Spring Security.

• O que é o SS

o Uma funcionalidade voltada para segurança que oferece soluções rápidas e iniciais para que voce consiga dar o passo inicial em deixar sua aplicação segura, além de restringir e permitir acessos a diferentes módulos de sua aplicação Web.

o Spring Security não é apenas para aplicações Spring, na realidade é possível interligar qualquer aplicação Web com SS, evidentemente que se sua aplicação for desenvolvida com Spring você terá muito mais funcionalidades do SS.

o Além disso, usando o SS para autenticação e autorização nós já estamos protegendo nossa aplicação de diversos ataques como session fixation e o cross site request forgery

• O que veremos?

o O que é Spring Security

o Como configurar o Spring Security

o Configurar autenticação em memória

o Como fazer autenticação via JDBC

o Como fazer autenticação via JPA utilizando a interface UserDetailsService

o Criar uma página de login customizada

o A função “lembrar-me”

o Criar a funcionalidade de logout

o Como adicionar permissões (autorização) em nossas páginas

• Configurando o SS

Adicionar as Dependências do SS no pom.xml

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-web</artifactId>

<!-- <version>4.2.1.RELEASE</version> -->

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-config</artifactId>

<!-- <version>4.2.1.RELEASE</version> -->

</dependency>

Criar uma classe que estenda AbstractSecurityWebApplicationInitializer, essa classe fará o filtro do SS e servirá como inicializadora do SS em nossa aplicação

Para o nosso exemplo criarmos:

**package** com.algaworks.festa;

**import** org.springframework.security.web.context.AbstractSecurityWebApplicationInitializer;

**public** **class** SpringWebSecurityInitializer **extends** AbstractSecurityWebApplicationInitializer {

}

* HTTP Basic

A primeira forma de autenticação que o SS permite é a HTTP Basic, nela o Spring não irá exigir qualquer permissão de acesso, apenas uma autenticação global para acesso a nossa aplicação.

Para isso vamos estender a SecurityWebConfigAdapter e sobrescrever seu metodo configure.

**package** com.algaworks.festa.config;

**import** org.springframework.security.config.annotation.web.builders.HttpSecurity;

**import** org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

**import** org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@EnableWebSecurity

**public** **class** SecurityWebConfig **extends** WebSecurityConfigurerAdapter {

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.authorizeRequests()

// Para qualquer requisição (anyRequest) é preciso estar

// autenticado (authenticated).

.anyRequest().authenticated()

.and()

.httpBasic();

}

// usuario padrao: user

// senha padrao será exibida no console

}

Para essa classe criada não podemos esquecer da notação @EnableWebSecurity

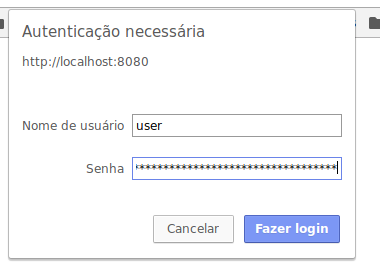
Ela é quem diz ao Spring para habilitar os recursos inerentes a segurança em nossa aplicação

Neste ponto nossa aplicação já consegue solicitar a autenticação no ato de sua inicialização. A Implementação do configure apenas diz ao Spring que para qualquer requisição que ocorra será necessário estar autenticado.

Mesmo sem termos definido qualquer usuário ou senha o Spring, como sempre, já nos fornece o primeiro passo, se executarmos a aplicação neste ponto ele exibirá no console uma senha gerada para o usuario padrão “user”. Você verá algo assim :

Using default security password: 1d9fb37c-d5f4-4752-842d-a2ff1a47336f

Evidentemente que isso não é nada comercial, mas pode ter sua utilidade em casos onde precisamos ganhar tempo.



* Autenticação em Memória

Só para se ter uma ideia da simplicidade que o SS fornece, neste ponto já é possível dizer que já configuramos 50% de tudo, porém, toda aplicação precisa ter seus usuários, vamos parametrizar isso agora.

