



Instituto Politécnico de Castelo Branco
Escola Superior de Tecnologia

Spring Security

Unidade Curricular

Aplicações Internet Distribuídas / Aplicações Distribuídas

Licenciatura em Engenharia Informática

Mestrado em Engenharia Informática - Desenvolvimento de Software e Sistemas Interactivos

UTC Informática

Est-IPCB

Ano Letivo 2022/2023

Prof.º Doutor Alexandre Fonte
(adf@ipcb.pt)

Versão: 3 novembro de 2022

Sumário

- Spring Security
- REST API security com Spring Security
- Spring Security MVC + Thymeleaf

Declaração de Direitos de Autor

É Proibida a cópia, difusão, ou uso destes materiais não seja no âmbito exclusivo das Unidade Curriculares de Aplicações Internet Distribuídas / Aplicações Distribuídas dos cursos de Mestrado e Licenciatura em Engenharia Informática- IPCB

Principais Vulnerabilidades em Aplicações Web ou API REST

- O documento **10 Top Owasp** apresenta os principais risco de segurança aplicacional (<https://owasp.org/www-project-top-ten>)
- Vulnerabilidades mais comuns identificadas
 - Falhas na Autenticação e Autorização
 - Falhas na Criptografia
 - Cross Site Request Forgery (CSRF)
 - Server-Side Request Forgery (**NOVA**)
 - Problemas nas Configurações de Segurança
 - Utilização de dependências/componentes desatualizados e/com vulnerabilidades conhecidas
 - Falhas Logging e Monitoring

Spring Security (atualmente (2/11/22) na versão 5.7)

- Projecto ecossistema Spring especializado nos mecanismos de segurança (<https://spring.io/projects/spring-security>)

Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications.

Spring Security is a framework that focuses on providing both authentication and authorization to Java applications. Like all Spring projects, the real power of Spring Security is found in how easily it can be extended to meet custom requirements

Oferece:

- Comprehensive and extensible support for both Authentication and Authorization
- Protection against attacks like session fixation, clickjacking, cross site request forgery, etc
- Servlet API integration
- Optional integration with Spring Web MVC

E muito mais..

Obtenção da última versão (5.7), no Spring Boot 2.7.5

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

Spring Security

- De seguida abordaremos os mecanismos mais essenciais a implementar na:
 - Segurança de APIs
 - Autenticação Básica
 - SSL
 - Segurança de Aplicação Web MVC
 - Autenticação Básica in-memory ou base de dados
 - Autorização
 - Proteção contra ataques CSRF
 - Integração Spring Security + Thymeleaf
 - SSL
- Outros mecanismos como autenticação Bearer, Tokens JWS, Message Digest, autorização OAuth, autenticação com LDAP, etc também podem ser equacionados, mas está fora do âmbito desta UC

Autenticação Básica (Basic Auth)

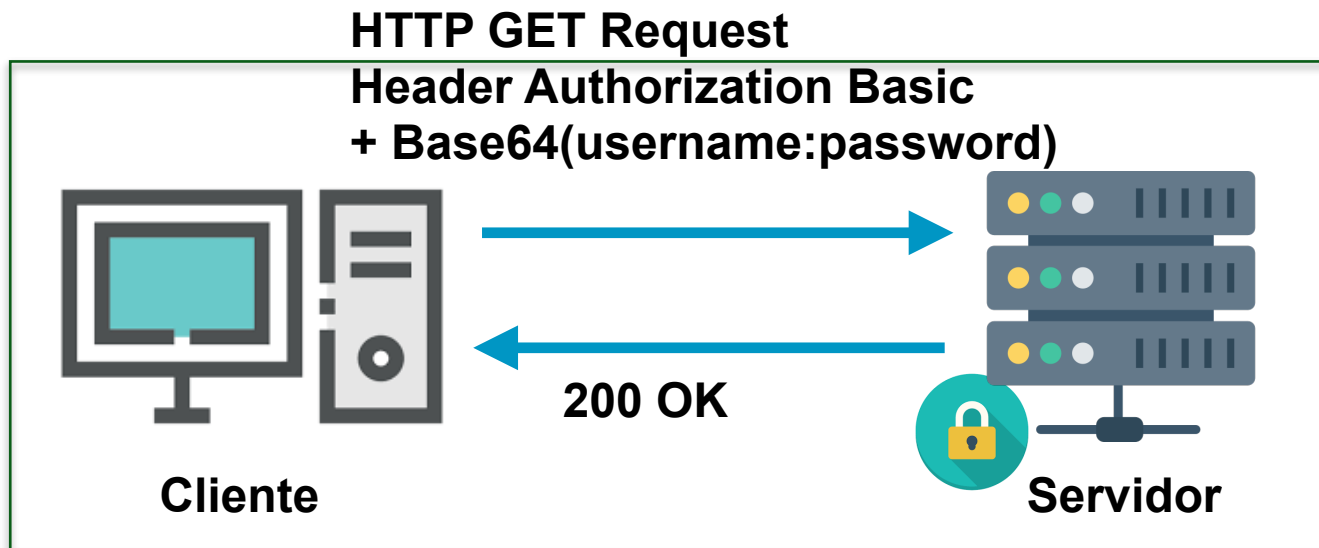
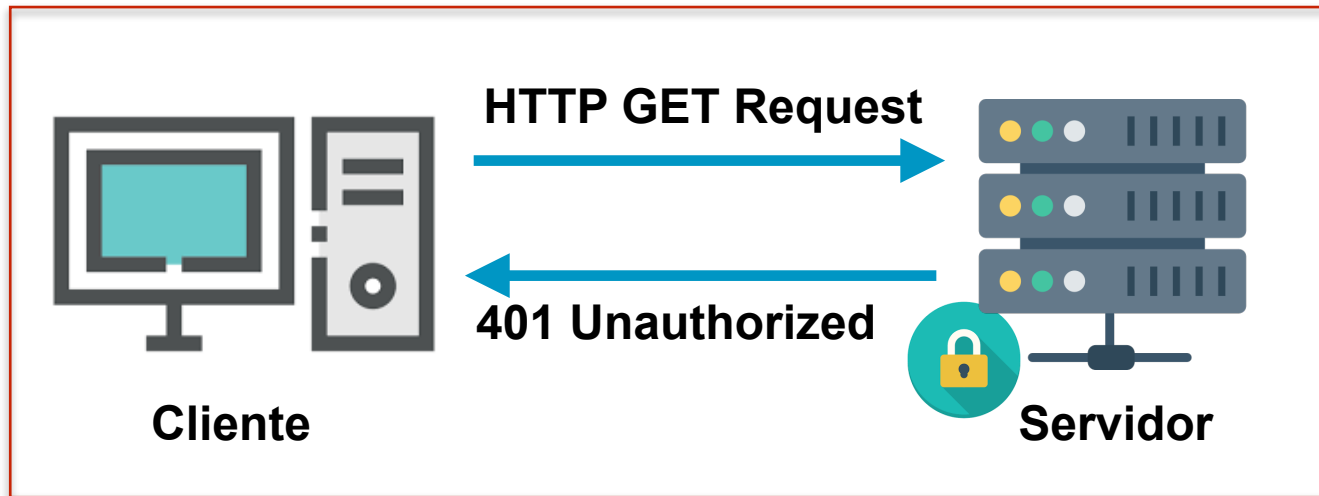
- É um método muito utilizado para autenticação simples: **utilizador/palavra-passe**.
- Este tipo de mecanismo não oferece qualquer protecção de confidencialidade para as credenciais transmitidas.
- A autenticação básica é tipicamente utilizada em concertação com HTTPS para fornecer confidencialidade/segurança.

```
GET /customers/333 HTTP/1.1  
Authorization: Basic YmJlcmtlOmdlaGVpbQ==
```



