");
        }
        return "login";
    }

}
```

→ **login.html**

32

32

login.html




```
...
  <body onload='document.loginForm.username.focus();'>

    <h1>Spring Security Custom Login Form (Annotation)</h1>

    <div id="login-box">

      <h3>Login with Username and Password</h3>

      <div th:if="${error}" class="error"
                                th:text="${error}"></div>

      <div th:if="${msg}" class="msg"
                                th:text="${msg}"></div>
```

33

33

login.html

```
<form th:action="@{/login}" method="post">
 <table>
        <tr>
            <td>User:</td>
            <td><input type='text' name='username' value=''></td>
        </tr>
        <tr>
            <td>Password:</td>
            <td><input type='password' name='password' /></td>
        </tr>
        <tr>
            <td colspan='2'>
           <button name="submit" type="submit">Login</button></td>
        </tr>
 </table>

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

</form>
</div>
</body>
</html>
```

"username" and "password" are the default names.

Ensure that you include the CSRF token in all POST methods. By including the <input> element with the _csrf.token attribute, you are ensuring that the CSRF token is included in the form submission, and that the server-side code can validate it and prevent any potential CSRF attacks.

34

34

Customize the login-form fields

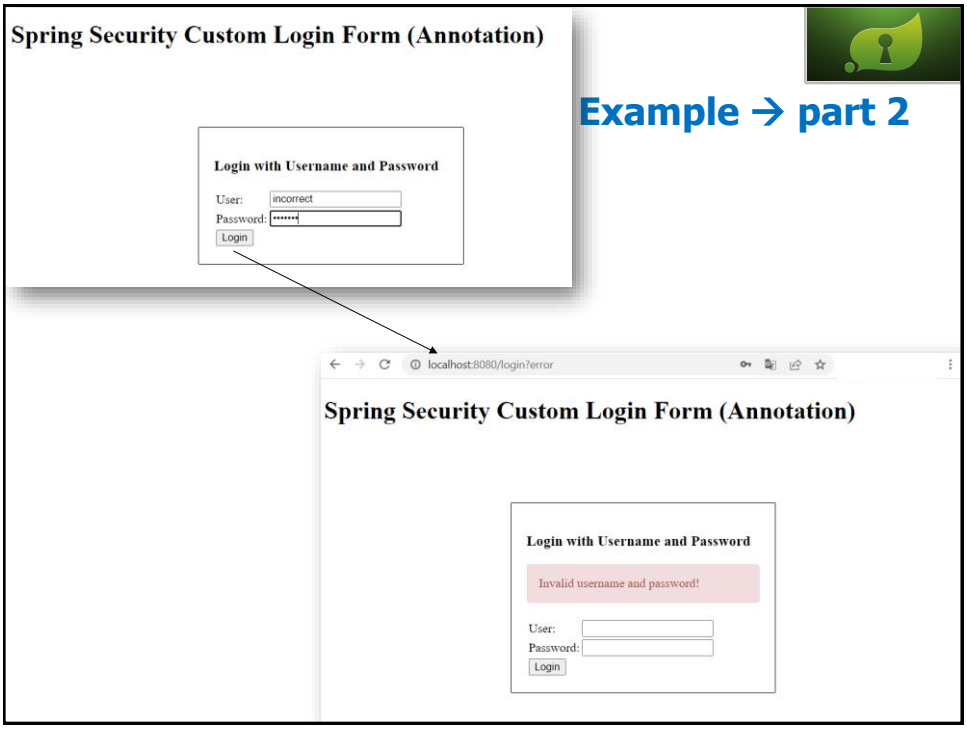
For example, we want to use "email" and "password2" in login.html:

```
<tr>
    <td>User:</td>
    <td><input type='text' name='email' value=''></td>
</tr>
<tr>
    <td>Password:</td>
    <td><input type='password' name='password2' /></td>
```

```
@Bean
SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http...
        formLogin(form ->
                form
                .usernameParameter("email")
                .passwordParameter("password2")
                .defaultSuccessUrl("/welcome", true)
                .loginPage("/login"));

        return http.build(); }
```

35

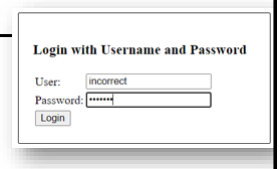
35



36

NOT .withUser("nameUser").password(encoder.encode("12345678")).roles("USER");
➔
loginPage("/**login**"));

Spring configuration file



LoginController

