Spring Security

Víctor Herrero Cazurro

Temario

- Introducción
- Configuración
- Seguridad en la capa Web
  - Recursos no asegurados
  - Formularios de Login
  - Permiso insuficiente
  - Criptografía
  - Conexión segura SSL
  - Sesiones
  - Librería de etiquetas
- Seguridad en los Servicios
- Seguridad Jerárquica
  - Votantes
  - AccessDecisionManager
- Cadena de filtros

Introducción

- La seguridad en Spring:
  - Basada en otorgar.
  - Jerárquica y perimetral. Aplica niveles.
  - Transportable.

Introducción

- Existe una especificación estándar para la seguridad en JEE.
- La seguridad en JEE:
  - Basada en restricciones. Todo es accesible hasta que se restringe el acceso.
  - Perimetral. Estas dentro o no.
  - No es tan estándar, no es fácil migrar

Introducción

- Arquitectura

Configuración

- Añadir las dependencias al proyecto.
- Con Maven

```
<dependency>
		<groupId>org.springframework.security</groupId>
		<artifactId>spring-security-taglibs</artifactId>
		<version>3.2.4.RELEASE</version>
</dependency>
<dependency>
		<groupId>org.springframework.security</groupId>
		<artifactId>spring-security-config</artifactId>
		<version>3.2.4.RELEASE</version>
</dependency>
<dependency>
		<groupId>org.springframework.security</groupId>
		<artifactId>spring-security-crypto</artifactId>
		<version>3.2.4.RELEASE</version>
</dependency>
```

Configuración

- Definir el contexto de Spring.

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
                classpath:aplicacion.xml,
                /WEB-INF/seguridad.xml
        </param-value>
</context-param>
<listener>
        <listener-class>
                org.springframework.web.context.ContextLoaderListener
        </listener-class>
</listener>
```

Configuración

- Definir el filtro Web que asegurará las peticiones Web.

```
<filter>
        <display-name>springSecurityFilterChain</display-name>
        <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>
```

Configuración

- Normalmente en un fichero de contexto de Spring independiente, definir las características de seguridad, como mínimo estas incluyen
  - Configuración de los filtros de seguridad.
    - Asociación de **Roles** a **Recursos**.
  - Declaración de **AuthenticationManager**, **AuthenticationProvider** y **UserDetailService**.

Configuración

- Configuración de los filtros de seguridad,
  - Para ello se tiene la etiqueta **http** dentro de la librería de etiquetas de Spring Security.
  - Con la siguiente configuración se establece un configuración por defecto de los filtros.

```
<http auto-config="true" use-expressions="true">
```

- Asociación de **Roles** a **Recursos**.

```
<http auto-config="true">
        <intercept-url pattern="/**" access="ROLE_USER"/>
</http>
```

Configuración

- Declaración de **AuthenticationManager**.
  - Existen distintas implementaciones, la más básica es la definición de usuarios en el ámbito de Spring.

```
<authentication-manager>
        <authentication-provider>
                <user-service>
                        <user name="victor"
                        password="victor" authorities="ROLE_USER"/>
                </user-service>
        </authentication-provider>
</authentication-manager>
```

  - Otras implementaciones abarcan LDAP, JDBC, …

Recursos No Asegurados

- Para excluir una serie de recursos de la seguridad, se ha de definir otro patrón.

```
<intercept-url pattern="/pages/*" access="IS_AUTHENTICATED_ANONYMOUSLY"/>
```

- Otra opción en versiones a partir de la 3.1

```
<http pattern=“/pages/*” security=“none”/>
```

Formulario Login

- Por defecto la configuración de Spring incluye un formulario para loguearse y una pagina de error de logueo.
- Para configurar unas paginas personalizadas, se ha de añadir a la etiqueta <**http**>.