Uma autenticação em memória não é utilizada em produção, ao menos não deveria ser, mas é muito útil, além de viável, o seu uso em Desenvolvimento! Lembre-se que a palavra aqui é TEMPO.

Para isso faremos algumas mudanças na classe SecurityWebConfig , vamos sobrescrever o método configure para que ele receba uma instancia de AuthenticationManagerBuilder. Vamos usar esse objeto para criar nossos usuários estáticos.

**package** com.algaworks.festa.config;

**import** org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

**import** org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

**import** org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@EnableWebSecurity

**public** **class** SecurityWebConfig **extends** WebSecurityConfigurerAdapter {

@Override

**public** **void** configure(AuthenticationManagerBuilder builder) **throws** Exception {

builder

.inMemoryAuthentication()

.withUser("Lula").password("13")

.roles("USER")

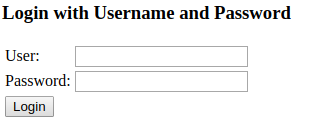
.and()

.withUser("Bolsonaro").password("51")

.roles("USER");

}

}



* Autenticação via JDBC

Essa já uma alterativa que pode ser utilizada em produção sem nenhum problema. Para usarmos essa autenticação no spring tudo que iremos necessitar é de uma base de dados SQL referente as consultas que buscam o usuário no banco.

Lembrando que JAMAIS devemos deixar as senhas serem gravadas no BD no formato de texto limpo! O correto é usar algum algoritmo de criptografia para encriptar e desencriptar a senha a cada vez que ela for utilizada. Para este fim utilizaremos o PasswordEncoder, mas você poderia usar outras formas de criptografia que achar conveniente.

Pois bem, para JDBC vamos alterar a classe SecurityWebConfig inserindo atributos estáticos que farão a consulta ao nosso BD.

@EnableWebSecurity

**public** **class** SecurityWebConfig **extends** WebSecurityConfigurerAdapter {

**private** **static** **final** String ***USUARIO\_POR\_LOGIN*** = "SELECT login, senha, ativo, nome FROM usuario" + " WHERE login = ?";

**private** **static** **final** String ***PERMISSOES\_POR\_USUARIO*** = "SELECT u.login, p.nome FROM usuario\_permissoes up"

+ " JOIN usuario u ON u.id = up.usuarios\_id"

+ " JOIN permissao p ON p.id = up.permissoes\_id"

+ " WHERE u.login = ?";

**private** **static** **final** String ***PERMISSOES\_POR\_GRUPO*** = "SELECT g.id, g.nome, p.nome FROM grupo\_permissoes gp"

+ " JOIN grupo g ON g.id = gp.grupos\_id"

+ " JOIN permissao p ON p.id = gp.permissoes\_id"

+ " JOIN usuario\_grupos ug ON ug.grupos\_id = g.id"

+ " JOIN usuario u ON u.id = ug.usuarios\_id"

+ " WHERE u.login = ?";

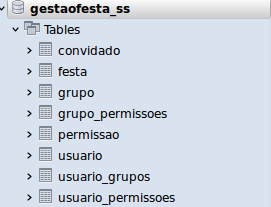
Parece assustador, mas é apenas uma consulta SQL. Que diz o que deverá ser consultado pelo SS no ato da autenticação .

O próximo passo será relacionarmos essa classe com a DataSource do pacote tomcat.jdbc.pool.DataSource ela será utilizada como uma fonte de dados genérica para prover uma interface ao usuário. Caso voce faça este exemplo utilizando o banco de dados H2 o datasource consegue gerar automaticamente o banco pra você.

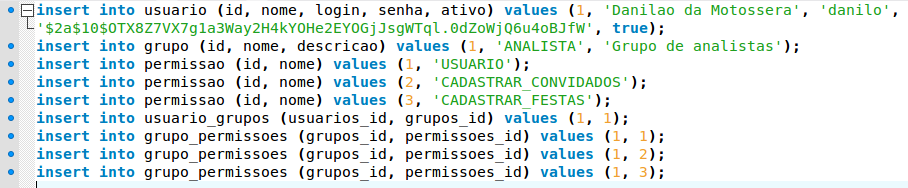
@Autowired

**private** DataSource dataSource;

É importante lembrar que para testar essa implementação você irá necessitar das tabelas usuários e suas permissões. Você poderia criar as entidades Usuário, Grupo e Permissao normalmente e deixar que o Spring crie as tabelas (mas nós iremos fazer isso na proxima implementação quando mostrarmos a autenticação com JPA) ou mesmo criar as tabelas previamente direto no seu banco de dados.



