



# SEGURANÇA

Inserindo autenticação e autorização na aplicação

Visão Geral

Autenticação

Autorização

Logout/Login

Autenticação com BD

Verificação de Aprendizado

## 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

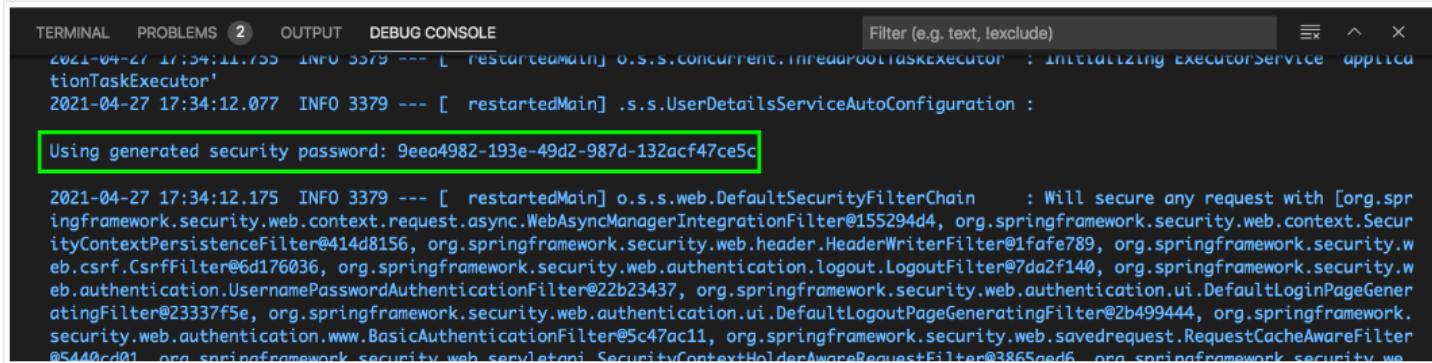
Autenticação é o processo de verificar se alguém é quem diz ser. Isso pode ser feito de diferentes formas, como usando uma senha ou por biometria. Autenticação é um pré-requisito da autorização. Autorização significa que alguém tem acesso a algum recurso.

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.

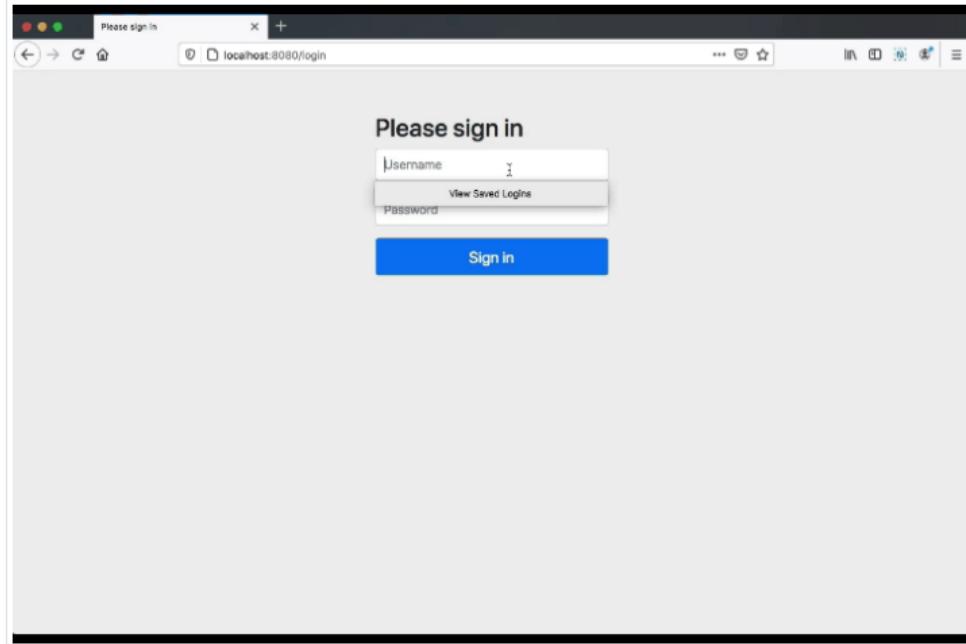
Mas como assim? Eu nem criei o usuário. É verdade, mas o Spring Boot cria um usuário de exemplo (`user`) pra você e coloca toda a aplicação sob essa segurança. Você pode consultar na console do sistema a senha criada automaticamente pelo Spring Boot.



```
TERMINAL PROBLEMS 2 OUTPUT DEBUG CONSOLE
Filter (e.g. text, exclude)
2021-04-27 17:54:11.755 INFO 3379 --- [ restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService applicationTaskExecutor'
2021-04-27 17:34:12.077 INFO 3379 --- [ restartedMain] .s.s.UserDetailsServiceAutoConfiguration :
Using generated security password: 9eeda4982-193e-49d2-987d-132acf47ce5c

2021-04-27 17:34:12.175 INFO 3379 --- [ restartedMain] o.s.s.web.DefaultSecurityFilterChain : Will secure any request with [org.springframework.security.web.context.request.async.WebAsyncManagerIntegrationFilter@155294d4, org.springframework.security.web.context.SecurityContextPersistenceFilter@414d8156, org.springframework.security.web.header.HeaderWriterFilter@1fafef89, org.springframework.security.web.csrf.CsrfFilter@6d176036, org.springframework.security.web.authentication.logout.LogoutFilter@7da2f140, org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter@22b23437, org.springframework.security.web.authentication.ui.DefaultLoginPageGeneratingFilter@2b499444, org.springframework.security.web.authentication.www.BasicAuthenticationFilter@5c47ac11, org.springframework.security.web.savedrequest.RequestCacheAwareFilter@5440cd01, org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter@3865aed6, org.springframework.security.we...
```

