04

# Usuário no Postgres

### Transcrição

Uma solução simples é criar uma *URL* bem estranha e difícil de acertar aleatóriamente que executará em um *controller* a ação de adicionar um usuário ao banco de dados. No `HomeController` por exemplo, podemos fazer o seguinte:

```
@Transactional
@ResponseBody
@RequestMapping("/url-magica-maluca-oajksfbvad6584i57j54f9684nvi658efnoewfmnvowefnoeijn")
public String urlMagicaMaluca() {
    Usuario usuario = new Usuario();
    usuario.setNome("Admin");
    usuario.setEmail("admin@casadocodigo.com.br");
    usuario.setSenha("$2a$10$lt7pS7Kxxe5JfP.vjLNSyOXP11eHgh7RoPxo5fvvbMCZkCUss2DGu");
    usuario.setRoles(Arrays.asList(new Role("ROLE_ADMIN")));

    usuarioDao.gravar(usuario);

    return "Url Mágica executada";
}
```

Perceba que estamos simplesmente um usuário comum e assinando o método como transacional para que este possa gravar o usuário no banco de dados. A anotação `@ResponseBody` já vimos que esta faz com que a resposta da requisição seja apenas o texto retornado pelo método.

A senha parece bem maluca, mas é apenas o resultado da criptografia dos digitos de 1 à 6 usando a ferramenta *BCrypt* que também vimos anteriormente. O objeto `usuarioDao` precisa ser criado como atributo desta classe e anotado com `@Autowired` para que o *Spring* se encarregue de cria-lo. Na classe `HomeController` então adicionaremos mais um atributo.

```
@Autowired
private UsuarioDAO usuarioDao;
```

A classe `Role` não tem um construtor que aceite o nome da *role*. Criaremos o mesmo então e também um outro, sendo padrão, para que não tenhamos problemas em outras partes da aplicação.

```
public Role(){
}

public Role(String nome) {
  this.nome = nome;
}
```

Lembre-se que o *Spring Security* está ativo e pode bloquear a *url* que criamos para adicionar nosso usuário padrão. Para evitarmos este problema, adicionaremos esta *url* nas regras de permissão do *Spring Security*. No método `configure` da classe `SecurityConfiguration` teremos:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
  http.authorizeRequests()
  .antMatchers("/produtos/form").hasRole("ADMIN")
  .antMatchers("/carrinho/**").permitAll()
  .antMatchers("/pagamento/**").permitAll()
  .antMatchers(HttpMethod.GET, "/produtos").hasRole("ADMIN")
  .antMatchers(HttpMethod.POST, "/produtos").hasRole("ADMIN")
  .antMatchers("/produtos/**").permitAll()
  .antMatchers("/resources/**").permitAll()
  .antMatchers("/").permitAll()

  .antMatchers("/url-magica-maluca-oajksfbvad6584i57j54f9684nvi658efnoewfmnvowefnoeijn").permit

  .anyRequest().authenticated()
  .and().formLogin().loginPage("/login").permitAll()
  .and().logout().logoutRequestMatcher(new AntPathRequestMatcher("/logout"));
}
```

E por último e não menos importante, criar o método `gravar` na classe `UsuarioDAO`. Pois este não existe. O código é bem simples, apenas usar o método `persist` do objeto `manager` para salvar o usuário.

```
public void gravar(Usuario usuario) {
  manager.persist(usuario);
}
```

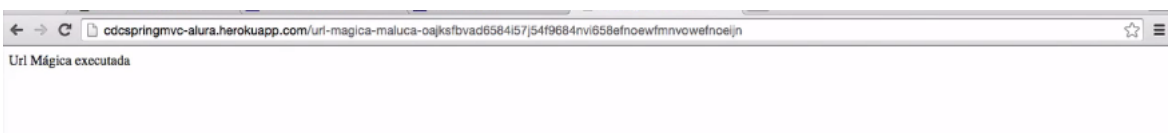
Uma alteração e observação a ser feita, é que da forma que estamos salvando o usuário, não temos garantia de que sua *role* será gravada também junta com este. Para garantirmos isso, precisamos na classe `Usuario` alterar a anotação `OneToMany` do atributo `roles` para que esta persista também os outros objetos relacionados ao usuário, quando este for persistido.