Vamos alimentar nosso arquivo IMPORT.SQL para popular nossas tabelas incialmente:



Você também poderia considerar nossa própria entidade Convidado como sendo o “usuario” da aplicação, porém optamos por não contextualizar dessa forma e deixando a Entidade usuário como sendo independente de convidado.

Bom agora vamos sobrescrever o método configure outra vez. caso voce tenha notado a primeira linha o builder deste método passou a ser .jdbc.Authentication() , ao invés de estar como autenticação em memória como no condigo anterior

@Override

**protected** **void** configure(AuthenticationManagerBuilder builder) **throws** Exception {

builder

.jdbcAuthentication()

.dataSource(dataSource)

.passwordEncoder(**new** BCryptPasswordEncoder())

.usersByUsernameQuery(***USUARIO\_POR\_LOGIN***)

.authoritiesByUsernameQuery(***PERMISSOES\_POR\_USUARIO***)

.groupAuthoritiesByUsername(***PERMISSOES\_POR\_GRUPO***)

.rolePrefix("ROLE\_");

}

O Datasource e o PasswordEncore são injetados diretamente no método para que o Spring faça o restante as configurações e em seguida vem o nosso SQL. Onde nós falamos para o SS como buscar os usuarios, grupos e permissões.

Para a encriptação, apenas caso voce queira ver qual é o hash gerado a partir da senha inserida, no caso aqui usei 123, podemos deixar um método main dentro dessa mesma classe.

**public** **static** **void** main(String[] args) {

System.***out***.println(**new** BCryptPasswordEncoder().encode("123"));

}

Com isso já é possivel fazermos o login usando a autenticação via banco de dados.

Nossa classe final ficará assim:

**package** com.algaworks.festa.config;

**import** org.apache.tomcat.jdbc.pool.DataSource;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

**import** org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

**import** org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

**import** org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@EnableWebSecurity

**public** **class** SecurityWebConfig **extends** WebSecurityConfigurerAdapter {

**private** **static** **final** String ***USUARIO\_POR\_LOGIN*** = "SELECT login, senha, ativo, nome FROM usuario"

+ " WHERE login = ?";

**private** **static** **final** String ***PERMISSOES\_POR\_USUARIO*** = "SELECT u.login, p.nome FROM usuario\_permissoes up"

+ " JOIN usuario u ON u.id = up.usuarios\_id"

+ " JOIN permissao p ON p.id = up.permissoes\_id"

+ " WHERE u.login = ?";

**private** **static** **final** String ***PERMISSOES\_POR\_GRUPO*** = "SELECT g.id, g.nome, p.nome FROM grupo\_permissoes gp"

+ " JOIN grupo g ON g.id = gp.grupos\_id"

+ " JOIN permissao p ON p.id = gp.permissoes\_id"

+ " JOIN usuario\_grupos ug ON ug.grupos\_id = g.id"

+ " JOIN usuario u ON u.id = ug.usuarios\_id"

+ " WHERE u.login = ?";

@Autowired

**private** DataSource dataSource;

@Override

**protected** **void** configure(AuthenticationManagerBuilder builder) **throws** Exception {

builder

.jdbcAuthentication()

.dataSource(dataSource)

.passwordEncoder(**new** BCryptPasswordEncoder())

.usersByUsernameQuery(***USUARIO\_POR\_LOGIN***)

.authoritiesByUsernameQuery(***PERMISSOES\_POR\_USUARIO***)

.groupAuthoritiesByUsername(***PERMISSOES\_POR\_GRUPO***)

.rolePrefix("ROLE\_");

}

**public** **static** **void** main(String[] args) {

System.***out***.println(**new** BCryptPasswordEncoder().encode("123"));

}

}

* Autenticação via JPA com interface UserDetailsService

A terceira forma de autenticarmos um usuário é via JPA ela permite que seja possível uma maior liberdade na hora de personalizarmos a interface UserDetailsService. Ela é uma interface do próprio SS a qual vamos implementar.