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 nessa aplicação. Para isso, crie uma classe em `br.edu.utfpr.cp.espjava.crudcidades.SecurityConfig` e insira a anotação `org.springframework.security.config.annotation.web.configuration.EnableWebSecurity` 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.

O framework também já fornece os métodos que precisamos sobrescrever para configurar nossa própria autenticação – basta estender a classe `org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter`.

Essa classe fornece o método `configure(AuthenticationManagerBuilder)`, que vamos sobrescrever para definir nossos usuários. Veja como está o código até agora.

```
11  @EnableWebSecurity
12  @Configuration
13  public class SecurityConfig extends WebSecurityConfigurerAdapter {
14
15      @Override
16      protected void configure(AuthenticationManagerBuilder auth) throws Exception {
17
18      }
19
20  }
```

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 começar criando um método que retorna um objeto `org.springframework.security.crypto.password.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 precisamos criar nada. Para esse exemplo, vamos usar o `org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder`. Tudo que precisamos fazer é retornar uma nova instância dessa classe. Para finalizar, vamos adicionar a anotação `org.springframework.context.annotation.Bean` imediatamente antes do método. Isso faz com o que o Spring Boot gerencie automaticamente a configuração definida pelo método.

```
11 @EnableWebSecurity
12 @Configuration
13 public class SecurityConfig extends WebSecurityConfigurerAdapter {
14
15     @Override
16     protected void configure(AuthenticationManagerBuilder auth) throws Exception {
17
18     }
19
20     @Bean
21     public PasswordEncoder cifrador() {
22         return new BCryptPasswordEncoder();
23     }
24 }
```

Finalmente, podemos associar nossos usuários e o conjunto de *autorizações* que eles têm acesso. Uma *autorização* ou *papel* define um grupo de autorizações/restricções que um ou mais usuários possuem. Os usuários e seus 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 dos dois.

Para isso, vamos usar a variável `auth`, parâmetro do método `configure()`. Essa variável dá acesso aos métodos do objeto `org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder`. Esse objeto define o método `inMemoryAuthentication()`, que vamos usar para definir nossos usuários. Veja como fica o código na Figura abaixo.

```
11 @EnableWebSecurity
12 @Configuration
13 public class SecurityConfig extends WebSecurityConfigurerAdapter {
14
15     @Override
16     protected void configure(AuthenticationManagerBuilder auth) throws Exception {
17         auth.inMemoryAuthentication()
18             .withUser("john")
19             .password(cifrador().encode("test123"))
20             .authorities("listar")
21             .and()
22             .withUser("anna")
23             .password(cifrador().encode("test123"))
24             .authorities("admin");
25     }
26
27     @Bean
28     public PasswordEncoder cifrador() {
29         return new BCryptPasswordEncoder();
30     }
31 }
```

Cada usuário é definido com o método `withUser(String)`. Nesse exemplo, temos dois usuários: john e anna. Cada usuário tem sua senha cifrada usando o algoritmo definido no método `cifrador()`. A senha é definida pelo método `password()`. O valor passado para o método `encode(String)`, definido no objeto `PasswordEncoder`, é uma string simples, que é cifrada e usada pelo Spring Security.

