



Formación
BDI



Curso Spring Framework



Módulo 8 Spring Security



Andrés Guzmán F.
Formación BDI TI
Bolsadeideas.com



Spring Security



¿Qué es Spring
Security?

¿Porqué Spring
Security?

Esquema de Base
de Datos

Autenticación
Base de Datos

Configuración

Form y login
básico

Logout



Temas

*¿Porqué
Spring Security?*

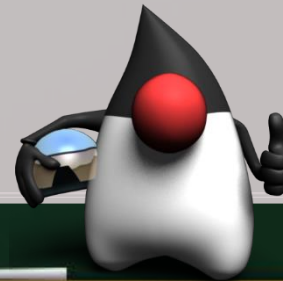


Problemas del API de Seguridad Java EE

- Las características de Seguridad en la especificación EJB y en la especificación Servlet carecen de profundidad para ciertos escenarios en aplicaciones empresariales y son más complejas de configurar
- Además no son portables a nivel de WAR o EAR
- Requiere reconfigurar toda la seguridad en caso de emigrar a una nueva plataforma o servidor de aplicaciones.

Spring security

- Una solución mucho más portable y fácil de configurar
- Soluciona las deficiencias de seguridad de Java EE mencionados anteriormente



*¿Qué es
Spring Security?*



¿Qué es Spring Security?

Spring Security es un subproyecto del framework Spring, que permite gestionar completamente la seguridad de nuestras aplicaciones Java, y cuyas ventajas principales son las siguientes:

¿Qué es Spring Security?

La configuración de la seguridad es portable de un servidor a otro, ya que se encuentra dentro del WAR o el EAR de nuestras aplicaciones

Soporta muchos modelos de identificación de los usuarios (HTTP BASIC, HTTP Digest, basada en formulario, LDAP, OpenID, JAAS y muchos más). Además podemos ampliar estos mecanismos implementando nuestras propias clases que extiendan el modelo de Spring Security

Es capaz de gestionar seguridad en varios niveles: URLs que se solicitan al servidor, acceso a métodos y clases Java, y acceso a instancias concretas de las clases

Permite separar la lógica de nuestras aplicaciones del control de la seguridad, utilizando filtros para las peticiones al servidor de aplicaciones o aspectos para la seguridad en clases y métodos

Características Spring Security

Provee
características de
seguridad para
aplicaciones
empresariales Java
EE



Maneja
componentes de
"Autenticación" y
"Autorización"

Spring Security

Autenticación

Autorización
(control de acceso)

Características Spring Security

Autenticación: se refiere al proceso de establecer un principal (un principal significa un usuario, dispositivo o algún otro sistema el cual puede ejecutar alguna acción en nuestro sistema), en general permite a los principal autenticarse en base a cualquier proveedor de seguridad por ejemplo LDAP, Base de datos relacional principalmente y Autenticación HTTP

Autorización: se refiere al proceso de decidir si se otorga acceso a un usuario para realizar una acción dentro de la aplicación, es decir para controlar el acceso a los recursos de la aplicación por medio de la asignación de roles y permisos a grupos de usuarios

Soporte en Spring Authentication

Autenticación HTTP BASIC

Autenticación Form (Login de usuarios plataforma web)

Autenticación HTTP Digest

HTTP X.509 (client certificate exchange)

Autenticación automática vía "remember-me"

Java Authentication and Authorization Service (JAAS)

Autenticación contenedor Java EE

LDAP

OpenID

Kerberos

Soporte en Spring Authorization

Tres áreas
en las que
se puede
aplicar
autorización

- Autorización en peticiones web HTTP (HTTP Requests)
- Autorización cuando los métodos son invocados
- Autorización cuando se accede a objetos del dominio (instancias)

Configuración Spring Security



Configurar web.xml

Provee una forma estándar para configurar Spring Security en Aplicaciones Web Java EE

La clase DelegatingFilterProxy (de Spring Framework) es la encargada del arranque de Spring Security, quién delega la seguridad en una implementación de un Filtro Servlet

```
<!-- Spring Security -->
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```



XML configuración de Contexto Spring Security

Además necesitamos habilitar el archivo XML configuración de Contexto Spring Security, donde se centralizan todas las configuraciones (reglas y definiciones de seguridad)

Para ello se agrega el archivo **applicationContext-security.xml** en el parámetro de contexto "**contextConfigLocation**" (context-param) en el web.xml

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/spring/root-context.xml,
    /WEB-INF/spring/applicationContext-security.xml
  </param-value>
</context-param>
```



