Seguridad en aplicaciones web

Índice

1 Autentificaciones....2
2 Seguridad del servidor....3
2.1 Políticas de seguridad....3
2.2 Autentificación en Tomcat: realms....4
3 Seguridad en aplicaciones web....7
3.1 Tipologías de seguridad y autentificación....7
3.2 Autentificación basada en formularios....8
3.3 Autentificación basic....11
3.4 Anotaciones relacionadas con la seguridad....12
3.5 Acceso a la información de seguridad....14

Cuando hablamos de seguridad en aplicaciones web podemos distinguir dos aspectos: por un lado la seguridad del servidor, es decir, cómo configurar las restricciones de seguridad que va a tener la propia aplicación (qué operaciones va a poder realizar y qué operaciones no, independientemente del usuario que esté accediendo). Por otro, como autentificar a los usuarios y cómo aplicar las restricciones a las operaciones que puedan realizar dentro de la aplicación. Hablaremos aquí de estos dos aspectos, centrándonos sobre todo en el segundo.

1. Autentificaciones

Veremos ahora algunos mecanismos que pueden emplearse con HTTP para autentificar (validar) al usuario que intenta acceder a un determinado recurso.

Autentificaciones elementales

El protocolo HTTP incorpora un mecanismo de autentificación básico (**basic**) basado en cabeceras de autentificación para solicitar datos del usuario (el servidor) y para enviar los datos del usuario (el cliente), de forma que comprobando la exactitud de los datos se permitirá o no al usuario acceder a los recursos. Esta autentificación no proporciona confidencialidad ni integridad, sólo se emplea una codificación Base64.

Una variante de esto es la autentificación **digest**, donde, en lugar de transmitir el password por la red, se emplea un password codificado. Dicha codificación se realiza tomando el login, password, URI, método HTTP y un valor generado aleatoriamente, y todo ello se combina utilizando el método de encriptado MD5, muy seguro. De este modo, ambas partes de la comunicación conocen el password, y a partir de él pueden comprobar si los datos enviados son correctos. Sin embargo, algunos servidores no soportan este tipo de autentificación.

Certificados digitales y SSL

Las aplicaciones reales pueden requerir un nivel de seguridad mayor que el proporcionado por las autentificaciones *basic* o *digest*. También pueden requerir confidencialidad e integridad aseguradas. Todo esto se consigue mediante los **certificados digitales**.

- **Criptografía de clave pública**:La clave de los certificados digitales reside en la **criptografía de clave pública**, mediante la cual cada participante en el proceso tiene dos claves, que le permiten encriptar y desencriptar la información. Una es la clave pública, que se distribuye libremente. La otra es la clave privada, que se mantiene secreta. Este par de claves es asimétrico, es decir, una clave sirve para desencriptar algo codificado con la otra. Por ejemplo, supongamos que A quiere enviar datos encriptados a B. Para ello, hay dos posibilidades:
  - A toma la clave pública de B, codifica con ella los datos y se los envía. Luego B utiliza su clave privada (que sólo él conoce) para desencriptar los datos.

    - A toma su clave privada, codifica los datos y se los envía a B, que toma la clave pública de A para descodificarlos. Con esto, B sabe que A es el remitente de los datos.

  El encriptado con clave pública se basa normalmente en el algoritmo RSA, que emplea números primos grandes para obtener un par de claves asimétricas. Las claves pueden darse con varias longitudes; así, son comunes claves de 1024 o 2048 bits.
- **Certificados digitales**: Lógicamente, no es práctico teclear las claves del sistema de clave pública, pues son muy largas. Lo que se hace en su lugar es almacenar estas claves en disco en forma de **certificados digitales**. Estos certificados pueden cargarse por muchas aplicaciones (servidores web, navegadores, gestores de correo, etc).

