Seção 04 - Resumo

Perfil de Administrador

Quando desenvolvemos uma aplicação baseada em um sistema de autenticação as credenciais do administrador devem ser inseridas no banco de dados via script SQL ou por meio de um gerenciador, por exemplo, o MySQL Workbench. A partir desse primeiro administrador você poderá inserir novos usuários via aplicação.

Segue abaixo um exemplo script de inserção para o perfil de ADMIN no sistema que estamos trabalhando:

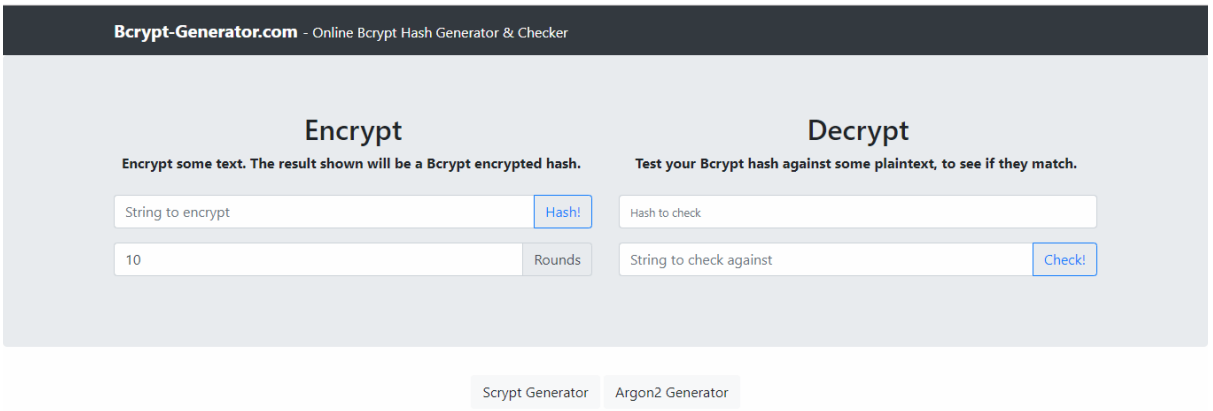
```
INSERT INTO `usuarios` (`id`,`ativo`,`email`,`senha`)
VALUES
(1,1,'admin@clinica.com.br','$2a$10$.F46G/xyDa./MGzFaGAocedxR07U9OILRf3JdOfs
e0WsFXPW5SbNS');
```

A senha do usuário já deve estar criptografada e por conta disso você pode usar a classe main da aplicação com a seguinte operação para ter acesso a senha desejada:

```
public class DemoSecurityApplication {
	public static void main(String[] args) {
		System.out.println(new
BCryptPasswordEncoder().encode("123456"));
	}
}
```

A classe `BCryptPasswordEncoder` do pacote `org.springframework.security.crypto.bcrypt` fornece o método `encode()` no qual você passa o valor da senha a ser criptografado e a saída no console será a senha já criptografada.

Outra forma de ter acesso a uma senha criptografa via algoritmo BCrypt é pelo site https://bcrypt-generator.com/.



BCrypt Generator Password

Neste site você pode adicionar o valor a ser criptografado, mas precisa selecionar a opção `Rounds` com o valor `10`, valor usado pelo sistema de criptografia no Spring Security. Esse valor é referente a complexidade do algoritmo BCrypt e quanto maior, mais tempo de processamento para criar e verificar a senha.

O Rounds varia entre `4` e `31` e, caso queira alterá-lo no Spring Security, use a seguinte sobrecarga do método construtor da classe `BCryptPasswordEncoder`:

```
public BCryptPasswordEncoder(int strength) {
	this(strength, null);
}
```

Relação entre Usuários e Perfis

No sistema trabalhado no curso o sistema de credenciais permite que um usuário tenha dois perfis, por conta disso, temos uma relacionamento muitos para muitos. A cadastrar um usuário administrador via script SQL, será ainda necessário vincular o usuário ao seu perfil. Esse vinculo é criado na tabela de relacionamentos `usuarios_tem_perfis`:

```
INSERT INTO `usuarios_tem_perfis` (`usuario_id`,`perfil_id`) VALUES (1,1);
```

Autenticação != Autorização

Quando trabalhamos com o Spring Security estamos trabalhando com dois conceitos que são: Autenticação e Autorização.

