Spring Framework

Parte 05 – Spring Security

Introdução

- Spring Security é um framework para facilitar a incorporação de funcionalidades relativas ao controle de acesso necessárias às aplicações.
- Possui recursos de integração com a API de servlets, permitindo seu uso em conjunto com qualquer framework baseado nesta API como Spring MVC e JSF.

Conceitos

- **Autenticação**: refere-se à identificação de um usuário, ou seja, garantir que um usuário é realmente quem ele diz ser. A identificação é realizada com base nas credenciais apresentadas pelo usuário (e.g, login e senha, certificação digital, dados biométricos, etc).
- **Autorização**: Um usuário autenticado possui um conjunto de permissões que indicam as operações e telas disponíveis a este usuário. Autorização é o processo que libera ou bloqueia o acesso a um recurso de acordo com as permissões do usuário.

Configuração

- Adicionar dependências no **pom.xml**:

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-taglibs</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-config</artifactId>
</dependency>
```

Configuração

- Classe com as configurações do Spring Security (1):

```
package cursoSpring.revenda_veiculos.config;

import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.authentication.builde
rs.AuthenticationManagerBuilder;
import
org.springframework.security.config.annotation.web.configuration.Ena
bleWebMvcSecurity;
import
org.springframework.security.config.annotation.web.configuration.Web
SecurityConfigurerAdapter;
```

Configuração

- Classe com as configurações do Spring Security (2):

```
@Configuration
@EnableWebMvcSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter{

  @Override
  public void configure(AuthenticationManagerBuilder auth)
  throws Exception{
    auth.inMemoryAuthentication()
        .withUser("admin").password("admin").roles("ADMIN")
        .and().withUser("user").password("user").roles("USER");
  }
}
```

Configuração

- Classe com as configurações do Spring Security (2):

```
@Configuration
@EnableWebMvcSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter{

  @Override
  public void configure(AuthenticationManagerBuilder auth)
  throws Exception{
    auth.inMemoryAuthentication()
        withUser("admin").password("admin").roles("ADMIN")

}
```

A adição de **@EnableWebMvcSecurity** e o uso de **WebSecurityConfigurerAdapter** como superclasse definem uma série de funcionalidades de segurança: autenticação obrigatória em todas as URLs da aplicação; geração de um formulário de login; prevenção de ataques CSRF; integração com cabeçalhos HTTP de segurança; integração com a API de servlets.

Configuração

- Classe com as configurações do Spring Security (2):

```
@Configuration
@EnableWebMvcSecurity
public class SecurityConfig e                                    er{

  @Override
  public void configure(AuthenticationManagerBuilder auth)
  throws Exception{
    auth.inMemoryAuthentication()
        .withUser("admin").password("admin").roles("ADMIN")
        .and().withUser("user").password("user").roles("USER");
  }
}
```

Define dois usuários para a aplicação com suas respectivas credenciais e níveis (roles) de autorização. Neste caso, estes usuários são armazenados em memória.

Configuração

- Classe **SpringMVCServlet**:

```
package cursoSpring.revenda_veiculos.config;

import org.springframework.core.annotation.Order;
...

@Order(1)
public class SpringMVCServlet extends
AbstractAnnotationConfigDispatcherServletInitializer {

  @Override
  protected Class<?>[] getRootConfigClasses() {
    return new Class[]{AppConfig.class, SecurityConfig.class};
  }
  ...
}
```

Configuração

- Classe **SpringMVCServlet**:

```
package cursoSpring.revenda.veiculos.config;

import org.spri
...

@Order(1)
public class SpringMVCServlet extends
AbstractAnnotationConfigDispatcherServletInitializer {

  @Override
  protected Class<?>[] getRootConfigClasses() {
    return new Class[]{AppConfig.class, SecurityConfig.class};
  }
  ...
}
```

Indica que este deve ser o primeiro Servlet com configurações do Spring a ser carregado. A seguir iremos definir um servlet específico ao Spring Security.

Faz com que a classe **SecuirtyConfig** seja carregada pelo contexto do Spring.

Configuração

- Classe **SecurityInitializer**:

```
package cursoSpring.revenda_veiculos.config;

import org.springframework.core.annotation.Order;
import
org.springframework.security.web.context.AbstractSecurityWebApplicat
ionInitializer;