```
<form-login
        login-page="/pages/login.jsp"
        authentication-failure-url="/pages/error.html"/>
```

- Hay que tener en cuenta que se hace una redirección y por tanto el recurso no se ha de verse afectado por la seguridad, deben ser dos paginas publicas.

Formulario Login

- Dentro del formulario, se han de definir
  - **action** =“/j_spring_security_check"
  - **name**="j_username"
  - **name**=“j_password"

```
<form action="../j_spring_security_check" method="POST">
            <input type="text" name="j_username"  />
            <input type="password" name="j_password" />
            <input name="submit" type="submit“ value="Login" />
</form>
```

Logout

- En general todas las aplicaciones con Login, necesitan poder realizar Logout.
- Para controlarlo Spring ofrece un Filtro, y para configurar este, se ha de añadir

```
<logout logout-success-url="/pages/disconnected.jsp" delete-cookies="JSESSIONID" />
```

- Donde se indica la pagina a la que se redirige al hacer Logout y si en el proceso hay que borrar las Cookies.
- El Filtro escuchara peticiones a la url

```
/j_spring_security_logout
```

Remember Me

- En la definición del formulario, se puede incluir un checkbox, para que el navegador recuerde al usuario y no pida de nuevo el login.

```
<input type="checkbox" name="_spring_security_remember_me"  />
```

- Además habrá que indicar en la configuración, que se ha de revisar la cookie correspondiente para validar al usuario

```
<remember-me key=“miApp-rememberMe" token-validity-seconds="86400" />
```

Permiso insuficiente

- Cuando el usuario se ha logado, pero no tiene los permisos necesarios para acceder un recuso, se lanzará una excepción, esta puede ser capturada y configurar una pagina a mostrar en ese caso.

```
<access-denied-handler  error-page="/pages/access-denied.jsp"/>
```

Criptografía

- Spring proporciona un API criptográfico, que permite encriptar las contraseñas de los usuarios en el medio de almacenamiento.
- Para emplearlo, habrá que configurar la lectura de la contraseña y la escritura de la contraseña.
- Para la escritura, se ha de emplear las clases del API, en concreto aquellas que hereden de **BasePasswordEncoder**.

```
Md5PasswordEncoder pe = new Md5PasswordEncoder();
pe.encodePassword("mipassword", "salt");
```

Criptografía

- Una clase auxiliar que permite generar los Hash

```
public class SaltedPasswordEncrypter {
        public static void main(String[] args) {
                Scanner scanner = new Scanner(System.in);
                System.out.println("tell me user name (salt)?");
                // we're gonna use username as salt...
                String salt = scanner.nextLine();
                System.out.println("tell me the password?");
                String password = scanner.nextLine();
                Md5PasswordEncoder encoder = new Md5PasswordEncoder();
                String encryptedPassword =
                        encoder.encodePassword(password, salt);
                System.out.println(encryptedPassword);
        }
}
```

Criptografía

- En el proceso de **codificación/decodificación**, se incluye un elemento **Salt** para hacerlo mas seguro.
- Para la lectura, se ha de configurar el **AuthenticationManager**, indicando el **algoritmo de encriptación** y el **Salt**.

```
<authentication-manager>
        <authentication-provider>
                <password-encoder hash="md5">
                        <salt-source user-property="username"/>
                </password-encoder>
…
        </authentication-provider>
</authentication-manager>
```

Conexión segura SSL

- Activar SSL en el servidor. Para Tomcat.

```
<Service name="Catalina">
        <Executor name="tomcatThreadPool" namePrefix="catalina-exec-"
                maxThreads="150" minSpareThreads="4"/>
        <Connector connectionTimeout="20000" port="8080" protocol="HTTP/1.1"
                redirectPort="8443“executor="tomcatThreadPool"/>
        <Connector
                executor="tomcatThreadPool"
                port="8443"  redirectPort="8443"
                protocol="org.apache.coyote.http11.Http11Protocol"
                connectionTimeout="20000"
                acceptCount="100“  maxKeepAliveRequest="15"
                keystoreFile="${catalina.base}/conf/tcserver.keystore"
                keystorePass="changeme“ keyAlias="tcserver"
                SSLEnabled="true" scheme="https" secure="true" />
        <Engine defaultHost="localhost" name="Catalina">
                …
        </Engine>
 </Service>
```

Conexión segura SSL

- Indicar que las peticiones que intercepta Spring, han de ser con el protocolo **https**.

```
<intercept-url pattern="/**" access="ROLE_USER" requires-channel="https"/>
```

- Se pueden redirigir las peticiones que lleguen con **http** hacia **https**.

```
<http>
        ...
        <port-mappings>
                <port-mapping http="8080" https="8443"/>
        </port-mappings>
        ...
</http>
```

Sesiones

- Se podrá configurar el comportamiento de la aplicación con respecto a las sesiones creadas.
  - Controlando lo que sucede cuando expiran.