Credenciais codificadas em Base64

Fluxo Autenticação Básica / Basic Authentication



Autenticação Bearer

- Método Bearer Auth (uso de tokens de segurança)
 - A autenticação ao “portador” (também chamada autenticação token) é um esquema de autenticação HTTP que envolve tokens de segurança.
 - O termo "Autenticação ao portador" pode ser entendido como "dar acesso ao portador deste token".
 - O token permite o acesso a um determinado recurso ou URL.
 - O cliente deve enviar este token no cabeçalho da Autorização ao fazer pedidos a recursos protegidos: `Authorization: Beared <token>`

GET /customers/333 HTTP/1.1

Authorization: Beared YmJlcmtlOmdlaGVpbQ==

- O esquema de autenticação Bearer foi originalmente criado como parte do OAuth 2.0 no RFC-6750, mas por vezes é também utilizado por si só.
- Tal como a autenticação Básica, a autenticação Bearer só deve ser utilizada sobre HTTPS (SSL).

Spring Security - Basic Authentication + SSL

Spring Security

- *Spring Security is a framework that provides **authentication, authorization, and protection against common attacks.***
- *With first class support for securing both imperative and reactive applications, it is the de-facto standard for securing Spring-based applications.*
- Atualmente, já vai na versão Spring Security 5.x

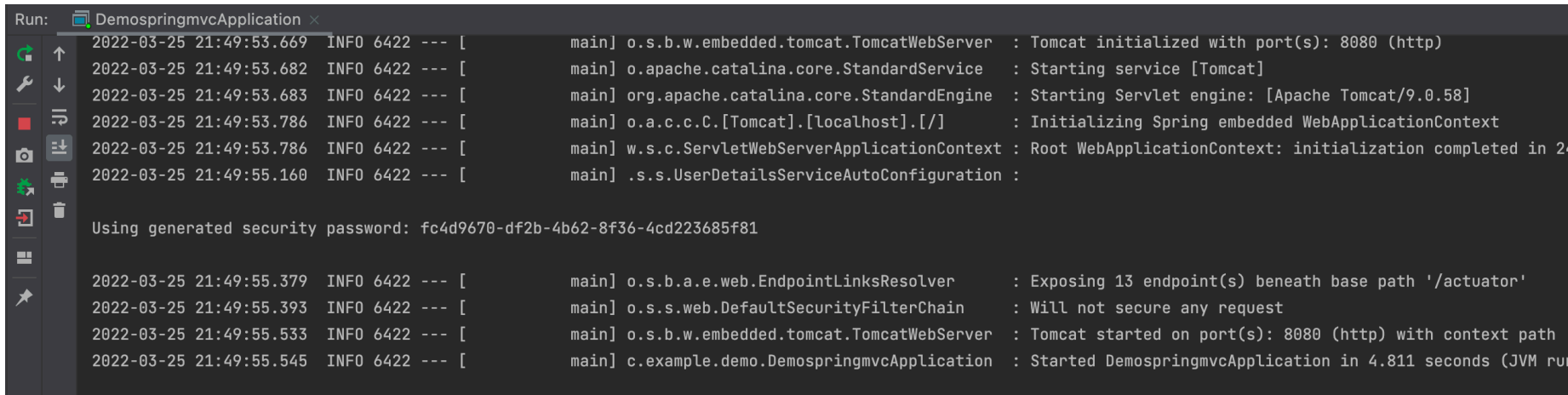
Consultar: <https://docs.spring.io/spring-security/reference/index.html>

REST API Security

Basic Authentication with Spring Security (1)

- The addition of Basic Auth in Spring Boot is almost trivial.
- First add the following dependency, and run de application. Below we can see an exemple of a default password created by Spring:

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



```
Run: DemospringmvcApplication x  
2022-03-25 21:49:53.669 INFO 6422 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)  
2022-03-25 21:49:53.682 INFO 6422 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]  
2022-03-25 21:49:53.683 INFO 6422 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.58]  
2022-03-25 21:49:53.786 INFO 6422 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext  
2022-03-25 21:49:53.786 INFO 6422 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 2  
2022-03-25 21:49:55.160 INFO 6422 --- [main] .s.s.UserDetailsServiceAutoConfiguration :  
  
Using generated security password: fc4d9670-df2b-4b62-8f36-4cd223685f81  
  
2022-03-25 21:49:55.379 INFO 6422 --- [main] o.s.b.a.e.web.EndpointLinksResolver : Exposing 13 endpoint(s) beneath base path '/actuator'  
2022-03-25 21:49:55.393 INFO 6422 --- [main] o.s.s.web.DefaultSecurityFilterChain : Will not secure any request  
2022-03-25 21:49:55.533 INFO 6422 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path  
2022-03-25 21:49:55.545 INFO 6422 --- [main] c.example.demo.DemospringmvcApplication : Started DemospringmvcApplication in 4.811 seconds (JVM ru
```

REST API Security

Basic Authentication with Spring Security (2)

- If you would like to customize your Api credentials fill on the application.properties the following:

```
spring.security.user.name=admin  
spring.security.user.password=123456
```

The screenshot displays two instances of the Postman interface. The left instance shows the 'Authorization' tab with 'Basic A...' selected, and the 'Body' tab showing a 401 Unauthorized response. The right instance shows the 'Authorization' tab with 'Basic A...' selected, and the 'Body' tab showing a 200 OK response. The response body is a JSON object with links to the self, beans, and actuator endpoints.

Left Panel (401 Unauthorized):

- Method: GET
- URL: localhost:8080/actuator
- Authorization: Basic A...
- Body: 401 Unauthorized 838 ms 557 B

Right Panel (200 OK):

- Method: GET
- URL: localhost:8080/actuator
- Authorization: Basic A...
- Body: 200 OK 642 ms 2.17 KB

Response Body (JSON):

```
{  
  "_links": {  
    "self": {  
      "href": "http://localhost:8080/actuator",  
      "templated": false  
    },  
    "beans": {  
      "href": "http://localhost:8080/actuator/beans",  
      "templated": false  
    }  
  }  
}
```

REST API Security

How to Secure a Rest API with HTTPs

- **Step 1: Create a X.509 Certificate**
 - In production-grade applications, certificates are issued from renowned Certification Authorities (CA) to ensure that our application is a trusted entity.
 - However, we can create a Self-Signed Certificate for our application by using Java **keytool** available the JDK_HOME/bin directory

```
keytool -genkey -keyalg RSA -alias mycertificatex509 -keystore mycertificatex509.jks -storepass password -validity 365 -keysize 4096 -storetype pkcs12
```

- Using the RSA algorithm
- Providing an alias name as mycertificatex509
- Naming the Keystore file as mycertificatex509.jks
- Validity for one year: 365
- Storepass: password

REST API Security

How to Secure a Rest API with HTTPs

- **Step 2:** Add the Certificate to the Project
 - Copy to the directory src/main/resources at the classpath

- **Step 3:** Add ssl properties to application.properties

```
server.ssl.key-store=classpath:mycertificatex509.jks
server.ssl.key-store-type=pkcs12
server.ssl.key-store-password=password
server.ssl.key-password=password
server.ssl.key-alias=mycertificatex509
server.port=8443
```

- **Step 4:** Teste your API with Postman
 - But first Postman Disable Certificate Verification

REST API Security

How to Secure a Rest API with HTTPs

- Illustration:

GET https://localhost:8443/getconta?id=1 Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Type Basic A...

The authorization header will be automatically generated when you send the request.
[Learn more about authorization](#)

Username admin

Password

☐ Show Password

! Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. [Learn more about variables](#)

Body Cookies (1) Headers (15) Test Results 200 OK 9 ms 595 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 1,
3   "titular": "alexandre",
4   "morada": "Castelo Branco",
5   "nif": 123456,
6   "pin": 1234,
7   "saldo": 2000.0
8 }
```


Spring Security - MVC + Thymeleaf

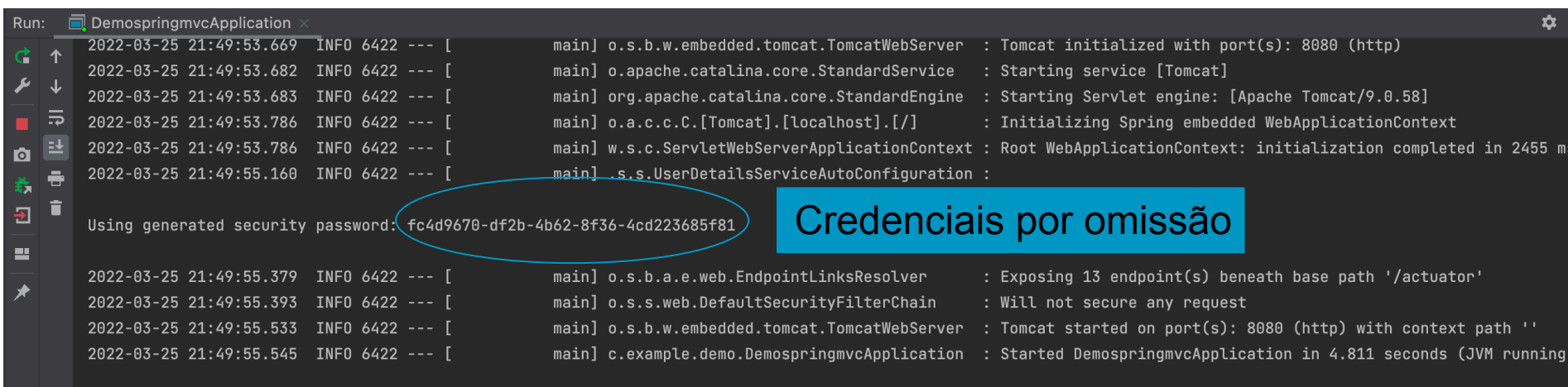
Spring Security - MVC + Thymeleaf

- At the end of current section we will provide Spring Security and Thymeleaf integration custom login and logout pages with CSRF token using JavaConfig.
- We can create our login and logout form using Thymeleaf.
- Spring Security enables CSRF protection by default.
- Thymeleaf includes CSRF token within form automatically.
- Logout form must be submitted as POST when using CSRF protection.
- Spring Boot configure all need Spring Security and Thymeleaf dependencies, including Thymeleaf extras.
- Users can access only after successful authentication.
- User' credentials can be stored on in-memory or on a database. Therefore, we will see how to implement a database for User Authentication

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

- **Passo 1:** Add to pom.xml the spring-boot-starter-security.

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



Run: DemospringmvcApplication x

