



LISTENERS, SESSÃO E COOKIES

Complementando a aplicação

Visão Geral

Listeners

Sessão

Cookies

Verificação de Aprendizado

VISÃO GERAL

Chegamos ao final do nosso curso. Nessa última aula vamos falar sobre três tópicos complementares no contexto do aplicativo que estamos usando como exemplo, mas importantes no desenvolvimento de aplicações Web.

Listeners permitem empregar o padrão *publisher-subscriber (observer)* de uma forma muito simples com o Spring Boot. Sessões e *cookies* permitem armazenar dados temporariamente. Enquanto o primeiro guarda dados em um espaço na memória do servidor, o outro guarda dados diretamente no navegador.

Você pode aprender mais sobre esses tópicos na Seções seguintes, ou assistir o vídeo abaixo para ter uma ideia de como tudo funciona.

LISTENERS

Listeners são parte do [gerenciamento de eventos](#) do Spring. Um evento é algo que acontece, como o login na aplicação, o acesso ao banco de dados, ou a inicialização da aplicação. Esses eventos emitem notificações que são captadas por *listeners*.

Assim como quase tudo no Spring, eventos e *listeners* podem ser estendidos para atender necessidades específicas. Nesta seção, vamos ver como usar um evento e um *listener* para identificar o usuário atualmente logado na aplicação.

Um *listener* atua sob um método que você definiu. Por isso, vamos começar criando um método. Nosso método será criado na classe `crudcidades.SecurityConfig`, porque esse método está relacionado com a autenticação.

Observe que o método poderia ser criado em qualquer classe gerenciada pelo Spring Boot, como a `cidade.CidadeController`, por exemplo.

Crie o método público `printUsuarioAtual()` na classe `crudcidades.SecurityConfig`. O método deve receber um parâmetro do tipo `org.springframework.security.authentication.event.InteractiveAuthenticationSuccessEvent`, que vamos mapear com a variável `event`. Esse objeto carrega valores e métodos que são entregues pelo `evento` que foi disparado ao *listener*. Podemos usar esse objeto para extrair o nome do usuário atualmente logado. Como de costume, o Spring Boot se encarrega de instanciar o objeto, por meio da *injeção de dependências*.

```
43     public void printUsuarioAtual(InteractiveAuthenticationSuccessEvent event) {  
44  
45 }
```

No corpo do método, vamos usar a variável `event` para ter acesso ao método `getAuthentication()`, do objeto `org.springframework.security.authentication.event.InteractiveAuthenticationSuccessEvent`. Esse método retorna o objeto que representa a autenticação. Por meio desse objeto, podemos ter acesso ao método `getName()`, que retorna o nome do usuário atual. Após ter identificado o usuário, vamos imprimir o nome do usuário na console do sistema.

```
43     public void printUsuarioAtual(InteractiveAuthenticationSuccessEvent event) {  
44  
45         var usuario = event.getAuthentication().getName();  
46  
47         System.out.println(usuario);  
48     }
```

Como o esse método sabe o nome do usuário atual? Você já informou pra ele! Quando? Quando implementou a interface `UserDetails`. Esse é um dos grandes benefícios do uso de interfaces.

O último passo é identificar que esse método deve ser acionado por um *listener*. Para isso, vamos adicionar a anotação `org.springframework.context.event.EventListener` imediatamente antes do método. Adicionalmente, vamos usar a classe `org.springframework.security.authentication.event.InteractiveAuthenticationSuccessEvent.class` como argumento da anotação.

```
42     @EventListener(InteractiveAuthenticationSuccessEvent.class)
43     public void printUsuarioAtual(InteractiveAuthenticationSuccessEvent event) {
44
45         var usuario = event.getAuthentication().getName();
46
47         System.out.println(usuario);
48     }
```

A classe `org.springframework.security.authentication.event.InteractiveAuthenticationSuccessEvent` representa um evento que é disparado pelo Spring Boot quando o usuário loga na aplicação (por isso `AuthenticationSuccess` no nome da classe). A anotação serve como um *inscrito*, que *ouve* esse tipo de evento. O próprio framework cuida de tudo e, quando o evento é disparado, o método que definimos entra em ação, imprimindo o nome do usuário na console do sistema.

Contudo, imprimir algo na console do sistema costuma ser pouco útil para uma aplicação Web. Por isso, na próxima seção vamos salvar o nome do usuário na *sessão da aplicação* para podermos apresentar na tela principal.

