EST – IPCB

LICENCIATURA EM ENGENHARIA INFORMÁTICA

Aplicações Distribuídas

A minha primeira Aplicação Spring Boot + Spring Security + Thymeleaf

3º Ano / 1º Semestre – 2022/2023

Atividade Prática n.º7 - A Minha Primeira Aplicação Spring Boot Security + Thymeleaf

Ao longo desta atividade, pretende-se que ao realizar os passos seguintes experimente diversas funcionalidades do Spring Security e possa ficar a conhecer como blindar uma Aplicação Spring Boot MVC, incluindo extras de Segurança do Thymeleaf.

Para melhor compreensão dos conteúdos deste guião sugere-se a consulta do bloco de slides: AD-2023-Modulo-5.3-Spring-Security.pdf.

Durante a realização desta atividade serão implementados pequenos projetos, em sequência até ao caso de uma implementação mais completa e personalizada.

Caso 1: Spring Boot security por Omissão (Default Login Form) usando Credenciais por Omissão.

Clone o diretório do projeto DemoMVC-Thymeleaf da Atividade PL n.º2 e dê-lhe o nome **DemoMVC-Thymeleaf-Spring-Security-Default-Authentication.**

Passo 1: Adicione ao ficheiro pom.xml o Starter Spring Security.

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

Passo 2: Ative as configurações de segurança anotando a classe principal com **@EnableWebSecurity.** Poderá ser necessário adicionar a dependência maven e realizar o import.

Passo 3: Execute a aplicação e registe as credenciais criadas por emissão.

Passo 4: Verificar no application.properties a porta utilizada, aceder à aplicação em **http://localhost** e preencher as credenciais utilizador **user** + password no Login Form auto-gerado pelo Spring Boot.

Se o pedido não enviar as credenciais o Dispatcher Controller devolve 401 not allowed

Prosseguir para o Caso 2 (próxima página)

Caso 2: Spring Boot security Basic Authentication + Default Popup Login Form usando Credenciais de utilizador guardadas diretamente no ficheiro application.properties.

Clone o diretório do projeto do caso 1 e dê-lhe o nome DemoMVC-Thymeleaf-Spring-Security-Basic-Authentication, remova a anotação da @EnableWebSecurity.

Passo 1: Criar uma classe chamada **ConfiguracoesSeguranca** anotada com @EnableWebSecurity que estende a classe abstracta WebSecurityConfigurerAdapter com as Configurações de Segurança.

```
@Configuration
@EnableWebSecurity // (1)
public class ConfiguracoesSeguranca extends
WebSecurityConfigurerAdapter { // (1)

 }
```

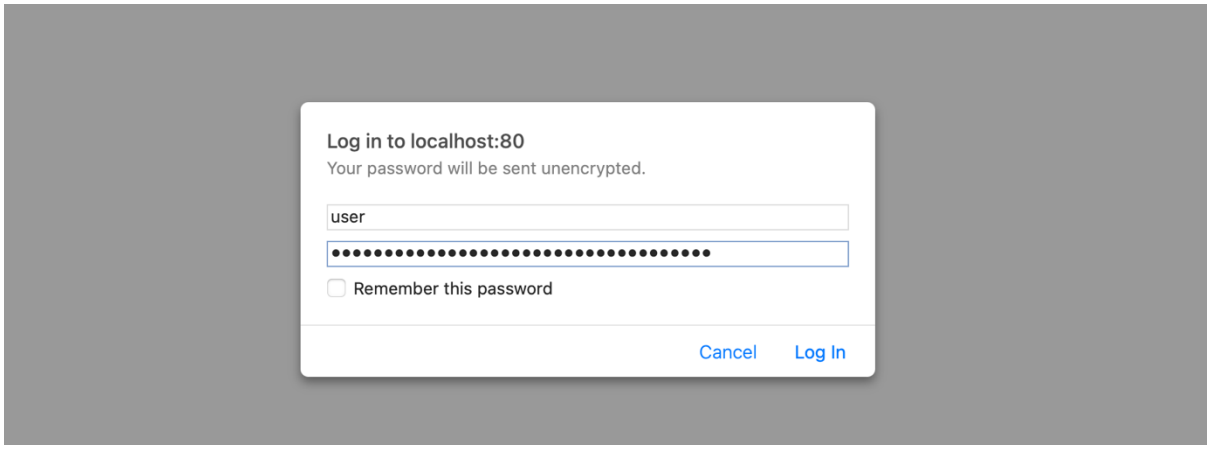
Passo 2: Sobrepor o método **configure()** que permite configurar a autorização dos pedidos http à aplicação.