O último argumento definido para um usuário é o conjunto de autorizações que ele deve assumir. Isso é feito com método `authorities(String...)`, que recebe um conjunto de `String` definindo um ou mais conjuntos de *autorizações*. Após o primeiro usuário ser definido, é possível definir mais usuários usando o método `and()`, conforme é feito na linha 21 da Figura acima.

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.

### Please sign in

john	I
Password	
→ 0 No username (12 Mar 2021)	
From this website	
<a href="#">View Saved Logins</a>	

O código desenvolvido nesta Seção está disponível no [Github](#), na branch [semana07-10-autenticacao](#).

## AUTORIZAÇÃO

Com o usuário autenticado, o próximo passo é definir as autorizações. Para definir as autorizações, vamos sobreescrver o método `configure(org.springframework.security.config.annotation.web.builders.HttpSecurity)`, herdado da classe `org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter`. Usando o parâmetro `http`, vamos começar desabilitando o csrf.

```
28 protected void configure(HttpSecurity http) throws Exception {
29     http
30         .csrf().disable()
31 }
32 }
```

Agora, vamos começar a mapear as autorizações com as URLs da aplicação. Para isso, vamos usar o método `authorizeRequests()`, seguido das URLs que queremos proteger e dos papéis autorizados para cada URL. Para isso usamos o método `antMatchers(String)`, e `hasAnyAuthority(String...)` ou `hasAuthority(String)`, conforme mostra a Figura a seguir.

```
28 protected void configure(HttpSecurity http) throws Exception {
29     http
30         .csrf().disable()
31         .authorizeRequests()
32             .antMatchers("/").hasAnyAuthority("listar", "admin")
33             .antMatchers("/criar").hasAuthority("admin")
34             .antMatchers("/excluir").hasAuthority("admin")
35             .antMatchers("/preparaAlterar").hasAuthority("admin")
36             .antMatchers("/alterar").hasAuthority("admin")
37 }
```

Na linha 32 definimos que a URL raiz (`antMatchers("/")`), mapeada no `CidadeController` com o método `listar()`, por ser acessada por qualquer usuário com os papéis `listar` ou `admin` (`hasAnyAuthority("listar", "admin")`). No nosso exemplo, isso inclui os dois usuários definidos na seção anterior.

As linhas 33 a 36 seguem a mesma lógica, porém, definindo apenas um papel por URL (`hasAuthority(String)`). 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.

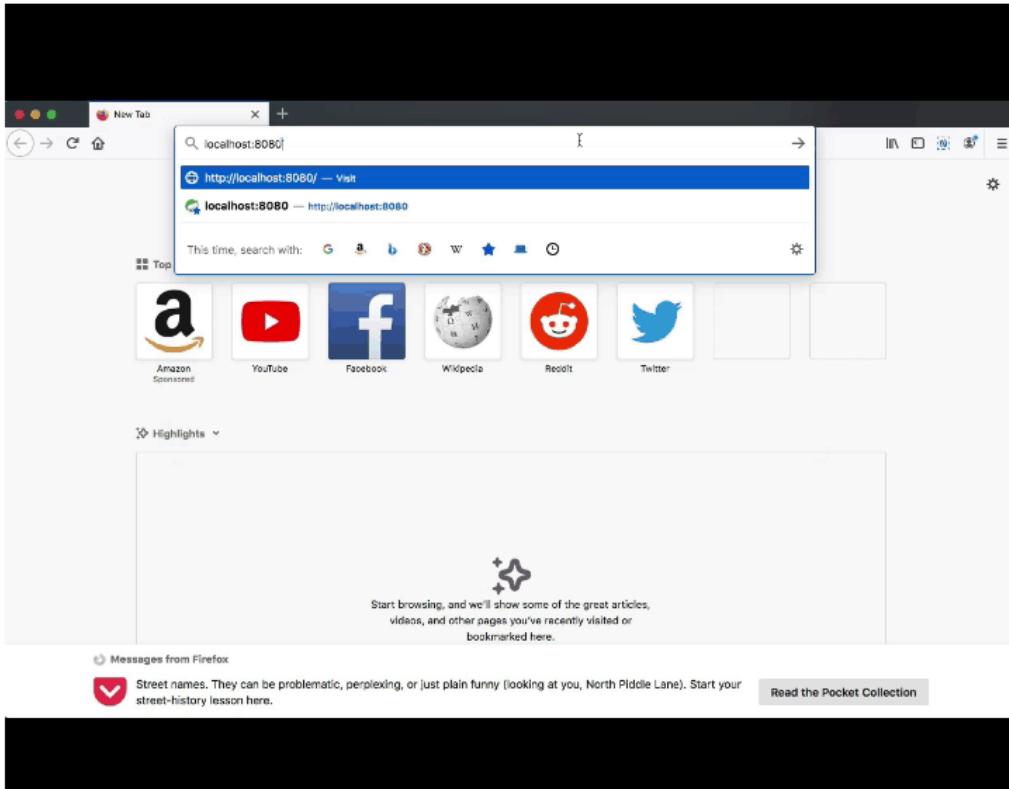
Para evitar que qualquer URL não definida anteriormente seja deixada aberta para acesso, vamos negar o acesso a qualquer outra URL não definida explicitamente usando `anyRequest().denyAll()`.