@Order(2)
public class SecurityInitializer extends
AbstractSecurityWebApplicationInitializer{}
```

Configuração

- Classe **SecurityInitializer**:

```
package cursoSpring.revenda_veiculos.config;

import org.springframework.core.annotation.Order;
import
org.springframework.security.web.context.AbstractSecurityWebApplicat
ionInitializer;

@Order(2)
public class SecurityInitializer extends
AbstractSecurityWebApplicationInitializer{}
```

Esta classe registra o filtro do Spring Security responsável por capturar as requisições HTTP para então aplicar as restrições de segurança. Para garantir que esse filtro seja o primeiro da fila de filtros registrados, esta classe deve ser a última a ser carregada, por isso a anotação @Order(2).

Logout

- Como a proteção CSRF está ativada o procedimento de logout exige:
  - uma requisição POST;
  - a presença do token CSRF (deve estar presente em todas as requisições PATCH, PUT, POST e DELETE).
- Nas páginas JSP, o token pode ser acessado através do objeto **_crsf**:
  - **${_csrf.parameterName}**
  - **${_csrf.token}**
- Ao utilizar as tags de formulário do Spring MVC, os tokens são adicionados automaticamente.

Logout

- **index.jsp**:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://www.springframework.org/tags/form"
prefix="form" %>
<html>
<body>
  <h2>Hello World!</h2>
  <c:url var="logoutUrl" value="/logout" />
  <form:form action="${logoutUrl}" method="post" id="logoutForm">
    <input type="submit" value="logout">
  </form:form>
</body>
</html>
```

Teste 1

- Acesse qualquer URL da aplicação e verifique que a autenticação é exigida.
- Realize login e logout com os dois usuários definidos.

Controlando o acesso às URLs

- Classe **SecurityConfig**:

```
...
import
org.springframework.security.config.annotation.web.builders.HttpSecu
rity;
...
public class SecurityConfig extends WebSecurityConfigurerAdapter{
  @Override
  public void configure(HttpSecurity http) throws Exception{
    http.formLogin().and().logout()
      .and().authorizeRequests()
        .antMatchers("/olamundo").anonymous()
        .antMatchers("/calcular").permitAll()
        .antMatchers("/fabricantes*").hasRole("ADMIN")
        .antMatchers("/veiculos*").hasAnyRole("ADMIN", "USER")
        .antMatchers("/*").authenticated();
  }
  ...
```

Controlando o acesso às URLs

- Classe **SecurityConfig**:

```
...
import
org.springframework.securi                                    cu
rity;
...
public class SecurityConfig
  @Override
  public void configure(HttpSecurity http) throws Exception{
    http.formLogin().and().logout()
      .and().authorizeRequests()
        .antMatchers("/olamundo").anonymous()
        .antMatchers("/calcular").permitAll()
        .antMatchers("/fabricantes*").hasRole("ADMIN")
        .antMatchers("/veiculos*").hasAnyRole("ADMIN", "USER")
        .antMatchers("/*").authenticated();
  }
  ...
```

A sobrescrita deste método desabilita algumas configurações padrão. Estas chamadas habilitam o formulário automático de login e a URL de logout.

Controlando o acesso às URLs

- Classe **Security**

```
...
import
org.springframework.se                                    cu
rity;
...
public class SecurityCo
  @Override
  public void configure(HttpSecurity http) throws Exception{
    http.formLogin().and().logout()
      .and().authorizeRequests()
        .antMatchers("/olamundo").anonymous()
        .antMatchers("/calcular").permitAll()
        .antMatchers("/fabricantes*").hasRole("ADMIN")
        .antMatchers("/veiculos*").hasAnyRole("ADMIN", "USER")
        .antMatchers("/*").authenticated();
  }
  ...
```

As declarações possuem uma ordem de prioridade definida pela sequência de declaração (as primeiras declarações se sobrepõem às últimas). No exemplo, o acesso anônimo a **/olamundo** sobrepõe-se ao acesso controlado do padrão **/***. Logo, as restrições mais específicas deve ser declaradas antes das restrições mais genéricas.

Teste 2

- Verifique o acesso às URLs de acordo com cada usuário autenticado.
- Também teste o acesso anônimo.