```
2022-03-25 21:49:53.669 INFO 6422 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2022-03-25 21:49:53.682 INFO 6422 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2022-03-25 21:49:53.683 INFO 6422 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.58]
2022-03-25 21:49:53.786 INFO 6422 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2022-03-25 21:49:53.786 INFO 6422 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 2455 ms
2022-03-25 21:49:55.160 INFO 6422 --- [main] .s.s.UserDetailsServiceAutoConfiguration :

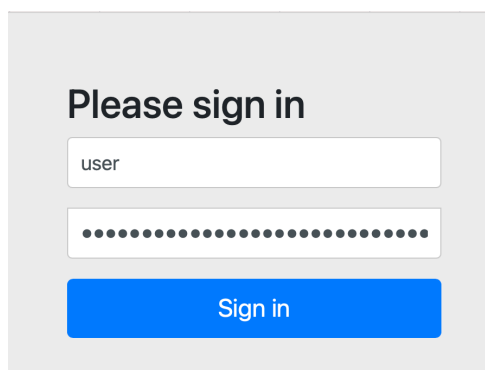
Using generated security password: fc4d9670-df2b-4b62-8f36-4cd223685f81

2022-03-25 21:49:55.379 INFO 6422 --- [main] o.s.b.a.e.web.EndpointLinksResolver : Exposing 13 endpoint(s) beneath base path '/actuator'
2022-03-25 21:49:55.393 INFO 6422 --- [main] o.s.s.web.DefaultSecurityFilterChain : Will not secure any request
2022-03-25 21:49:55.533 INFO 6422 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2022-03-25 21:49:55.545 INFO 6422 --- [main] c.example.demo.DemospringmvcApplication : Started DemospringmvcApplication in 4.811 seconds (JVM running)
```

Credenciais por omissão

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

- **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:** Run App and fill generated security password on auto-generated Login form.

A screenshot of the default Spring Boot login form. It features a light gray background with the text "Please sign in" at the top. Below this, there are two input fields: the first is labeled "user" and contains the text "user"; the second is a password field filled with dots. At the bottom of the form is a blue button with the text "Sign in".

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

Caso 2: Spring Boot basic authentication popup login usando o ficheiro application.properties

- Spring boot basic http authentication popup is a traditional & easy way to authenticate.
- If you have a single login user only, then you can use properties files to save the user credentials directly.
- **You don't need to implement a database or in-memory authentication provider.**
- **Passo 1:** Criar uma classe anotada com **@EnableWebSecurity** que estende a classe abstracta **WebSecurityConfigurerAdapter** com as Configurações de Segurança.

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

}
```

Classe Abstracta WebSecurityConfigurerAdapter

- WebSecurityConfigurerAdapter is an abstract class provided by the Spring Security.
- Generally, we use it to override its **configure()** methods in order to define our security configuration class.
- Typically, we use two configure() methods. One is used to declare **authentication** related configurations whereas the other one is to declare **authorization** related configurations.

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

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {

        // configure Authentication .....
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        // configure Authorization .....
    }
}
```

Caso 2: Spring Boot basic authentication popup login usando o ficheiro application.properties

- **Passo 2:** Fazer o override do método configure() para configurar a autorização dos pedidos http à aplicação.
- **Passo 3:** Configure HttpSecurity security to authenticate all requests and apply HttpBasic authentication using httpBasic() method.

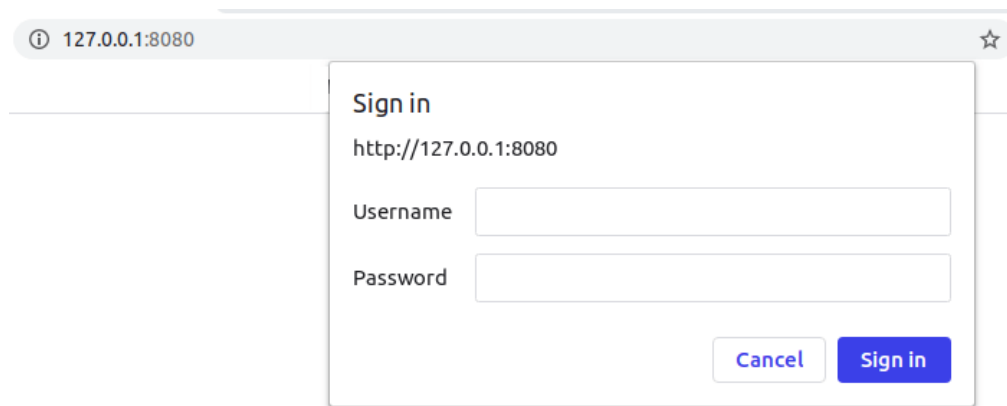
```
@Configuration
@EnableWebSecurity // (1)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter { // (1)
    @Override
    protected void configure(HttpSecurity http) throws Exception { // (2)
        http.authorizeRequests() //(3)
            .anyRequest().authenticated()
         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

- **Passo 4:** Definir as credenciais do utilizador e tipo de utilizador no application.properties para o Spring Security comparar durante o acesso.

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

Caso 2: Spring Boot basic authentication popup login usando o ficheiro application.properties

- **Passo 5:** Executar e Testar o acesso. Deverá ser levantado um formulário de autenticação popup



Caso 3: Spring Boot Security 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.*
- **Passos 1 a 3 (tal como antes):** Criar a classe anotada com **@EnableWebSecurity** com as Configurações de Segurança de autenticação.

