Seção 02 - Resumo

Spring Security

Quando desejamos começar a trabalhar com qualquer recurso da família Spring, o ideal é ter sempre como uma especie de livro de bolso, a documentação oficial. Por conta disso, sempre busque por informações sobre o Spring Security no guia de referencia, que lá você encontrará tudo que o Spring Security oferece.

Para começar a utilizar o Spring Security junto ao Spring Boot, devemos adicionar no arquivo *pom.xml* a dependência referente ao starter do Spring Security, conforme o exemplo a seguir:

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

Com esse starter o Spring Boot já habilita algumas configurações iniciais e sua aplicação já estará protegida. Por conta disso, a única página a qual terá acesso será a de login. Para que possa acessar as demais páginas de sua aplicação você pode usar um login e senha fornecidos pelo Spring Boot na inicialização da aplicação. O login é `user` e a senha deve ser encontrada no log do console, conforme a figura a seguir:

Log no console com a senha para login user.

Quando o Spring Security é configurado, toda a aplicação passa a ser bloqueada. A configuração inicial, ou seja, aquela básica fornecida pelo Spring Boot vai apenas bloquear o acesso as páginas, mas uma configuração realizada por você vai mais além, bloqueando página e recursos estáticos como arquivos de imagens, css e js.

CSRF

Quando trabalhamos com o Spring Security junto a uma aplicação baseada em páginas web o sistema de segurança vai por padrão adicionar na página de login um token de proteção do tipo Cross Site Request Forgery(CSRF). Esse token tem como objetivo garantir

que a requisição de autenticação está sendo realizada por um navegador que esteja realmente conectado ao contexto da aplicação. Isso vai impedir, por exemplo, que aplicações baseadas em bots consigam logar na aplicação.
Quando a página de login abrir no navegador, você pode visualizar o código fonte e procurar por um `input` do tipo `hidden` com o `name="_csrf"`, como o exemplo a seguir:

```
<input type="hidden"
    name="_csrf"
    value="af50-5065-age6-er06-605r"/>
```

O código no atributo `value` será automaticamente gerado e quando o submit do formulário de login chegar ao lado servidor o Spring Security vai verificar se esse código corresponde ao código que ele gerou e só em caso positivo a verificação de login será realizada. Caso contrário, um exceção será lançada.

A geração do código pode ser desabilitada se assim desejar, para isso, na classe de configuração do Spring Security que vamos ver mais a frente, use no corpo do método `configure()` a instrução:

```
http.csrf().disable();
```

Não é recomendada a ação de desabilitar essa instrução, a não ser que sua aplicação seja do tipo RESTful.

Classe de configuração

Como citado, o Spring Boot auto configura de forma básica o Spring Security. Porém, essa configuração não é suficiente para um trabalho avançado de segurança. Por isso, devemos sempre criar nossa própria classe de configuração e sobrescrever a configuração padrão.

A sua classe de configuração pode ter qualquer nome, mas deve ser anotada com `@EnableWebSecurity`, para que o Spring Security identifique a classe. Além disso, é preciso estender a classe `WebSecurityConfigurerAdapter`. A partir dessa classe vamos sobrescrever o método `configure()` para incluir nossas regras de autenticação e autorização.

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

        @Override
    protected void configure(HttpSecurity http) throws Exception {
        // regras de segurança
    }
 }
```

Regras de acesso

No método `configure()` usamos a variável `http`, de `HttpSecurity`, para definir as regras de acessos. Entre essas regras devemos tornar público o acesso aos arquivos estáticos da aplicação. Como vimos em aula, esses arquivos acabam sendo bloqueados pelo Spring Security quando iniciamos nossa própria configuração.

Para realizar o desbloqueio trabalhe com o método `antMatchers()`, no qual adicionamos uma lista de URIs que dão acesso a tais arquivos. A partir da variável `http` teremos acesso acesso ao método `authorizeRequests()`, o qual, seu próprio nome já deixa claro que estaremos autorizando requisições e a partir dele vamos acessar o `antMatchers()`.

Além disso, muitas vezes é necessário liberar publicamente não apenas os arquivos estáticos, mas algumas página que não necessitam autenticação, como a página home ou index da aplicação. Para liberar essas páginas públicas, também usamos o `antMatchers()`.

Mas para que o acesso público seja realmente liberado, o `antMatchers()` deve ser seguido pelo método `permitAll()`, como no exemplo a seguir:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                // acessos públicos liberados
                .antMatchers("/webjars/**", "/css/**", "/image/**", "/js/**").permitAll()
                .antMatchers("/", "/home").permitAll()
                .anyRequest().authenticated()
}
```

A configuração deve ser encerrada pelos métodos `anyRequest().authenticated()`. Esses métodos informam que qualquer solicitação a aplicação deve estar autenticada, a menos é claro, aquelas que foram liberadas como públicas.