Configuración Mínima de Seguridad <http>

Todo lo mínimo necesario para habilitar la seguridad web (HTTP Request) es configurar lo siguiente en el archivo de configuración de contexto de Spring Security (applicationContext-security.xml)

```
<!-- Todas las URLs quedarán seguras, requiere el ROLE_USER para el acceso -->  
<http auto-config='true'  
  <intercept-url pattern="/*" access="hasRole('ROLE_USER')" />  
</http>
```


Elemento `<intercept-url>`

El elemento `<intercept-url>` define un patrón el cual debe coincidir contra las URLs proveniente del request utilizando una sintaxis de path.

También podemos usar expresiones regulares para definir el patrón

Podemos usar multiples elementos `<intercept-url>` para definir los diferentes accesos para las diferentes URLs aque requieran aplicar seguridad, serán evaluadas en el orden listado y aplicará la primera regla de seguridad que coincida con el patrón y la URL.

También podemos agregar el atributo "method" para limitar las coincidencias hacia un método específico HTTP (GET, POST, PUT etc.).

Atributo auth-config

- El atributo *auto-config="true"* es un atajo para lo siguiente:

```
<http>  
  <form-login />  
  <logout />  
</http>
```

- Representan el manejo de un formulario de login (form-login) y cierre de sesión (logout) respectivamente
- Cada uno de ellos tienen atributos que pueden utilizarse para personalizar su comportamiento e implementación

<authentication-manager> & <authentication-provider>

Podemos tener más de un elemento <authentication-provider> para definir diferentes tipos u origen de autenticación, cada uno se consultará por turno

Usamos <user-service> para definir credenciales de usuarios las cuales serán usadas por el Authentication Manager de Spring para procesar la autenticación

```
<authentication-manager>  
  <authentication-provider>  
    <user-service>  
      <user name="andres" password="andres" authorities="ROLE_USER,  
                                                         ROLE_ADMIN" />  
      <user name="john" password="1234" authorities="ROLE_USER" />  
    </user-service>  
  </authentication-provider>  
</authentication-manager>
```

<authentication-manager> & <authentication-provider>

Para agregar usuarios, podemos definir un set de información de ejemplo directamente en el namespace del xml

```
<authentication-manager>  
  <authentication-provider>  
    <user-service>  
      <user name="andres" password="andres" authorities="ROLE_USER,  
                                                         ROLE_ADMIN" />  
      <user name="john" password="1234" authorities="ROLE_USER" />  
    </user-service>  
  </authentication-provider>  
</authentication-manager>
```

La configuración de arriba define dos usuarios guardados en memoria (In Memory), con sus contraseñas y sus roles dentro de la aplicación (las cuales serán usadas para el control de acceso).

También es posible cargar la información de los usuarios desde un archivo properties usando el atributo properties en el elemento user-service

Password Encoding MD5

- También podemos crear un codificador hash md5 digest para la password usando el API Jacksum, ejemplo:

```
java -jar jacksum.jar -a md5 -q "txt:miClaveSecreta"
```

```
<authentication-manager>  
  <authentication-provider>  
    <password-encoder hash="md5"/>  
    <user-service>  
      <user name="andres" password="231badb19b93e44f47da1bd64a8147f2"  
        authorities="ROLE_USER, ROLE_ADMIN" />  
      <user name="john" password="81dc9bdb52d04dc20036dbd8313ed055"  
        authorities="ROLE_USER" />  
    </user-service>  
  </authentication-provider>  
</authentication-manager>
```

Password Encoding SHA

Similar al ejemplo MD5, el valor del password puede ser codificado usando algoritmo hash SHA en <password-encoder>

Usamos el atributo hash para especificar el algoritmo codificador a SHA:

```
<authentication-manager>
  <authentication-provider>
    <password-encoder hash="sha"/>
    <user-service>
      <user name="andres"
        password="883768b6dd2c42aea0031b24be8a2da40fef4b64"
        authorities="ROLE_USER, ROLE_ADMIN" />
      <user name="john"
        password="7110eda4d09e062aa5e4a390b0a572ac0d2c0220"
        authorities="ROLE_USER" />
    </user-service>
  </authentication-provider>
</authentication-manager>
```

Concepto de Role



¿Qué es un Role?