O código desenvolvido nesta Seção está disponível no [Github](#), na branch `semana08-10-misc-listener`.

SESSÃO

Vamos usar o método `listar()` da classe `cidade.CidadeController` para identificar o usuário atual e também salvar seu nome na sessão atual. Para isso, vamos adicionar dois parâmetros nesse método. O primeiro parâmetro é do tipo `java.security.Principal`. Esse objeto representa o usuário logado na aplicação. O segundo parâmetro é do tipo `javax.servlet.http.HttpSession`. Esse objeto representa a `sessão` atual. Ambos objetos são instanciados automaticamente pelo Spring Boot como parte da injeção de dependências.

Vários objetos podem ser injetados automaticamente como parâmetros. Você pode consultar a [documentação oficial](#) para conhecer esses objetos.

```
25  @GetMapping("/")
26  public String listar(Model memoria, Principal usuario, HttpSession sessao) {
```

No corpo do método `listar()`, podemos usar a variável `usuario` para acessar o método `getName()`, que retorna o nome do usuário logado no sistema. Também podemos usar a variável `sessao` para acessar o método `setAttribute()`. Esse método recebe dois parâmetros. O primeiro é uma `String` que define o nome do atributo que será criado na sessão. O segundo parâmetro é o objeto que será armazenado na sessão.

```
25  @GetMapping("/")
26  public String listar(Model memoria, Principal usuario, HttpSession sessao) {
27
28      memoria.addAttribute("listaCidades", repository
29                      .findAll()
30                      .stream()
31                      .map(Cidade::clonar)
32                      .collect(Collectors.toList()));
33
34      sessao.setAttribute("usuarioAtual", usuario.getName());
35
36      return "/crud";
37  }
```

A linha 34 na Figura acima usa a variável `sessao` para criar um atributo na sessão chamado `usuarioAtual`. O nome do usuário atual, extraído com `usuario.getName()`, é então armazenado como o valor do atributo `usuarioAtual`.

O último passo é atualizar nossa página `crud.ftl` para que ela apresente o valor armazenado no atributo de sessão `usuarioAtual`. Vamos adicionar o nome do usuário na barra de menu, conforme pode ser visto na linha 15 da Figura a seguir. O Freemarker permite acessar a sessão usando o objeto `Session`. Do lado direito do objeto, vamos usar o nome do atributo criado na sessão. Tudo isso precisa estar dentro da sintaxe de interpolação (`${}`).

```
13 <body>
14     <nav class="navbar navbar-expand-sm bg-dark">
15         <span class="navbar-brand text-white">${Session.usuarioAtual}</span>
16             <ul class="navbar-nav ml-auto">
17                 <li class="nav-item">
18                     <a
19                         href="/logout"
20                         class="nav-link btn btn-secondary"
21                         >Sair da aplicação</a>
22                 </li>
23             </ul>
24         </nav>
```

Pronto! Agora você pode executar e logar na sua aplicação para ver o resultado.

The screenshot shows a web application interface. At the top is a dark navigation bar with the user name "anna" on the left and a "Sair da aplicação" (Logout) button on the right. A green arrow points from the "anna" text to the user name field in the code above. Below the navigation bar is a light gray header section containing the title "GERENCIAMENTO DE CIDADES" and the subtitle "UM CRUD PARA CRIAR, ALTERAR, EXCLUIR E LISTAR CIDADES". The main content area has two input fields: "Nome" (Name) and "Estado" (State), both with placeholder text "Informe o nome da cidade" (Enter city name) and "Informe o estado ao qual a cidade pertence" (Enter the state to which the city belongs). Below these is a blue "CRIAR" (Create) button. At the bottom is a dark table header row with columns labeled "Nome" (Name), "Estado" (State), and "Ações" (Actions).

O código desenvolvido nesta Seção está disponível no [Github](#), na branch `semana08-20-misc-sessao`.

COOKIES

Cookies representam informações que são armazenadas como um par de valores do tipo texto no navegador do usuário. Enquanto um valor representa o nome do cookie, o outro valor representa o conteúdo armazenado no cookie.

Cookies são muito comuns em páginas Web. Com certeza você já acessou algum website que te informa que seus dados serão armazenados para 'melhorar sua experiência no site'. Normalmente, esses dados são salvos em cookies.