Personalizando a página de login

- **login.jsp** (1):

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html>
<html>
<head>
  <title>Login</title>
</head>
<body>
  <c:url var="loginUrl" value="/login" />
  <form action="${loginUrl}" method="post">
    <label for="username">Username</label>
    <input type="text" name="username"><br>
    <label for="password">Password</label>
    <input type="password" name="password">
```

Personalizando a página de login

- **login.jsp** (2):

```
    <input type="hidden"
        name="${_csrf.parameterName}"
        value="${_csrf.token}"/>
    <br>
    <button type="submit">Entrar</button>
  </form>
  <c:if test="${param.error != null}">
    <p style="color: red;">
      ${SPRING_SECURITY_LAST_EXCEPTION.message}
    </p>
  </c:if>
</body>
</html>
```

Personalizando a página de login

- Classe **SecurityConfig**:

```
@Override
public void configure(HttpSecurity http) throws Exception{
  http
    .formLogin()
      .loginPage("/login.jsp")
      .permitAll()
      .loginProcessingUrl("/login")
    .and().logout()
    .and().authorizeRequests()
      .antMatchers("/resources/**").permitAll()
      .antMatchers("/fabricantes*").hasRole("ADMIN")
      .antMatchers("/veiculos*").hasAnyRole("ADMIN", "USER")
      .antMatchers("/*").authenticated();
}
```

Personalizando a página de login

- Classe **SecurityConfig**:

```
@Override
public void configure(HttpSecurity http) throws Exception{
  http
    .formLogin()
      .loginPage("/login.jsp")
      .permitAll()
      .loginProcessingUrl("/login")
    .and().logout()
    .and().authorizeRequests()
      .antMatchers("/resources/**").permitAll()
      .antMatchers("/fabricantes*").hasRole("ADMIN")
      .antMatchers("/veiculos*").hasAnyRole("ADMIN", "USER")
      .antMatchers("/*").authenticated();
}
```

URL para a qual o formulário de login deve ser submetido.

Gerenciador e provedores de autenticação

- O provedor de autenticação é o objeto que verifica a autenticidade das credenciais de um usuário.
- O gerenciador de autenticação é o componente responsável por receber as credenciais de um usuário e verificar sua autenticidade junto aos provedores de autenticação registrados. Basta que um provedor confirme a validade das credenciais para que o usuário seja autenticado pelo Spring Security.
- O Spring Security disponibiliza provedores de autenticação para diversas situações de armazenamento ou uso de credenciais: memória, banco de dados relacionais, LDAP, redes sociais, OpenId, entre outros.
  - Um provedor de autenticação é uma classe que implementa a interface **AuthenticationProvider**.

Criando um provedor de autenticação

- Principais interfaces:
  - **GrantedAuthority**: representa uma permissão (role) do sistema.
  - **Authentication**: armazena os dados de autenticação do usuário, incluindo sua lista de permissões.
  - **AuthenticationProvider**: valida as credenciais de usuário.

Criando um provedor de autenticação

- Classe **Usuario**:

```
package cursoSpring.revenda_veiculos.dominio;

public class Usuario {
  private String login;
  private String nome;
  public Usuario() {}
  public Usuario(String login, String nome) {
    this.login = login;
    this.nome = nome;
  }
  //getters and setters
}
```

Criando um provedor de autenticação

- Classe **AppAuthenticationProvider** (1):

```
package cursoSpring.revenda_veiculos.infra;

import java.util.ArrayList;
import java.util.List;
import
org.springframework.security.authentication.AuthenticationProvider;
import
org.springframework.security.authentication.UsernamePasswordAuthenti
cationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.core.GrantedAuthority;
import
org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.stereotype.Component;
import cursoSpring.revenda_veiculos.dominio.Usuario;
```

Criando um provedor de autenticação

- Classe **AppAuthenticationProvider** (2):

```
@Component
public class AppAuthenticationProvider
implements AuthenticationProvider{

  @Override
  public boolean supports(Class<?> clazz) {
    return UsernamePasswordAuthenticationToken
              .class.isAssignableFrom(clazz);
  }
```

Criando um provedor de autenticação

- Classe **AppAuthenticationProvider** (2):

```
@Component
public class AppAuthenticationProvider
implements AuthenticationProvider{