Muitos confundem esses conceitos ou acham que significam a mesma coisa, mas não é bem assim. Embora sejam conceitos simples eles tem significados diferentes.

- Autenticação - é o conceito de segurança baseado em credenciais como login e senha. A autenticação se faz necessária para que o usuário tenha acesso a área privada de uma aplicação segura. Quando a aplicação segura recebe uma requisição a uma área privada sem que o usuário tenha autenticado o erro a ser lançado será o `HTTP 401 Unauthorized`.
- Autorização - é o conceito baseado em permissões que os usuários estão restritos dentro de um sistema de autenticação. A autorização pode ser definida como regras de permissões de acessos baseados em perfis ou papeis. Normalmente ouvimos falar em perfis de usuário, administrador, root ou master, mas cada aplicação pode ter definida sua próprio lista e nomenclatura de perfis. Quando um usuário autenticado tenta acessar uma área restrita a seu perfil o servidor lançará um erro do tipo `HTTP 403 Forbidden`.

Definindo autorizações no Spring Security

Quando configuramos um sistema de autenticação com o Spring Security o usuário logado terá acesso a todos os links e recursos da aplicação. Sendo assim, para limitar o acesso de cada usuário a apenas algumas áreas da aplicação, trabalhamos com o sistema de autorização do Spring Security.

A autorização vai definir qual área da aplicação cada perfil de usuário terá permissão de acesso. Como sempre, o Spring Security não se limita a uma única forma para se configurar essas permissões. Entretanto, as formas mais comuns são via métodos `hasRole([role])` ou `hasAuthority([authority])` e suas variações `hasAnyRole([role1,role2])` e `hasAnyAuthority([authority1,authority2])`.

A principal diferença entre o `hasRole([role])` e `hasAuthority([authority])` é como o Spring Security lida com esses dois métodos. Quando trabalhamos com o `hasRole` o perfil deverá ser criado no banco de dados com o prefixo `ROLE_`: `ROLE_ADMIN`, `ROLE_USER`, `ROLE_READER`, … Entretanto, na classe de configuração onde vamos definir os perfis que acessam as diferentes áreas da aplicação o `ROLE_` deve ser omitido: `hasRole("ADMIN"), hasAnyRole("USER", "READER")`

Já nas páginas se você desejar usar algum recurso que acesse o perfil e faça alguma operação lógica baseada nele, os métodos `hasRole` devem conter o prefixo `ROLE_.`:

```
<div sec:authorize="hasRole('ROLE_USER')">
  This content is only shown to users.
</div>
```

Por conta dessas diferenças, minha preferencia fica pelo uso dos métodos `hasAuthority` e `hasAnyAuthority`. Com esses métodos a forma como o perfil foi declarado no banco de dados será a forma usada em qualquer parte da sua aplicação, ou seja, sem o uso esporádico do prefixo `ROLE_`.