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.

```
28     protected void configure(HttpSecurity http) throws Exception {  
29         http  
30             .csrf().disable()  
31             .authorizeRequests()  
32                 .antMatchers("/").hasAnyAuthority("listar", "admin")  
33                 .antMatchers("/criar").hasAuthority("admin")  
34                 .antMatchers("/excluir").hasAuthority("admin")  
35                 .antMatchers("/preparaAlterar").hasAuthority("admin")  
36                 .antMatchers("/alterar").hasAuthority("admin")  
37             .anyRequest().denyAll()  
38             .and()  
39             .formLogin().permitAll();  
40     }
```

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.ftl` e insira a barra exatamente antes do início 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
18                     href="/logout"
19                     class="nav-link btn btn-secondary"
20                     >Sair da aplicação</a>
21             </li>
22         </ul>
23     </nav>
24
25     <div class="container-fluid">
26         <div class="jumbotron mt-5">
27             <h1>GERENCIAMENTO DE CIDADES</h1>
28             <p>UM CRUD PARA CRIAR, ALTERAR, EXCLUIR E LISTAR CIDADES</p>
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.

# GERENCIAMENTO DE CIDADES

UM CRUD PARA CRIAR, ALTERAR, EXCLUIR E LISTAR CIDADES

Cidade:

Informe o nome da cidade

Estado:

Informe o estado ao qual a cidade pertence

**CRIAR**

Nome	Estado	Ações

Outra coisa que provavelmente você vai 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 do 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..

```

4 <head>
5     <meta charset="UTF-8">
6     <meta http-equiv="X-UA-Compatible" content="IE=edge">
7     <meta name="viewport" content="width=device-width, initial-scale=1.0">
8     <title>CRUD Cidades - Login</title>
9
10    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
11 </head>
12
13 <body class="bg-light">
14     <main class="container-fluid">
15         <section class="col-md-12 d-flex justify-content-center mt-5">
16             <div class="card">
17                 <div class="card-header bg-primary text-white">Dados para Login</div>
18                 <div class="card-body">
19                     <form method="POST">
20                         <div class="form-group">
21                             <label for="usuario">Usuário:</label>
22                             <input name="username" type="text" class="form-control" placeholder="Informe usuário" id="usuario">
23                         </div>
24                         <div class="form-group">
25                             <label for="senha">Senha:</label>
26                             <input name="password" type="password" class="form-control" placeholder="Informe senha" id="senha">
27                         </div>
28                         <button type="submit" class="btn btn-primary">Login</button>
29                     </form>
30                 </div>
31             </div>
32         </section>
33     </main>
34 </body>

```

Não 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 `crudcidades.SecurityConfig` para configurar nossa nova página como página de login. Logo após o método `formLogin()`, vamos adicionar o método `loginPage(String)`. Como parâmetro do método, vamos definir a página de login `login.html`. O método `permitAll()` deve então ser anexado ao método `loginPage(String)`, significando que qualquer um pode acessar a página de login.