  Notar que con este sistema se garantiza la **confidencialidad** (porque los datos van encriptados), y la **integridad** (porque si los datos se desencriptan bien, indica que son correctos). Sin embargo, no proporciona **autentificación** (B no sabe que los datos se los ha enviado A), a menos que A utilice su clave privada para encriptar los datos, y luego B utilice la clave pública de A para desencriptarlos. Así, B descodifica primero el mensaje con su clave privada, y luego con la pública de A. Si el proceso tiene éxito, los datos se sabe que han sido enviados por A, porque sólo A conoce su clave privada.
- **SSL**: SSL (*Secure Socket Layer*) es una capa situada entre el protocolo a nivel de aplicación (HTTP, en este caso) y el protocolo a nivel de transporte (TCP/IP). Se encarga de gestionar la seguridad mediante criptografía de clave pública que encripta la comunicación entre cliente y servidor. La versión 2.0 de SSL (la primera mundialmente aceptada), proporciona autentificación en la parte del servidor, confidencialidad e integridad. Funciona como sigue:
    - Un cliente se conecta a un lugar seguro utilizando el protocolo HTTPS (HTTP + SSL). Podemos detectar estos sitios porque las URLs comienzan con `https://`
    - El servidor envía su clave pública al cliente.
    - El navegador comprueba si la clave está firmada por un certificado de confianza. Si no es así, pregunta al cliente si quiere confiar en la clave proporcionada.

  SSL 3.0 proporciona también soporte para certificados y autentificación del cliente. Funcionan de la misma forma que los explicados para el servidor, pero residiendo en el cliente.

2. Seguridad del servidor

2.1. Políticas de seguridad

Si los usuarios del servidor pueden colocar en él sus propias aplicaciones web es necesario asegurar que dichas aplicaciones no van a comprometer la seguridad de las demás, del servidor o del propio sistema. Aunque confiemos en la buena fe del que

desarrolla las aplicaciones, alguien podría aprovechar un "bug" en una de ellas para efectuar operaciones no permitidas.

Tomcat utiliza los mecanismos estándar de seguridad de Java para asegurar que las clases ejecutadas en el servidor cumplen una serie de restricciones. Dichas restricciones se definen en el fichero de políticas de seguridad `conf/catalina.policy`. No obstante, por defecto estas restricciones no están activadas salvo que se arranque Tomcat con la opción `-security`.

Políticas de seguridad por defecto

Si se observa el contenido del fichero `conf/catalina.policy` se verá que por defecto, a las clases que componen Tomcat y a las pertenecientes al JDK se les asignan todos los permisos posibles. A continuación se muestra un extracto de dicho fichero:

```
...
// Permisos para las clases del JDK
// (están dentro de JAVA_HOME)
grant codeBase "file:${java.home}/lib/-" {
        permission java.security.AllPermission;
};
...
// Permisos para las clases de Tomcat
// (están dentro de CATALINA_HOME)
grant codeBase "file:${catalina.home}/bin/bootstrap.jar" {
        permission java.security.AllPermission;
};
...
// Permisos para aplicaciones web
grant {
        // Required for JNDI lookup of named JDBC DataSource's and
        // javamail named MimePart DataSource used to send mail
        permission java.util.PropertyPermission "java.home", "read";
        ...
```

Como puede observarse si se examina detenidamente el fichero, a las clases que componen una aplicación web cualquiera se les dan permisos adecuados para poder acceder a JNDI, leer ciertas propiedades del sistema, etc.

Cambiar las políticas de seguridad

Cambiando el fichero de políticas podemos asignar permisos especiales a ciertas aplicaciones web que lo requieran, manteniendo las restricciones en las demás. Por ejemplo, para dar ciertos permisos a la aplicación j2ee habrá que introducir un código similar al siguiente en el fichero de políticas

```
grant codeBase "file:${catalina.home}/webapps/j2ee/-" {
        permission ...
}
```

2.2. Autentificación en Tomcat: realms

Cualquier aplicación medianamente compleja tendrá que autentificar a los usuarios que acceden a ella, y en función de quiénes son, permitirles o no la ejecución de ciertas

operaciones

Los mecanismos de autentificación en Tomcat se basan en el concepto de `realm`. Un `realm` es un conjunto de usuarios, cada uno con un *password* y uno o más *roles*. Los roles determinan qué permisos tiene el usuario en una aplicación web (esto es configurable en cada aplicación a través del descriptor de despliegue, `web.xml`).

Tomcat proporciona distintas implementaciones para los realms, que básicamente se diferencian en dónde están almacenados los datos de *logins*, *passwords* y *roles*. En la configuración de Tomcat por defecto, dichos datos están en el fichero `conf/tomcat-users.xml`, pero pueden almacenarse en una base de datos con JDBC, tomarse de un directorio LDAP u obtenerse mediante JAAS, el API estándar de Java para autentificación.