```
<session-management invalid-session-url="/pages/session-expired.jsp“/>
```

  - Controlando el numero de sesiones abiertas por usuario

```
<session-management invalid-session-url="/pages/session-expired.jsp">
        <concurrency-control max-sessions="1"
                                        error-if-maximum-exceeded="true"/>
</session-management>
```

Sesiones

- Para controlar la concurrencia, y que la anterior configuración tenga efecto, se ha de configurar un **Listener** Web.

```
<listener>
        <listener-class>
        org.springframework.security.web.session.HttpSessionEventPublisher
        </listener-class>
</listener>
```

Librería de Etiquetas

- Spring proporciona una librería de etiquetas para incluir funcionalidades relacionadas con la seguridad.
- Esta librería se encuentra en un jar independiente, por lo que habrá que añadirlo.
- Con Maven

```
<dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-taglibs</artifactId>
        <version>4.0.3.RELEASE</version>
</dependency>
```

Librería de Etiquetas

- Una vez se tiene las librería, para emplearla, añadir a las JSP.

```
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags"%>
```

- La librería ofrece etiquetas como
  - **authentication**: Que permite acceder al objeto Principal y sus características.
  - **authorize**: Que permite a través de una expresión booleana indicar si se ha de pintar o no una parte del JSP.

Librería de Etiquetas

- Ejemplo de uso de **authentication**

```
<sec:authentication property="principal.username" />
```

- Ejemplo de uso de **authorize**

```
<sec:authorize access="hasAnyRole('ROLE_USER')">
        <a href="<c:url value='/privado'/>">privado</a>
</sec:authorize>
```

SpEL

- Dentro de las etiquetas se puede emplear **SpEL**, el lenguaje de Expresiones de Spring.
- Este lenguaje de expresiones aporta una serie de funciones
  - **hasRole**('ROLE'): True si el principal tiene el Rol
  - **hasAnyRole**({'ROLE‘, 'ROLE'}): True si el Principal tiene alguno de los Roles
  - **isAuthenticated**(): True si el usuario no es **Anónimo**
  - **isAnonymous**(): True si el Principal es **Anónimo**
  - **isRememberMe**(): True si Principal es **RememberMe**
  - **isFullyAuthenticated**(): True si el Principal no es **Anónimo** o **RememberMe**.

SpEL

- Y una serie de objetos
  - **principal**: Referencia la objeto **Principal**.
  - **authentication**: Referencia al objeto **Authentication** obtenido de **SecurityContext**.
  - **permitAll**: Siempre retorna True
  - **denyAll**: Siempre retorna False

AOP

- La seguridad en los servicios esta basada en AOP.
- La idea general de la AOP, es poder crear proxies, que envuelvan una determinada funcionalidad, suplantándola de cara a los objetos que hacen uso de ella, permitiendo cambiar el comportamiento de la funcionalidad.
- Aplicaciones típicas de la AOP son:
  - Seguridad
  - Logger
  - Transacciones

AOP

- Los actores principales en AOP son
  - **Aspecto**: Funcionalidad que se quiere ejecutar cuando se ejecuta un método.
  - **JointPoint**: Método envuelto por el proxy para permitir la ejecución del aspecto.
  - **Advice**: Momento en el que se aplica el aspecto (antes, después, …)
  - **Target**: Objeto envuelto por uno o mas proxies.
  - **PointCut**: Expresión de AspectJ que selecciona métodos para ser envueltos.

Activar Proxies con @

- Para activar la intercepción de métodos con anotaciones, se ha de añadir

```
<global-method-security
        pre-post-annotations="enabled"
        secured-annotations="enabled“
        jsr250-annotations="enabled"/>
```

- Se pueden indicar varios tipos de estrategias en la definición de **JointPoint**.
  - **pre-post-annotations**: Anotaciones @**PreFilter**, @**PreAuthorize**, @**PostFilter**, @**PostAuthorize**
  - **secured-annotations**: Anotaciones @**Secured**
  - **jsr250-annotations**: Anotaciones @**RolesAllowed**

Activar Proxies con @

- El siguiente paso es incluir dichas anotaciones en los métodos de los servicios, indicando con expresiones, la condición que se ha de cumplir.
- Las anotaciones @**PreAuthorize** y @**PostAuthorize**, permiten trabajar con los objetos recibidos y retornado por los métodos, siendo en las expresiones
  - **#<parametro>**: El parámetro que llega al método
  - **returnObject** : El objeto retornado por el método
  - **authentication**: El usuario autenticado

Activar Proxies con @

- Las anotaciones @**PreFilter** y @**PostFilter**, permiten trabajar con objetos tanto recibidos como retornados que sean **Collection** o **Array**, siendo en las expresiones:
  - **filterObject**: Cada uno de los objetos incluidos en la Collection o Array.
- Las anotaciones @**Secured** y @**RolesAllowed**, no aceptan expresiones, sino una lista de Roles que han de cumplirse para ejecutar el método.

Activar Proxies con @

- Algunas expresiones