Passo 3: Configure a segurança Http (HttpSecurity) de autorização solicitando que todos os pedidos sejam autenticados e (and) se aplique autenticação básica usando o método httpBasic().

Exemplo Ilustrativo:

```
@Configuration
@EnableWebSecurity // (1)
public class ConfiguracoesSeguranca extends WebSecurityConfigurerAdapter {
// (1)
@Override
  protected void configure(HttpSecurity http) throws Exception {// (2)
     http.authorizeRequests() //(3)
           .anyRequest().authenticated()
           .and()
           .httpBasic();
  }
}
```

Passo 4: Execute e testar a aplicação. Registe as credenciais criadas por omissão e introduza-as no popup Login Form.



Passo 5: Finalmente, experimente personalizar as credenciais preenchendo no ficheiro **application.properties** as seguinte configuração:

```
spring.security.user.name=admin
spring.security.user.password=admin
```

Prosseguir para o Caso 3 (próxima página)

Caso 3: Spring Boot Security Basic Authentication usando Autenticação in-memory e Default Login Form.

Autenticação in-memory é o mecanismo de manter as user credentials na memória da JVM. *If you are trying to test something in spring boot or building some kind of proof of concept then usually in-memory authentication is used.*

Clone o diretório do projeto do caso anterior e dê-lhe o nome **DemoMVC-Thymeleaf-Spring-Security-Basic-Authentication-In-memory-credentials.**

Passos 1 a 3 (tal como antes): Criar a classe anotada com **@EnableWebSecurity** com as Configurações de Segurança de que todos os pedidos Http têm que ser autenticados (HttpSecurity).

Passo 5: Na classe das Configurações de Segurança configurar as autenticações, sobrepondo o método **configure()** com um **AuthenticationManagerBuilder** e defina as credenciais in-memory e a função do utilizador.

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
throws Exception {
   auth.inMemoryAuthentication()
        .withUser("admin")
            .password("admin")
            .roles("ADMIN")
        .and()
        .withUser("utilizador")
            .password("123456")
            .roles("USER")
        ;
}
```

É importante referir que o Spring Security neste caso sobrepõe automaticamente as funções ADMIN→ ROLE_ADMIN e USER→ROLE_USER.

Passo 6: Executar e testar a aplicação, introduzindo as credenciais admin:admin ou utilizador:123456. Analise o *output*.

O Spring Security não permite o Log in e retorna uma exceção indicando que foi realizado um acesso Ilegal.

Uma vez que vamos utilizar a autenticação in-memory. Recomenda-se a utilização de um codificador de palavras-chave para guardar as palavras-chave na memória ou na base de dados. O Spring Security dá suporte a um dos codificadores mais recomendados: BCryptPasswordEncoder.

Passo 7: Defina um método Bean chamado "**codificador.bcrypt**" que retorna um Bean do tipo **BCryptPasswordEncoder**. Depois, inclua-o na implementação do passo 5 invocando este método durante a parametrização da password para criar o Bean e invocando o método **encode()** para codificar a password

Passo 6: Executar e testar NOVAMENTE a aplicação.
Prosseguir para o Caso 4 (próxima página)

Caso 4: Spring Boot Security usando Autenticação in-memory e Login Form Personalizado.

Clone o diretório do projeto do caso anterior e dê-lhe o nome **DemoMVC-Thymeleaf-Spring-Security-Basic-Authentication-In-memory-credentials-mylogin-form.**

Neste caso a **página Login é criada inteiramente pelo Developer e personalizada para as suas necessidades**

Passos 1 a 7: Mantém-se iguais ao do caso anterior.

Passo 8: Criar um Login Form usando a Thymeleaf com CSRF Token.

A Spring Security permite a proteção CSRF por defeito, contudo é preciso submeter o formulário utilizando o método POST.

O token CSRF será incluído em runtime pela Thymeleaf devido à anotação @EnableWebSecurity.

- Para realizar este passo pode duplicar o ficheiro formregistar.html para login.html e adaptá-lo por forma a manter apenas os inputs nome e password do utilizador.

- Mude a ação para "/login".

- Depois inclua as **divs** (a amarelo) para capturar as mensagens de alerta/erro.

Exemplo de Formulário Login Personalizado