Diferente da sessão, um cookie armazena os dados no navegador do usuário em vez de um espaço na memória do servidor. Isso é importante porque a memória do servidor é encerrada no logout, enquanto o cookie pode permanecer no navegador. Outra diferença é que o cookie armazena um par de **Strings**, enquanto uma sessão pode armazenar um objeto.

É claro que o usuário pode sempre se livrar dos cookies limpando o cache do navegador, ou com a expiração dos cookies.

Dessa forma, sempre que você acessar uma determinada página, seu aplicativo pode verificar pela existência de cookies criados previamente. Se eles existirem, você pode aproveitar os dados para melhorar a experiência do usuário e indicar produtos relacionados com pesquisas prévias, por exemplo.

O gerenciamento de cookies no Spring Boot é muito similar ao da sessão. O cookie também é injetado com injeção de dependência. Dessa forma, tudo que precisamos fazer é acessar o objeto e adicionar valores.

Para mostrar o uso de cookies, vamos gravar em um cookie o momento em que cada operação do CRUD foi acessada pela última vez. Para isso, vamos precisar alterar os métodos que representam o CRUD na classe **cidade.CidadeController**.

As alterações são sempre as mesmas: (i) adicionar um parâmetro do tipo **javax.servlet.http.HttpServletResponse**, e (ii) usar esse objeto para adicionar um cookie com os valores definidos.

O objeto **javax.servlet.http.HttpServletResponse** representa o resultado de uma solicitação que é enviada como retorno para o navegador. É esse objeto tem o método **addCookie()**. Esse método recebe como parâmetro um objeto do tipo **javax.servlet.http.Cookie**, representando um par de valores **String**, sendo que o primeiro é nome do cookie e o segundo é o valor que o cookie representa.

```

28     @GetMapping("/")
29     public String listar(
30         Model memoria,
31         Principal usuario,
32         HttpSession sessao,
33         HttpServletResponse response) {
34
35         response.addCookie(new Cookie("listar", LocalDateTime.now().toString()));

```

Na Figura acima podemos ver como o método **listar()** ficou após a modificação. A linha 33 mostra a definição da variável **response**, como parâmetro do tipo **javax.servlet.http.HttpServletResponse**. Já a linha 35 mostra como o método **addCookie()** é usado para adicionar um novo cookie.

Nesse exemplo, o nome do cookie é **listar**, e o valor armazenado no cookie é extraído do objeto **java.time.LocalDateTime**, representando o dia e hora atuais.

Veja como ficou a replicação dessa ação no método `criar()`. Nesse trecho de código, as mudanças estão nas linhas 53 e 55.

```
48     @PostMapping("/criar")
49     public String criar(
50         @Valid Cidade cidade,
51         BindingResult validacao,
52         Model memoria,
53         HttpServletResponse response) {
54
55         response.addCookie(new Cookie("criar", LocalDateTime.now().toString()));
```

