



## SEGURANÇA

Inserindo autenticação e autorização na aplicação

## VISÃO GERAL

A maioria das aplicações Web têm algum tipo de segurança. Segurança é um termo amplo que abrange vários aspectos, como conexão segura, cifragem de dados entre outros. Nesta seção, nós vamos focar em dois dos mecanismos mais comuns de segurança: autenticação e autorização.

Esse tópico é tão grande que o Spring tem um projeto inteiro só pra cuidar disso: o Spring Security. Essa seção se concentra naquilo que é fundamental para os mecanismos de autenticação e autorização, incluindo a definição de usuários em um banco de dados e suas permissões.

Cada seção detalha os aspectos de autenticação e autorização em aplicações Web. Se você tem pressa, talvez prefira ver o vídeo abaixo.

## AUTENTICAÇÃO

Autenticação é o processo de verificar se alguém é quem diz ser. Isso pode ser feito de diferentes formas, como usando uma senha ou por biometria. Autenticação é um pré-requisito da autorização. Autorização significa que alguém tem acesso a algum recurso.

A forma mais simples de inserir autenticação em um aplicativo Spring Boot é simplesmente adicionar a dependência `spring-boot-starter-security` no seu `pom.xml`.

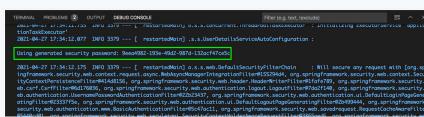
```

52      <dependency>
53          <groupId>org.springframework.boot</groupId>
54          <artifactId>spring-boot-starter-data-jpa</artifactId>
55      </dependency>
56
57      <dependency>
58          <groupId>org.springframework.boot</groupId>
59          <artifactId>spring-boot-starter-security</artifactId>
60      </dependency>
61  </dependencies>

```

Ao executar a aplicação você vai perceber que o Spring Boot vai te redirecionar para a tela de login.

Mas como assim? Eu nem criei o usuário! Verdade, mas o Spring Boot cria um usuário de exemplo (`user`) pra você e coloca toda a aplicação sob essa segurança. Você pode consultar na consola do sistema a senha criada automaticamente pelo Spring Boot.



```

2023-04-27 22:34:12.477 190339 [main] INFO org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext 139 - Root WebApplicationContext: started in 0.021 seconds
2023-04-27 22:34:12.477 190339 [main] INFO org.springframework.security.web.DefaultSecurityFilterChain 217 - Will secure any request with [org.springframework.security.web.context.request.async.WebAsyncManagerIntegrationFilter, org.springframework.security.web.header.HeaderWriterFilter, org.springframework.security.web.context.HttpSessionContextIntegrationFilter, org.springframework.security.web.authentication.logout.LogoutFilter, org.springframework.security.web.savedrequest.RequestCacheAwareFilter, org.springframework.security.web.authentication.www.BrowserAuthCookieFilter, org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter, org.springframework.security.web.authentication.www.BasicAuthenticationFilter, org.springframework.security.web.authentication.logout.LogoutHandler]

```

Mas, é claro, normalmente, o que queremos é definir usuários e autorizações para esses usuários. Nesta Seção, vamos tratar da autenticação, enquanto na Seção seguinte detalhamos o processo de autorização.



Nosso primeiro passo é indicar para o Spring Boot que vamos usar os recursos de autenticação e autorização dessa aplicação. Para isso, crie uma classe em `src/main/java/com.example.demo.config.SecurityConfig` e insira a anotação `@EnableWebSecurity` e `@Configuration`, comentando-a com `//Início da configuração` imediatamente antes da definição da classe. O nome da anotação descreve muito bem o que ela faz – habilita o uso dos recursos do Spring Security.

Também precisamos adicionar a anotação `org.springframework.context.annotation.Configuration` imediatamente antes da definição da classe. Essa anotação indica que essa classe carrega configurações que devem ser usadas pelo Spring Boot.