  @Override
  public boolean supports(Class<?> clazz) {
    return UsernamePasswordAuthenticationToken
              .class.isAssignableFrom(clazz);
  }
```

O método **supports** é utilizado para indicar os tipos de objetos de autenticação suportados pelo provedor. Spring Security fornece algumas classes destes objetos, sendo **UsernamePasswordAuthenteicationToken** a mais comum por representar uma autenticação por login e senha.

Criando um provedor de autenticação

- Classe **AppAuthenticationProvider** (3):

```
@Override
public Authentication authenticate(Authentication auth)
throws AuthenticationException {
  String username = auth.getName();
  String password = auth.getCredentials().toString();
  if(!"senha".equals(password))
    return null;
  List<GrantedAuthority> roles = new ArrayList<>();
  Usuario usuario = null;
  if("admin".equals(username)){
    usuario = new Usuario(username, "ALEX FABIANO");
    roles.add(new SimpleGrantedAuthority("ROLE_ADMIN"));
  }
  else if("user".equals(username)){
    usuario = new Usuario(username, "ANA MARIA");
    roles.add(new SimpleGrantedAuthority("ROLE_USER"));
  }
```

Criando um provedor de autenticação

- Classe **AppAuthenticationProvider** (3):

```
@Override
public Authentication authenticate(Authentication auth)
throws AuthenticationException {
  String username = auth.getName();
  String password = auth.getCredentials().toString();
  if(!"senha".equals(password))
    return null;
  List<GrantedAuthority> roles = new ArrayList<>();
  Usuario usu
  if("admin"
    usuario
    roles.ad
  }
  else if("u
    usuario
    roles.add
  }
```

O método **authenticate** é utilizado para validar as credenciais do usuário. Quando as credenciais são válidas, o método deve retornar uma instância de **Authentication** preenchidas com as credenciais do usuário e sua lista de permissões. Caso contrário, deve retornar **null** ou uma **AuthenticationException** para indicar que a autenticação falhou.

Criando um provedor de autenticação

- Classe **AppAuthenticationProvider** (3):

```
@Override
public Authentication authent
throws AuthenticationExceptio
  String username = auth.getN
  String password = auth.getC
  if(!"senha".equals(password
    return null;
  List<GrantedAuthority> roles = new ArrayList<>();
  Usuario usuario = null;
  if("admin".equals(username)){
    usuario = new Usuario(username, "ALEX FABIANO");
    roles.add(new SimpleGrantedAuthority("ROLE_ADMIN"));
  }
  else if("user".equals(username)){
    usuario = new Usuario(username, "ANA MARIA");
    roles.add(new SimpleGrantedAuthority("ROLE_USER"));
  }
```

O objeto retornado pelo método pode encapsular um objeto específico da aplicação contendo os dados do usuário. No nosso exemplo, **usuario** é criado com esta finalidade.

Criando um provedor de autenticação

- Classe **AppAuthenticationProvider** (4):

```
    else
      return null;
    UsernamePasswordAuthenticationToken principal =
      new UsernamePasswordAuthenticationToken(username, password,
roles);
    principal.setDetails(usuario);
    return principal;
  }
}
```

Criando um provedor de autenticação

- Classe **SecurityConfig**:

```
...
import cursoSpring.revenda_veiculos.infra.AppAuthenticationProvider;

@Configuration
@EnableWebMvcSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter{
  ...
  @Autowired
  private AppAuthenticationProvider authenticationProvider;

  @Override
  public void configure(AuthenticationManagerBuilder auth)
  throws Exception{
    auth.authenticationProvider(authenticationProvider);
  }
}
```

Criando um provedor de autenticação

- Classe **AppConfig**:

```
...
import cursoSpring.revenda_veiculos.infra.AppAuthenticationProvider;