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

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

- **Passo 4:** Since we are going to use the in-memory authentication. It's recommended to use a password encoder for saving passwords in memory or database. Spring security has inbuilt support for BCryptPasswordEncoder (ver slide seguinte).
- Basta especificar na classe de configurações o método @Bean:

```
@Bean  
public BCryptPasswordEncoder bCryptPasswordEncoder() {  
    return new BCryptPasswordEncoder();  
}
```

Spring Security - Codificadores de Passwords

- Spring Security cannot magically guess your preferred password hashing algorithm. That's why you need to specify a @Bean, a PasswordEncoder.
- If you want to, say, use the BCrypt password hashing function (Spring Security's default) for all your passwords, you would specify this @Bean in your SecurityConfig.

```
@Bean
public BCryptPasswordEncoder bCryptPasswordEncoder() {
    return new BCryptPasswordEncoder();
}
```

```
@Bean
public PasswordEncoder passwordEncoder() {
    return PasswordEncoderFactories.createDelegatingPasswordEncoder();
}
```

Forma como os Hash são guardados - o prefixo indica o algoritmo:

```
{bcrypt}$2y$12$6t86Rpr3lIMANhCUt26oUen2WhvXr/A89Xo9zJion8W7gWgZ/zA0C
{sha256}5ffa39f5757a0dad5dfada519d02c6b71b61ab1df51b4ed1f3beed6abe0ff5f6
```

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

- **Passo 5:** Na WebSecurityConfig, configurar as autenticações um AuthenticationManagerBuilder para definir as credenciais in-memory e a função do utilizador

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {

    auth.inMemoryAuthentication()
        .passwordEncoder(new BCryptPasswordEncoder())
        .withUser("admin")
        .password(passwordEncoder().encode("admin123"))
        .roles("ADMIN")
        .and()
        .withUser("utilizador")
        .password(passwordEncoder().encode("123456"))
        .roles("USER")
    ;
}
```

- **Passo 6:** Executar e testar a aplicação

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

- Neste caso a página Login é criada inteiramente pelo Developer e personalizada para as suas necessidades
- **Passo 0:** Verificar se o starter Thymeleaf foi adicionado ao pom.xml
- **Passos 1 a 5:** Mantém-se iguais ao do caso anterior.
- **Passo 6:** Criar um Login Form usando a Thymeleaf com CSRF Token
 - The value for action, **user name and password** attributes used in authentication are the custom values which has been defined in our spring security configuration.
 - Spring Security enables CSRF protection by default. We need to submit form using **POST method**.
 - CSRF token will be included at runtime by Thymeleaf because of **@EnableWebSecurity** annotation. We are catching default **error** query parameter to display error message.

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

- Exemplo de Login personalizado

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

```
    <div th:if="${param.error}" class="alert alert-error">
      Invalid username and password.
    </div>
    <div th:if="${param.logout}" class="alert alert-success">
      You have been logged out, Please login again.
    </div>
```

```
    <form th:action="@{/appLogin}" 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>
```

Nota: podem ser personalizados, contudo se não se utilizar explicitamente um objeto UserDetails, por omissão são verificados e se forem diferentes o spring security retorna Bad Credentials

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

- **Passo 7:** Criar o Controlador que fornece os mapeamentos para as páginas *home* ("/") e login.

- Exemplo:

```
@Controller
public class HelloController {

    @GetMapping("/")
    public ModelAndView home(Principal principal) {
        return (new ModelAndView("home")).addObject("principal", principal);
    }

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

- **Passo 8:** Na classe WebSecurityConfig fazer as configurações de segurança. (Ver slides seguintes) por forma a mapear a página login.
- **Passo 9:** Executar a aplicação e testar

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

- How to configure Spring Security:
WebSecurityConfigurerAdapter (versões Spring Boot 2.6.x)
 - (1)** Criar classe de Configuração de Segurança anotada com `@EnableWebSecurity` que estende a classe abstrata `WebSecurityConfigurer`, que fornece os métodos para as configurações.
 - Com estes métodos, podemos especificar quais os URIs da aplicação proteger ou que protecções contra *exploits* podem ser activadas/desactivar

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

}
```

Créditos: <https://www.marcobehler.com/guides/spring-security>

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

- How to configure Spring Security: WebSecurityConfigurerAdapter (versões Spring Boot 2.6.x) - Exemplo

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

    @Override
    protected void configure(HttpSecurity http) throws Exception { // (2)
        http
            .authorizeRequests()
            .antMatchers("/", "/home").permitAll() // (3)
            .anyRequest().authenticated() // (4)
            .and()
            .formLogin() // (5)
            .loginPage("/login") // (5)
            .permitAll()
            .and()
            .logout() // (6)
            .permitAll()
            .and()
            .httpBasic(); // (7)
    }
}
```

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

- (1) A normal Spring `@Configuration` with the `@EnableWebSecurity` annotation, extending from `WebSecurityConfigurerAdapter`.
- (2) By overriding the adapter's `configure(HttpSecurity)` method, you get a nice little DSL with which you can configure your `FilterChain`.
- (3) All requests going to `/` and `/home` are allowed (permitted) - the user does not have to authenticate. You are using an `antMatcher`, which means you could have also used wildcards (`*`, `**`, `?`) in the string.
- (4) Any other request needs the user to be authenticated first, i.e. the user needs to login.
- (5) You are allowing form login (username/password in a form), with a custom `loginPage` (`/login`, i.e. not Spring Security's auto-generated one). Anyone should be able to access the login page, without having to log in first (`permitAll`; otherwise we would have a Catch-22!).
- (6) The same goes for the logout page
- (7) On top of that, you are also allowing Basic Auth, i.e. sending in an HTTP Basic Auth Header to authenticate.