```
@PreAuthorize("hasRole('ROLE_ASSISTANT_DIRECTOR')")
@PreAuthorize("#file.investigator == authentication.name")
@PostAuthorize ("returnObject.owner == authentication.name")

@PostFilter("not filterObject.report.contains(principal.username)")
@PostFilter ("filterObject.owner == authentication.name")
@PreFilter("filterObject.owner == authentication.name")

@Secured ({"ROLE_USER", "ROLE_ADMIN"})
@Secured ("ROLE_USER")

@RolesAllowed({"ROLE_REGULAR_USER","ROLE_ADMIN"})
```

- En las expresiones se pueden emplear **and**, **or** y **not**.

Activar Proxies con PointCut

- Se pueden definir expresiones de selección de métodos mas genéricas, sin necesidad de incluir una anotación en cada método, para ello hay que definir expresiones **AspectJ**.
- Las expresiones de AspectJ tiene la siguiente firma

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)
```

- Un ejemplo

```
execution(public com.ats.test.*.*(..))
```

Activar Proxies con PointCut

- El * es un comodín para los tipos de acceso, nombres de clase y métodos.
- Los .. en los argumentos de la llamada, indica que es con cualquier tipo de argumentos.
- Para argumentos en concreto se pone el tipo:

```
execution(* transfer(int,int))
```

- Se pueden combinar expresiones de pointcuts con and, or y not.

Activar Proxies con PointCut

- Tipos de **pointcut** soportados por Spring
  - **execution** – para seleccionar la ejecución de un método
  - **within** – para seleccionar ejecución de algún método del tipo especificado.

```
within(com.ats.test.*)
```

  - **this** – para seleccionar ejecución de algún método de una bean que implemente una interfaz del tipo especificado.

```
this(com.ats.AccountService)
```

Activar Proxies con PointCut

- Tipos de **pointcut** soportados por Spring
  - **target** - para seleccionar ejecución de algún método de una bean cuyo destino implemente una interfaz del tipo especificado.

```
target(com.ats.services.MyService).
```