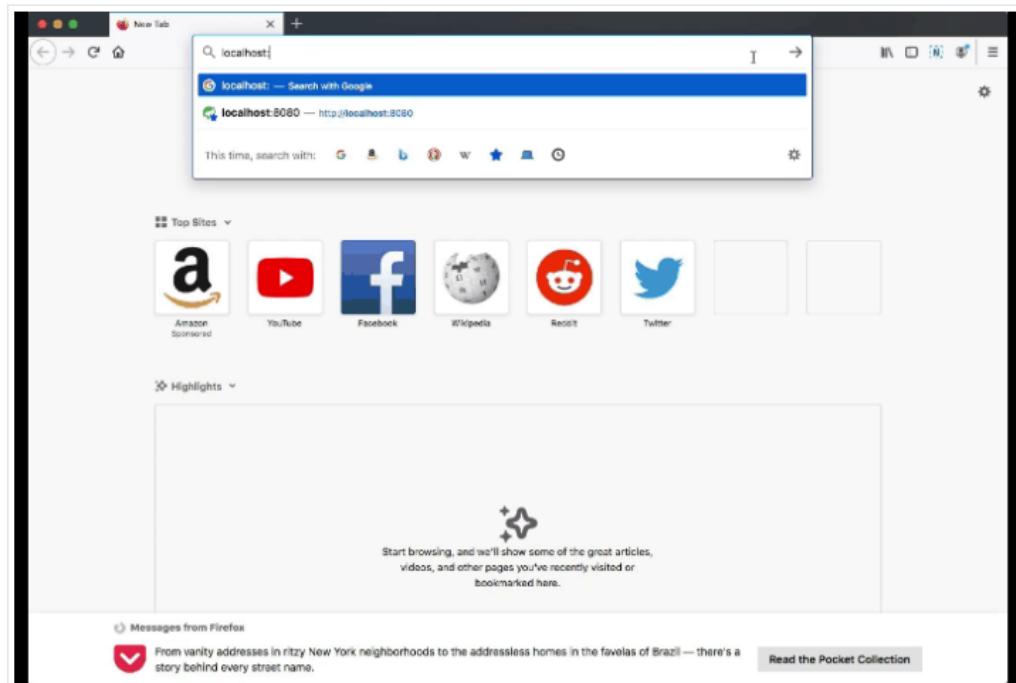
Também vamos adicionar (`and()`) uma permissão para acesso à página de logout (`logout().permitAll()`). Veja como ficou o código com as mudanças.

```

28     protected void configure(HttpSecurity http) throws Exception {
29         http
30             .csrf().disable()
31             .authorizeRequests()
32             .antMatchers("/").hasAnyAuthority("listar", "admin")
33             .antMatchers("/criar").hasAuthority("admin")
34             .antMatchers("/excluir").hasAuthority("admin")
35             .antMatchers("/preparaAlterar").hasAuthority("admin")
36             .antMatchers("/alterar").hasAuthority("admin")
37             .anyRequest().denyAll()
38             .and()
39             .formLogin()
40             .loginPage("/login.html").permitAll()
41             .and()
42             .logout().permitAll();
43     }

```

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



O código desenvolvido nesta Seção está disponível no [Github](#), na branch `semana07-30-logout-login-form`.

# AUTENTICAÇÃO COM BANCO DE DADOS

Da forma como está, a autenticação funciona bem para uma pequena aplicação com usuários estáticos. Contudo, normalmente você vai enfrentar cenários nos quais a criação de usuários precisa ser feita de forma dinâmica. Assim, um novo usuário não depende de ajustes no código para começar a usar seu aplicativo. Nesta seção, vamos ajustar o código atual para que nossa autenticação esteja baseada em usuários em um banco de dados, em vez de código direto na aplicação.

Como vamos precisar salvar os usuários no banco de dados, precisamos criar uma entidade persistente e um *repository* para gerenciar a persistência. Vamos começar criando um novo pacote (pasta) chamado `usuario`, na pasta `br.edu.utfpr.cp.espjava.crudcidades`. Dessa forma, continuamos usando a arquitetura definida em seções anteriores.

