



Spring Security com Spring Boot



**Certified
Developer**
The Ultimate Tech Degree

DigitalHouse >



Passos

**Configurar a
administração de
funções e usuários**

**Criptografar
a senha**

1

2

3

4

**Adicionar a
biblioteca**

**Configurar a autenticação e
segurança de URL**





1 - Adicionar a biblioteca, de Starter do Spring Security

Primeiro, precisamos assegurar de adicionar a dependência do Spring Security em nosso arquivo **pom.xml**.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```





2 - Configurar a administração de funções e usuários

Para tomar decisões sobre o acesso aos recursos, é necessário *identificar os diferentes usuários e quais funções possuem* para validar se tem autorização de acesso aos diferentes recursos da aplicação.

Para isso, devemos implementar a interface **UserDetailsService**. Esta interface descreve um objeto que realiza acesso a dados com um único método **loadUserByUsername** que retorna informações sobre um usuário com base em seu nome de usuário.





```
@Service
@Transactional
public class UserDetailsServiceImpl implements UserDetailsService {
    @Autowired
    UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        MyUser appUser = userRepository.findByUsername(username);

        Set<GrantedAuthority> grantList = new HashSet<GrantedAuthority>();

        for (Role role: appUser.getRoles()) {
            GrantedAuthority grantedAuthority = new SimpleGrantedAuthority(role.getDescription());
            grantList.add(grantedAuthority);
        }

        UserDetails user = null;
        user = (UserDetails) new User(username, appUser.getPassword(), grantList);
        return user;
    }
}
```

Busca o usuário pelo nome de usuário em nosso banco de dados



```
@Service
@Transactional
public class UserDetailsServiceImpl implements UserDetailsService {
    @Autowired
    UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        MyUser appUser = userRepository.findByUsername(username);

        Set<GrantedAuthority> grantList = new HashSet<GrantedAuthority>();

        for (Role role: appUser.getRoles()) {
            GrantedAuthority grantedAuthority = new SimpleGrantedAuthority(role.getName());
            grantList.add(grantedAuthority);
        }

        UserDetails user = null;
        user = (UserDetails) new User(username, appUser.getPassword(), grantList);
        return user;
    }
}
```

Cria uma lista de funções / acessos que o usuário possui



```
@Service
@Transactional
public class UserDetailsServiceImpl implements UserDetailsService {
    @Autowired
    UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        MyUser appUser = userRepository.findByUsername(username);
        Set<GrantedAuthority> grantList = new HashSet<GrantedAuthority>();

        for (Role role: appUser.getRoles()) {
            GrantedAuthority grantedAuthority = new SimpleGrantedAuthority(role.getDescription());
            grantList.add(grantedAuthority);
        }

        UserDetails user = null;

        user = (UserDetails) new User(username, appUser.getPassword(), grantList);
        return user;
    }
}
```

Cria e retorna o objeto de usuário suportado pelo Spring Security



3 - Configurar a autenticação e segurança de URL

Usando a anotação **@EnableWebSecurity** e estendendo a classe **WebSecurityConfigurerAdapter**, podemos configurar e ativar rapidamente a segurança para os diferentes usuários que efetuam login em nossa aplicação.

Por sua vez, **@EnableWebSecurity** habilita o suporte de segurança web do Spring Security e também proporciona a integração com o Spring MVC e **WebSecurityConfigurerAdapter** fornecendo um conjunto de métodos que são usados para habilitar uma configuração de segurança web específica.





Como iremos usar nossa própria configuração, devemos criar uma classe que herda de `WebSecurityConfigurerAdapter`, e nela sobrescrever o método **configure** onde iremos customizar nossa configuração de segurança.

Indica que a classe é uma classe de configuração e precisa ser carregada durante a inicialização do servidor.

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // nossa própria configuração de segurança.
    }
}
```



Como vimos em nossa classe de configuração de segurança, devemos substituir o método `configure()` para habilitar a proteção de URL.

```
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .anyRequest()  
        .authenticated()  
        .and()  
        .httpBasic();  
}
```

Ativa a proteção HTTP básica, normalmente o navegador mostra uma caixa onde seremos solicitados a inserir o nome de usuário e senha.





Vejamos um exemplo. Isso irá gerar automaticamente um formulário de login e é o que iremos utilizar.

```
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .antMatchers("/home").hasRole("USER")  
        .antMatchers("/vendas").hasRole("ADMIN")  
        .and().formLogin()  
        .and().logout();  
}
```





```
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .antMatchers("/home").hasRole("USER")  
        .antMatchers("/vendas").hasRole("ADMIN")  
        .and().formLogin()  
        .and().logout();  
}
```

Indica que todas as solicitações estarão protegidas, ou seja, precisamos nos autenticar para poder acessar qualquer parte do site.





```
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .antMatchers("/home").hasRole("USER")  
        .antMatchers("/vendas").hasRole("ADMIN")  
        .and().formLogin()  
        .and().logout();  
}
```

É utilizado para indicar que um usuário com a função “USER” pode entrar na home.





```
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .antMatchers("/home").hasRole("USER")  
        .antMatchers("/vendas").hasRole("ADMIN")  
        .and().formLogin()  
        .and().logout();  
}
```

É usada para indicar que um usuário com a função “ADMIN” pode entrar somente na página de vendas.





4 - Criptografar a senha

As senhas dos usuários podem ser criptografadas usando um algoritmo de criptografia.

Uma função de criptografia permite que qualquer texto seja transformando em um código que não possa ser revertido.

A criptografia/codificação de senhas nos permite armazená-las de forma segura.

O **Spring Security** disponibiliza várias implementações de codificação de senhas para você escolher. Cada um tem suas vantagens e desvantagens, e um desenvolvedor pode escolher qual usar, dependendo do requisito de autenticação da sua aplicação. Para fins práticos veremos o **BCryptPasswordEncoder**.





BCryptPasswordEncoder

Ao instanciar um objeto da classe **BCryptPasswordEncoder**, podemos criptografar/gerar um hash da senha, para isso devemos:

- 1) Criamos um encoder chamando o construtor de **BCryptPasswordEncoder** com o valor 12. Esse valor pode estar entre 4 e 31 e quanto maior, mais trabalho é necessário para calcular o hash.

```
BCryptPasswordEncoder encoder = new BCryptPasswordEncoder(12);
```

- 2) Invocamos o método **encode("senha")**, passando a senha que queremos criptografar. É assim que o hash da senha se parece:

encodedPassword: \$2a\$12\$DlfnjD4YgCNbDEtgd/ITeOj.jmUZpuz1i4gt51YzetW/iKY2O3bqa

```
String encodedPassword = encoder.encode("UserPassword");
```




Usando @PreAuthorize e @PostAuthorize

A anotação **@PreAuthorize** verifica a expressão antes de ingressar no método, para decidir se um usuário na sessão tem ou não acesso para usá-lo, enquanto a anotação **@PostAuthorize** verifica após a execução do método podendo alterar o resultado.

```
@PreAuthorize("hasAnyRole('ROLE_ADMIN','ROLE_USER')")  
public User updateUser(User formUser) throws Exception {  
    ...  
}
```

DigitalHouse>