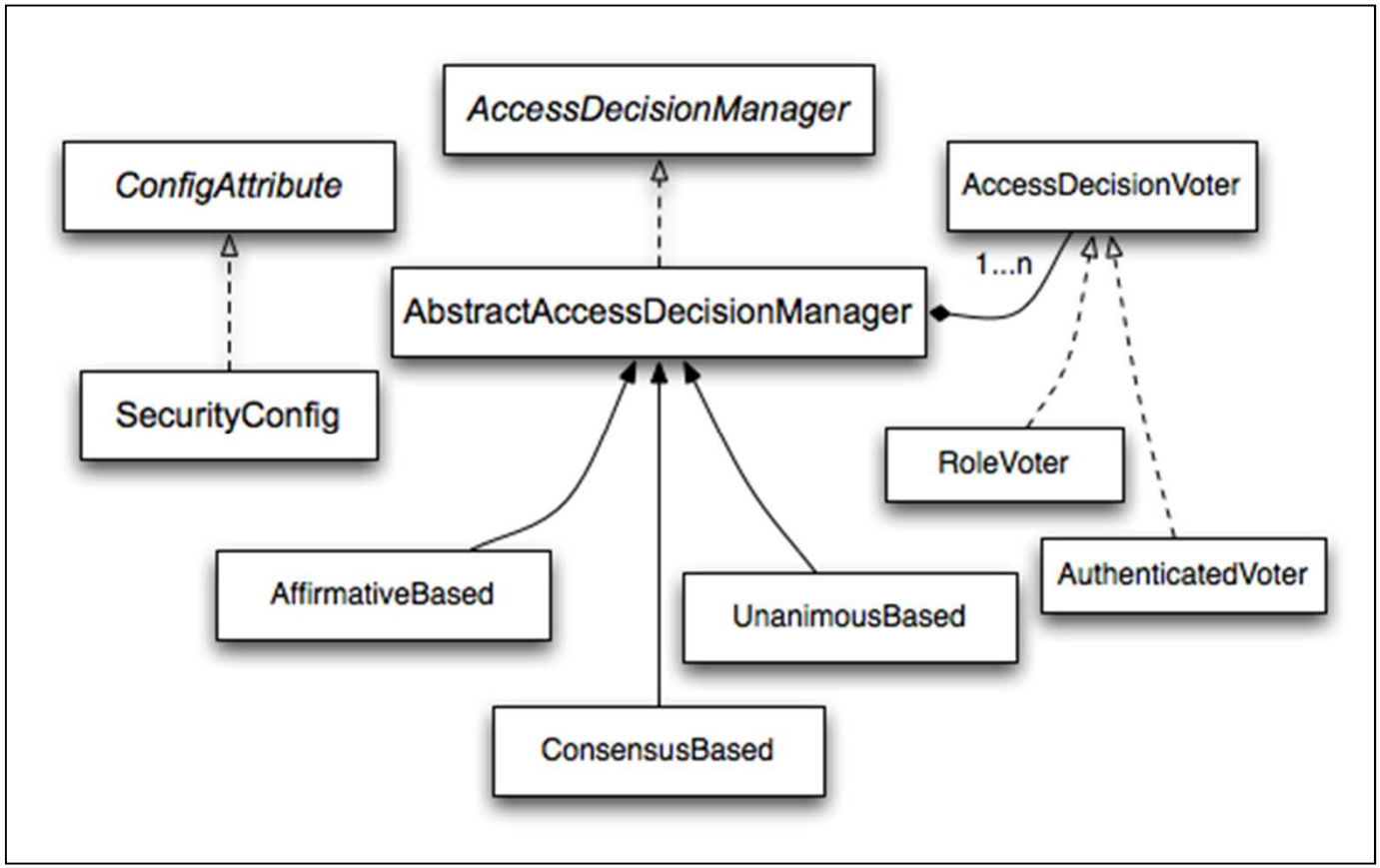
  - **args** - seleccionar ejecución de algún método de un tipo cuyos argumentos son instancias de los tipos dados.

Configurar Votantes

- El sistema de votantes, permite definir distintas estrategias de aceptación de las restricciones de seguridad.
- Las estrategias serán
  - Afirmativa
  - Unánime
  - Consenso
- El que se encarga de aplicar las estrategias es el **AccessDecisionManager**

AccessDecissionManager

- Estrategias de Control de Acceso.



AccessDecissionManager

- El **AccessDecisionManager** hay que indicarlo en la etiqueta <**http**>

```
<http use-expressions="true" access-decision-manager-ref="accessDecisionManager">
```

- O en <**global-method-security**>

```
<global-method-security access-decision-manager-ref="accessDecisionManager">
```

- De no indicar ninguno, se crea uno por defecto con
  - AffirmativeBased
  - RoleVoter y AuthenticatedVoter

AccessDecissionManager

- El **AccessDecisionManager** será un Bean de Spring, empelando las clases del paquete **org.springframework.security.access.vote**
  - **AffirmativeBased**. Con un afirmativo es suficiente.
  - **ConsesusBased**. Lo que opine la mayoría, ignorando abstenciones.
  - **UnanimousBased**. Todos afirmativo para aprobar.

AccessDecissionManager

- Un ejemplo de definición de **AccessDecisionManager**

```
<beans:bean id="accessDecisionManager"
            class="org.springframework.security.access.vote.UnanimousBased">
            <beans:constructor-arg>
            <beans:list>
                        <beans:bean
                        class="org.springframework.security.access.vote.RoleVoter">
                        <beans:property name="rolePrefix" value="ROLE_" />
                        </beans:bean>
                        <beans:bean
            class="org.springframework.security.access.vote.AuthenticatedVoter"/>
                        <beans:bean
class="org.springframework.security.web.access.expression.WebExpressionVoter"/>
                        <beans:bean class="com.ejemplo.DateVoter"/>
            </beans:list>
            </beans:constructor-arg>
</beans:bean>
```

AccessDecissionManager

- Estos Bean necesitan para su construcción un listado de Votantes, que serán clases que implementen **AccessDecisionVoter**.
- Estas clase retornarán en su método **vote**:
  - **ACCESS_ABSTAIN**
  - **ACCESS_GRANTED**
  - **ACCESS_DENIED**
- Los métodos de **supports**, permiten distinguir si una configuración ha de ser procesada por un votante o no.

