Seguridad

Índice

1 Conceptos básicos..... 2
2 Una configuración mínima para una aplicación web..... 3
3 Proveedores de autentificación: el DAO provider..... 5
4 Seguridad web..... 7
4.1 Autentificación con formulario..... 7
4.2 Autentificación BASIC..... 8
4.3 Recordar los datos del usuario..... 8
4.4 Logout..... 9
4.5 Internacionalización..... 9
5 Seguridad en ejecución de código..... 10
5.1 Seguridad con AOP..... 10
5.2 Seguridad con anotaciones..... 11

En este tema vamos a introducir *Spring Security*, un proyecto "hijo" de Spring que permite controlar de forma declarativa y totalmente configurable la seguridad de nuestra aplicación. Además, nuestro proyecto será totalmente portable entre servidores, a diferencia de la seguridad declarativa estándar de JavaEE, que no lo es en varios aspectos, por ejemplo, la definición de usuarios y roles.

En el momento de escribir estas líneas acaba de aparecer la versión 2 de Spring Security. Esta versión, además de incorporar numerosas mejoras, sobre todo simplifica bastante el proceso de configuración. En versiones anteriores, el proyecto no estaba totalmente integrado debajo del "paraguas" de Spring y se denominaba "Acegi". Por este nombre se puede encontrar en numerosos tutoriales y en varios libros. Si se consulta alguno de ellos, hay que tener en cuenta que se ha cambiado el "packaging" de todo el código para que esté dentro de "org.springframework.security" en lugar de "org.acegisecurity" como anteriormente. Además, como ya hemos dicho, muchas tareas se pueden realizar de forma más directa y sencilla en esta última versión.

1. Conceptos básicos

Lo primero que encuentra un usuario que intenta acceder a una aplicación segura es el mecanismo de **autentificación**. Para autentificarse, el usuario necesita un *principal*, que típicamente es un login y unas *credenciales*, normalmente un password. No siempre se usa login y password. El *principal* y las credenciales pueden proceder por ejemplo de un certificado digital o de otros mecanismos.

En Spring Security, el encargado de gestionar la autentificación es el *Authentication manager*. En versiones anteriores del *framework* era necesario definirlo explícitamente, pero ahora solo lo es si tenemos necesidades especiales de configuración. El *Authentication manager* depende de uno o varios *authentication providers*, que son los que efectivamente obtienen el *principal* y credenciales del usuario. Spring security tiene implementados un gran número de proveedores de autentificación: login con formulario web, login con HTTP BASIC (el navegador muestra una ventana propia para introducir login y password), servidor LDAP, certificados digitales, etc.

La autentificación demuestra que el usuario es quien dice ser, pero queda por ver si tiene permiso de acceso al recurso que ha solicitado. Esto se denomina **control de acceso**. Aquí entra en juego el *Access manager*, que en función de las credenciales, toma la decisión de permitir o no el acceso. Normalmente cada usuario tiene asociado una serie de roles o, como se dice en Spring, de *authorities*, que se asocian a los recursos para permitir o no el acceso.

En una aplicación normalmente solo hay un *access manager*, aunque Spring permite el uso simultáneo de varios, que "por consenso" o "por votación" decidirán si conceden el acceso al recurso

En aplicaciones web sencillas el control de accesos declarativo suele ser una cuestión de

"todo o nada" para un determinado rol. Una forma más avanzada de control de accesos es algo muy común en sistemas operativos: las *Access Control Lists* (ACL) que especifican qué operaciones (acceso/modificación/borrado) puede realizar cada usuario sobre cada recurso. Las aplicaciones con requerimientos de seguridad avanzados pueden asignar a cada recurso un ACL que controlará Spring Security, lo que proporciona una enorme flexibilidad de configuración.

Hay otros tipos de gestores de seguridad en Spring Security, como los *run-as managers*, que permiten ejecutar ciertas tareas cambiando temporalmente el rol del usuario (al estilo de `su` en UNIX) o los *after invocation managers*, que controlan que todo es correcto *después* de acceder al recurso. No obstante, quedan fuera del ámbito de estos apuntes. Aunque la documentación de Spring Security no es tan exhaustiva como la del propio Spring, es bastante aceptable y pueden consultarse en ella todos estos conceptos.