Spring Security - Codificadores de Passwords

- Spring Security cannot magically guess your preferred password hashing algorithm. That's why you need to specify a @Bean, a PasswordEncoder.
- If you want to, say, use the BCrypt password hashing function (Spring Security's default) for all your passwords, you would specify this @Bean in your SecurityConfig.

```
@Bean
public BCryptPasswordEncoder bCryptPasswordEncoder() {
    return new BCryptPasswordEncoder();
}
```

```
@Bean
public PasswordEncoder passwordEncoder() {
    return PasswordEncoderFactories.createDelegatingPasswordEncoder();
}
```

Forma como os Hash são guardados - o prefixo indica o algoritmo:

```
{bcrypt}$2y$12$6t86Rpr3lIMANhCUt26oUen2WhvXr/A89Xo9zJion8W7gWgZ/zA0C
{sha256}5ffa39f5757a0dad5dfada519d02c6b71b61ab1df51b4ed1f3beed6abe0ff5f6
```

Spring Security - Cross-Site-Request-Forgery: CSRF

- (...) By default Spring Security protects any incoming POST (or PUT/DELETE/PATCH) request with a valid CSRF token.

```
<form action="/transfer" method="post"> <!-- 1 -->
  <input type="text" name="amount"/>
  <input type="text" name="routingNumber"/>
  <input type="text" name="account"/>
  <input type="submit" value="Transfer"/>
</form>
```

With Spring Security enabled, you **won't be able to submit that form anymore**. Because Spring Security's CSRFFilter is looking for an *additional hidden parameter* on **any POST** (PUT/DELETE) request: a so-called CSRF token.

It generates such a token, by default, *per HTTP session* and stores it there. And you need to make sure to inject it into any of your HTML forms.

Spring Security - Cross-Site-Request-Forgery: CSRF + Thymeleaf

- if you are using "th:action" for your form, Thymeleaf will automatically inject that hidden field for you, without having to do it manually.

```
<form action="/transfer" method="post"> <!-- 1 -->
  <input type="text" name="amount"/>
  <input type="text" name="routingNumber"/>
  <input type="text" name="account"/>
  <input type="submit" value="Transfer"/>
  <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
</form>
```

Injeção "Manual"

```
<!-- OR -->
```

```
<form th:action="/transfer" method="post"> <!-- 2 -->
  <input type="text" name="amount"/>
  <input type="text" name="routingNumber"/>
  <input type="text" name="account"/>
  <input type="submit" value="Transfer"/>
</form>
```

Injeção "Automática"

Spring Security - Disabling CSRF

- **If you are only providing a stateless REST API** where CSRF protection does not make any sense, you would completely disable CSRF protection. This is how you would do it

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

```
    @Override  
    protected void configure(HttpSecurity http) throws Exception {  
        http  
            .csrf().disable();  
    }  
}
```

Spring Security & Thymeleaf Extras

- Spring Security integrates well with Thymeleaf. It offers a special Spring Security Thymeleaf dialect, which allows you to put security expressions directly into your Thymeleaf HTML templates.

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

```
<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>
```

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

- Spring boot security with database authentication is the most preferred way in standard applications.
- We will implement a custom login with H2 database authentication.
- Grosso modo os passos anteriores do caso 3 são válidos, com a exceção do método das configurações de autenticação ainda que veremos ser necessário realizar vários passos específicos.

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

```
EM VEZ DE:
auth.inMemoryAuthentication()
    .passwordEncoder(new BCryptPasswordEncoder())
    .withUser("admin")
    .password(passwordEncoder().encode("admin123"))
    .roles("ADMIN")
    .and()
    .withUser("utilizador")
    .password(passwordEncoder().encode("123456"))
    .roles("USER")
;
```


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

- **Passos adicionais:**
- **Passo 1:** Adicionar os starter da Spring Data JPA e da Base de dados.

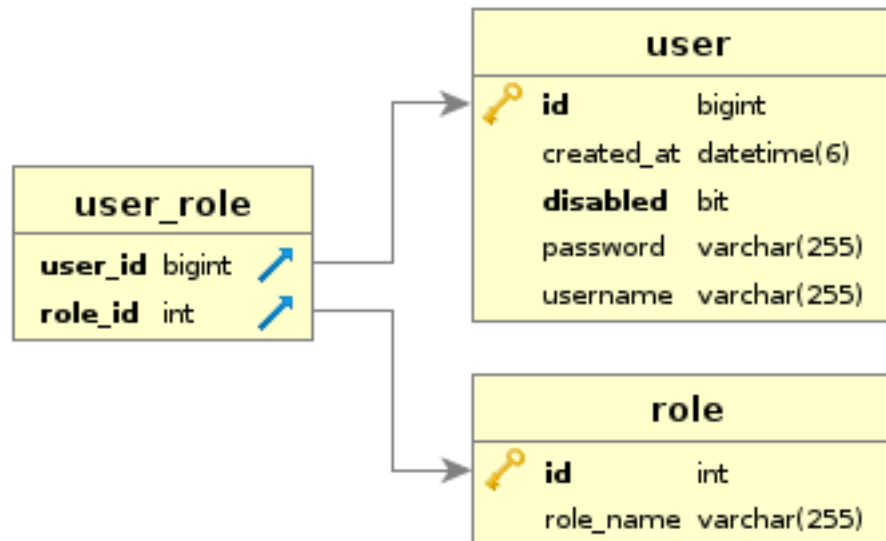
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

```
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
```

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

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

- **Passos adicionais (cont.):**
- **Passo 2:** Criar as classes entidade Utilizador (User) e da função (Role), e desejavelmente a adição do mapeamento many-to-many entre o utilizador e a função (Role)



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

- **Passos adicionais (cont.):**
- **Passo 2:** Criar as classes entidade do Utilizador e da função (Role), e desejavelmente a adição do mapeamento many-to-many entre o utilizador e a função (Role)

Nas aulas PL faremos uma implementação em que a classe User extends a classe UserDetails

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

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

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

- **Passos adicionais (cont.):**
- **Passo 2:** Criar as classes entidade do Utilizador e da função (Role), e desejavelmente a adição do mapeamento many-to-many entre o utilizador e a função (Role)

```
@Entity
@Data
public class Role {
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Id
    private Long id;
    private String roleName;
    ...
}
```

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

- **Passos adicionais (cont.):**
- **Passo 3:** Criar os repositórios de dados RoleRepository e UserRepository.
- O RoleRepository não precisa mais do que os método *default*. No caso do UserRepository é preciso adicionar um método que procure pelo username (disable no exemplo é opcional)

```
public interface RoleRepository extends JpaRepository<Role, Long> {  
}
```

```
public interface UserRepository extends JpaRepository<User, Long> {  
    User findByUsernameAndDisabled(String username, boolean disabled);  
}
```

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

- Passos adicionais (cont.):
- **Passo 4:** Criar um serviço que implementa a classe Spring UserDetailsService e fazer o override do método **loadUserByUsername()** que implementa a procura pelo utilizador.