Agora, precisamos da classe que representa a entidade *usuário*. Para isso, crie uma classe `usuario.Usuario`. Nessa classe, vamos definir quatro atributos privados: `Long id`, `String nome`, `String senha` e `List papeis`, além dos gets/sets. Não esqueça de adicionar as anotações do JPA.

Diferente do que fizemos na classe anterior, aqui temos uma relação de muitos para muitos com papéis. Dessa forma, um usuário pode ter vários papéis, e cada papel pode ser usado por vários usuários. Em vez de usar a anotação `ManyToMany`, vamos usar um `@ElementCollection(fetch = FetchType.EAGER)`. Essa anotação cria automaticamente uma relação entre a entidade (`Usuario`) e a lista. Nesse caso, não temos outras entidades se relacionando com a lista de papéis, portanto, esse é uma solução simples e apropriada para esse exemplo.

```

19  @Entity
20  public class Usuario implements Serializable {
21
22      @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
23      private Long id;
24
25      private String nome;
26
27      @ElementCollection(fetch = FetchType.EAGER)
28      private List<String> papeis;
```

Além da classe de entidade, precisamos da interface que gerencia as operações de persistência. Assim como fizemos antes, vamos criar uma interface `usuario.UsuarioRepository`. Também vamos criar um método que retorna um usuário com base no nome informado (`findByName(String)`). Vamos precisar desse método para integrar com o mecanismo de autenticação do Spring Security.

```

1 package br.edu.utfpr.cp.espjava.crudcidades.usuario;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4
5 public interface UsuarioRepository extends JpaRepository<Usuario, Long>{
6
7     public Usuario findByName(String nome);
8 }
```

Uma das coisas que torna o framework Spring extensível é o uso de interfaces. Para marcar nossa classe `usuario.Usuario` como um usuário do sistema que o Spring Security entende, tudo que precisamos fazer é implementar a interface `org.springframework.security.core.userdetails.UserDetails`. Essa interface define um conjunto de métodos que precisam ser implementados por um usuário do sistema. Vamos ver cada um deles.

- `String getUsername()` retorna o nome do usuário. Na classe `usuario.Usuario`, o nome do usuário é definido no atributo `String nome`. Por isso, tudo que precisamos fazer para implementar esse método é retornar o valor armazenado no atributo `nome`.
- `Collection<? extends GrantedAuthority> getAuthorities()` retorna uma lista de papéis na qual o usuário tem permissões. Na classe `usuario.Usuario`, os papéis estão definidos no atributo `List papeis`. Por isso, precisamos implementar esse método retornando o valor armazenado no atributo `papeis`. Mas não é só isso, o atributo `papeis` é uma lista de `String`, enquanto o método `getAuthorities()` retorna uma coleção de `org.springframework.security.core.GrantedAuthority`. Por isso, precisamos converter cada papel do tipo `String` por um `org.springframework.security.core.authority.SimpleGrantedAuthority`. Veja o código na Figura logo abaixo para ver como fazer isso.
- `String getPassword()` retorna a senha do usuário. Na classe `usuario.Usuario`, a senha está definida no atributo `senha`. Por isso, tudo que precisamos fazer para implementar esse método é retornar o valor armazenado no atributo `senha`.
- `boolean isCredentialsNonExpired(), boolean isEnabled(), boolean isAccountNonExpired(), boolean isAccountNonLocked()` fazem exatamente o que seus nomes definem. Por exemplo, `isEnabled()` retorna verdadeiro ou falso dependendo se a conta para esse usuário está ativa ou não. Em um cenário real, precisaríamos de atributos adicionais na nossa classe para armazenar esses valores. Contudo, nesse exemplo, vamos simplesmente retornar o valor fixo `true`, indicando que as credenciais não estão expiradas, a conta está habilitada, a conta não está expirada, e a conta não está travada, respectivamente.