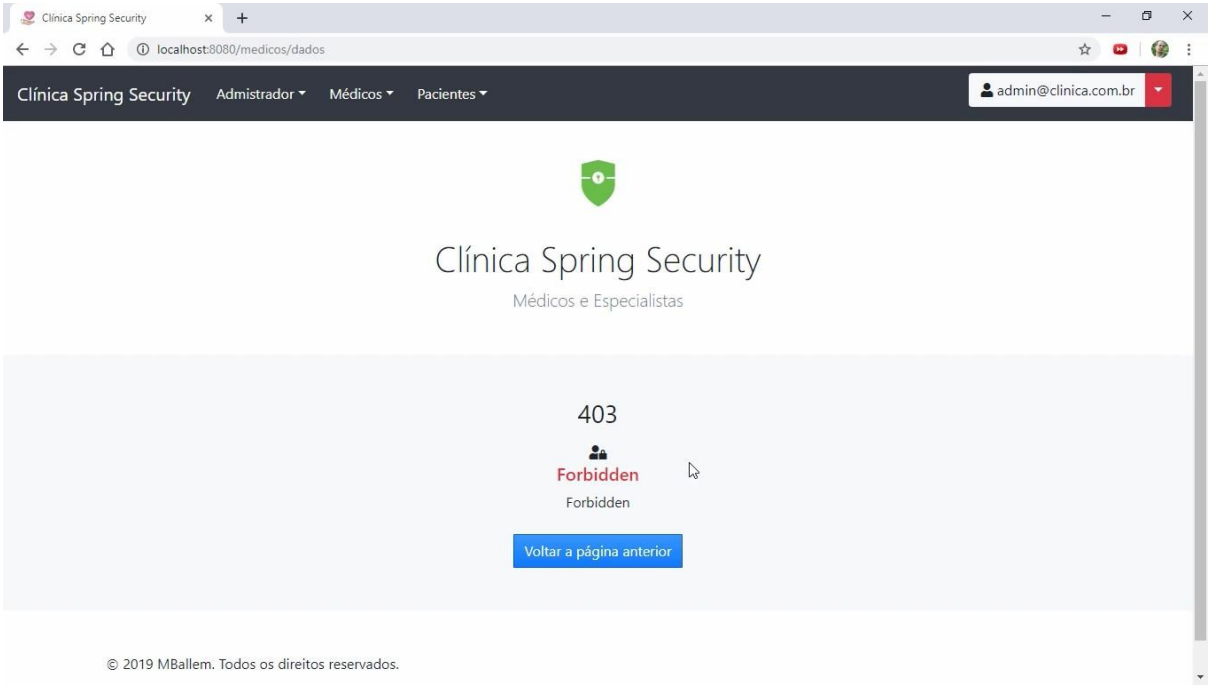
Observe um breve exemplo de como liberar o acesso a áreas da aplicação via instrução de perfis de usuários:

```
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .antMatchers("/resources/**", "/signup", "/about").permitAll()
[1]         .antMatchers("/administrador/**").hasAuthority("ADMIN")
[2]         .antMatchers("/medico/**").hasAnyAuthority("MEDICO",
"ESPECIALISTA")
[3]         .antMatchers("/paciente/*/consultas").hasAuthority("ADMIN")
            .anyRequest().authenticated()
            .and()
        // ...
        .formLogin();
}
```

1. Usamos o método `hasAuthority` para restringir o acesso a área de administrador apenas ao perfil `ADMIN`. Todas as URIs precedidas por `/administrador` estão bloqueadas para qualquer outro perfil de usuário que não `ADMIN`, inclusive a própria `/administrador`.
2. Neste trecho de código o acesso está sendo restringido a dois diferentes perfis que são: `MEDICO` e `ESPECIALISTA`. Nem o `ADMIN` teria acesso as URIs baseadas no prefixo `/medico`.
3. Aqui estamos restringindo o acesso ao perfil de `ADMIN` a URI `/paciente/*/consultas`. O asterisco solitário indica um *path* na URI baseado em um parâmetro que poderia ser um `id`, `nome` ou qualquer outro como: `/paciente/10/consultas` ou `/paciente/bruna/consultas`.