Para nosso exemplo foi criado às três entidades citadas anteriormente: Usuário, Grupo e Permissão. A implementação delas é exatamente idêntica ao que já foi feito em convidados e festas. Sem segredos.

Além dessas entidades vamos criar uma nova classe UsuarioSistema, que é uma implementação de User. Precisamos dela para conseguir dar um retorno para o método loadUserByUsername, que vamos implementar a seguir na classe FestaUserDetailsService.

Para a UsuárioSistema nossa classe ficará assim:

**package** com.algaworks.festa.security;

**import** java.util.Collection;

**import** org.springframework.security.core.GrantedAuthority;

**import** org.springframework.security.core.userdetails.User;

**public** **class** UsuarioSistema **extends** User {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** String nome;

**public** UsuarioSistema(String nome, String username, String password, Collection<? **extends** GrantedAuthority> authorities) {

**super**(username, password, authorities);

**this**.nome = nome;

}

**public** String getNome() {

**return** nome;

}

}

Devemos criar a FestaUserDetailsService que é a Classe que irá implementar a interface UserDetails vamos usa-la para conseguirmos dar um retorno para o método loadUserByUsername, falaremos dele a seguir.

**package** com.algaworks.festa.security;

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.security.core.GrantedAuthority;

**import** org.springframework.security.core.authority.SimpleGrantedAuthority;

**import** org.springframework.security.core.userdetails.UserDetails;

**import** org.springframework.security.core.userdetails.UserDetailsService;

**import** org.springframework.security.core.userdetails.UsernameNotFoundException;

**import** org.springframework.stereotype.Component;

**import** com.algaworks.festa.model.Grupo;

**import** com.algaworks.festa.model.Permissao;

**import** com.algaworks.festa.model.Usuario;

**import** com.algaworks.festa.repository.Grupos;

**import** com.algaworks.festa.repository.Permissoes;

**import** com.algaworks.festa.repository.Usuarios;

@Component

**public** **class** FestaUserDetailsService **implements** UserDetailsService {

@Autowired

**private** Usuarios usuarios;

@Autowired

**private** Grupos grupos;

@Autowired

**private** Permissoes permissoes;

@Override

**public** UserDetails loadUserByUsername(String username) **throws** UsernameNotFoundException {

Usuario usuario = usuarios.findByLogin(username);

**if** (usuario == **null**) {

**throw** **new** UsernameNotFoundException("Usuário não encontrado!");

}

**return** **new** UsuarioSistema(usuario.getNome(), usuario.getLogin(), usuario.getSenha(), authorities(usuario));

}

**public** Collection<? **extends** GrantedAuthority> authorities(Usuario usuario) {

**return** authorities(grupos.findByUsuariosIn(usuario));

}

**public** Collection<? **extends** GrantedAuthority> authorities(List<Grupo> grupos) {

Collection<GrantedAuthority> auths = **new** ArrayList<>();

**for** (Grupo grupo: grupos) {

List<Permissao> lista = permissoes.findByGruposIn(grupo);

**for** (Permissao permissao: lista) { auths.add(**new** SimpleGrantedAuthority("ROLE\_" + permissao.getNome()));

}

}

**return** auths;

}

}

Esse método existe apenas para que possamos buscar o usuario pelo seu username. Não é preciso nem mesmo fazer qualquer validação de senha nele, pois o SS faz isso pra você.

Essa implementação é uma de varias formas de se realizar essa autenticação, contudo ela pode ser reaproveitada em qualquer aplicação a qual você utilize, uma vez que configurada não precisaremos mais alterar essa classe e ainda vamos simplificar a classe SecurityWebCOnfig.

Ou seja, utilizamos desta forma o SS ainda eleva a Reusabilidade de nosso código para futuros projetos.