Por fim, vamos liberar e indicar como o acesso deve ser realizado para o processo de login. Para isso, analise o método as instruções a seguir:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                // acessos públicos liberados
                .antMatchers("/webjars/**", "/css/**", "/image/**", "/js/**").permitAll()
                .antMatchers("/", "/home").permitAll()
                .anyRequest().authenticated()
                        .and() (1)
                                .formLogin() (2)
                                .loginPage("/login") (3)
                                .defaultSuccessUrl("/", true) (4)
                                .failureUrl("/login-error") (5)
                                .permitAll() (6)
                        .and() (7)
                                .logout() (8)
                                .logoutSuccessUrl("/"); (9)
}
```

Após o `anyRequest().authenticated()` usamos o método `and()` para incluir uma nova configuração. Essa configuração será referente ao login:

1. Método usado para iniciar uma nova etapa de configurações;
2. O método indica que a configuração será para o processo de login;
3. Método usado para indicar qual a URI de acesso ao login. Essa URI deve ser a `action` do formulário de autenticação;
4. Instrução usada para informar o caminho para onde a aplicação deve seguir após o login ser validado;
5. Em caso de falha, ou seja, o login foi recusado indicamos para onde a aplicação deve ser redirecionada;
6. O processo de login é finalizado com o `permitAll()` para que ele seja público.

Em seguida, vamos configurar o processo de logout:

1. Como o logout é uma nova configuração, deve vir após o uso do método `and()`;
2. Indicamos que será configurada a etapa de logout;
3. Método que indica para qual página a aplicação será redirecionada após o usuário logado solicitar o logout.

Para concluir a configuração de login vamos incluir um método que abra a página de login. Este método deve estar presente em um controller qualquer:

```
@GetMapping("/login")
public String login() {
   return "login";
}
```

Ao clicar no link de login da página, a requisição será recebida pelo método `login()` e como resposta será aberta a página de *login.html*.

Na etapa de login ainda temos a possibilidade das credenciais estarem invalidas. Nesse caso, o Spring Security vai direcionar a aplicação para o método referente a URI: `/login-error`, conforme definida na configuração da etapa de login. O método será similar ao descrito a seguir:

```
@GetMapping("/login-error")
public String loginError(ModelMap model) {
    model.addAttribute("mensagem", "Login Inválido");
    return "login";
}
```

O método `loginError()` vai receber a requisição do Spring Security e uma mensagem pode ser enviada para a página de login para informar ao usuário que suas credenciais são invalidas.

Página de login

A página de login deve ter algumas características que precisam ser seguidas. Entre elas, o método do formulário deve ser do tipo `POST` e action deve ser setada com a URI indicada no método `loginPage()` da etapa de login.

```
<form th:action="@{/login}" method="post">
        <p th:if="${mensagem != null}" th:text="${mensagem}"/>
    <p>
        <label for="username">Username</label>
        <input type="text" id="username" name="username"/>
    </p>
    <p>
        <label for="password">Password</label>
        <input type="password" id="password" name="password"/>
    </p>
    <button type="submit" class="btn">Log in</button>
</form>
```

Outro ponto importante são os atributos `name` dos componentes de `input` para nome de usuário e senha. Por padrão o Spring Security vai procurar na requisição por `username` para nome de usuário e `password` para senha. Caso tenha desejo de alterar o `name` do componentes, você deve informa por quais variáveis o Spring Security deve procurar na requisição. Isso pode ser feito na etapa de login, setando os métodos `usernameParameter()` e `passwordParameter()` com os nomes de cada `name` informados como parâmetros nos respectivos métodos.

```
.usernameParameter("web_login")
.passwordParameter("web_senha")
```

Referencias

- Spring Security - https://spring.io/projects/spring-security#learn

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.