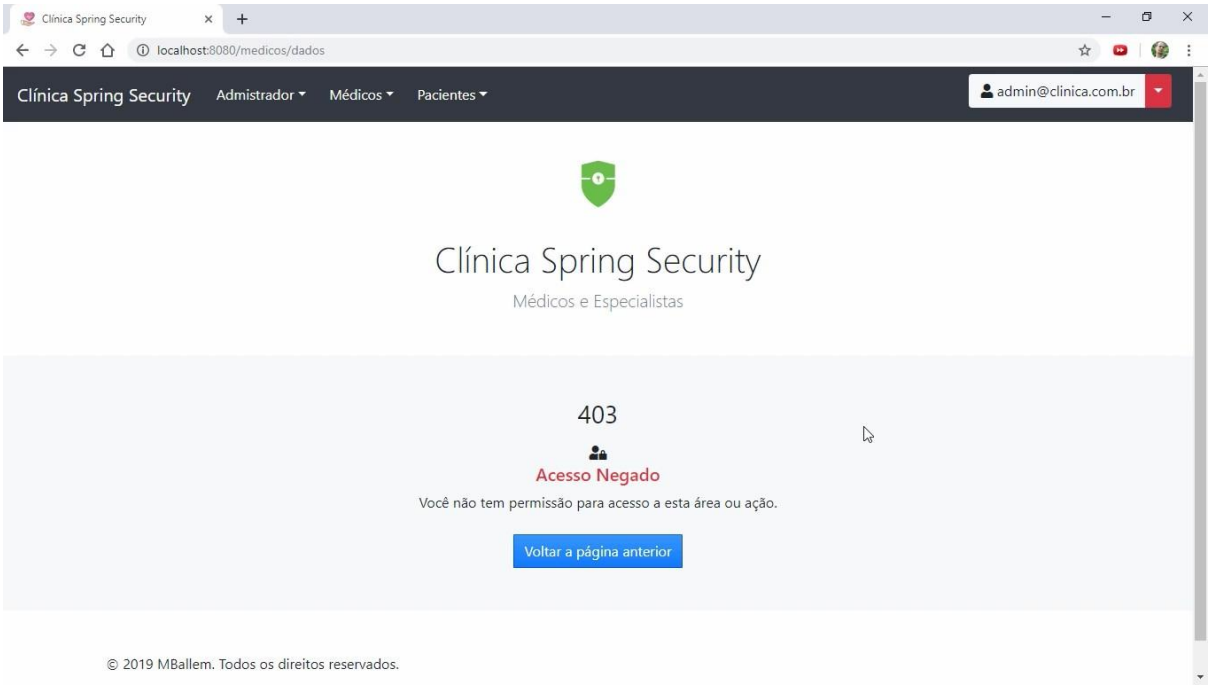
Tratando a exceção para acesso restrito

Quando um usuário logado acessar uma área a qual ele não tem permissão de acesso o Spring Security vai lançar uma exceção com status 403, conforme imagem abaixo:



Acesso negado - Exceção Padrão

Porém, essa exceção terá a mensagem padrão desse status. Sendo assim, podemos tratá-la com o auxilio de uma simples configuração na classe SecurityConfig, a qual resultará em um aspecto como este:



Acesso negado - Exceção Tratada

Para este tratamento devemos adicionar no método `configure()` a instrução `and()` seguida pela chamada aos métodos `exceptionHandling()`, para dizer que vamos tratar a exceção e ao método `accessDeniedPage()` para informar onde vamos tratá-la. Veja um breve exemplo a seguir:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
	http.authorizeRequests()
		// ...
		.anyRequest().authenticated()
		.and()
			//...
		.and()
			.exceptionHandling()
			.accessDeniedPage("/acesso-negado");
}
```

Observe que o método `accessDeniedPage()` recebe como parâmetro uma string contendo um URI. Essa URI é um método declarado no controller, onde você poderá por meio dele, tratar as mensagens exibidas na tela.

```
@GetMapping({"/acesso-negado"})
public String acessoNegado(ModelMap model, HttpServletResponse resp) {
	model.addAttribute("status", resp.getStatus());
	model.addAttribute("error", "Acesso Negado");
	model.addAttribute("message", "Você não tem permissão de acesso a esta
área.");
	return "error";
}
```

No método `acessoNegado()` é declarada sobre sua assinatura a anotação `@GetMapping` com a mesma URI usada em `accessDeniedPage()`, assim, o Spring Security encontrará o método que vai tratar a exceção.

No corpo do método adicionamos as variáveis que o Thymeleaf receberá na página `erro.html` com as devidas mensagens.

Referencias

- Spring Security, Password Encoding
- Spring Security, Expression-Based Access Control
- Thymeleaf + Spring Security integration basics

Código Fonte

Caso tenha tido algum tipo de dificuldade para acompanhar a desenvolvimento do código fonte até o final desta seção, ele está disponível na área de arquivo para download.

Resources for this lecture