```
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <title> Spring Security + Thymeleaf Integration Example
</title>
    </head>
    <body>
        <legend>Please Login here</legend>

    <!-- Check for login error -->
    <div th:if="${param.error}">
        <div class="alert alert-danger col-xs-offset-1 col-xs-10">
            Invalid username and password.
        </div>
    </div>
    <!-- Check for logout -->
    <div th:if="${param.logout}">
```

```
        <div class="alert alert-success col-xs-offset-1 col-xs-10">
            You have been logged out.
        </div>
    </div>

        <form th:action="@{/login}" method="POST">
            Nome Utilizador: <input type="text" name="username"/>
<br/><br/>
            Password: <input type="password" name="password"/>
<br/><br/>
            <input type="submit" value="Login"/>
        </form>
    </body>
</html>
```

Passo 9: Criar um Controlador de Login que fornece os mapeamentos para as páginas home ("/") e /login. Remova do ControladorUtilizador o mapeamento ("/")

Exemplo de Controlador

```
@Controller
public class ControladorLogin {

  @GetMapping(value={"/login","/"})
  public ModelAndView login() {
     return new ModelAndView("login");
  }
}
```

Passo 10: Na classe ConfiguracoesSeguranca fazer as configurações de segurança por forma a mapear a página login.

As politicas de segurança são:

- Permitir que todos os utilizadores atinjam as páginas "/" e "/login
- Após o utilizador ser autenticado é carregada a página que lista todos os utilizadores "/listar"
- Permitir que todos os utilizadores façam "/logout"

```
@Configuration
@EnableWebSecurity
public class ConfiguracoesSeguranca extends
WebSecurityConfigurerAdapter {

  @Override
  protected void configure(HttpSecurity http) throws Exception
{
        http.authorizeRequests()
             .antMatchers("/","/login").permitAll()
             .anyRequest().authenticated()
             .and()
```

```
                .formLogin().loginPage("/login").permitAll()
                    .defaultSuccessUrl("/listar")
                .and()
                .logout().logoutUrl("/logout").permitAll()
                .and()
                .httpBasic();  }
}
```

Passo 11: Incluir na página de listar-utilizadores.html um botão para logout, bem como uma div que indique quem é neste momento o utilizador logado.

```
<div>
    Logged in user: <b th:text="${principal.name}">nome
utilizador</b>
</div>
<a class="btn btn-primary" href="/logout"> Logout</a>
```

Para passar esta informação à vista listar-utilizadores.html, é preciso definir um parâmetro do tipo Principal no método listar do ControladorUtilizador mapeado com `@GetMapping("/listar")` e passá-lo ao Model como um atributo.

Passo 12: Executar a aplicação e testar.

Provavelmente, chegou novamente à página de Login, mas sem qualquer feedback do que ocorreu.

Passo 13: Passe o alerta de logout à página de login/logout.

Basta incluir o seguinte na configuração de logout:

```
/* Linha para enviar mensagem ao param.logout */
.logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
```

Passo 14: Neste passo final, vamos definir duas áreas reservadas e duas áreas permitidas a qualquer tipo de utilizador:

- Registar utilizador – permitida a apenas a utilizadores com a função ROLE_ADMIN
- Remover utilizador – permitida a apenas a utilizadores com a função ROLE_ADMIN
- Editar – acessível a qualquer utilizador (AnyRole)
- Listar - acess(AnyRole)

Para realizar este passo precisa de definir permissões do género:

```
.antMatchers("formregistar")
```

```
                .hasRole("ADMIN")
```

Passo 15: Para dar melhor feedback crie uma página acesso-negado.html, com a mensagem e dois botões para voltar a listar ou fazer logout.

```
<hr>
<h2>Access Denied - You are not authorized to access this
resource.</h2>
<hr>

<a th:href="@{/listar}">Listar Todos os Utilizados</a>
<a class="btn btn-primary" href="/logout"> Logout</a>
```

Passo 16: Para configurar o redireccionamento faça o seguinte:

- No parâmetro Httpsecurity adicione o redireccionamento em caso de acesso negado para /acesso-negado.