Un rol es un grupo o tipo de usuario que se le otorgan ciertos privilegios para llevar a cabo una o varias acciones dentro de una aplicación

Son construcciones simples, que consta de un nombre como "admin", "usuario", "cliente", etc

Pueden concederse ya sea a los usuarios y en algunos casos a otros roles

Se utilizan para crear grupos lógicos de usuarios para la asignación adecuada de los privilegios en la aplicación

Logout



Elemento <logout> para manejar el cierre de sesión

El elemento <logout> da soporte para cerrar la sesión de usuario vía URL.

- Atributo **logout-url**: URL que causa el logout (será procesada por el filter). URL por defecto **"/logout"**.
- El atributo **logout-success-url**: URL a la cual será re-direccionado el usuario después del logout. URL por defecto **"/"**.
- El atributo **invalidate-session**: Valor por defecto **"true"**, la sesión será invalidada en el logout.

Elemento <logout> para manejar el cierre de sesión

```
<http use-expressions="true">  
  <intercept-url pattern="/" access="permitAll"/>  
  <intercept-url pattern="/static/**" filters="none" />  
  <intercept-url pattern="/**" access="isAuthenticated()" />  
  <form-login />  
  <logout logout-success-url="/mi_pagina_logout"/>  
</http>
```

Ejemplo de Logout

```
<form id="logoutForm"
action="${pageContext.request.contextPath}/logout"
method="post">
  <input type="submit" value="Log out" />
  <input type="hidden" name="${_csrf.parameterName}"
value="${_csrf.token}" />
</form>
```

Por defecto viene habilitada la protección Cross Site Request Forgery (CSRF)

Al tenerla habilitada, necesitamos incluir un campo oculto `_csrf.token` en cada formulario que tengamos en nuestra aplicación, sobre todo en los formularios de login y logout.

Formulario Login



Formulario de Login

Spring Security genera un form automáticamente, basado en las características habilitadas y usando valores por defecto como la URL que procesa el login, la URL donde será re-direccionado el usuario despues del login, etc.

Login

Definir nuestra propia página de login

Podemos configurar nuestra propia página formulario de login vía `<form-login>`

A pesar de que podríamos tener el auto-config, el elemento `<form-login>` sobrescribe las configuraciones por defecto

```
<http auto-config="true">  
  <intercept-url pattern="/login*"  
    access="isAnonymous()" />  
  <intercept-url pattern="/*" access="hasRole('ROLE_USER')" />  
  <form-login login-page="/login" />  
</http>
```

Acceso a usuarios anónimos o invitados

Anonymous authentication nos permite darle acceso a un usuarios anónimos a secciones del sistema, digamos que la pagina de registro debe ser de acceso anónimo, usuarios que no están registrados en el sistema, lo mismo sucede con la página de login o inicio de sesión:



Página de Login con acceso a usuarios anónimos o invitados



Cualquier petición a la página de login debe tener acceso público, es decir para usuarios invitados o anónimos, básicamente para permitirles iniciar sesión:

```
<http auto-config="true">  
  <intercept-url pattern="/login*"  
    access="isAnonymous()"/>  
  <intercept-url pattern="/*" access="hasRole('ROLE_USER')"/>  
  <form-login login-page="/login"/>  
</http>
```

Omitir el Security Filter Chain



También es posible para toda petición (request) que coincidan con cierto patrón puedan saltarse completamente el filtro de seguridad de spring, por ejemplo recursos estáticos como imagenes, css, javascript etc:

```
<http auto-config='true'>  
  <intercept-url pattern="/css/**" filters="none" />  
  <intercept-url pattern="/login.jsp*" filters="none" />  
  <intercept-url pattern="/**" access="hasRole('ROLE_USER')" />  
  <form-login login-page='/login.jsp' />  
</http>
```