A Figura abaixo mostra o código completo da classe `usuario.Usuario` após implementar a interface `org.springframework.security.core.userdetails.UserDetails`.

```

19 @Entity
20 public class Usuario implements Serializable, UserDetails {
21
22     @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
23     private Long id;
24     private String nome;
25     private String senha;
26
27     @ElementCollection(fetch = FetchType.EAGER)
28     private List<String> papeis;
29
30     public Long getId() { return id; }
31     public String getNome() { return nome; }
32     public List<String> getPapeis() { return papeis; }
33     public String getSenha() { return senha; }
34     public void setId(Long id) { this.id = id; }
35     public void setNome(String nome) { this.nome = nome; }
36     public void setPapeis(List<String> papeis) { this.papeis = papeis; }
37     public void setSenha(String senha) { this.senha = senha; }
38
39     @Override public String getUsername() { return this.nome; }
40
41     @Override
42     public Collection<? extends GrantedAuthority> getAuthorities() {
43         return this.papeis
44             .stream()
45             .map(papelAtual -> new SimpleGrantedAuthority(papelAtual))
46             .collect(Collectors.toList());
47     }
48
49     @Override public String getPassword() { return this.senha; }
50     @Override public boolean isCredentialsNonExpired() { return true; }
51     @Override public boolean isEnabled() { return true; }
52     @Override public boolean isAccountNonExpired() { return true; }
53     @Override public boolean isAccountNonLocked() { return true; }
54 }
```

Agora o Spring Security entende a classe `usuario.Usuario` como uma classe que representa um usuário do sistema. O próximo passo é definir uma implementação do serviço que verifica se a existência do usuário. Para isso, vamos criar a classe `usuario.UsuarioDetailsService`, que implementa `org.springframework.security.core.userdetails.UserDetailsService`. Essa classe deve ser anotada com a anotação `org.springframework.stereotype.Service`, indicando ao Spring Boot que essa classe deve ser gerenciada pelo framework.

`org.springframework.security.core.userdetails.UserDetailsService` define o método `UserDetails loadUserByUsername(String)`. Nossa usuário está no banco de dados, por isso, precisamos verificar se o usuário passado como parâmetro nesse método existe no nosso banco. Já criamos um método que faz isso no `usuario.UsuarioRepository`. Para usar o `usuario.UsuarioRepository`, precisamos criar um construtor para que o Spring Boot faça a injeção de dependência. Por fim, se o usuário não existir, nós disparamos a exceção `org.springframework.security.core.userdetails.UsernameNotFoundException`, que é usada pelo Spring Security para dizer que houve algum problema na autenticação. Veja como ficou o código completo.

```
8 @Service
9 public class UsuarioDetailsService implements UserDetailsService {
10
11     private final UsuarioRepository usuarioRepository;
12
13     public UsuarioDetailsService(final UsuarioRepository usuarioRepository) {
14         this.usuarioRepository = usuarioRepository;
15     }
16
17     @Override
18     public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
19
20         var usuario = usuarioRepository.findByNome(username);
21
22         if (usuario == null)
23             throw new UsernameNotFoundException("Usuário não encontrado!");
24
25         return usuario;
26     }
27 }
```

Nesse ponto, precisamos lembrar que estamos usando um algoritmo de cifragem para garantir que as senhas não são armazenadas como texto puro. Por isso, para criar um usuário no banco, precisamos cifrar a senha antes de salvá-la. Nesse exemplo, vamos criar o usuário diretamente no banco, mas é importante lembrar disso quando estiver criando uma aplicação real.

Vamos criar o método `printSenhas` na classe `SecurityConfig`. O objetivo desse método é usar o algoritmo de cifragem para cifrar a senha de exemplo e imprimir na console do sistema. Assim, podemos pegar essa senha e salvar direto no banco de dados.

```
50     @EventListener(ApplicationReadyEvent.class)
51     public void printSenhas() {
52         System.out.println(this.cifrador().encode("test123"));
53     }
```

A novidade nesse código é a anotação `org.springframework.context.event.EventListener`. Mas não se preocupe com isso agora, vamos falar sobre essa anotação na próxima aula.