Para garantir que as senhas são armazenadas de forma segura, o Spring Security depende de um algoritmo de cifragem. Para definir qual algoritmo iremos usar, vamos criar um método e dizer para o Spring Boot usar o algoritmo definido nesse método.

Vamos, então, conceber, criando, um método que retorna um objeto `org.springframework.security.crypto.encrypt.PasswordEncoder`. Esse objeto define uma interface que todos os algoritmos de cifragem devem seguir. Vamos chamar o método de `cifrador()`:

No corpo do método vamos definir o algoritmo de cifragem que será usado. O Spring Security já possui alguns cifradores, por isso, não precisaremos criar nada. Para esse exemplo, vamos usar o `org.springframework.security.crypto.encrypt.BCryptPasswordEncoder`, só que precisamos fazer uma nova instância dessa classe. Para finalizar, vamos adicionar a anotação `org.springframework.context.annotation.Bean` imediatamente antes do método. Isto faz com que o Spring Boot gerencie automaticamente a configuração definida nesse método.

```

1  package com.example.demo.config;
2
3  import org.springframework.context.annotation.Bean;
4  import org.springframework.context.annotation.Configuration;
5  import org.springframework.security.config.annotation.web.builders.HttpSecurity;
6  import org.springframework.security.core.userdetails.UserDetailsService;
7  import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
8
9  @Configuration
10 @EnableWebSecurity
11 @Configuration
12 public class SecurityConfig {
13
14     @Bean
15     public PasswordEncoder cifrador() {
16         return new BCryptPasswordEncoder();
17     }
18 }

```

Finalmente, podemos associar nossos usuários e o conjunto de papéis que eles têm acesso. Uma **autoração ou papel** define qual conjunto de recursos ou operações que os usuários possuem. Os usuários e suas **papéis/autorizações** podem ser armazenados em diferentes locais, como na memória da aplicação, ou no banco de dados. Para esse primeiro exemplo, vamos definir os usuários na memória da aplicação.

**Papéis (roles) e autorizações (authorities)** podem ser usados de forma intercambiável em alguns materiais. Contudo, o Spring Security implementa o conceito de papéis e autorizações **como duas coisas diferentes**. Aqui, estamos usando ambos como sinônimos, por simplicidade. Mas em configurações de segurança mais complexas, talvez você precise usar os dois.

Para isso, vamos criar um novo método (`configure()`, linha 17, na Figura abaixo). Esse método retorna um objeto `org.springframework.security.core.userdetails.UserDetailsService`, que é responsável pelo gerenciamento como o gerenciamento de usuários é feito. Nesse primeiro exemplo, os usuários estarão armazenados em memória. Em uma尧尧 posterior, vamos armazenar os usuários em um banco de dados. Veja como fica o código na Figura abaixo.

Cada usuário é definido programmaticamente, como uma instância da classe `org.springframework.security.core.userdetails.UserDetails`. A classe `UserDetails` fornece um método útil que permite criar um novo usuário (`withUsername(String)`), sua senha (`password(String)`), e papéis associados (`roles(String)`).

Observe o uso do método `cifrador()`, para atribuir a senha do usuário, nas linhas 19 e 24 da Figura abaixo. Veja como o mesmo cifrador usado para criar a senha nas linhas 19 e 24 também será usado pelo Spring Security para verificar a senha – por isso a necessidade da anotação `@Bean` no método `cifrador()`. Veja como fica o código na Figura abaixo.