Múltiples elementos <http>

http://



Desde la versión Spring Security 3.1 se pueden usar múltiples elementos http para definir reglas de seguridad independientes para diferentes patrones URL del request:

```
<http pattern="/css/**" security="none"/>
<http pattern="/login.jsp*" security="none"/>
<http auto-config='true'>
  <intercept-url pattern="/**" access="hasRole('ROLE_USER')"/>
  <form-login login-page='/login'/>
</http>
```

Configurar Página Post-Login

El atributo default-target-url define la URL de destino después de que el usuario haya realizado sesión, por defecto es la página raíz: "/".

```
<http auto-config="true">
  <intercept-url pattern="/miapp/admin*"
    access="hasRole('ROLE_SUPERVISOR')" />
  <intercept-url pattern="/miapp*"
    access="hasRole('ROLE_USER')" />
  <form-login login-page="/mi_pagina_login"
    default-target-url="/mi_pagina_despues_login"
    always-use-default-target='true'
    authentication-failure-url="/mi_pagina_error_login" />
  <logout logout-success-url="/mi_pagina_logout" />
</http>
```


Configurar Página Post-Login

Luego se define un controlador genérico parametrizable que carga la vista o se implementa uno completamente, ejemplo de uno parametrizable para hacerlo más simple:

```
<view-controller path="/mi_pagina_despues_login" view-name="mi_pagina_despues_login"/>
```

La vista mi_pagina_despues_login.jsp: :

```
<body>
<h3>Esto es la primera página justo después del login de inicio de
sesión
Todo el mundo después de iniciar sesión, verá esta página.
</h3>
<p><a href="index.jsp">volver al index.jsp</a></p>
<p><a href="logout">Logout</a></p>
</body>
```

Configurar Página Post-Login

También podemos configurar otras opciones de modo que el usuario siempre terminará en la página post-login (independientemente si el inicio de sesión fue "on-demand" (implícito) o el usuario lo hizo explícitamente) estableciendo el atributo `always-use-default-target` a `"true"` .

Esto es útil si la aplicación requiere que el usuario siempre inicie en la misma página `"/mi_pagina_despues_login"`, por ejemplo

```
<http auto-config="true">
  <intercept-url pattern="/miapp/admin*"
    access="hasRole('ROLE_SUPERVISOR')" />
  <intercept-url pattern="/miapp*" access="hasRole('ROLE_USER')" />
  <form-login login-page="/mi_pagina_login"
    default-target-url="/mi_pagina_despues_login"
    always-use-default-target='true'
    authentication-failure-url="/mi_pagina_error_login" />
  <logout logout-success-url="/mi_pagina_logout" />
</http>
```

Configurar Página Error Login

También es posible indicar la pagina de error del login si falla la autenticación:

```
<http auto-config="true">
  <intercept-url pattern="/miapp/admin*"
    access="hasRole('ROLE_SUPERVISOR')" />
  <intercept-url pattern="/miapp*"
    access="hasRole('ROLE_USER')" />
  <form-login login-page="/mi_pagina_login"
    default-target-url="/mi_pagina_despues_login"
    always-use-default-target='true'
    authentication-failure-url="/mi_pagina_error_login" />
  <logout logout-success-url="/mi_pagina_logout" />
</http>
```

Configurar Página Error Login

Luego se define un controlador genérico parametrizable con la vista o se implementa uno completamente, ejemplo parametrizable:

```
<view-controller path="/mi_pagina_error_login" view-name="mi_pagina_error_login"/>
```

La vista mi_pagina_error_login.jsp:

```
<body>
<h2>Esta es mi_pagina_error_login.jsp. Indica que el login ha fallado. </h2>
<a href="index.jsp">Ir a index.jsp</a></p>
</body>
```

Página de acceso denegado 403

También es posible indicar la pagina de acceso prohibido cuando el usuario no tiene los permisos necesarios:

```
<http auto-config="true">
  <access-denied-handler error-page="/mi_pagina_403" />
  <intercept-url pattern="/miapp/admin*"
    access="hasRole('ROLE_SUPERVISOR')" />
  <intercept-url pattern="/miapp*"
    access="hasRole('ROLE_USER')" />
  <form-login login-page="/mi_pagina_login"
    default-target-url="/mi_pagina_despues_login"
    always-use-default-target='true'
    authentication-failure-url="/mi_pagina_error_login" />
  <logout logout-success-url="/mi_pagina_logout" />
</http>
```