Los *realms* se definen en el fichero de configuración `server.xml` introduciendo el elemento `Realm` con un atributo `className` que indique la implementación que deseamos utilizar. El resto de atributos dependen de la implementación en concreto. Podemos definir *realms* en distintos puntos del fichero, de manera que afecten a todo el servidor, a un *host* o solo a una aplicación. En cada caso, siempre se utilizará el *realm* aplicable más bajo en la "jerarquía de configuración", de modo que si se configura para una aplicación en concreto, el general deja de utilizarse.

UserDatabaseRealm

Es la implementación que utiliza por defecto la distribución de Tomcat, y como se ha comentado, toma los datos de un fichero. Este fichero se lee cuando arranca el servidor, de modo que cualquier cambio en el mismo requiere el rearranque de Tomcat para tener efecto. La definición del *realm* en el fichero de configuración se realiza de la siguiente manera:

```
<Realm className="org.apache.catalina.realm.UserDatabaseRealm"
       debug="0" resourceName="UserDatabase"/>
```

Donde el atributo `resourceName` determina el recurso del que se obtiene la información. Este está definido dentro de GlobalNamingResources para que apunte al fichero `conf/tomcat-users.xml`.

Dicho fichero tiene un formato similar al siguiente:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<tomcat-users>
        <role rolename="usuario"/>
        <role rolename="admin"/>
        <user username="pepe" password="pepepw" roles="usuario"/>
        <user username="manuel" password="manolo" roles="admin"/>
        <user username="toni" password="toni" roles="usuario, admin"/>
</tomcat-users>
```

Así, por ejemplo, para un recurso (URL) al que sólo puedan acceder roles de tipo *admin*, podrían acceder los usuarios *manuel* y *toni*. Notar también que los passwords están visibles en un fichero de texto fácilmente accesible por casi cualquiera, con lo que no es

una buena forma de gestionar los passwords para una aplicación profesional.

JDBCRealm

Esta implementación almacena los datos de los usuarios en una base de datos con JDBC, lo cual es mucho más flexible y escalable que el mecanismo anterior y además no requiere el rearranque de Tomcat cada vez que se cambia algún dato. Para configurar Tomcat con esta implementación de *realm* se puede descomentar uno que ya viene definido en el `server.xml`, a nivel global dentro del elemento `Engine`. Los atributos de este *realm* indican cómo efectuar la conexión JDBC y cuáles son los campos de la base de datos que contienen *logins*, *passwords* y roles.

Atributo	Significado
`className`	clase Java que implementa este *realm*. Debe ser `org.apache.catalina.realm.JDBCRealm`
`connectionName`	nombre de usuario para la conexión JDBC
`connectionPassword`	password para la conexión JDBC
`connectionURL`	URL de la base de datos
`debug`	nivel de depuración. Por defecto 0 (ninguno). Valores más altos indican más detalle.
`digest`	Algoritmo de "digest" (puede ser SHA, MD2 o MD5). Por defecto es `cleartext`
`driverName`	clase Java que implementa el *driver* de la B.D.
`roleNameCol`	nombre del campo que almacena los roles
`userNameCol`	nombre del campo que almacena los *logins*
`userCredCol`	nombre del campo que almacena los *passwords*
`userRoleTable`	nombre de la tabla que almacena la relación entre *login* y roles
`userTable`	nombre de la tabla que almacena la relación entre *login* y *password*

Un ejemplo de etiqueta con este tipo de `Realm` (fichero `conf/server.xml`) sería:

```
<Realm  className="org.apache.catalina.realm.JDBCRealm"
 driverName="com.mysql.jdbc.Driver"
     connectionURL="jdbc:mysql://localhost/authority"
     connectionName="test" connectionPassword="test"
     userTable="users" userNameCol="user_name" userCredCol="user_pass"
     userRoleTable="user_roles" roleNameCol="role_name" />
```

Nota

Una práctica recomendable es cifrar los passwords en nuestra base de datos, guardando sólo su huella digital (*digest*, por ejemplo MD5 o SHA), para mantener la confidencialidad sobre estos datos. Al insertar los datos en la base de datos podemos utilizar funciones de SQL como `md5()` o `sha()`: `insert into usuarios(login, password) values('pepe', sha('passwordpepe'))`. Utilizando el atributo `digest` de `Realm` podemos hacer que en lugar de comparar el password directamente, compruebe su huella digital, para así poder validar los usuarios a pesar de tener almacenada únicamente la huella digital de los password.

3. Seguridad en aplicaciones web

3.1. Tipologías de seguridad y autentificación

Podemos tener básicamente dos motivos para proteger una aplicación web:

- Evitar que usuarios no autorizados accedan a determinados recursos.
- Prevenir que se acceda a los datos que se intercambian en una transferencia a lo largo de la red.

Para cubrir estos agujeros, un sistema de seguridad se apoya en tres aspectos importantes:

- **Autentificación**: medios para identificar a los elementos que intervienen en el acceso a recursos.
- **Confidencialidad**: asegurar que sólo los elementos que intervienen entienden el proceso de comunicación establecido.
- **Integridad**: verificar que el contenido de la comunicación no se modifica durante la transmisión.

Control de la seguridad

Desde el punto de vista de quién controla la seguridad en una aplicación web, existen dos formas de implantación:

- **Seguridad declarativa**: Aquella estructura de seguridad sobre una aplicación que es externa a dicha aplicación. Con ella, no tendremos que preocuparnos de gestionar la seguridad en ningún servlet, página JSP, etc, de nuestra aplicación, sino que el propio servidor Web se encarga de todo. Así, ante cada petición, comprueba si el usuario se ha autentificado ya, y si no le pide login y password para ver si puede acceder al recurso solicitado. Todo esto se realiza de forma transparente al usuario. Mediante el descriptor de la aplicación principalmente (fichero *web.xml* en Tomcat), comprueba la configuración de seguridad que queremos dar.
- **Seguridad programada**: Mediante la seguridad programada, son los servlets y páginas JSP quienes, al menos parcialmente, controlan la seguridad de la aplicación.

En este tema veremos la seguridad declarativa, que es la que puede configurar el

administrador del servidor web, dejando para más adelante la seguridad programada

Autentificación

Tenemos distintos tipos de autentificación que podemos emplear en una aplicación web:

- **Autentificación *basic***: Con HTTP se proporciona un mecanismo de autentificación básico, basado en cabeceras de autentificación para solicitar datos del usuario (el servidor) y para enviar los datos del usuario (el cliente). Esta autentificación no proporciona confidencialidad ni integridad, sólo se emplea una codificación Base64.
- **Autentificación *digest***: Existe una variante de lo anterior, la autentificación **digest**, donde, en lugar de transmitir el password por la red, se emplea un password codificado utilizando el método de encriptado MD5. Sin embargo, algunos servidores no soportan este tipo de autentificación.
- **Autentificación basada en formularios**: Con este tipo de autentificación, el usuario introduce su login y password mediante un formulario HTML (y no con un cuadro de diálogo, como las anteriores). El fichero descriptor contiene para ello entradas que indican la página con el formulario de autentificación y una página de error. Tiene el mismo inconveniente que la autentificación *basic*: el password se codifica con un mecanismo muy pobre.
- **Certificados digitales y SSL**: Con HTTP también se permite el uso de SSL y los certificados digitales, apoyados en los sistemas de criptografía de clave pública. Así, la capa SSL, trabajando entre TCP/IP y HTTP, asegura, mediante criptografía de clave pública, la integridad, confidencialidad y autentificación.

3.2. Autentificación basada en formularios

Veremos ahora con más profundidad la autentificación basada en formularios comentada anteriormente. Esta es la forma más comúnmente usada para imponer seguridad en una aplicación, puesto que se emplean **formularios HTML**.

El programador emplea el descriptor de despliegue para identificar los recursos a proteger, e indicar la página con el formulario a mostrar, y la página con el error a mostrar en caso de autentificación incorrecta. Así, un usuario que intente acceder a la parte restringida es redirigido automáticamente a la página del formulario, si no ha sido autentificado previamente. Si se autentifica correctamente accede al recurso, y si no se le muestra la página de error. Todo este proceso lo controla el servidor automáticamente.

Este tipo de autentificación no se garantiza que funcione cuando se emplea reescritura de URLs en el seguimiento de sesiones. También podemos incorporar SSL a este proceso, de forma que no se vea modificado el funcionamiento aparente del mismo.

Para utilizar la autentificación basada en formularios, se siguen los pasos que veremos a continuación. Sólo el primero es dependiente del servidor que se utilice.

1. Establecer los logins, passwords y roles

En este paso definiríamos un *realm* del modo que se ha explicado en apartados anteriores.

2. Indicar al servlet que se empleará autentificación basada en formularios, e indicar las páginas de formulario y error.

Se coloca para ello una etiqueta `<login-config>` en el descriptor de despliegue. Dentro, se emplean las subetiquetas:

- `<auth-method>` que en general puede valer:
  - `FORM`: para autentificación basada en formularios (como es el caso)
  - `BASIC`: para autentificación BASIC
  - `DIGEST`: para autentificación DIGEST
  - `CLIENT-CERT`: para SSL
- `<form-login-config>` que indica las dos páginas HTML (la del formulario y la de error) con las etiquetas:
  - `<form-login-page>` (para la de autentificación)
  - `<form-error-page>` (para la página de error).

Por ejemplo, podemos tener las siguientes líneas en el descriptor de despliegue:

```
<web-app>
        ...
        <login-config>
                <auth-method>FORM</auth-method>
                <form-login-config>
                        <form-login-page>
                                /login.jsp
                        </form-login-page>
                        <form-error-page>
                                /error.html
                        </form-error-page>
                </form-login-config>
        </login-config>
        ...
</web-app>
```

3. Crear la página de login

El formulario de esta página debe contener campos para introducir el login y el password, que deben llamarse *j_username* y *j_password*. La acción del formulario debe ser *j_security_check*, y el METHOD = POST (para no mostrar los datos de identificación en la barra del explorador). Por ejemplo, podríamos tener la página:

```
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<body>
        <form action="j_security_check" METHOD="POST">
        <table>
        <tr>
                <td>
                        Login:<input type="text" name="j_username"/>
                </td>
        </tr>
```

```
        <tr>
                <td>
                        Password:<input type="text" name="j_password"/>
                </td>
        </tr>
        <tr>
                <td>
                        <input type="submit" value="Enviar"/>
                </td>
        </tr>
        </table>
        </form>
</body>
</html>
```

4. Crear la página de error

La página puede tener el mensaje de error que se quiera. Ante fallos de autentificación, se redirigirá a esta página con un código 401. Un ejemplo de página sería:

```
<!DOCTYPE HTML PUBLIC
"-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<body>
        <h1>ERROR AL AUTENTIFICAR USUARIO</h1>
</body>
</html>
```

5. Indicar qué direcciones deben protegerse con autentificación

Para ello utilizamos etiquetas <security-constraint> en el descriptor de despliegue. Dichos elementos debe ir inmediatamente antes de *<login-config>*, y utilizan las subetiquetas:

- `<display-name>` para dar un nombre identificativo a emplear (opcional)
- `<web-resource-collection>` para especificar los patrones de URL que se protegen (requerido). Se permiten varias entradas de este tipo para especificar recursos de varios lugares. Cada uno contiene:
  - Una etiqueta `<web-resource-name>` que da un nombre identificativo arbitrario al recurso o recursos
  - Una etiqueta `<url-pattern>` que indica las URLs que deben protegerse
  - Una etiqueta `<http-method>` que indica el método o métodos HTTP a los que se aplicará la restricción (opcional)
  - Una etiqueta `<description>` con documentación sobre el conjunto de recursos a proteger (opcional)

  NOTA: este modo de restricción se aplica sólo cuando se accede al recurso directamente, no a través de arquitecturas MVC (Modelo-Vista-Controlador), con un *RequestDispatcher*. Es decir, si por ejemplo un servlet accede a una página JSP protegida, este mecanismo no tiene efecto, pero sí cuando se intenta a acceder a la página JSP directamente.
- `<auth-constraint>` indica los roles de usuario que pueden acceder a los recursos indicados (opcional) Contiene:

- Uno o varios subelementos `<role-name>` indicando cada rol que tiene permiso de acceso. Si queremos dar permiso a todos los roles, utilizamos una etiqueta *`<role-name>*</role-name>`*.
- Una etiqueta `<description>` indicando la descripción de los mismos.

En teoría esta etiqueta es opcional, pero omitiéndola indicamos que ningún rol tiene permiso de acceso. Aunque esto puede parecer absurdo, recordar que este sistema sólo se aplica al acceso directo a las URLs (no a través de un modelo MVC), con lo que puede tener su utilidad.

Añadimos alguna dirección protegida al fichero que vamos construyendo:

```
<web-app>
        <security-constraint>
                <web-resource-collection>
                        <web-resource-name>
                                Prueba
                        </web-resource-name>
                        <url-pattern>
                                /prueba/*
                        </url-pattern>
                </web-resource-collection>
                <auth-constraint>
                        <role-name>admin</role-name>
                        <role-name>subadmin</role-name>
                </auth-constraint>
        </security-constraint>

        <login-config>
                ...
</web-app>
```

En este caso protegemos todas las URLs de la forma *`http//host/ruta_aplicacion/prueba/*`*, de forma que sólo los usuarios que tengan roles de *admin* o de *subadmin* podrán acceder a ellas.

3.3. Autentificación basic

El método de autentificación basada en formularios tiene algunos inconvenientes: si el navegador no soporta cookies, el proceso tiene que hacerse mediante reescritura de URLs, con lo que no se garantiza el funcionamiento.

Por ello, una alternativa es utilizar el modelo de autentificación *basic* de HTTP, donde se emplea un cuadro de diálogo para que el usuario introduzca su login y password, y se emplea la cabecera *Authorization* de petición para recordar qué usuarios han sido autorizados y cuáles no. Una diferencia con respecto al método anterior es que es difícil entrar como un usuario distinto una vez que hemos entrado como un determinado usuario (habría que cerrar el navegador y volverlo a abrir).

Al igual que en el caso anterior, podemos utilizar SSL sin ver modificado el resto del esquema del proceso.

El método de autentificación *basic* consta de los siguientes pasos:

1. Establecer los logins, passwords y roles

Este paso es exactamente igual que el visto para la autentificación basada en formularios.

2. Indicar al servlet que se empleará autentificación BASIC, y designar los dominios

Se utiliza la misma etiqueta `<login-config>` vista antes, pero ahora una etiqueta `<auth-method>` con valor BASIC. Se emplea una subetiqueta `<realm-name>` para indicar qué dominio se empleará en la autorización. Por ejemplo:

```
<web-app>
        ...
        <login-config>
                <auth-method>BASIC</auth-method>
                <realm-name>dominio</realm-name>
        </login-config>
        ...
</web-app>
```

3. Indicar qué direcciones deben protegerse con autentificación

Este paso también es idéntico al visto en la autentificación basada en formularios.

3.4. Anotaciones relacionadas con la seguridad

Con la especificación 3.0 de servlets se introduce la posibilidad de configurar los permisos de acceso mediante anotaciones en lugar de en el `web.xml`, lo que hace la configuración menos farragosa y más clara. La anotación principal que utilizaremos es `@ServletSecurity`, que toma dos parámetros:

- `value`: Define la restricción de seguridad, mediante una anotación de tipo `@HttpConstraint`.
- `httpMethodConstraints`: Permite definir una lista de restrincciones sobre los métodos HTTP que pueden acceder al servlet. Se definen mediante una anotación de tipo `@HttpMethodConstraint`. De esta manera no damos una restricción general para todos los métodos, sino que podemos personalizar las restricciones que se aplicarán para cada uno de ellos.

La anotación `@HttpConstraint` acepta los siguientes parámetros:

- `rolesAllowed`: Nos permite definir la lista de roles que pueden acceder al servlet.
- `transportGuarantee`: Puede tomar los valores `TransportGuarantee.NONE` o `TransportGuarantee.CONFIDENTIAL`. Con el segundo de ellos sólo estaremos permitiendo acceder al servlet si la conexión se realiza mediante SSL.
- `value`: Nos permite establecer la política de acceso a llevar a cabo independientemente del rol del usuario. Las opciones son admitir todas las peticiones (`EmptyRoleSemantic.PERMIT`) o denegarlas (`EmptyRoleSemantic.DENY`). Definiremos estas políticas cuando no se especifique una lista de roles.

Un caso común es aquel en el que queremos permitir acceso al servlet a unos roles determinados:

```
@WebServlet("/MiServlet")
@ServletSecurity(@HttpConstraint(rolesAllowed={"rol1","rol2"}))
public class MiServlet extends HttpServlet { ... }
```

Si queremos permitir el acceso a cualquier usuario (aunque no esté autentificado), pero siempre mediante SSL, podemos indicarlo de la siguiente forma:

```
@WebServlet("/MiServlet")
@ServletSecurity(@HttpConstraint(value=EmptyRoleSemantic.PERMIT,
             transportGuarantee=TransportGuarantee.CONFIDENTIAL)
public class MiServlet extends HttpServlet { ... }
```

Como alternativa, podemos definir diferentes restricciones para cada método HTTP. Para ello utilizaremos la anotación `@HttpMethodConstraint`, que podrá tomar los siguientes parámetros:

- `value`: Indica el método para el que vamos a definir las restricciones de acceso. Por ejemplo `"GET"`, `"POST"`, etc.
- `rolesAllowed`: Define la lista de roles que pueden acceder al servlet mediante el método indicado.
- `transportGuarantee`: Indica si sólo se permite acceder al método indicado mediante SSL. Se define de la misma forma que en el caso de `@HttpConstraint`.
- `emptyRoleSemantic`: Nos permite establecer la política de acceso a llevar a cabo independientemente del rol del usuario para el método especificado. Se define de la misma forma que en el caso de `@HttpConstraint`.

Si sólo se especifica el método (sin añadir más parámetros), se considera que siempre se permite el acceso mediante dicho método. Por ejemplo, podemos definir políticas diferentes para los métodos `GET`, `POST` y `PUT`:

```
@WebServlet("/MiServlet")
@ServletSecurity(httpMethodConstraints={
   @HttpMethodConstraint("GET"),
   @HttpMethodConstraint(value="POST",rolesAllowed="admin"),
   @HttpMethodConstraint(value="PUT",
                         emptyRoleSemantic=EmptyRoleSemantic.DENY)})
public class MiServlet extends HttpServlet { ... }
```

En este ejemplo, se permite acceder a todos los usuarios con `GET`, pero con `POST` sólo podrán acceder los que tengan rol `admin`, y con `PUT` no podrá acceder nadie.

También podemos combinar una política particular para una serie de métodos, y una política general para el resto:

```
@ServletSecurity(
   value=@HttpConstraint(EmptyRoleSemantic.PERMIT),
   httpMethodConstraints={
     @HttpMethodConstraint(value="POST",rolesAllowed="admin")
     @HttpMethodConstraint(value="PUT",
         transportGuarantee=TransportGuarantee.CONFIDENTIAL)})
public class MiServlet extends HttpServlet { ... }
```

En este caso sólo los usuarios con rol `admin` podrán acceder mediante `POST`, y sólo se podrán establecer conexiones `PUT` mediante SSL, pero para el resto de métodos se permitirá acceder a cualquier usuario sin necesidad de utilizar SSL.

3.5. Acceso a la información de seguridad

Es muy probable que necesitemos acceder desde el código de la aplicación a información del contexto de seguridad del servidor, como puede ser el nombre del usuario autentificado actualmente, o sus roles. Podemos acceder a esta información a través del objeto `HttpServletRequest`.

En primer lugar, podemos obtener los datos del usuario autentificado actualmente con el método `getUserPrincipal()` de la petición. Esto nos devolverá un objeto de tipo `Principal`, del que podremos sacar el nombre del usuario. De forma alternativa, este nombre también se puede obtener mediante el método `getRemoteUser()` del mismo objeto.

```
Principal p = request.getUserPrincipal();
if(p!=null) {
    out.println("El usuario autentificado es " + p.getName());
} else {
    out.println("No hay ningun usuario autentificado");
}
```

También puede interesarnos comprobar si el usuario actual pertenece a un determinado rol, para así saber si debemos darle permiso o no para realizar una operación dada. Esto lo podemos realizar con el método `isUserInRole(rol)` del objeto petición.

```
if(request.isUserInRole("admin")) {
    usuarioDao.altaUsuario(usuario);
} else {
    out.println("Solo los administradores pueden realizar altas");
}
```

Otros métodos que nos aportan información sobre el contexto de seguridad son `getAuthType()`, que nos dice el tipo de autentificación que estamos utilizando (`BASIC`, `DIGEST`, `FORM`, `CLIENT-CERT`), y `isSecure()` que nos indica si estamos realizando una conexión segura (SSL) o no.

Para finalizar, si estamos utilizando seguridad basada en formulario podremos cerrar la sesión de forma sencilla llamando al método `invalidate()` del objeto `HttpSession`.