```
@Controller
@RequestMapping("/login")
public class LoginController {

    @GetMapping
    public String login(String error, String logout, Model model) {

        if (error != null) {
            model.addAttribute("error", "Invalid username and password!");
        }
        if (logout != null) {...}
        return "login";
    }
}
```

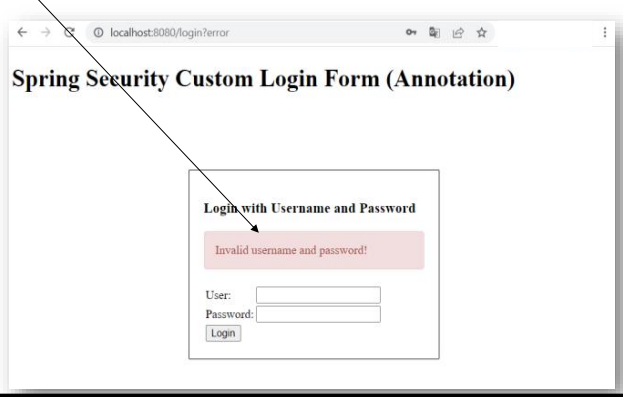
→ login.html

37

37

login.html

```
...
<body onload='document.loginForm.username.focus();'>

    <h1>Spring Security Custom Login Form (Annotation)</h1>
    <div id="login-box">
        <h3>Login with Username and Password</h3>

        <div th:if="${error}" class="error"
                th:text="${error}"></div>
```

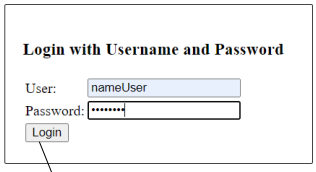


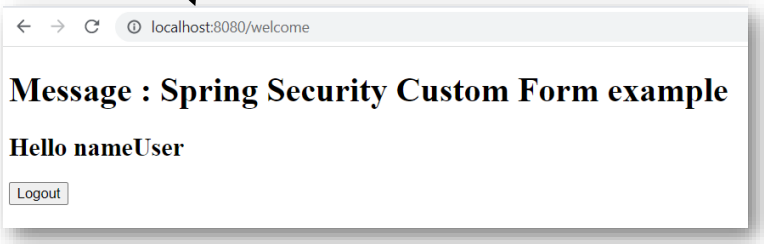
38

38

Spring Security Custom Login Form (Annotation)

Example → part 3

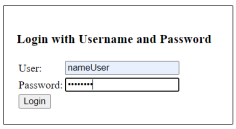




39

39

Spring Security Custom Login Form (Annotation)



login ok

Spring configuration file

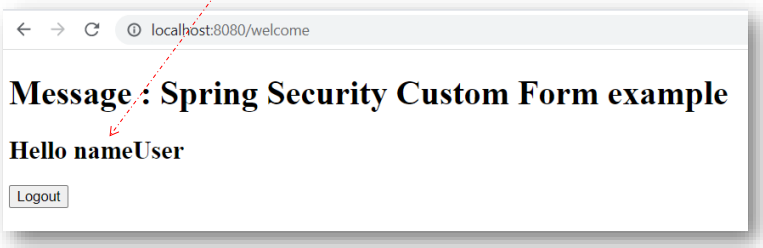
```
@Bean
SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

        http…
        formLogin(form -> form.defaultSuccessUrl("/welcome", true)
                .loginPage("/login"));

        return http.build();