2. Una configuración mínima para una aplicación web

La configuración de la versión 1 de Spring Security se realizaba definiendo diversos *beans* en el XML de Spring. Recordemos que para definir un bean en XML necesitamos la clase que lo implementa y pasarle los valores de las propiedades. La versión 2 incorpora una serie de etiquetas XML propias que definen estos beans de manera implícita y permiten realizar una configuración mucho más concisa. Las aplicaciones con requerimientos de seguridad menos convencionales tendrán que recurrir a la definición directa de los beans.

Veamos una configuración de seguridad mínima para una aplicación web. Spring Security usa filtros de servlets de manera extensiva, por ello lo primero es declararlos en el fichero `web.xml`. Lo más sencillo es usar un filtro de la clase `DelegatingFilterProxy`, que hará de interfaz entre el mecanismo estándar de filtros y los beans de Spring

```
(fragmento del web.xml...)

<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>
```

Aviso:

El nombre del filtro debe ser `springSecurityFilterChain`, ya que es lo que espera la configuración estándar, aunque por supuesto esto es modificable con algo de configuración adicional.

La configuración de la seguridad se hace en el fichero de definición de beans de Spring (`applicationContext.xml` o similar)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:security="http://www.springframework.org/schema/security"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-2.0.xsd">


    <security:http auto-config="true">
        <security:intercept-url pattern="/admin/**"
access="ROLE_ADMIN"/>
        <security:intercept-url pattern="/**"
access="ROLE_REGISTRADO, ROLE_ADMIN"/>
    </security:http>

    <security:authentication-provider >
        <security:user-service>
            <security:user name="spring" password="spring"
                authorities="ROLE_REGISTRADO" />
        </security:user-service>
    </security:authentication-provider>
</beans>
```

Aquí hemos usado el prefijo `security` para destacar que son etiquetas de Spring Security, aunque por supuesto esto es arbitrario y configurable a través del `xmlns` de la etiqueta raíz del XML.

La etiqueta `http` establece cuáles son las URLs protegidas. Cada `intercept-url` especifica un patrón de URLs, con la sintaxis que usa ant para los paths, y los roles que pueden acceder al recurso. En este caso, para acceder a cualquier URL ("/**") hay que tener rol `ROLE_REGISTRADO` o `ROLE_ADMIN`. Cuando hay varios patrones de URL se debe poner siempre primero el más específico, ya que ante una solicitud HTTP Spring aplicará la protección del primer `intercept-url` que encaje. Así, en el ejemplo anterior, dentro del directorio "admin" solo se podría acceder siendo `ROLE_ADMIN`.

Aviso:

Por defecto, **los nombres de los roles deben comenzar por "`ROLE_`"**, otros nombres no serán considerados como válidos. Consultar la documentación para ver cómo cambiar dicho prefijo.

El atributo `auto-config="true"` activa por defecto los servicios de autentificación BASIC, autentificación a través de formulario generado por Spring, gestión de logout y uso de cookies para "recordar" la autentificación. Iremos viendo dichos servicios con más detalle en los siguientes apartados.

Por otro lado, hay que especificar qué proveedor de autentificación usaremos. Esto se

hace mediante la etiqueta `authentication-provider`. El proveedor de autentificación es el responsable de almacenar y comprobar *principal*, credenciales y roles de cada usuario. Para hacer pruebas, lo más rápido es colocar esta información directamente en el XML, como hemos hecho en el ejemplo. Cada usuario viene identificado por un `user`, con su login, password y lista de roles separados por comas.

Si desplegamos una aplicación web basada en la configuración anterior e intentamos acceder a cualquier URL, el resultado será un formulario de login automáticamente generado por Spring:

login form

Podemos hacer pruebas con este formulario, aunque lo habitual será definir el nuestro propio. Obsérvese que automáticamente se ofrece la posibilidad de "recordar" el login, usando cookies.

3. Proveedores de autentificación: el DAO provider

Spring ofrece una variedad bastante amplia de proveedores de autentificación, que obtendrán el principal y credenciales de distintas fuentes. Lo más habitual en aplicaciones web es almacenar logins y passwords en la base de datos, lo que permite modificarlos de manera sencilla. En Spring se usa un *DAO authentication provider* para esta tarea. Hay diversas clases que nos van a simplificar la tarea de implementar este DAO, de las que la más sencilla de usar es `JdbcDaoImpl` gracias a la que no tenemos que implementar código, tan solo configurar ciertas propiedades. En nuestro XML cambiaríamos el anterior `authentication-provider` por uno nuevo:

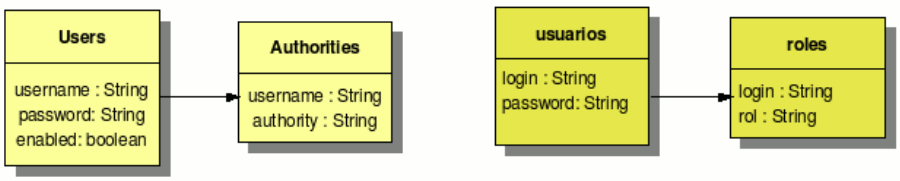
```
<security:authentication-provider
user-service-ref="miJDBCUserDetails"/>