@Configuration
@ComponentScan(basePackageClasses={FabricanteDAO.class,
CompraService.class, AppAuthenticationProvider.class})
@EnableTransactionManagement(proxyTargetClass=true)
public class AppConfig {
  ...
}
```

Teste 3

- Novamente, acesse a aplicação experimentando os diversos usuários e URLs.

Taglibs do Spring Security

- Prefixo:
  - http://www.springframework.org/security/tags
- Tag **authorize**:
  - Utilizada para renderização condicional, a qual poder ser determinada a partir das permissões dos usuário ou a partir de uma URL.
- Tag **authentication**:
  - Tag de acesso ao objeto **Authentication** retornado pelo provedor de autenticação.

Taglibs do Spring Security

- **index.jsp**:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://www.springframework.org/tags/form"
prefix="form" %>
<%@ taglib uri="http://www.springframework.org/security/tags"
prefix="sec" %>
<html>
<body>
  <h2>Hello World!</h2>
  Olá <sec:authentication property="details.nome"/><br>
  <sec:authorize ifAnyGranted="ROLE_ADMIN">
    <c:url var="fabricantesUrl" value="/fabricantes" />
    <a href="${fabircantesUrl}">Fabricantes</a><br>
  </sec:authorize>
  ...
</body>
</html>
```

Taglibs do Spring Security

- **index.jsp**:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://www.spri
prefix="form" %>
<%@ taglib uri="http://www.spri
prefix="sec" %>
<html>
<body>
  <h2>Hello World!</h2>
  Olá <sec:authentication property="details.nome"/><br>
  <sec:authorize ifAnyGranted="ROLE_ADMIN">
    <c:url var="fabricantesUrl" value="/fabricantes" />
    <a href="${fabircantesUrl}">Fabricantes</a><br>
  </sec:authorize>
  ...
</body>
</html>
```

Também poderia utilizar **<sec:authorize url="/fabricantes">**. Neste caso a renderização ocorreria com base nas permissões definidas para a URL.

Restringindo execução de métodos

- Spring Security oferece a anotação **@Secure** para controlar a execução de métodos.
- Para que o recurso funcione, é necessária sua habilitação e que o objeto anotado com **@Secure** seja um bean Spring.

Restringindo execução de métodos

- Classe **SecurityConf**:

```
...
import
org.springframework.security.config.annotation.method.configuration.
EnableGlobalMethodSecurity;

@Configuration
@EnableWebMvcSecurity
@EnableGlobalMethodSecurity(securedEnabled=true)
public class SecurityConfig extends WebSecurityConfigurerAdapter{
  ...
  @Bean @Override
  public AuthenticationManager authenticationManagerBean()
  throws Exception {
    return super.authenticationManagerBean();
  }
}
```

Restringindo execução de métodos

- Classe **VeiculoDAO**:

```
...
import org.springframework.security.access.annotation.Secured;

@Repository
@Transactional
public class VeiculoDAO implements VeiculoRepositorio {
  ...
  @Override
  @Secured("ROLE_ADMIN")
  public List<Veiculo> todos() {
    Session session = sessionFactory.getCurrentSession();
    return session.createQuery("from Veiculo").list();
  }
  ...
}
```

Referências

- Alex, Ben; Taylor, Luke; Winch, Rob. **Spring Security Reference**. Disponível em <http://docs.spring.io/spring-security/site/docs/3.2.7.RELEASE/reference/htmlsingle/#authorize-requests>.
- Deinum, Marten et al. **Spring recipes: a problem solution aproach**. 3ª ed. Apress, 2014.
- Johnson, Rod et al. **Spring Framework Reference Documentation**, 4.2.1 release. Disponível em <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/>.
- Macedo, José A.. **Spring Security 3: parte 2**. Revista Java Magazine 88, 2010.

Referências

- Mkyong.com. **Spring Security Tutorial**. Disponível em <http://www.mkyong.com/tutorials/spring-security-tutorials/>
- Souza, Alberto. **Spring MVC:** domine o principal framework web Java. São Paulo: Casa do Código, 2015.
- Weissmann, Henrique L. **Vire o jogo com Spring Framework**. São Paulo: Casa do Código, 2013.
- Winch, Rob. **Spring Security guides**. Disponível em <http://docs.spring.io/spring-security/site/docs/3.2.x/guides/index.html>.
- Zanini, Michel. **Spring Security**. Revista Java Magazine 69, 2009.

Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte

Campus Natal Central

Diretoria Acadêmica de Gestão e Tecnologia da Informação

Curso de formação em Spring Framework 4

Parte 05 – Spring Security

Autor: Alexandre Gomes de Lima

Natal, outubro de 2015.