Vou utilizar JPA para buscar o usuário do banco de dados, mas é importante destacar que não importa de onde ele vem – banco de dados, memória, etc – o que importa é devolver uma implementação de UserDetails. Poderíamos utilizar aqui uma consulta JDBC com queries personalizadas, se quiséssemos.

Com o nosso UserDetailsService em mãos, podemos voltar na classe SecurityWebConfig e terminar a configuração

**package** com.algaworks.festa.config;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;

**import** org.springframework.security.config.annotation.web.builders.HttpSecurity;

**import** org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

**import** org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

**import** org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

**import** com.algaworks.festa.security.FestaUserDetailsService;

@EnableWebSecurity

**public** **class** SecurityWebConfig **extends** WebSecurityConfigurerAdapter {

@Autowired

**private** FestaUserDetailsService ssUserDetailsService;

@Override

**protected** **void** configure(AuthenticationManagerBuilder builder) **throws** Exception {

builder

.userDetailsService(ssUserDetailsService)

.passwordEncoder(**new** BCryptPasswordEncoder());

}

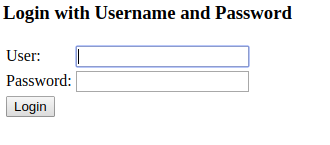
**public** **static** **void** main(String[] args) {

System.***out***.println(**new** BCryptPasswordEncoder().encode("123"));

}

}

Pecebe como nossa classe fica muito mais limpa? Neste ponto já é possivel até mesmo subir a aplicação e autenticarmos sem maiores problemas.



* Customizando a Pagina de Login

Se você olha o código fonte da aplicação neste momento você irá notar que não temos nenhuma Viewer para login, e ainda sim vemos um formulário para autenticação! Acontece que , novamente, o Spring Security faz você ganhar tempo e já te fornece uma pagina de login básica a qual você pode utilizar. Ela é simples, mas muito funcional,

Agora para customizarmos nossa própria pagina de login.

Vamos criar um novo método configure, mas deixe o anterior lá, vamos utilizar um novo método com assinatura diferente.  
 @Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.authorizeRequests()

.anyRequest().authenticated()

.and()

.formLogin()

// Aqui dizemos que temos uma página customizada.

.loginPage("/login")

// Mesmo sendo a página de login, precisamos avisar

// ao Spring Security para liberar o acesso a ela.

.permitAll();

}

Não esqueça de mapear o /login dentro do controller. E de criar seu formulario de login.

No nosso exemplo usamos este:

<form th:action=*"@{/login}"* method=*"post"*>

<input name=*"username"* class=*"form-control"* placeholder=*"Usuário"*/>

<input type=*"password"* name=*"password"* class=*"form-control"*placeholder=*"Senha"*/>

<button class=*"btn btn-primary btn-block"*>Entrar</button>

</form>

Esse é o coração do seu form de login

Neste Momento se iniciarmos a aplicação veremos o nossa tela sem qualquer elemento visual, pois não fizemos liberações para o CSS, JS e Bootstrapp no nosso SS. Para fazermos isto para modificarmos nosso método configure

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.authorizeRequests()

.antMatchers("/resources/\*\*","/css/\*\*","/fonts/\*\*","/images/\*\*","/js/\*\*","/index","/").permitAll()

.anyRequest().authenticated()

.and()

.formLogin()

// Aqui dizemos que temos uma página customizada.

.loginPage("/login")

// Mesmo sendo a página de login, precisamos avisar

// ao Spring Security para liberar o acesso a ela.

.permitAll();

}

A linha .antMatchers("/resources/\*\*","/css/\*\*","/fonts/\*\*","/images/\*\*","/js/\*\*","/index","/").permitAll() Eh quem libera a aparecia a nossa viewer

A ação é /login e o método HTTP que será usado é o POST. Isso é porque a ação tem o mesmo path da URL que abre a página de login. Só que a requisição para página é feita com um GET para /login e a submissão do formulário é feita com um POST.

