CETEJ35 - Java Web - JAVA_XXX (2024_01)

Segurança

✓ **Feito:** Ver ✓ **Feito:** Gastar pelo menos 20 minutos na atividade **A fazer:** Passar pela atividade até o fim

Fecha: segunda-feira, 2 dez. 2024, 00:00

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.

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      private String nome;
25      private String senha;
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 (`findByNome(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 findByNome(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, em especial, atente para a linha 46.

- `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`.

Usuario.java

```java
@Entity
public class Usuario implements Serializable, UserDetails {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
    private String senha;

    @ElementCollection(fetch = FetchType.EAGER)
    private List<String> papeis;

    public Long getId() { return id; }
    public String getNome() { return nome; }
    public List<String> getPapeis() { return papeis; }
    public String getSenha() { return senha; }
    public void setId(Long id) { this.id = id; }
    public void setNome(String nome) { this.nome = nome; }
    public void setPapeis(List<String> papeis) { this.papeis = papeis; }
    public void setSenha(String senha) { this.senha = senha; }

    @Override
    public String getUsername() { return this.nome; }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return this.papeis
                    .stream()
                    .map(papelAtual -> new SimpleGrantedAuthority("ROLE_" + papelAtual))
                    .collect(Collectors.toList());
    }

    @Override
    public String getPassword() { return this.senha; }

    @Override
    public boolean isCredentialsNonExpired() { return true; }

    @Override
    public boolean isEnabled() { return true; }

    @Override
    public boolean isAccountNonExpired() { return true; }

    @Override
    public boolean isAccountNonLocked() { return true; }
}
```

Snipped

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)`. Nosso 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.

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.

```
INSERT INTO `usuario` VALUES (1,'john','$2a$10$/.KKJU.G71/Fl.4wKo.lJOfDhxhHPo0o.DPxIy98IuKSaK74WXUy2');
INSERT INTO `usuario` VALUES (2,'anna','$2a$10$/.KKJU.G71/Fl.4wKo.lJOfDhxhHPo0o.DPxIy98IuKSaK74WXUy2');

INSERT INTO `usuario_papeis` VALUES (1, 'listar');
INSERT INTO `usuario_papeis` VALUES (2, 'listar');
INSERT INTO `usuario_papeis` VALUES (2, 'admin');
```

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

Adicione a instrução `spring.jpa.defer-datasource-initialization=true` no arquivo `application.properties`. Dessa forma, o arquivo `data.sql` será executado *antes* da aplicação entrar em ação - garantindo que os usuários e permissões já existem quando a aplicação tentar autenticar.

Por fim, podemos voltar na classe `SecurityConfig` e apagar o método `InMemoryUserDetailsManager configure()`, 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`.

Retroceder Avançar

◄ Atividade II WebConf

Seguir para...

Listeners & Cookies ►

Contate o suporte do site

Você acessou como RAFAEL ROCHA DA SILVA PROENCA (Sair)
CETEJ35 - Web (2024_01)

Tema
Adaptable
Boost
Clássico

Campus
Apucarana
Campo Mourão
Cornélio Procópio
Curitiba
Dois Vizinhos
Francisco Beltrão
Guarapuava
Londrina
Medianeira
Pato Branco
Ponta Grossa
Reitoria
Santa Helena
Toledo

UTFPR
Ajuda
Chat UTFPR
Calendário Acadêmico
Biblioteca
e-Mail
Nuvem (OwnCloud)
Produção Acadêmica
Secretaria Acadêmica
Sistemas Corporativos
Sistema Eletrônico de Informação - SEI
Suporte ao usuário
Criação de curso
Comunidade
Português - Brasil (pt_br)
Deutsch (de)
English (en)
Português - Brasil (pt_br)

Resumo de retenção de dados

Baixar o aplicativo móvel.

Dê um feedback sobre este software

Universidade Tecnológica Federal do Paraná - UTFPR
Suporte ao usuário