```
.exceptionHandling().accessDeniedPage("/accesso-negado")
```

- Adicione ao ControladorLogin um método que retorna a página acesso-negado.

```
@GetMapping("/acesso-negado")
public String showAccessDenied() {
   return "acesso-negado";
}
```

Prosseguir para o Caso 5 (próxima página)

Caso 5: Spring Boot Security usando Autenticação com uma Base de Dados e Login Form Personalizado.

Neste caso para substituição da autenticação in-memory podem ser realizados os seguintes passos adicionais:

Passo 1: Adicionar os starter da Spring Data JPA e da Base de dados H2, e definir no application.properties a ligação à BD.

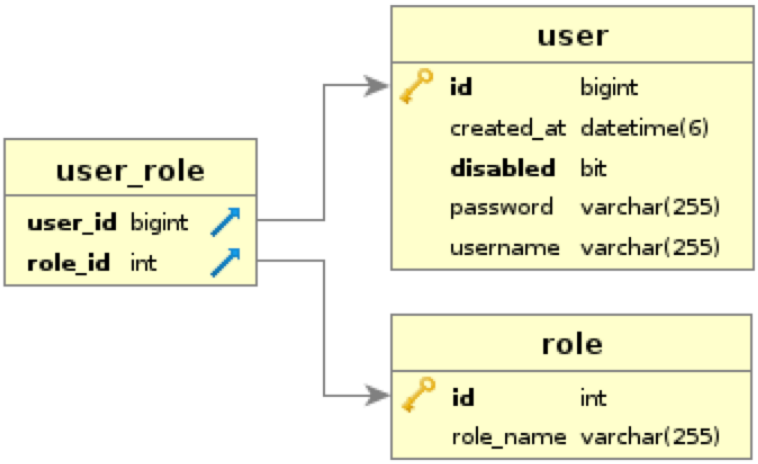
```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
	<groupId>com.h2database</groupId>
	<artifactId>h2</artifactId>
	<scope>runtime</scope>
</dependency>
```

```
//Application.properties
spring.datasource.url=jdbc:h2:mem:testedb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=admin
spring.datasource.password=admin
spring.datasource.initialization-mode=always
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
```

Passo 2: Considere o modelo de dados e o relacionamento Many-to-Many entre a entidade User (Utilizador) e a entidade Role (função).

Crie a classe entidade Utilizador (User) – no nosso caso já está criada e anote-a devidamente. Inclua os campos adicionais indicados.



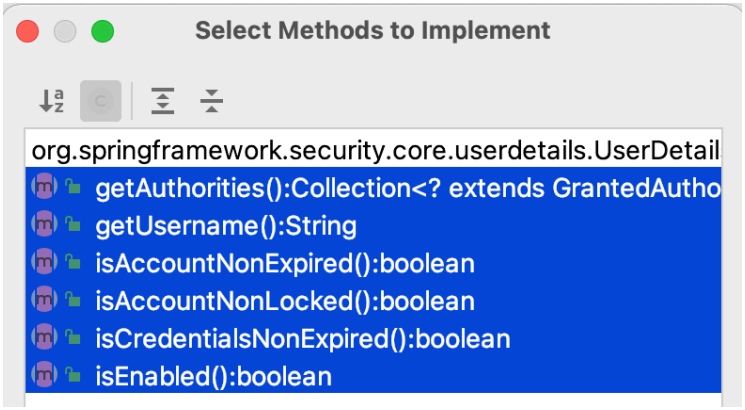
Exemplo:

```
@Entity
@Data
public class Utilizador {
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Id
  private Long id;
  private String username;
  private String password;
  private String email;
  private boolean disabled = false;
  private LocalDateTime createdAt = LocalDateTime.now();

…}
```

Passo 3: A classe Utilizador para poder guardar na Base de Dados as passwords e a Spring Security as consultar durante a autenticação precisa de *extender* a Classe UserDetails:

- Extenda a classe UserDetails
- De seguida precisa de fazer o override e implementação dos métodos da UserDetails:



Passo 4: Para testes iniciais implemente o método getAuthorities() por enquanto assumindo que ainda NÃO fizemos o relacionamento com a entidade Role. Todos os utilizadores serão terão a função USER ou ADMIN.