O campo “Usuário” precisa ter o nome username. Ele pode ser configurado explicitamente, mas eu preferi utilizar o padrão. Da mesma forma é o campo “Senha” de nome password.

Com as alterações acima já é possível fazer o login através da nossa página. Só que ainda temos um problema com os arquivos JS e CSS utilizados para montar o layout dela:

Lembrar-me

Configurar a opção de “lembra-me” por meio do SS é algo muito simples. Voce literalmente precisa de duas etapas para já ter essa função ativa em sua aplicação.   
A primeira é criar o checkbox no seu formulário:

<input type=*"checkbox"* id=*"remember-me"* name=*"remember-me"* />

<label for=*"remember-me"*>Oi, você vem sempre aqui?</label>

O segredo esta no nome do checkbox, ele precisa ser “remember-me” para que o spring saiba do que se trata, esse nome também pode ser alterado caso você deseja, mas vamos deixar ele no padrão.

O segundo passo é simplesmente personalizar nosso método configure para que ele habilite essa função.

.and()

.rememberMe(); // faz o spring lembrar de voce ;-)

O nosso método ficará assim:

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.authorizeRequests()

.antMatchers("/resources/\*\*","/css/\*\*","/fonts/\*\*","/images/\*\*","/js/\*\*").permitAll()

.anyRequest().authenticated()

.and()

.formLogin().loginPage("/login")// Aqui dizemos que temos uma página customizada.

.permitAll()// Mesmo sendo a página de login, precisamos avisar ao Spring Security para liberar o acesso a ela.

.and()

.rememberMe(); // faz o spring lembrar de voce ;-)

}

Pronto. Com isso o SS não irá mais solicitar seu login e senha a cada nova sessão

Fazendo Logout

Por mais que você goste da aplicação uma hora você vai precisar sair dela, da forma atual a única maneira que você consegue sair de sua aplicação é fechando totalmente o navegador, para resolvermos isso vamos implementar o logout.

Opa, mas estamos novamente com SS e isso já está implementado tudo que precisamos fazer é adicionar um methodo post para /logout dentro da nossa view. No nosso exemplo isso foi inserido dentro da navbar em nosso cabeçalho.

<form method=*"post"* class=*"navbar-form navbar-right"* th:action=*"@{/logout}"*>

<button type=*"submit"* class=*"btn btn-default"*>

<span class=*"glyphicon glyphicon-log-out"*></span>Sair

</button>

</form>

O path “/logout” também pode ser personalizado caso você queira.

Autorização

Como já temos tudo configurado e já ate adicionamos duas funcionalidades extras (lembrar e logout) o SS ainda facilita na hora de liberar permissões especificas para determinados usuários, faremos isso a seguir:

Vamos identificar no nosso configure quais as permissões necessárias que precisam haver no banco de dados para liberar uma determinada URL.

No nosso caso temos 3 usuarios :

1 ) Danilo com permissão total (ele é membro dos 3 grupos possíveis de permissões)

2) Luana com permissão de Organizador (consegue cadastrar as festas)

3) Higor com permissão de Convidado ( Consegue cadastrar os convidados )

Lembrando que essas permissões foram injetadas diretamente no BD para que não precisássemos implementar telas de cadastro para elas. (vide arquivo import.sql)

Vamos ver como fica nosso método configure:

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

http

.authorizeRequests()

.antMatchers("/resources/\*\*","/css/\*\*","/fonts/\*\*","/images/\*\*","/js/\*\*").permitAll()

// libera acesso a convidados quem tiver a permissao de convidados

.antMatchers("/convidados").hasRole("CADASTRAR\_CONVIDADOS")

// libera acesso a /festas quem tiver a permissao CADASTRAR\_FESTAS

.antMatchers("/festas").hasRole("CADASTRAR\_FESTAS")

.anyRequest().authenticated()

.and()

.formLogin().loginPage("/login")// Aqui dizemos que temos uma página customizada.

.permitAll()// Mesmo sendo a página de login, precisamos avisar ao Spring Security para liberar o acesso a ela.

.and()

.rememberMe(); // faz o spring lembrar de voce ;-)

}