}
```

40

40

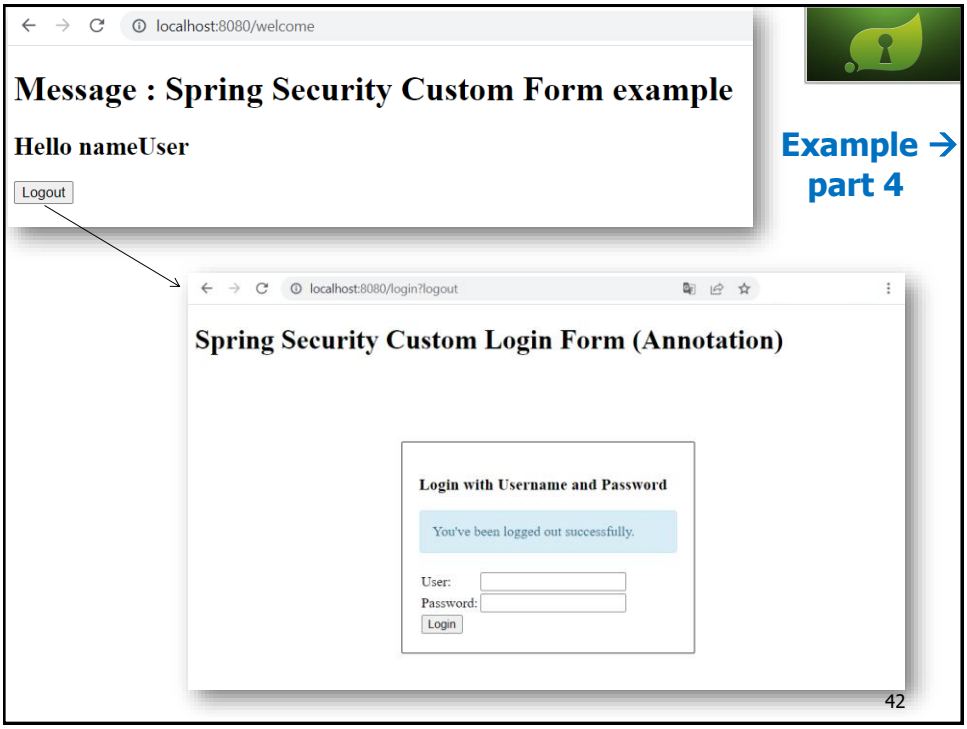
WelcomeController

```
@Controller
public class WelcomeController {

    @GetMapping(value = "/welcome")
    public String printWelcome(Model model, Principal principal) {
        model.addAttribute("username", principal.getName());
        model.addAttribute("message",
                                    "Spring Security Custom Form example");
        return "hello";
    }
```




```
.withUser("nameUser").password(encoder.encode("12345678")).roles("USER");
```

41



42

hello.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
<link rel="stylesheet" th:href="@{/css/style.css}" />
</head>
<body>
        <h1 th:text="|Message : ${message}|"></h1>
        <h2 th:text="|Hello ${username}|"></h2>

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

</body>
</html>
```

Ensure that you include the CSRF token in all POST methods.

43

43

LoginController

localhost:8080/login?logout

```
@Controller
@RequestMapping("/login")
public class LoginController {

    @GetMapping
    public String login(String error, String logout, Model model) {

        if (error != null) {
            model.addAttribute("error", "Invalid username and password!");
        }
        if (logout != null) {
            model.addAttribute("msg",
                    "You've been logged out successfully.");
        }
        return "login";
    }

}
```

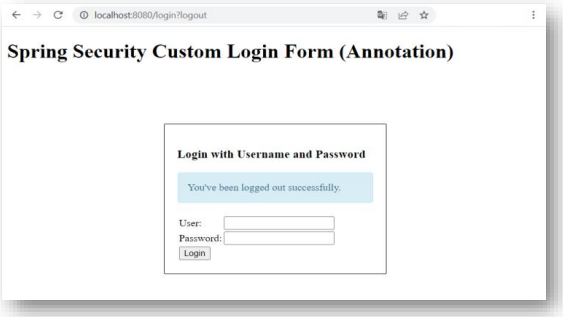
login.html

44

44

login.html

```
...
<body onload='document.loginForm.username.focus();'>
        <h1>Spring Security Custom Login Form (Annotation)</h1>

        <div id="login-box">

                <h3>Login with Username and Password</h3>
                ...
                <div th:if="${msg}" class="msg"
                        th:text="${msg}"></div>
```



45

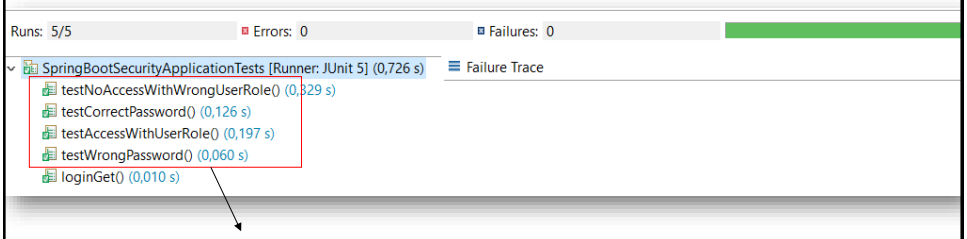
45

```
package com.springBoot.security;
import static org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestBuilders.formLogin;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.model;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.redirectedUrl;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.view;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.context.annotation.Import;
import org.springframework.security.test.context.support.WithMockUser;
import org.springframework.test.web.servlet.MockMvc;

@WebMvcTest
@Import(SecurityConfig.class)
class SpringBootSecurityApplicationTests {

        @Autowired
        private MockMvc mockMvc;
```

46

46

```
@Test
void loginGet() throws Exception {
        mockMvc.perform(get("/login"))
        .andExpect(status().isOk())
        .andExpect(view().name("login"));
}
```



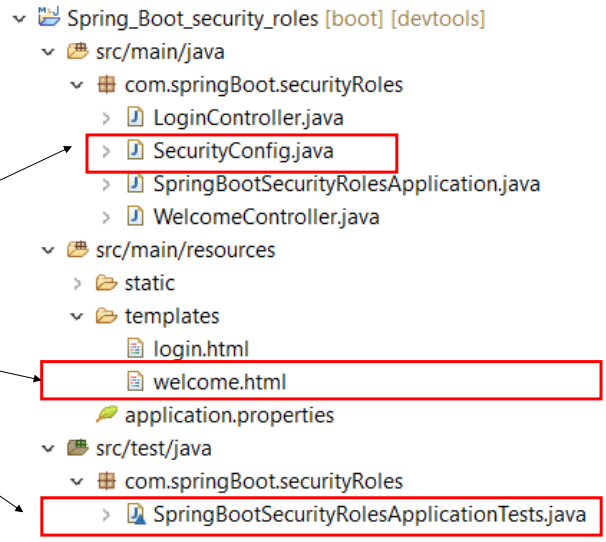
These methods are the same as Form-based login service
See slides 21 to 24.