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

    public UserDetailsServiceImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Override
    public UserDetails loadUserByUsername(String userName) throws UsernameNotFoundException {
        User user = userRepository.findByUsernameAndDisabled(userName, false);
        if (user == null) {
            throw new UsernameNotFoundException("User is not Found");
        }

        return new org.springframework.security.core.userdetails.User(user.getUsername(),
            user.getPassword(),
            mapRolesToAuthorities(user.getRoles()));
    }

    private Collection<? extends GrantedAuthority> mapRolesToAuthorities(List<Role> roles) {
        return roles.stream()
            .map(role -> new SimpleGrantedAuthority(role.getRoleName()))
            .collect(Collectors.toList());
    }
}
```

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

- Passos adicionais (cont.):
- **Passo 5:** (Em parte já visto) Incorporar na classe **WebSecurityConfig** a classe **UserDetailsServiceImpl**

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

```
    @Autowired
    UserRepository userRepository;
```

```
    @Bean
    public UserDetailsService userDetailsService() {
        return new UserDetailsServiceImpl(userRepository);
    }
```

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

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

- **Passos adicionais (cont.):**
- **Passo 6:** Criar o(s) as funções e o(s) utilizador e Testar a aplicação.
- Exemplo (colocar no Application Runner):

```
System.out.println("Saving demo data");  
Role roleAdmin = new Role("ADMIN");  
Role roleUser = new Role("USER");  
roleRepository.save(roleAdmin);  
roleRepository.save(roleUser);
```

```
User userAdmin = new User("technicalsand", passwordEncoder.encode("password1"),  
roleAdmin);  
User userReader = new User("reader", passwordEncoder.encode("reader1"), roleUser);
```

```
userRepository.save(userAdmin);  
userRepository.save(userReader);
```

```
System.out.println("Demo data saved successfully.");
```

- **Passo 7:** Executar e Testar a aplicação. Comece por verificar o conteúdo da BD

Notas Finais

- Utilizamos na implementação das configurações de segurança a classe **WebSecurityConfigurerAdapter**, **contudo** esta foi recentemente deprecated no Spring Boot 2.7.x o qual usa automaticamente o Spring Security 5.7.x

[-https://spring.io/blog/2022/02/21/spring-security-without-the-websecurityconfigureradapter](https://spring.io/blog/2022/02/21/spring-security-without-the-websecurityconfigureradapter)

- A migração para o novo formato é relativamente simples. Uma das principais diferenças é a criação de um Bean SecurityFilterChain

ANTES:

```
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests((authz) -> authz
                .anyRequest().authenticated()
            )
            .httpBasic(withDefaults());
    }
}
```

AGORA:

```
@Configuration
public class SecurityConfiguration {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests((authz) -> authz
                .anyRequest().authenticated()
            )
            .httpBasic(withDefaults());
        return http.build();
    }
}
```

Notas Finais

- Alterações In-Memory Authentication (agora exige um Bean InMemoryUserDetailsManager)

ANTES:

```
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("password")
            .roles("USER")
            .build();
        auth.inMemoryAuthentication()
            .withUser(user);
    }
}
```

AGORA:

```
@Configuration
public class SecurityConfiguration {
    @Bean
    public InMemoryUserDetailsManager userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("password")
            .roles("USER")
            .build();
        return new InMemoryUserDetailsManager(user);
    }
}
```

Notas Finais

- Alterações Database-based Authentication (agora exige um Bean AuthenticationManager)

ANTES:

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

    @Autowired
    UserDetailsService userDetailsService;

    @Override
    public void configure(AuthenticationManagerBuilder authenticationManagerBuilder) throws Exception {
        authenticationManagerBuilder.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

AGORA:

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig {

    @Bean
    AuthenticationManager authenticationManager(AuthenticationConfiguration
authenticationConfiguration) throws Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }
}
```