Ejemplo Formulario Login Personalizado

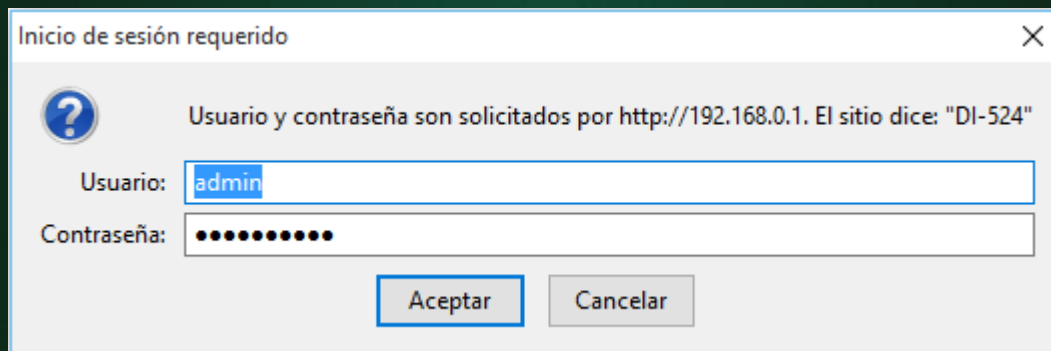
```
<form name='f' action="${pageContext.request.contextPath}/login"
      method='post'>
<table>
  <tr>
    <td>Usuario:</td>
    <td><input type='text' name='username' value=''></td>
  </tr>
  <tr>
    <td>Password:</td>
    <td><input type='password' name='password' /></td>
  </tr>
  <tr>
    <td colspan='2'>
      <input type="hidden" name="${_csrf.parameterName}"
        value="${_csrf.token}" />
      <td colspan='2'>
        <input name="submit" type="submit" value="submit" />
      </td>
    </tr>
</table>
</form>
```

Basic Login



HTTP BASIC authentication

Se refiere a un tipo de autenticación mediante cabeceras http, es decir HTTP Authorization con un sistema de autenticación básica y una codificación username/password simbólico:



A screenshot of a Windows login dialog box titled "Inicio de sesión requerido" (Login required). The dialog box has a question mark icon on the left and a close button (X) on the right. The main text reads: "Usuario y contraseña son solicitados por http://192.168.0.1. El sitio dice: 'DI-524'". Below this text are two input fields: "Usuario:" (Username) with the text "admin" entered, and "Contraseña:" (Password) with a masked password represented by ten dots. At the bottom of the dialog box are two buttons: "Aceptar" (Accept) and "Cancelar" (Cancel).

HTTP BASIC authentication

Si queremos usar autenticación básica (HTTP BASIC authentication) en vez de formulario de login, cambiamos a la siguiente configuración:

```
<http auto-config='true'>  
  <intercept-url pattern="/*" access="hasRole('ROLE_USER')" />  
  <http-basic />  
</http>
```

*Proveedor de
Autenticación
Base de Datos*



Proveedores de Autenticación

En la práctica necesitaremos proveedores más escalables para el manejo y registro de usuarios y sus credenciales, algo más robusto que agregar la información de usuarios en el XML de contexto Spring:

- Ejemplos de proveedores más robustos
 - ❖ Base de datos*
 - ❖ LDAP**
- Podemos usar múltiples <authentication-provider>, los cuales son consultados en el orden en que se listan*

Base de datos como Proveedor de Autenticación

En el ejemplo de abajo, "**securityDataSource**" corresponde al nombre del bean DataSource definido en XML root/application context de spring, que apunta a la base de datos que contiene el esquema estándar de tablas de spring security (users, authorities, groups etc):

```
<authentication-manager>  
  <authentication-provider>  
    <jdbc-user-service data-source-ref="securityDataSource"/>  
  </authentication-provider>  
</authentication-manager>
```


Base de datos como Proveedor de Autenticación

Ejemplo similar al anterior pero con un esquema de tablas que **NO es estándar de Spring** y usando consulta personalizadas de autenticación (de forma explícita) para consultar a los usuarios y roles desde la base de datos:

```
<authentication-manager>
  <authentication-provider>
    <jdbc-user-service data-source-ref="securityDataSource"
      users-by-username-query=
        "select username,password, enabled from users where username=?"
      authorities-by-username-query=
        "select username, role from user_roles where username =? " />
  </authentication-provider>
</authentication-manager>
```