```
@OneToMany(fetch=FetchType.EAGER, ascade=CascadeType.PERSIST)
private List<Role> roles = new ArrayList<Role>();
```

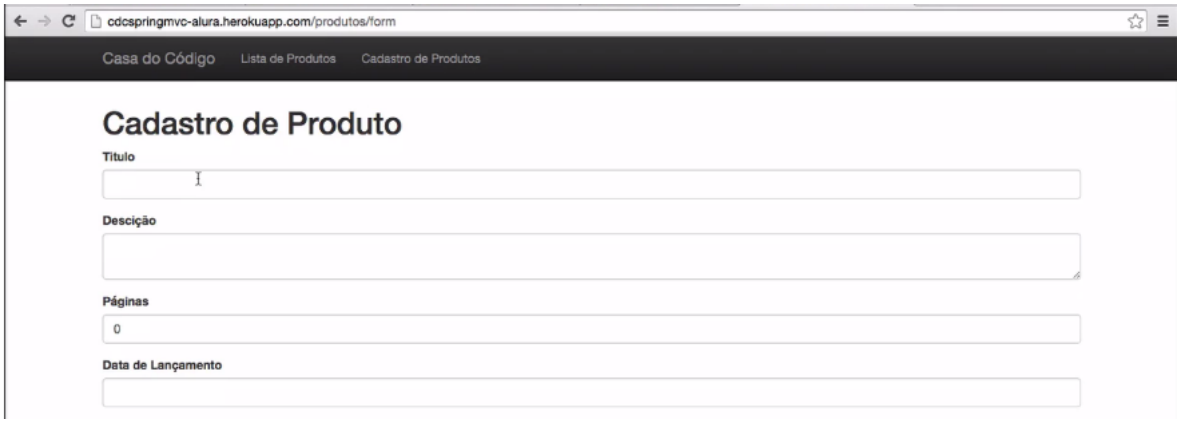
O atributo `cascade` faz exatamente o que queremos. Neste caso, ao persistir o usuário, garantimos que a *role* deste também será persistida. E assim, finalizam-se nossas alterações. Precisamos apenas enviar estas adições para o *Heroku*. Lembra quais são os comandos? São estes:

```
git add .
git commit -m "configurações de usuário padrão"
git push heroku master
```

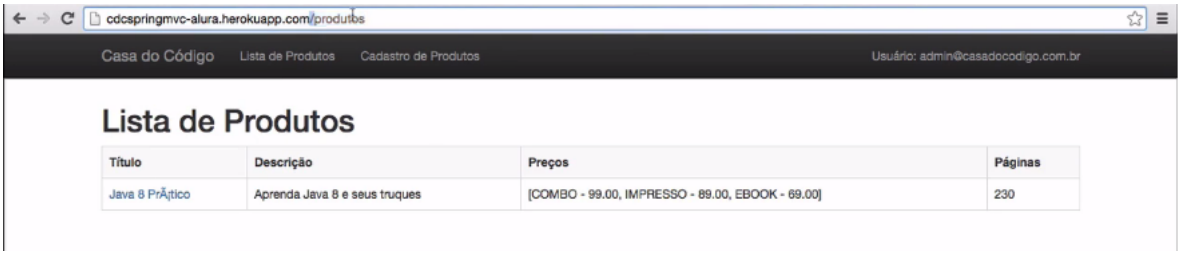
Após a mensagem de sucesso do *Heroku*, podemos acessar a aplicação e verificar que está tudo funcionando. Então podemos acessar a nossa *url maluca* para que o usuário *admin* seja adicionado ao banco de dados e como resposta a este acesso temos a página de sucesso.



E ao tentar fazer o login com os dados do *admin*, conseguimos acessar a página de cadastro de produtos. Indicando que tudo funcionou perfeitamente.




Os cadastros dos produtos também funcionam perfeitamente.




---

**Obsevação**: Você poderá encontrar diversos problemas ao enviar sua aplicação para o *Heroku*. Como este visivelmente na imagem acima, o `encoding` aparementemente está errado. O upload de arquivos dos sumários dos livros pode não funcionar por que o *Heroku* usa uma estratégia diferente para o armazenamento de arquivos.

O Envio de emails também pode não funcionar, porém para isto o *Heroku* disponibiliza ferramentas para configurar estas tarefas de forma simples e fácil de se fazer.

Nestes casos, você pode melhorar a aplicação e continuar a desenvolve-la afim de descobrir mais a respeito do *Spring* e também do *Heroku*. Fica o desafio para que encontre os problemas e soluções para o mesmo. Vale apena o desafio, aprenderá bastante no processo.

---

Com isto, chegamos ao final deste segundo módulo do curso de *Spring MVC*. Esperamos que tenha gostado e que você também não pare por aqui. Há muito ainda a se aprender, existem diversos blogs que falam sobre o *Spring* e você também sempre pode consultar a documentação oficial do mesmo.

Parabenizamos você por ter chegado até aqui e como sempre desejamos: **Bons estudos!**