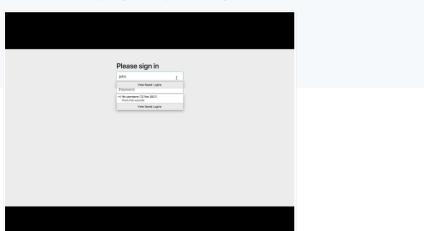
Por fim, basta retornar um novo objeto `InMemoryUserDetailsManager` com os dois usuários criados (linha 28). Veja como fica o código completo na Figura abaixo.

```

1  package com.example.demo.config;
2
3  import org.springframework.context.annotation.Bean;
4  import org.springframework.context.annotation.Configuration;
5  import org.springframework.security.config.annotation.web.builders.HttpSecurity;
6  import org.springframework.security.core.userdetails.UserDetailsService;
7  import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
8
9  @Configuration
10 @EnableWebSecurity
11 @Configuration
12 public class SecurityConfig {
13
14     @Bean
15     public InMemoryUserDetailsManager configure() throws Exception {
16         UserDetails john = User.withUsername("john")
17             .password(cifrador().encode("exit123"))
18             .roles("user")
19             .build();
20
21         UserDetails anna = User.withUsername("anna")
22             .password(cifrador().encode("exit123"))
23             .roles("admin")
24             .build();
25
26         return new InMemoryUserDetailsManager(john, anna);
27     }
28
29     @Bean
30     public PasswordEncoder cifrador() {
31         return new BCryptPasswordEncoder();
32     }
33 }

```

Ao executar a aplicação, a tela de login padrão do Spring Security é carregada. Ao inserir um usuário e senha conforme definidos na classe `SecurityConfig`, a tela do aplicativo é carregada.



O código desenvolvido nesta Seção está disponível no [Github](#), na branch `sistema87-10-autenticacao`.

## AUTORIZAÇÃO

Com o usuário autenticado, o próximo passo é definir as autorizações. Para definir as autorizações, vamos criar um novo método: `filter(org.springframework.security.config.annotation.web.builders.HttpSecurity http)`. O nome do método é irrelevante, mas o tipo de retorno e o parâmetro não podem ser alterados. O mecanismo de segurança do Spring Security é baseado na execução de uma cadeia de filtros. Os filtros são objetos que carregam regras de acesso, definindo as autorizações, por exemplo. O objeto `SecurityFilterChain` define esse método como mais um filtro que deve ser executado nessa cadeia. Já o objeto `HttpSecurity` permite acesso à métodos para definir as autorizações dos papéis dos usuários.

Quer saber mais sobre como funcionam os filtros de segurança? Que tal dar uma olhada na [documentação oficial](#).

O método `filter` precisa ser anotado com `@Bean`, assim o Spring se encarrega de cuidar do ciclo de vida do método e adicioná-lo no momento certo de execução. Com o método criado, podemos criar as regras de autorização. Por simplicidade, neste projeto vamos começar usando o parâmetro `http` para desabilitar o CSRF.

```
● ● ● SecurityConfig.java
33 @Bean
34 public SecurityFilterChain filter(HttpSecurity http) throws Exception {
35     return http
36         .csrf().disable()
37 }
```

Snipped

Agora, vamos começar a mapear as autorizações com as URLs da aplicação. Para isso, vamos usar o método `authorizeHttpRequests()`, seguido das URLs que queremos proteger e dos papéis autorizados para cada URL. Para isso usamos os métodos `requestMatchers(String)`, e `hasAnyRole(String...)` - para um conjunto de papéis, ou `hasRole(String)` - para um único papel, conforme mostra a Figura a seguir:

```
● ● ● SecurityConfig.java
33 @Bean
34 public SecurityFilterChain filter(HttpSecurity http) throws Exception {
35     return http
36         .csrf().disable()
37         .authorizeHttpRequests()
38         .requestMatchers("/").hasAnyRole("listar", "admin")
39         .requestMatchers("criar", "/excluir", "/alterar", "/preparaAlterar").hasRole("admin")
40 }
```

Snipped

Na linha 38 definimos que a URL raiz (`requestMatchers("")`), mapeada no `CidadeController` com o método `listar()`, pode ser acessada por qualquer usuário com os papéis `listar` ou `admin` (`hasAnyRole("listar", "admin")`). No nosso exemplo, isso inclui os dois usuários definidos na seção anterior.

A linhas 39 segue a mesma lógica, porém, definindo apenas um papel (`hasRole(String)`) para todas as URLs definidas. Observe que cada URL está mapeada com uma operação CRUD. Dessa forma, nosso usuário `john`, que tem o papel/autorização `listar`, tem permissão apenas para listar as cidades. Por outro lado, o usuário `anna`, que tem o papel `admin`, tem permissão para criar, alterar ou excluir cidades.