<bean id="miJDBCUserDetails"
class="org.springframework.security.userdetails.jdbc.JdbcDaoImpl">
   <property name="dataSource" ref="miDataSource"/>

    <!-- ¡¡¡todavía nos faltan algunas propiedades!!! -->
    ...
</bean>
```

Donde definimos un bean de la clase `JdbcDaoImpl`, le damos un nombre arbitrario y usamos ese nombre en el atributo `user-service-ref` de la etiqueta que define el proveedor de autentificación. La propiedad `dataSource` se usa para conectar con la base de datos donde se almacenan los datos de los usuarios.

No obstante, todavía queda una cuestión importante por configurar. Por defecto, `JdbcDaoImpl` supone que los login, password y roles están almacenados siguiendo un cierto esquema de base de datos. Lo más habitual es que no coincida con el de nuestra base de datos. En la siguiente figura se muestra a la izquierda el esquema de base de datos que presupone `JdbcDaoImpl` y a la derecha el hipotético esquema de nuestra aplicación.



esquemas de BD

`JdbcDaoImpl` usa una consulta SQL predefinida para obtener login y password de un usuario y otra para obtener los roles asociados. Las dos consultas por supuesto presuponen el esquema anterior. Lo que tendremos que hacer es suministrar consultas propias que devuelvan los resultados con los mismos nombres.

En primer lugar, para comprobar el password se hace:

```
SELECT username, password, enabled
  FROM users
  WHERE username = ?
```

Donde el campo `enabled`, del que carece nuestra base de datos, indica si el usuario está o no activado. Con nuestro esquema, para devolver los mismos resultados que la consulta anterior, haríamos:

```
SELECT login as username, password, true as enabled
  FROM usuarios
  WHERE login=?
```

Por otro lado, para obtener los roles (authorities) de un usuario, se hace:

```
SELECT username, authority
  FROM authorities
  WHERE username = ?
```

Con nuestro esquema de base de datos, haríamos:

```
SELECT login as username, rol as authority
   FROM roles
```

```
   WHERE login=?
```

Estas consultas se modifican a través de las propiedades `usersByUsernameQuery` y `authoritiesByUsernameQuery` de `JdbcDaoImpl`. Así, nuestro XML quedaría:

```
<security:authentication-provider
user-service-ref="miJDBCUserDetails"/>

<bean id="miJDBCUserDetails"
class="org.springframework.security.userdetails.jdbc.JdbcDaoImpl">
      <property name="dataSource" ref="miDataSource"/>
        <property name="usersByUsernameQuery"
                value="SELECT login as username, password, true as
enabled
                       FROM usuarios WHERE login=?"/>
      <property name="authoritiesByUsernameQuery"
                value="SELECT login as username, rol as authority
                       FROM roles WHERE login=?"/>
</bean>
```

Donde se destaca en negrita la nueva configuración introducida.

Como ya se ha comentado, Spring Security incorpora diversos proveedores de autentificación "listos para usar", basados en tecnologías tan diversas como LDAP, JAAS, sistemas single sign-on como CAS o OpenId,... No obstante, su uso y configuración queda fuera del ámbito de estos apuntes.

4. Seguridad web

Hemos visto que la etiqueta `http` con el atributo `auto-config="true"` configura automáticamente una serie de servicios de seguridad para aplicaciones web. Vamos a ver cómo afinar la configuración manualmente.

4.1. Autentificación con formulario

La mayoría de aplicaciones web usan un formulario HTML para que el usuario introduzca su login y password. Hemos visto que por defecto Spring crea automáticamente este formulario, pero lo habitual será que lo hagamos nosotros para poder darle el "look and feel" de la aplicación. Esto se consigue con la etiqueta `form-login`:

```
<security:http auto-config="true">
     <security:intercept-url pattern="/index.jsp" filters="none"/>
    <security:intercept-url pattern="/**"
                            access="ROLE_REGISTRADO, ROLE_ADMIN"/>
     <security:form-login login-page="/index.jsp"
                          default-target-url="/principal.jsp" />
</security:http>
```

Con el atributo `login-page` se especifica la página que contiene el formulario de login y con `default-target-url` la dirección a la que se saltará por defecto. En este punto, la autentificación con formulario de Spring se diferencia ligeramente de la seguridad declarativa estándar de JavaEE. En el estándar no se suele saltar directamente a la página de login, sino que esta se muestra automáticamente cuando el usuario intenta acceder a un recurso protegido. En Spring nada nos impide acceder directamente a la página de login, ya que se nos redirigirá una vez hecho login a la página indicada por `default-target-url`. Independientemente de ello, por supuesto, cuando en Spring se intenta acceder a un recurso protegido también "salta" la página de login.

Nótese que la existencia de la página de login nos obliga a desprotegerla para que los usuarios puedan acceder a ella. Esto se hace con el atributo `filters="none"` en el `intercept-url`.

La página de login contendrá un formulario HTML cuyos campos deben tener un nombre estándar, al estilo de los que se usan en seguridad declarativa JavaEE:

```
<form action="j_spring_security_check">
    login: <input type="text" name="j_username"/> <br/>
    password: <input type="text" name="j_password"/> <br/>
    <input type="submit" value="Entrar"/>
</form>
```

4.2. Autentificación BASIC

En la autentificación BASIC, el navegador muestra una ventana de tipo "popup" en la que introducir login y password. En realidad, la mayor utilidad de este mecanismo es para el acceso con un cliente de escritorio, ya que la forma de envío de login y password al servidor es sencilla de implementar y no requiere el mantenimiento de sesiones, a diferencia del login con formulario.

Para usar autentificación BASIC, simplemente colocaríamos la etiqueta `http-basic` en el XML:

```
<security:http auto-config="true">
    <security:intercept-url pattern="/**" access="ROLE_REGISTRADO"
/>
    <security:http-basic/>
</security:http>
```

4.3. Recordar los datos del usuario

La mayoría de aplicaciones web ofrecen la posibilidad de "ahorrarnos" el login en sucesivas visitas si accedemos desde la misma máquina. Esto se hace guardando en una cookie un *token* de autentificación, que asegura que en algún momento nos hemos autentificado. Spring llama a esta característica "remember-me" y se configura con la

etiqueta del mismo nombre. Normalmente no es necesario ponerla porque el atributo `auto-config="true"` la incluye por defecto. No obstante, si queremos configurar los detalles del servicio debemos ponerla explícitamente.

Por defecto, el campo de formulario que se asocia a esta característica debe llamarse `_spring_security_remember_me`. De este modo, nuestro formulario de login quedaría:

```
<form action="j_spring_security_check">
    login: <input type="text" name="j_username"/> <br/>
    password: <input type="text" name="j_password"/> <br/>
    <input type="checkbox" name="_spring_security_remember_me"/>
                  Recordar mi usuario y password <br/>
    <input type="submit" value="Entrar"/>
</form>
```

4.4. Logout

Spring nos ofrece un servicio de logout que se encarga de invalidar automáticamente la sesión HTTP y, si lo deseamos, redirigir al usuario a una página de "salida". Este servicio se configura con la etiqueta `logout`:

```
<security:http auto-config="true">
   ...
   <security:logout logout-url="/logout.jsp"
logout-success-url="/adios.jsp"/>
</security:http>
```

El atributo `logout-url` indica qué URL "disparará" el proceso. Por tanto, para que el usuario pueda hacer logout bastará con un enlace a esta URL en cualquier página. Por defecto la URL de logout es `/j_spring_security_logout`. Con `logout-success-url` indicamos a qué página se saltará tras invalidar la sesión. Por defecto es "/".

4.5. Internacionalización

Todas las excepciones que saltan cuando se produce algún problema de autorización tienen los mensajes externalizados en ficheros `.properties`. Podemos traducirlos tomando como base el fichero `messages.properties` que viene incluido en el paquete `org.springframework.security` del JAR de Spring Security. Basta con crear un fichero propio en el que se traduzcan los mensajes de error (hay del orden de 50) y decirle a Spring Security que lo cargue, usando la siguiente configuración:

```
...
<bean id="messageSource"
class="org.springframework.security.SpringSecurityMessageSource">
        <property name="basename"
value="es/ua/jtech/spring/securityMessages"/>
</bean>
...
```

Donde estamos suponiendo que el fichero `securityMessages` está en el mismo directorio que los fuentes java que están en el paquete `es.ua.jtech.spring`. Recordar que el nombre real del fichero se obtiene concatenando a este nombre el *locale* y la extensión. Es decir, para español el fichero se llamará `securityMessages_es.properties`

De este modo, podemos personalizar el mensaje que Spring nos devuelve cuando el login o el password son incorrectos. Primero debemos traducir el mensaje que está bajo la clave `AbstractUserDetailsAuthenticationProvider.badCredentials` y ya lo podemos mostrar teniendo en cuenta que Spring guarda la excepción con el error como un atributo de sesión bajo la clave definida a través de la constante `AbstractProcessingFilter.SPRING_SECURITY_LAST_EXCEPTION_KEY`. El siguiente *scriptlet* haría este trabajo:

```
<div style="color:red">
 <%
  Exception e = (Exception) session.getAttribute(
AbstractProcessingFilter.SPRING_SECURITY_LAST_EXCEPTION_KEY);
  if (e!=null) out.print(e.getMessage());
 %>
</div>
```

5. Seguridad en ejecución de código

Para mayor seguridad podemos controlar los permisos al ejecutar cualquier método. Los métodos restringidos se pueden especificar de dos formas: con anotaciones en el código fuente o con AOP. En ambos casos, la configuración se hace a través de la etiqueta `global-method-security`

En una aplicación web, el intento de ejecutar código sin permiso acabará generando una respuesta HTTP con código 403 (acceso denegado), gracias a los filtros de Spring. Esto nos permite tratar de manera uniforme las denegaciones de acceso sean por URL o por código.

5.1. Seguridad con AOP

La principal ventaja de esta forma de trabajar es que podemos cambiar la seguridad sin necesidad de tocar una sola línea de código. En el XML, con la etiqueta `protect-pointcut` podemos especificar un pointcut con sintaxis AspectJ e indicar a qué roles les será permitido el acceso.

```
...
<security:global-method-security>
    <security:protect-pointcut
                    expression="execution(* eliminarUsuario(..))"
```

```
                        access="ROLE_ADMIN"/>
</security:global-method-security>
...
```

Así, para ejecutar cualquier método llamado "eliminarUsuario" de cualquier clase, habrá que tener rol `ROLE_ADMIN`.

5.2. Seguridad con anotaciones

En algunos casos puede ser preferible incluir las restricciones de seguridad en el propio código fuente. Spring tiene una anotación propia para ello, `@Secured`, aunque también soporta las anotaciones típicas de EJB (`@RolesAllowed`, ...). Concretamente Spring soporta las anotaciones especificadas en el JSR-250 (de ahí que en temas anteriores viéramos también el uso de `@Resource`). Evidentemente, usar las anotaciones estándar aumentará la portabilidad de nuestro código, por lo que es el estilo recomendado.

En este caso, el XML de configuración quedaría:

```
...
<security:global-method-security jsr250-annotations="enabled"/>
...
```

Si quisiéramos usar la anotación `@Secured` deberíamos incluir el atributo `secured-annotations="enabled"`. Ambos tipos de anotaciones pueden usarse simultáneamente.

En el código a proteger, escribiríamos:

```
@RolesAllowed("ROLE_ADMIN")
public void eliminarUsuario {
      ...
}
```

Al igual que en EJBs, si colocamos la anotación delante de la clase, estamos protegiendo todos sus métodos.