Base de datos como Proveedor de Autenticación

Alternativamente, podemos configurar un bean **JdbcDaoImpl**, una clase DAO que viene como parte de Spring Security (listo para usar) usando el atributo `user-service-ref`:

```
<authentication-manager>
  <authentication-provider user-service-ref='userDetailsService' />
</authentication-manager>

<bean id="userDetailsService"
      class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">
  <property name="dataSource" ref="dataSource" />
</bean>
```

Implementar interface UserDetailsService

También podemos implementar nuestra propia clase UserDetailsService. En el ejemplo de abajo, **miUserService** corresponde al nombre de un bean (en root/application context) el cuál puede ser una clase personalizada DAO/Repository (o Service), propia de nosotros, que implementa la interfaz de UserDetailsService, por ejemplo con Hibernate, pero también se puede hacer con JPA, JDBC o cualquier otra tecnología de persistencia.

```
<authentication-manager>
  <authentication-provider user-service-ref='miUserService'>
    <password-encoder hash="bcrypt" />
  </authentication-provider>
</authentication-manager>

<bean id="miUserService" class="com.formacionbdi.security.
service.MiUserDetailsServiceHibernateImpl">
  <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

Implementar un personalizado bean AuthenticationProvider

En el ejemplo de abajo, **miProveedorAutenticacion** corresponde al nombre de un bean (en root/application context) el cuál implementa la interfaz de AuthenticationProvider para proveer un propio y personalizado Proveedor de autenticación

```
<authentication-manager>  
  <authentication-provider ref='miProveedorAutenticacion'/>  
</authentication-manager>
```

```
<bean id="miProveedorAutenticacion"  
class="com.formacionbdi.security.  
provider.MiAuthenticationProvider">  
</bean>
```


Esquema de Spring Security Base de Datos



Hay varios esquemas de base de datos usado por el framework, pero aquí pretendemos proporcionar un único punto de referencia para todos ellos

Solo tenemos que proveer las tablas necesarias para las áreas de funcionalidad que necesitamos

Esquema de tablas del usuario

- users,
- authorities
- groups
- group_authorities
- group_members

Esquema Login persistente (Remember-me)

- persistent_logins

Esquema ACL (Autorización)

- acl_sid, acl_class
- acl_object_identity
- acl_entry

Esquema de tablas del usuario

La implementación estandar JDBC del bean UserDetailsService (JdbcDaoImpl) requiere tablas para cargar las cuentas de usuarios con sus credenciales/password, estado de la cuenta (activado o desactivado) y una lista de los roles (authorities) para el usuario:

```
create table users(  
  username varchar_ignorecase(50) not null primary key,  
  password varchar_ignorecase(50) not null,  
  enabled boolean not null);
```

```
create table authorities (  
  username varchar_ignorecase(50) not null,  
  authority varchar_ignorecase(50) not null,  
  constraint fk_authorities_users foreign key(username)  
  references users(username));  
create unique index ix_auth_username on authorities  
  (username,authority);
```

Protocolo de Seguridad HTTP/HTTPS



Protocolo de Seguridad HTTPS

Si nuestra aplicación soporta ambos protocolos HTTP y HTTPS, y necesitamos que una particular URLs pueda acceder sobre HTTPS, entonces simplemente los configuramos usamos el atributo **requires-cannel** el elemento **<intercept-url>**

```
<http>
  <intercept-url pattern="/secure/**"
access="hasRole('ROLE_USER')"
    requires-channel="https"/>
  <intercept-url pattern="/**" access="hasRole('ROLE_USER')"
    requires-channel="any"/>
...
</http>
```

Protocolo de Seguridad HTTPS

Tener en cuenta, para que sea completamente seguro, una aplicación no debe utilizar HTTP para todo o hacer switch entre HTTP y HTTPS

Lo ideal es iniciar en HTTPS (con el usuario ingresando una URL HTTPS) y utilizar una conexión segura a lo largo de la aplicación, para evitar cualquier posibilidad de ataques man-in-the-middle

Session Management



Detectar Timeout

Podemos configurar Spring Security para detectar un ID de sesión inválida provocado por un timeout y redirigir al usuario a una determinada URL o página de login

Esto se configura en el elemento session-management