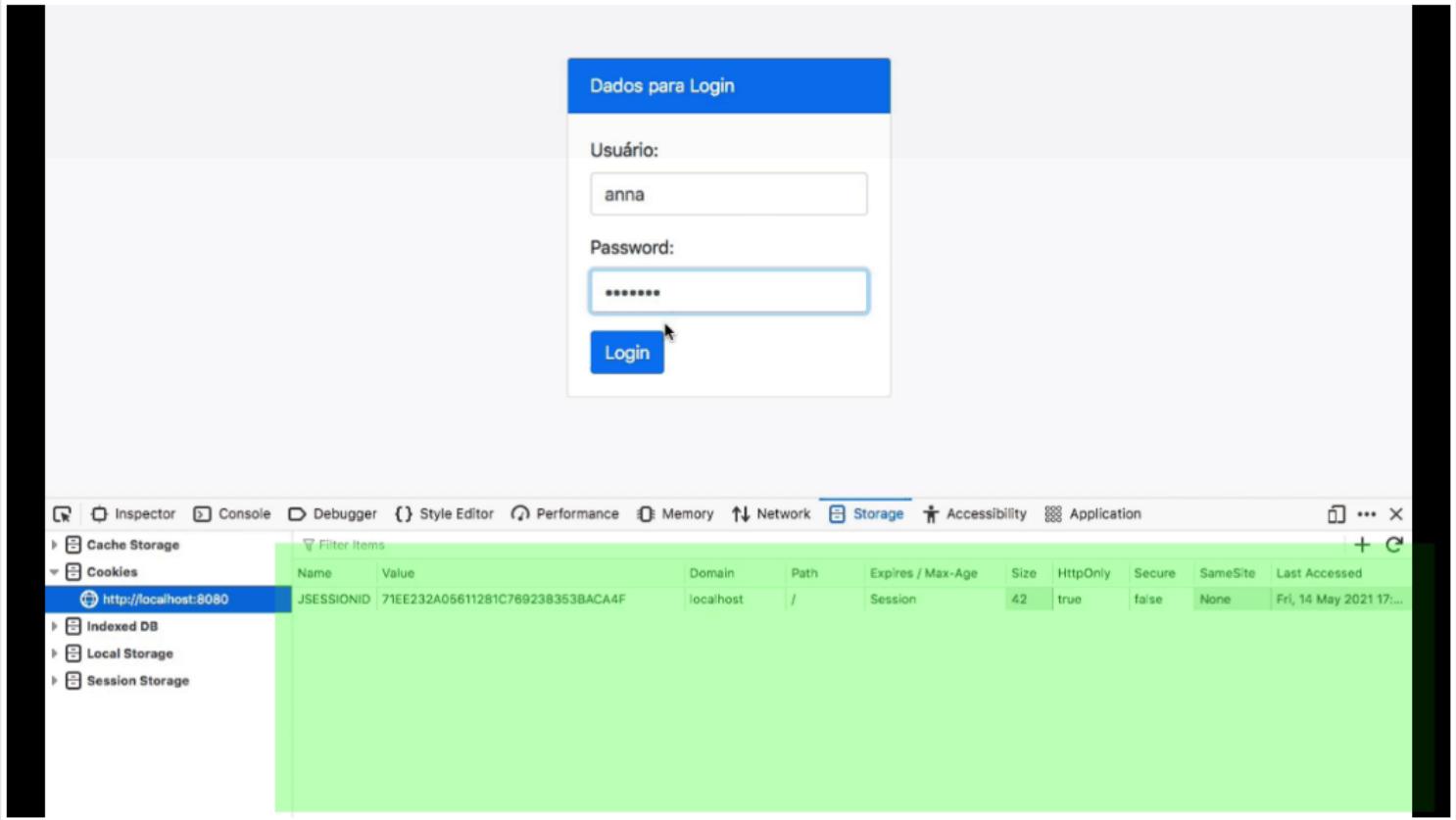
Veja como ficou a replicação dessa ação no método `excluir()`. Nesse trecho de código, as mudanças estão nas linhas 82 e 84.

```
78     @GetMapping("/excluir")
79     public String excluir(
80         @RequestParam String nome,
81         @RequestParam String estado,
82         HttpServletResponse response) {
83
84         response.addCookie(new Cookie("excluir", LocalDateTime.now().toString()));
```

Veja como ficou a replicação dessa ação no método `alterar()`. Nesse trecho de código, as mudanças estão nas linhas 114 e 116.

```
109    @PostMapping("/alterar")
110    public String alterar(
111        @RequestParam String nomeAtual,
112        @RequestParam String estadoAtual,
113        Cidade cidade,
114        HttpServletResponse response) {
115
116        response.addCookie(new Cookie("alterar", LocalDateTime.now().toString()));
```

Feito isso, é possível acessar o aplicativo e verificar, diretamente do navegador, os cookies que são adicionados à medida que os métodos são acessados.



Para mostrar como ler um cookie, vamos criar um novo método nessa classe chamado `mostrar()`, que retorna uma `String`, representando o resultado que deve ser apresentando na página Web. Esse método recebe uma `String` como parâmetro, representando o valor armazenado no cookie. O nome da variável de acesso ao parâmetro deve ser o mesmo usado para nomear o cookie. Nesse exemplo, vamos usar `listar`, que é o nome do cookie criado para guardar o horário do último acesso ao método `listar()`.

