07

Personalizando Login e Logout

Transcrição

Não foi comentado anteriormente mas é curioso. De onde veio o formulário de login que utilizamos para testar e autenticar usuários em nossa aplicação? Do próprio *Spring*! Este que apesar de funcionar, não é o ideal. Queremos nosso próprio formulário com os estilos da casa do código. Vamos criar então uma nova página chamada `loginForm` dentro da pasta `webapp/WEB_INF/views` e dentro deste novo *JSP* usaremos o seguinte código:

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


<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Livros de Java, Android, iPhone, Ruby, PHP e muito mais - Casa do Código</title>
    <c:url value="/resources/css" var="cssPath" />
    <link rel="stylesheet" href="${cssPath}/bootstrap.min.css" />
    <link rel="stylesheet" href="${cssPath}/bootstrap-theme.min.css" />
    <style type="text/css">
        body{
            padding: 60px 0px;
        }
    </style>
</head>
<body>
    <div class="container">
        <h1>Login Casa do Código</h1>
        <form:form servletRelativeAction="/login" method="post">
            <div class="form-group">
                <label>Nome</label>
                <input type="text" name="username" class="form-control" />
            </div>
            <div class="form-group">
                <label>Senha</label>
                <input type="password" name="password" class="form-control" />
            </div>
            <button type="submit" class="btn btn-primary">Logar</button>
        </form:form>
    </div>
</body>
</html>
```

E para que possamos utilizar essa nova **View**, devemos criar um novo **Controller** que chamaremos de `LoginController` que será anotado com `@Controller` e terá o método `loginForm` que retorna apenas o nome da *view* do login e será mapeado para `/login` aceitando apenas requisições do tipo **GET**.

```
@Controller
public class LoginController {

    @RequestMapping(value="/login", method=RequestMethod.GET)
    public String loginForm(){
        return "loginForm";
    }

}
```

Fazer isso ainda não é o suficiente, precisamos de alguma forma informar o *Spring* de que ele deve usar a nossa página e não a página padrão dele. Para isso, precisamos fazer mais configurações. Na classe `SecurityConfiguration` no método `configure` que recebe o `HttpSecurity` temos o seguinte código.

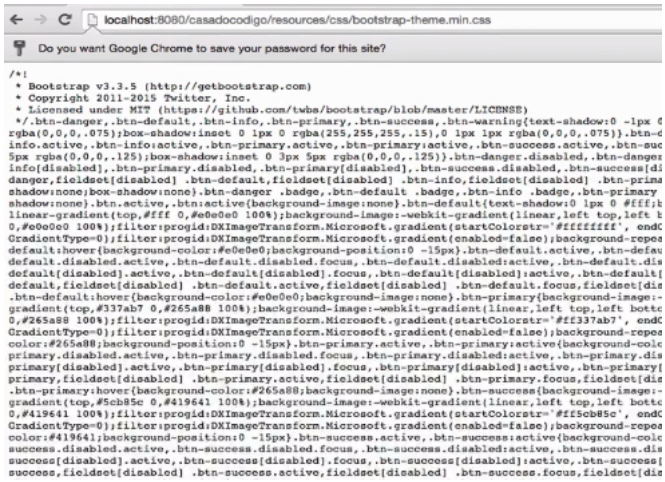
```
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
    .antMatchers("/produtos/form").hasRole("ADMIN")
    .antMatchers("/carrinho/**").permitAll()
    .antMatchers(HttpMethod.GET, "/produtos").permitAll()
    .antMatchers(HttpMethod.POST, "/produtos").hasRole("ADMIN")
    .antMatchers("/produtos/**").permitAll()
    .antMatchers("/").permitAll()
    .anyRequest().authenticated()
    .and().formLogin();
}
```

O objeto `HttpSecurity` tem um método chamado `loginPage` no qual passamos o caminho que será atendido pelo `LoginController` que no caso é `/login` . E neste caso precisaremos usar o `permitAll` para que o *Spring* não bloqueie a página, já que esta não está inclusa nos métodos `Matchers` . Assim teremos:

```
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
    .antMatchers("/produtos/form").hasRole("ADMIN")
    .antMatchers("/carrinho/**").permitAll()
    .antMatchers(HttpMethod.GET, "/produtos").permitAll()
    .antMatchers(HttpMethod.POST, "/produtos").hasRole("ADMIN")
    .antMatchers("/produtos/**").permitAll()
    .antMatchers("/").permitAll()
    .anyRequest().authenticated()
    .and().formLogin().loginPage("/login").permitAll();
}
```

Já podemos reiniciar o servidor e tentar acessar uma das páginas bloqueadas, como por exemplo `localhost:8080/casadocodigo/produtos/` e seremos levados a nossa página de login personalizada.

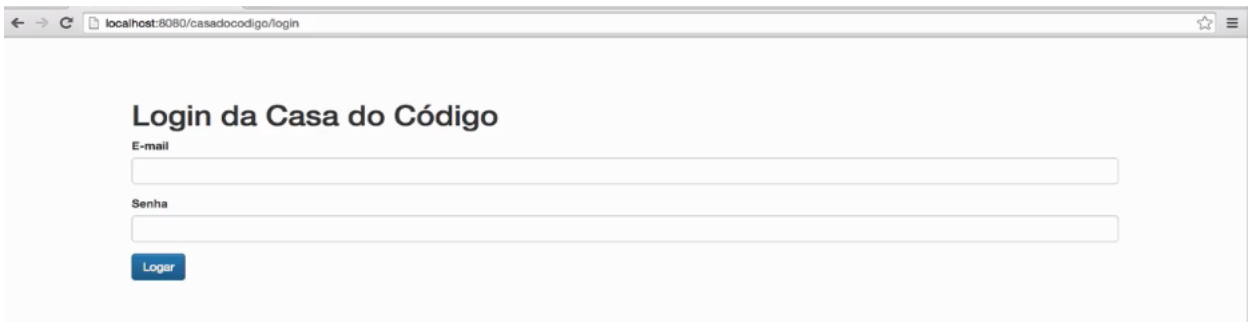
Mas ao tentarmos usar o usuário e senha que cadastramos no banco de dados, percebmos que funciona, mas há algo estranho, o código do **Bootstrap** foi impresso na página ao invés de sermos direcionados para a listagem de produtos. Mas porque?



Isso acontece porque o *Spring* redireciona para a última página ou recurso em que o acesso foi bloqueado e por incrível que pareça, deixamos as configurações do **Spring Security** bloqueando os arquivos de **CSS** e **JS** do *Bootstrap*. Precisamos usar o método `Matchers` com o valor `/resources/**` para desbloquear estes arquivos na classe `SecurityConfiguration`. Então teremos:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
    .antMatchers("/produtos/form").hasRole("ADMIN")
    .antMatchers("/carrinho/**").permitAll()
    .antMatchers(HttpMethod.GET, "/produtos").permitAll()
    .antMatchers(HttpMethod.POST, "/produtos").hasRole("ADMIN")
    .antMatchers("/produtos/**").permitAll()
    .antMatchers("/resources/**").permitAll()
    .antMatchers("/").permitAll()
    .anyRequest().authenticated()
    .and().formLogin().loginPage("/login").permitAll();
}
```

Após esta alteração e reinicialização do servidor, já podemos ver visualmente alguma diferença no formulário de login.



O que falta fazermos agora para que o login funcione completamente é sua segunda parte, o **logout**. Acontece que o *logout* já está pronto, porém, por padrão, o *Spring* só aceita requisições do tipo **POST** para o *logout* que pode ser feito no **path** `/logout`.

Para deixarmos o logout um pouco mais simples, faremos com que o mesmo possa ser feito através de requisições do tipo **GET**. Assim, precisaremos apenas de mais algumas configurações na classe `SecurityConfiguration`.

No objeto `HttpSecurity` precisaremos apenas usar o método `and` para adicionamos mais uma informação na cadeia de chamadas de métodos, este que por sua vez será seguido pela chamadas dos métodos `logout` e `logoutRequestMatcher` que receberá um objeto do tipo `AntPathRequestMatcher` indicando qual o caminho no qual o *Spring* ao receber uma requisisção, deverá fazer logout do usuário corrente. O método `configure` da classe `SecurityConfiguration` deve ficar da seguinte forma:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
    .antMatchers("/produtos/form").hasRole("ADMIN")
    .antMatchers("/carrinho/**").permitAll()
    .antMatchers(HttpMethod.GET, "/produtos").permitAll()
    .antMatchers(HttpMethod.POST, "/produtos").hasRole("ADMIN")
    .antMatchers("/produtos/**").permitAll()
    .antMatchers("/resources/**").permitAll()
    .antMatchers("/").permitAll()
    .anyRequest().authenticated()
    .and().formLogin().loginPage("/login").permitAll()
    .and().logout().logoutRequestMatcher(new AntPathRequestMatcher("/logout"));
}
```

Assim ao logarmos podemos ver todos os links antes ocultos e cadastrar novos produtos normalmente e para sairmos da aplicação pasta digitar `logout` na barra de endereços do navegador, sendo que o caminho completo deve ficar da seguinte forma: `localhost:8080/casadocodigo/logout`.

Recapitulando

Nesta aula, vimos como restringir partes da aplicação para usuários autenticados, como fazer o login e logout, particularidades sobre segurança e até aprendemos como evitar ataques como o *CSRF*. Fizemos também o *Spring* validar os usuários e criamos as *roles* que representam as permissões dos usuários. Aprendemos a usar algumas das tags da *security taglib* do *Spring* e vamos aprender muito mais sobre *Spring* nas próximas aulas.