```
Basta que este retorne:
Collections.<GrantedAuthority>singletonList(new
SimpleGrantedAuthority("USER"));

Ou

Collections.<GrantedAuthority>singletonList(new
SimpleGrantedAuthority("ADMIN"));
```

Passo 5: Crie o repositório de dados Jpa para a entidade Utilizador e defina um método Query que procura o utilizador pelo nome.

Passo 6: Crie o serviço de segurança ServicoUserDetails que implementa a interface UserDetailsService.

- Faça o *override* e implemente o método **loadUserByUsername()** da interface **UserDetailsService;**
- Este método deverá retornar o utilizador procurado pelo nome. No caso de o utilizar não ser encontrado será retornada uma excepção **UsernameNotFoundException**.

Passo 7: Na classe de Configurações de segurança desative, comentando, o método **configure()** com as configurações in-memory e faça uma nova implementação que utiliza o **ServicoUserDetails** antes criado e Bean CrypB para codificar/descodificar as passwords.

Passo 8: Num ApplicationRunner, crie dois utilizadores invocando o método **save()** do repositório de dados RepositorioUtizador. Não se esqueça de puxar o Bean "`codificador.bcrypt"` e codificar a password.

Teste o acesso à aplicação.

PASSOS FINAIS do caso 5.

Passo 9: Crie a classe entidade Role e defina na classe Utilizador o mapeamento many-to-many:

```
@ManyToMany(fetch = FetchType.EAGER)
  @JoinTable(name = "user_role",
     joinColumns = @JoinColumn(name = "user_id"),
     inverseJoinColumns = @JoinColumn(name = "role_id"))
  private List<Role> roles;
```

Passo 10: De seguida, crie o repositório **RepositorioRole**. Neste defina um método (**findRoleByName**) que permite procurar a Role pelo nome.

Passo 11: Reveja o passo 4, por forma a incluir o relacionamento many-to-many. Basta criar um **List<GrantedAuthority> listaRoles** e passar cada elemento da **List<Role> roles** para esta num SimpleGrantedAuthority:
Exemplo

```
listaRoles.add(new SimpleGrantedAuthority(roles.get(0));
```

Passo 12: Adapte o ApplicationRunner:

- Instancie dois Role com as String Autoridade ROLE_ADMIN e ROLE_USER e grave-os na base de dados.
- Ao utilizador **admin**, atribua ambas as funções ADMIN e USER
- Ao utilizador **utilizador**, atribua a função USER.

Teste o acesso à aplicação.

Melhorias da Integração do Spring Security com a Thymeleaf (adição de extras de segurança)

O Spring Security e a Thymeleaf estão bem integrados, oferendo um "dialeto" que permite a adições de expressões de segurança diretamento nos modelos html Thymeleaf.

Passo 1: Adicione ao pom.xml do projecto a dependência dos extras de segurança do Thymeleaf.

```
<dependency>
      <groupId>org.thymeleaf.extras</groupId>
      <artifactId>thymeleaf-extras-springsecurity5</artifactId>
</dependency>
```

Em todos os templates html adicione no elemento htlm, o namespace: **xmlns:sec=http://www.thymeleaf.org/thymeleaf-extras-springsecurity5**

Exemplos:

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

Logged user: <span sec:authentication="name">Bob</span>
Roles: <span sec:authentication="principal.authorities">[ROLE_USER,
ROLE_ADMIN]</span>
```

Passo 2: Adicione às Templates html a div seguinte para ecoar o nome do ultilizador e a(s) sua(s) função(ões):

```
<div>
Logged User: <b sec:authentication="name"></b>
<br>
Role(s): <b sec:authentication="principal.authorities"></b>
<br>
</div>
```

Passo 3: Experimente agora os dois casos seguintes:

- O botão "registar novo utilizador" deve ficar acessível apenas a utilizador com o ROLE_ADMIN usando o atributo:

  ```
  sec:authorize="hasRole('ROLE_ADMIN')"
  ```

- A seguinte div deve ficar acessível apenas a utilizador com o ROLE_USER usando o atributo:

  ```
  <div sec:authorize="hasRole('ROLE_USER')">

   Um utilizador com a função ROLE_USER tem apenas acesso a algumas opções.
  </div>
  ```

FIM