```
<http>
```

```
...
```

```
<session-management invalid-session-url="/sesion-invalida.htm" />
```

```
</http>
```


Detectar Timeout

Hay que tener en cuenta que si usamos el mecanismo para detectar session timeouts, puede reportar falsamente un error si el usuario cierra la sesión y luego se vuelve a conectar sin cerrar el navegador

Esto se debe a que la cookie de sesión no se borra cuando se invalida la sesión y volverá a reenviar incluso si el usuario ha cerrado la sesión, sin embargo hay una opción que nos permite eliminar de forma explícita la cookie JSESSIONID en el cierre de sesión:

```
<http>
```

```
...
```

```
<session-management invalid-session-url="/sesion-invalida.htm" />
```

```
<logout delete-cookies="JSESSIONID" />
```

```
</http>
```

Seguridad en las vistas



Seguridad en las vistas

Spring Security permite implementar condicionales como elementos taglib (etiquetas) para comprobar el acceso.

Basadas en los privilegios del usuario, utilizando las mismas expresiones EL de siempre, pero esta vez relacionadas a la seguridad usando Spring Security JSP Taglib

```
<%@ taglib prefix="sec"
uri="http://www.springframework.org/security/tags"%>
<sec:authorize access="hasRole('ROLE_ADMIN')">
  <a class="btn btn-default" href="form.htm" >Crear Producto </a>
</sec:authorize>
```

```
<sec:authorize access="hasRole('ROLE_USUARIO')">
  <a href="carro/agregar.htm?id=${producto.id}">
    agregar al carro</a>
</sec:authorize>
```

La anotación @Secured



Anotación @Secured

Nuestras clases pueden ser restringida con la anotación @Secured tanto a nivel de clase como a nivel de método

Si se decoran ambos, tanto la clase como el método con @Secured, entonces la restricción del método tendrá prioridad por sobre la clase

Típicamente la usamos para controlar acceso en las clases @Controller y del modelo (@Services, @Repository/Dao)

Anotación @Secured

- *@Secured debe especificar el control de acceso o permiso por parámetro:*

```
@Secured("ROLE_ADMIN")
@RequestMapping(value = "/form.htm")
@ModelAttribute("producto")
public Producto setupForm(@RequestParam("id") int id) {
    Producto producto = null;
    if (id > 0) {
        producto = productoService.findById(id);
    } else {
        producto = new Producto();
    }
    return producto;
}
```

- *En el ejemplo, el usuario debe tener rol Admin (ROLE_ADMIN) para acceder al formulario de producto*

Habilitar Anotación @Secured

Para habilitar y poder utilizar la anotación @Secured, es necesario configurar el archivo de contexto de Spring, por ejemplo si la vamos a utilizar sobre controladores se debe agregar en el archivo de contexto web (servlet-context), mientras que si la vamos a utilizar sobre clases Dao o Services se agrega en el archivo de contexto root (root-context):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
... etc...
xmlns:sec="http://www.springframework.org/schema/security"
... etc...
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security.xsd">

<sec:global-method-security secured-annotations="enabled" />
...
</beans>
```

Dependencias Maven




```
<!-- Spring Security -->
```

```
<dependency>
```

```
  <groupId>org.springframework.security</groupId>
```

```
  <artifactId>spring-security-core</artifactId>
```

```
  <version>4.0.2.RELEASE</version>
```

```
</dependency>
```

```
<dependency>
```

```
  <groupId>org.springframework.security</groupId>
```

```
  <artifactId>spring-security-web</artifactId>
```

```
  <version>4.0.2.RELEASE</version>
```

```
</dependency>
```

```
<dependency>
```

```
  <groupId>org.springframework.security</groupId>
```

```
  <artifactId>spring-security-config</artifactId>
```

```
  <version>4.0.2.RELEASE</version>
```

```
</dependency>
```

```
<!-- Spring Security JSP Taglib -->
```

```
<dependency>
```

```
  <groupId>org.springframework.security</groupId>
```

```
  <artifactId>spring-security-taglibs</artifactId>
```

```
  <version>4.0.2.RELEASE</version>
```

```
</dependency>
```



GRACIAS!