Para evitar que qualquer URL não definida anteriormente seja deixada aberta para acesso, vamos negar o acesso a qualquer outra URL não definida explicitamente usando `anyRequest().denyAll()` (linha 40).

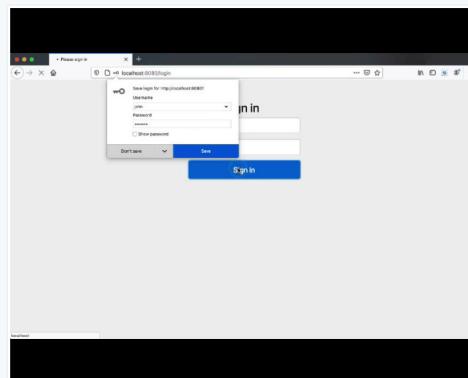
Para finalizar, vamos liberar o acesso à página de login usando `and().formLogin().permitAll()`. O método `and()` é usado como uma conjunção, permitindo adicionar mais um conjunto de regras. O acesso ao formulário de login (`formLogin()`) precisa estar liberado a todos (`permitAll()`), caso contrário os usuários não conseguem inserir suas credenciais para login. Veja como ficou o código completo.

```
● ● ● SecurityConfig.java
33 @Bean
34 public SecurityFilterChain filter(HttpSecurity http) throws Exception {
35     return http
36         .csrf().disable()
37         .authorizeHttpRequests()
38         .requestMatchers("/").hasAnyRole("listar", "admin")
39         .requestMatchers("criar", "/excluir", "/alterar", "/preparaAlterar").hasRole("admin")
40         .anyRequest().denyAll()
41         .and()
42         .formLogin().permitAll()
43         .and()
44         .build();
45 }
46 }
```

Snipped

Agora você pode executar a aplicação e ver o resultado.

Observe que ainda não temos um botão de logout. Por isso, para encerrar a sessão você precisa limpar a cache do navegador.



O código desenvolvido nesta Seção está disponível no [Github](#), na branch `semana07-20-autorizacao`.

## LOGOUT E TELA DE LOGIN

No momento, o aplicativo permite o login, mas não permite o logout. Se você quiser encerrar uma sessão, precisa limpar o cache do navegador. Para permitir que um usuário faça logout, vamos criar uma barra de navegação superior e um botão de logout.

Para isso, abra a página `crud.html` e insira a barra exatamente antes do inicio do `container` definido pelo Bootstrap. Veja o código a seguir:

```
13 <body>
14     <nav class="navbar navbar-expand-sm bg-dark">
15         <ul class="navbar-nav ml-auto">
16             <li class="nav-item">
17                 <a href="/logout"
18                     class="nav-link btn btn-secondary">
19                         Sair da aplicação</a>
20                 </li>
21             </ul>
22         </nav>
23
24         <div class="container-fluid">
25             <div class="jumbotron mt-5">
26                 <h1>GERENCIAMENTO DE CIDADES</h1>
27                 <p>UM CRUD PARA CRIAR, ALTERAR, EXCLUIR E LISTAR CIDADES</p>
28             </div>
29         </div>
```

As linhas 14 a 23 definem a barra de navegação, enquanto as linhas 17 a 20 definem o botão de logout. Observem que tudo que precisa ser feito para que o logout funcione, é criar um link para a URL `/logout` (linha 18). Quando o Spring Boot recebe a solicitação dessa URL, ele encerra a sessão do usuário atual e redireciona para a tela de login.



Outra coisa que provavelmente você querer fazer na sua aplicação Web é criar uma tela de login personalizada. Fazer isso é simples. Vamos começar criando uma nova página Web. A tela de login é uma página Web simples, por isso, vamos criar o arquivo `login.html` em `/resources/static/`.