Vamos mapear esse método com a URL `/mostrar`. Para isso, basta adicionar `@GetMapping("/mostrar")` imediatamente antes da definição do método. Como esse método retorna uma `String` em vez de uma página Web, precisamos adicionar a anotação `org.springframework.web.bind.annotation.ResponseBody`, imediatamente antes da definição do método. Veja como ficou o resultado.

```

134  @GetMapping("/mostrar")
135  @ResponseBody
136  public String mostraCookieAlterar(@CookieValue String listar) {
137      return "Último acesso ao método listar(): " + listar;
138 }
```

A anotação `org.springframework.web.bind.annotation.CookieValue`, adicionada imediatamente antes da definição do parâmetro do método, é usada pelo Spring Boot para mapear o cookie cujo nome corresponde ao nome do parâmetro (nesse caso, `listar`). Por fim, o corpo do método simplesmente retorna o valor em `listar` concatenado com um texto indicando o que esse valor representa.

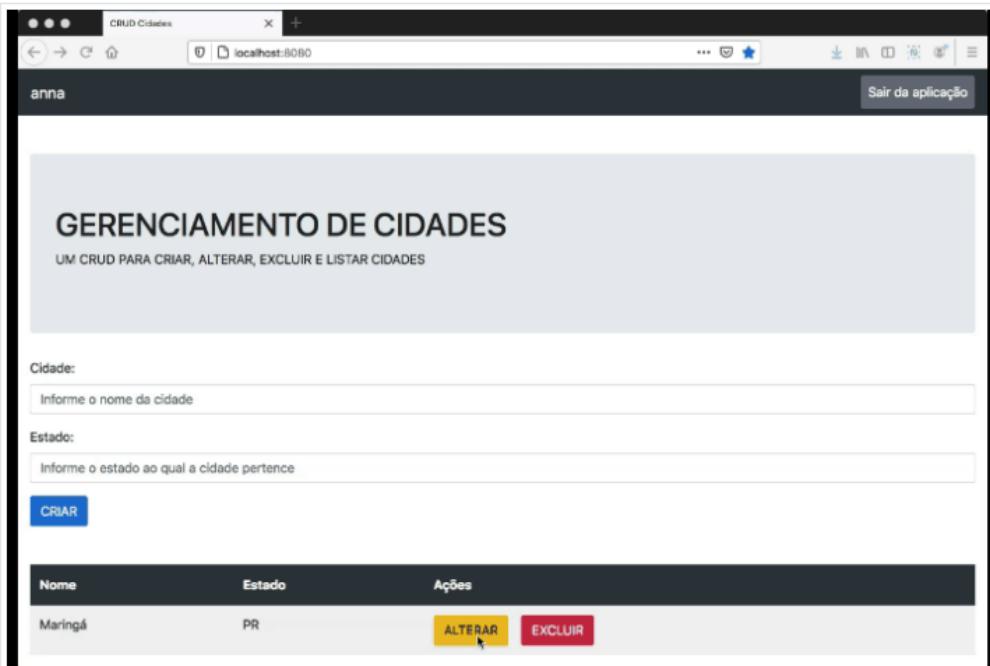
Como nossa aplicação está usando autenticação e autorização, precisamos adicionar uma regra permitindo acesso à URL `/mostrar`. Para isso, precisamos adicionar uma regra de permissão na classe `crudcidades.SecurityConfig`.

```

13  @EnableWebSecurity
14  @Configuration
15  public class SecurityConfig extends WebSecurityConfigurerAdapter {
16
17      protected void configure(HttpSecurity http) throws Exception {
18          http
19              .csrf().disable()
20              .authorizeRequests()
21              .antMatchers("/").hasAnyAuthority("listar", "admin")
22              .antMatchers("/criar").hasAuthority("admin")
23              .antMatchers("/excluir").hasAuthority("admin")
24              .antMatchers("/preparaAlterar").hasAuthority("admin")
25              .antMatchers("/alterar").hasAuthority("admin")
26              .antMatchers("/mostrar").authenticated()
27              .anyRequest().denyAll()
28                  .and()
29              .formLogin()
30              .loginPage("/login.html").permitAll()
31              .defaultSuccessUrl("/", false)
32                  .and()
33              .logout().permitAll();
34      }

```

Adicionamos a linha 26. Nessa linha, além de definir a nova URL, também adicionamos uma permissão dizendo que qualquer usuário autenticado tem acesso à URL. Agora é só executar a aplicação, acessar o a página principal e depois acessar a nova URL.



O código desenvolvido nesta Seção está disponível no [Github](#), na branch `semana08-30-misc-cookies`.