Notas Resumo

Segurança de APIs/Serviços REST

Segurança de Endpoints API e Serviços REST

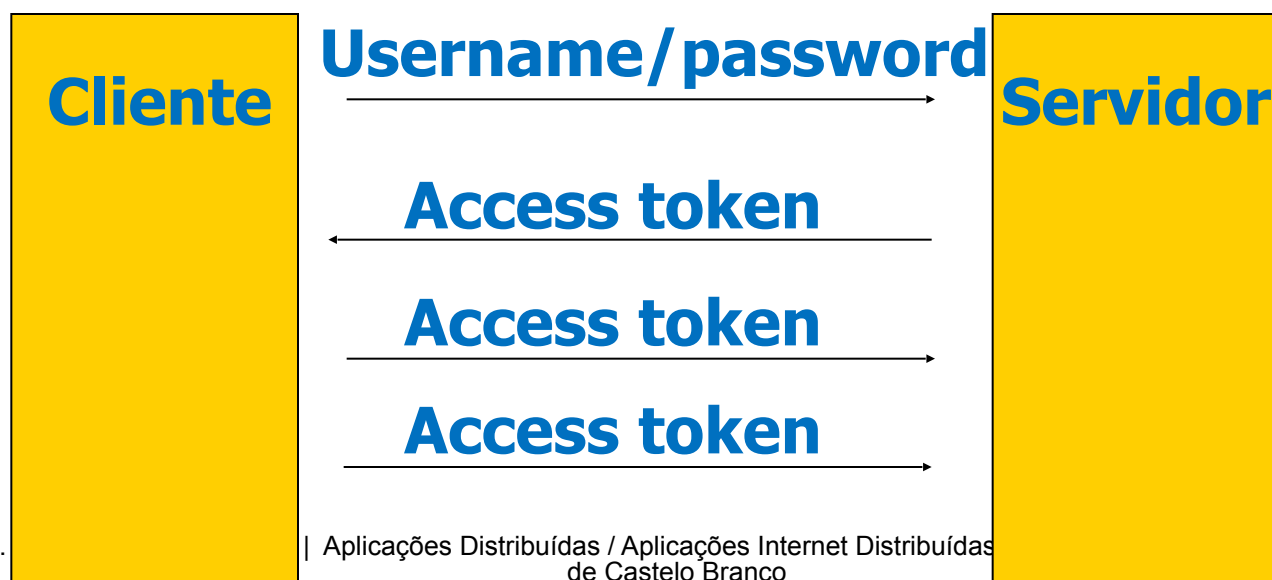
- Medidas preliminares

1. Proteger os Endpoints com HTTPS
2. Limitar os métodos HTTP que são utilizados e não passar informação sensível nos URLs
3. Utilizar códigos de Estado corretos (não apenas 200 OK) sem revelarem demasiada informação
4. Validar os tipos de conteúdo nas mensagens HTTP

https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html

Segurança de Endpoints API e Serviços REST

- Primeira Medida: Proteger os Endpoints com HTTPS
 - Protege as credenciais de autenticação em trânsito, por exemplo palavras-passe, API keys ou JSON Web Tokens.
 - Garante a integridade dos dados transmitidos.
 - Considere a utilização de certificados mutuamente autenticados do lado do cliente para fornecer protecção adicional para serviços web altamente privilegiados



Segurança de Endpoints API e Serviços REST

- Segunda Medida: Limitar os métodos HTTP e não passar informação sensível nos URLs
 - Aplicar uma lista de métodos HTTP permitidos, por exemplo, GET, POST, PUT.
 - Rejeitar todos os pedidos que não correspondam à lista de permissões com o código de resposta HTTP 405 Método não permitido.
 - Certifique-se de que o cliente/autor do pedido está autorizado a utilizar o método HTTP para o acesso as recursos, ações, e registo.
 - De momento, isto é difícil de implementar em Jakarta EE (ver https://cheatsheetseries.owasp.org/assets/REST_Security_Cheat_Sheet_Bypassing_VBAAC_with_HTTP_Verb_Tampering.pdf)

Segurança de Endpoints API e Serviços REST

- Segunda Medida: Limitar os métodos HTTP e não passar informação sensível nos URLs (Continuação...)
 - Não passar informação sensível (palavras-passe, API keys, Tokens) nos URLs pois podem ser capturadas pelos Logs Web:
 - Nos pedidos POST/PUT os dados sensíveis devem ser transferidos no body do request ou nos cabeçalhos.
 - Nos pedidos GET os dados sensíveis devem ser transferidos num cabeçalho HTTP.

OKAY:

<https://exemplo.com/coleccaorecursos/1/>

<https://twitter.com/vanderaj/lists>

ERRADO:

<https://exemplo.com/controlador/123/action?apiKey=a53f435643de32>

Errado pois a API Key está no URL!

Segurança de Endpoints API e Serviços REST

- Terceira Medida: Utilizar códigos de Estado corretos (não apenas 200 OK) sem revelarem demasiada informação
 - De seguida, está uma seleção não exaustiva de códigos de estado para as REST API relacionados com a segurança da API.

Code	Message	Description
200	OK	Response to a successful REST API action. The HTTP method can be GET, POST, PUT, PATCH or DELETE.
201	Created	The request has been fulfilled and resource created. A URI for the created resource is returned in the Location header.
202	Accepted	The request has been accepted for processing, but processing is not yet complete.
301	Moved Permanently	Permanent redirection.
304	Not Modified	Caching related response that returned when the client has the same copy of the resource as the server.
307	Temporary Redirect	Temporary redirection of resource.

400	Bad Request	The request is malformed, such as message body format error.
401	Unauthorized	Wrong or no authentication ID/password provided.
403	Forbidden	It's used when the authentication succeeded but authenticated user doesn't have permission to the request resource.
404	Not Found	When a non-existent resource is requested.
405	Method Not Acceptable	The error for an unexpected HTTP method. For example, the REST API is expecting HTTP GET, but HTTP PUT is used.
406	Unacceptable	The client presented a content type in the Accept header which is not supported by the server API.
413	Payload too large	Use it to signal that the request size exceeded the given limit e.g. regarding file uploads.
415	Unsupported Media Type	The requested content type is not supported by the REST service.
429	Too Many Requests	The error is used when there may be DOS attack detected or the request is rejected due to rate limiting.
500	Internal Server Error	An unexpected condition prevented the server from fulfilling the request. Be aware that the response should not reveal internal information that helps an attacker, e.g. detailed error messages or stack traces.
501	Not Implemented	The REST service does not implement the requested operation yet.
503	Service Unavailable	The REST service is temporarily unable to process the request. Used to inform the client it should retry at a later time.

Segurança de Endpoints API e Serviços REST

- Quarta Medida: Validar os tipos de conteúdo nas mensagens HTTP

- Documentar sempre todos os tipos de conteúdos suportados pela API.
- O conteúdo no body de pedido ou resposta REST tem que corresponder ao tipo de content type no cabeçalho. Senão, pode levar a má interpretação ou a ataques de injeção/execução de Código malicioso.
- Rejeitar todas as mensagens com conteúdo não esperado ou ausente e responder com HTTP response status 406 Unacceptable ou 415 Unsupported Media Type.
- Evitar a exposição accidental não intencional de tipos de conteúdo, por exemplo, @consumes("application/json"); @produces("application/json").
- Rejeitar os pedidos (idealmente com 406 Not Acceptable response) se o header Accept não conter especificamente um dos tipos permitidos.
- Serviços que incluem Código de script (e.g. JavaScript) na suas respostas devem ter um tratamento cuidado para defesa contra ataques de injeção de cabeçalhos.
- Assegurar que os content type headers nas respostas correspondem ao que é enviado no body e.g. application/json e não application/javascript

https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html

Questões