AccessDecissionManager

- Dicho método recibe
  - **Authentication**: Objeto que representa el invocante de la funcionalidad.
  - **Object**: El objeto asegurado.
  - **Collection<ConfigAttribute>**: Atributos de configuración que afectan al método que va a ser invocado.

AccessDecissionManager

- En el API se proporcionan
  - **AclEntryVoter**
  - **LabelBasedAclVoter**
  - **AuthenticatedVoter**
  - **Jsr250Voter**
  - **PreInvocationAuthorizationAdviceVoter**
  - **RoleVoter**
  - **RoleHierarchyVoter**
  - **WebExpresionVoter**

AccessDecissionManager

- Una configuración habitual podría ser
  - **RoleVoter**: Permite Votar sobre configuraciones con atributos de texto que empiecen por “ROLE_”
  - **AuthenticatedVoter**: Permite Votar sobre configuraciones con IS_AUTHENTICATED_FULLY, IS_AUTHENTICATED_REMEMBERED o IS_AUTHENTICATED_ANONYMOUSLY.
  - **PreInvocationAuthorizationAdviceVoter**: Permite Votar según las configuraciones de @**PreAuthorize** o @**PreFilter**.
  - **WebExpressionVoter**: Permite Votar sobre Objetos de tipo <**intercept-url**>.

AccessDecissionManager

- Ejemplo de un votante

```
public class DateVoter implements AccessDecisionVoter<Object> {
@Override
public boolean supports(ConfigAttribute configAttribute) {return true;}
@Override
public boolean supports(Class<?> clazz) {return true;}
@Override
public int vote(Authentication authentication, Object object,
                                        Collection<ConfigAttribute> configAttributes) {
        int vote=AccessDecisionVoter.ACCESS_DENIED;
        int actualMinute = Calendar.getInstance().get(Calendar.MINUTE);
        if (actualMinute % 2 == 0){
                vote=AccessDecisionVoter.ACCESS_GRANTED;
        }
        System.out.println("Voting: "+vote +" for minute "+actualMinute);
        return vote;
}
}
```

Cadena de Filtros

- Cuando se añade <**http auto-config="true"**>, se esta definiendo una cadena de Filtros por defecto. Y al añadir configuraciones, se amplia.
- Una posible configuración sería:
  - ChannelProcessingFilter
  - SecurityContextPersistenceFilter
  - ConcurrentSessionFilter
  - WebAsyncManagerIntegrationFilter
  - LogoutFilter

Cadena de Filtros

- UsernamePasswordAuthenticationFilter
- BasicAuthenticationFilter
- RequestCacheAwareFilter
- SecurityContextHolderAwareRequestFilter
- RememberMeAuthenticationFilter
- AnonymousAuthenticationFilter
- SessionManagementFilter
- ExceptionTranslationFilter
- FilterSecurityInterceptor

Cadena de Filtros

- Se puede personalizar la cadena de filtros de seguridad incluyendo algún Filtro a mayores de los que ya pre-configuran, indicando en <**http**>

```
<http>
        …
        <custom-filter ref="rememberLoginFilter" before="FORM_LOGIN_FILTER"/>
        …
</http>
```

- Se hace referencia a un Bean de Spring que extiende la clase abstracta **OncePerRequestFilter**

Cadena de Filtros

- Una ejemplo de **OncePerRequestFilter**

```
public class RemerberLoginFilter extends OncePerRequestFilter {
        @Override
        protected void doFilterInternal(HttpServletRequest request,
                        HttpServletResponse response, FilterChain chain)
                        throws ServletException, IOException {
                String login = request.getParameter("j_username");
                if(login != null && !login.isEmpty()){
                        response.addCookie(new Cookie(
                                "SPRING_SECURITY_LAST_USERNAME“,
                                login));
                }
                chain.doFilter(request, response);
        }
}
```

@VictorHerrero1

Víctor Herrero Cazurro

victorherrerocazurro

victorherrerocazurro