47

47

1.6 Different view depending on the role

Example



48

48

welcome.html

If user:

localhost:8080/welcome

Hello nameUser

[ROLE_USER]

GEEN ADMIN

Logout

If admin:

localhost:8080/welcome

Hello nameAdmin

[ROLE_ADMIN, ROLE_USER]

ADMIN

Logout

SecurityConfig.java

```
.withUser("nameUser").password(encoder.encode("12345678")).roles("USER").and()
.withUser("nameAdmin").password(encoder.encode("admin1234")).roles("USER","ADMIN");
```

49

49

```
@Controller
@RequestMapping("/welcome")
public class WelcomeController {

   @GetMapping
   public String listStudents(Model model,
                              Authentication authentication) {

        List<String> listRoles = authentication.getAuthorities()
        .stream().map(GrantedAuthority::getAuthority).toList();
```

The Authentication interface represents the current user's authentication information, which includes the **Principal** and **a collection of GrantedAuthority objects**.

The **getAuthorities()** method returns a collection of the user's GrantedAuthority objects.

A **GrantedAuthority** represents a granted authority or permission that a user has, such as a role**. A user can have multiple GrantedAuthority objects**.

50

50

```
    model.addAttribute("username", authentication.getName());
    model.addAttribute("userListRoles", listRoles);

    return "welcome";
  }
}
```

returns the name of the authenticated user

welcome.html

```
<h2 th:text="|Hello ${username}|"></h2>
<h2 th:text="${userListRoles}"></h2>
```

localhost:8080/welcome

Hello nameUser

[ROLE_USER]

localhost:8080/welcome

Hello nameAdmin

[ROLE_ADMIN, ROLE_USER]

51

welcome.html

```
<th:block th:if="${#authorization.expression('hasRole(''ADMIN'')')}">
    <h2>ADMIN</h2>
</th:block>
<th:block th:unless="${#authorization.expression('hasRole(''ADMIN'')')}">
    <h3>GEEN ADMIN</h3>
</th:block>
```

If the user has the "ADMIN" role,
an <h2> heading saying "ADMIN" is displayed,
else,
an <h3> heading saying "GEEN ADMIN" is displayed.

52

52

1.7 Displaying Logged-In Username Across Multiple Screens

1.7.1 Single Controller for Multiple Screens

Use @ModelAttribute to Avoid Duplicate Code

```
@Controller
@RequestMapping("/welcome")
public class WelcomeController {
    @ModelAttribute("username")
    public String populateUsername(Authentication authentication) {
            return authentication.getName();
    }

    @GetMapping
    public String detailWelcome(Model model/*, Authentication authentication*/) {
        ...
        //model.addAttribute("username", authentication.getName());
        model.addAttribute("userListRoles", listRoles);
        return "welcome";
    }

    @GetMapping("/next")
    public String detailWelcomeNext(Model model/*, Authentication authentication*/) {
        //model.addAttribute("username", authentication.getName());
        return "welcomeNext";
    }
}
```

1.7.2 Multiple Controller for Multiple Screens

```
@Controller
@RequestMapping("/welcome")
public class WelcomeController {

    @GetMapping
    public String detailWelcome(Model model/*, Authentication authentication*/) {
      ...
      //model.addAttribute("username", authentication.getName());
      model.addAttribute("userListRoles", listRoles);

      return "welcome";
    }
```

```
@Controller
@RequestMapping("/welcomeNext")
public class WelcomeNextController {

  @GetMapping
   public String listStudents(Model model/*, Authentication authentication*/) {
        //model.addAttribute("username", authentication.getName());
        return "welcomeNext";
  }
}
```

55

55

1.7.2 Multiple Controller for Multiple Screens

If you want **@ModelAttribute**, **@ExceptionHandler**, and other such annotations to apply **more globally** (across multiple controllers), you can declare them in a class annotated with **@ControllerAdvice**.

```
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.security.core.Authentication;
import org.springframework.web.bind.annotation.ControllerAdvice;

@ControllerAdvice
public class GlobalControllerAdvice {

   @ModelAttribute("username")
   public String populateColors(Authentication authentication) {
      return authentication == null?"": authentication.getName();
   }

}
```

56

56