Vamos usar o HTML e a formatação do Bootstrap para criar um componente central com um formulário. No formulário, vamos colocar uma entrada de texto e uma entrada de tipo senha. Também precisamos de um botão para submeter o formulário. Veja o código na figura abaixo.

Se não quiser perder tempo digitando toda essa formatação, você pode consultar o código direto no repositório no Github. A branch usada está descrita no final desta seção.

```
1 <head>
2     <meta charset="UTF-8">
3     <meta http-equiv="X-Content-Type-Options" content="IE-Edge">
4     <meta http-equiv="Content-Security-Policy" content="width=device-width, initial-scale=1.0">
5     <title>CRUD Cidades - Login</title>
6
7     <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
8
9 </head>
10
11 <body class="bg-light">
12     <main class="container-fluid">
13         <section class="col-md-12 d-flex justify-content-center mt-5">
14             <div class="card w-100" style="background-color: #f2f2f2; border-radius: 15px; padding: 20px; text-align: center">
15                 <div class="card-header bg-primary text-white">Dados para Login</div>
16                 <div class="card-body">
17                     <form method="post" style="margin-bottom: 20px;">
18                         <div class="form-group">
19                             <label form="username">Usuário:</label>
20                             <input name="username" type="text" class="form-control" placeholder="Informe usuário" id="username">
21                         </div>
22                         <div class="form-group">
23                             <label form="password">Senha:</label>
24                             <input name="password" type="password" class="form-control" placeholder="Informe senha" id="senha">
25                         </div>
26                         <div style="text-align: right; margin-top: 10px;">
27                             <button type="submit" class="btn btn-primary">Login</button>
28                         </div>
29                     </form>
30                 </div>
31             </div>
32         </section>
33     </main>
34 </body>
```

Agora vamos entrar em detalhes da formatação usada pelo Bootstrap, já que esse não é nosso foco. O que é importante notar aqui está nas linhas 19, 22 e 27. Na linha 19, definimos o método de envio de dados do formulário como `POST`. Isso é fundamental, já que o Spring Security espera que os dados de login sejam sempre enviados por `POST`.

Na linha 22 e 27 definimos os componentes de entrada do nome do usuário (`<input type="text"`) e senha (`<input type="password"`), respectivamente. Para que o Spring Security entenda que estamos enviando o nome e a senha do usuário, o componente do nome do usuário deve ter o atributo `name` definido como `username`. Da mesma forma, o componente de senha deve ter o atributo `name` definido como `password`.

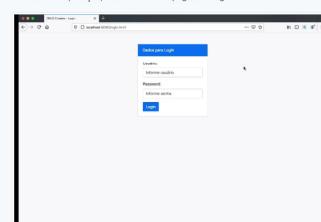
Agora, precisamos voltar para a classe `CrudCidadesSecurityConfig` para configurar nossa nova página como página de login. Lá, depois do método `formLogin()`, vamos adicionar o método `loginPage(String)`. Como parâmetro do método, vamos definir a página de `login.html`. O método `permitAll()` deve então ser anexado ao método `loginPage(String)`, significando que qualquer um pode acessar a página de login.

Também vamos adicionar (`and()`) uma permissão para acesso à página de logout (`logout().permitAll()`). Veja como ficou o código com as mudanças.

```
33 @Bean
34     protected SecurityFilterChain filter(HttpSecurity http) throws Exception {
35         return http
36             .csrf().disable()
37             .formLogin()
38             .requestMatchers("/*").hasAnyRole("listar", "admin")
39             .requestMatchers("/criar", "excluir", "/preparaAlterar", "alterar").hasRole("admin")
40             .anyRequest().denyAll()
41             .and()
42             .formLogin()
43             .loginPage("/login.html").permitAll()
44             .and()
45             .logout().permitAll()
46             .and()
47             .build();
48     }
49 }
```

Snipped

Ao executar a aplicação, você verá a nova página de login.



O código desenvolvido nesta Seção está disponível no [Github](#), na branch `semana07-3b-logout-login-form`.