Ao executar o código você verá a senha impressa na console do sistema. Copie esse valor pois vamos usá-lo logo em seguida.

```
2021-04-30 16:05:34.041 INFO 7163 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2021-04-30 16:05:34.050 INFO 7163 --- [ restartedMain] o.s.b.a.h2.H2ConsoleAutoConfiguration : H2 console available at '/h2-console'
'. Database available at 'jdbc:h2:mem:04ddae62-819d-4a43-a8e9-ab5e2a57bb78'
2021-04-30 16:05:34.357 INFO 7163 --- [ restartedMain] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [name: default]
2021-04-30 16:05:34.458 INFO 7163 --- [ restartedMain] org.hibernate.Version : HHH000412: Hibernate ORM core version 5.4.28.Final
2021-04-30 16:05:34.689 INFO 7163 --- [ restartedMain] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations 5.1.2.Final
2021-04-30 16:05:34.945 INFO 7163 --- [ restartedMain] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
2021-04-30 16:05:35.978 INFO 7163 --- [ restartedMain] o.h.e.t.j.p.i.JtaPlatformInitiator : HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]
2021-04-30 16:05:35.986 INFO 7163 --- [ restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2021-04-30 16:05:36.005 INFO 7163 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2021-04-30 16:05:36.262 WARN 7163 --- [ restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2021-04-30 16:05:37.359 INFO 7163 --- [ restartedMain] o.s.s.web.DefaultSecurityFilterChain : Will secure any request with org.springframework.security.web.context.request.async.WebAsyncManagerIntegrationFilter@6bf5794b, org.springframework.security.web.context.SecurityContextPersistenceFilter@1bc2959c, org.springframework.security.web.header.HeaderWriterFilter@4e028de2, org.springframework.security.web.authentication.logout.LogoutFilter@36cfecle, org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter@1bcc9346, org.springframework.security.web.savedrequest.RequestCacheAwareFilter@186efida, org.springframework.security.web.authentication.AuthenticationContextHolderAwareRequestFilter@5fle3e72, org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter@44932e8b, org.springframework.security.web.session.SessionManagementFilter@f0a918, org.springframework.security.web.access.ExceptionTranslationFilter@6bf7558a, org.springframework.security.web.access.intercept.FilterSecurityInterceptor@405cebb]
2021-04-30 16:05:37.599 INFO 7163 --- [ restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2021-04-30 16:05:38.120 INFO 7163 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2021-04-30 16:05:38.130 INFO 7163 --- [ restartedMain] e.u.c.e.c.CrudCidadesApplication : Started CrudCidadesApplication in 8.691 seconds (JVM running for 9.539)
$2as10$/.KKJU.G71/Fl.4wKo.lJ0fDhxhHPo0o.DPxIy98IuKSaK74WXUy2
```

Como estamos usando um banco de dados em memória, vamos criar o arquivo `data.sql` em `/src/main/resources/`. Vamos colocar nesse arquivo o código SQL para criar dois usuários e seus papéis.

Não altere o nome do arquivo. Ele precisa ser esse para que o Spring Boot use o conteúdo do arquivo para criar os usuários automaticamente.

```
1 INSERT INTO `usuario` VALUES (1,'john','$2a$10$/.KKJU.G71/Fl.4wKo.lJ0fDhxhHPo0o.DPxIy98IuKSaK74WXUy2');
2 INSERT INTO `usuario` VALUES (2,'anna','$2a$10$/.KKJU.G71/Fl.4wKo.lJ0fDhxhHPo0o.DPxIy98IuKSaK74WXUy2');
3
4 INSERT INTO `usuario_papeis` VALUES (1, 'listar');
5 INSERT INTO `usuario_papeis` VALUES (2, 'listar');
6 INSERT INTO `usuario_papeis` VALUES (2, 'admin');
```

Observe que o sinal empregado no nome da tabela é `backtick`, e **não** aspas simples.

Com a versão 2.5.4 do Spring Boot, houve uma [alteração na ordem de execução do script SQL](#). Por conta disso, precisamos adicionar a instrução `spring.jpa.defer-datasource-initialization=true` no arquivo `application.properties`.

Agora podemos voltar na classe `SecurityConfig` e apagar o método `configure(AuthenticationManagerBuilder)`, que definia os usuários. Podemos fazer isso porque agora nossos usuários estão no banco de dados.

Ao executar a aplicação você terá acesso à tela de login, assim como antes. Contudo, agora os usuários estão no banco de dados. Observe que nessa aula não vamos implementar um CRUD de usuários. Mas se você quiser fazer isso, basta repetir os mesmos passos usados para criar o CRUD de cidades. O único detalhe importante é que você precisa cifrar a senha informada pelo usuário antes de enviar para o banco de dados. É necessário para garantir tanto a segurança do usuário, como também para que o Spring Security consiga comparar a senha informada na tela de login com a senha cifrada armazenada no banco.

O código desenvolvido nesta Seção está disponível no [Github](#), na branch `semana07-40-autenticacao-db`.