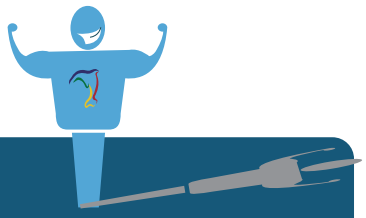
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

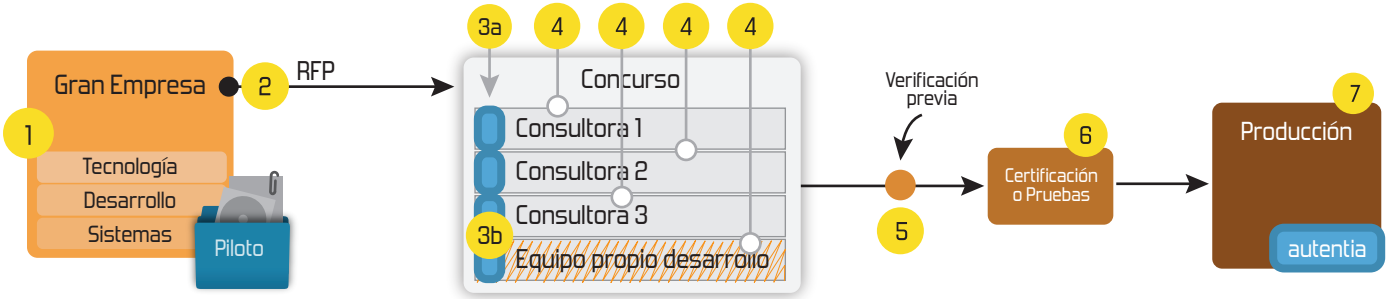
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
Gestor de contenidos (Alfresco)
Aplicaciones híbridas

Control de autenticación y acceso (Spring Security)
UDDI
Web Services
Rest Services
Social SSO
SSO (Cas)

JPA-Hibernate, MyBatis
Motor de búsqueda empresarial (Solr)
ETL (Talend)

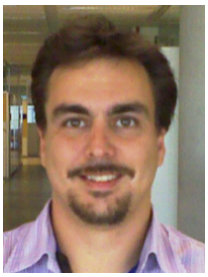
Dirección de Proyectos Informáticos.
Metodologías ágiles
Patrones de diseño
TDD

Tareas programadas (Quartz)
Gestor documental (Alfresco)
Inversión de control (Spring)

BPM (jBPM o Bonita)
Generación de informes (JasperReport)
ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com



DESARROLLADO POR:
Borja Lázaro de Rafael

Consultor tecnológico de desarrollo de proyectos informáticos.

Ingeniero en Informática

Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación

Somos expertos en Java/J2EE

Fecha de publicación del tutorial: 2011-09-29

Token con caducidad en Spring Security

0. Índice de contenidos.

1. Introducción

Hoy en día ya sabemos que los mecanismos de autenticación de las distintas aplicaciones pueden ser tan complejos o simples como podamos imáginar. Tenemos la autenticación básica que ofrecen los propios navegadores, la comunmente utilizada de un formulario de usuario y contraseña, contra LDAP o Active Directory, etc. Incluso podemos tener combinación de varios de estos mecanismos de seguridad o hacer extensiones de los mismos.

Por suerte, SpringSecurity nos ofrece ya una amplia variedad de mecanismos de autenticación que nos simplifican gran parte del trabajo. Así que sólo tenemos que configurarlos y/o hacer nuestras propias extensiones o modificaciones en caso de ser necesario.

El objetivo de este tutorial es la introducción de un token con un tiempo de caducidad para que el proceso de login se realice en un periodo de tiempo máximo. Esto es útil cuando tenemos mecanismos de autenticación automáticos, de forma que limitamos la ventana de tiempo para el acceso a nuestra aplicación.

Antes de seguir, tengo que dejar claro que esto por sí mismo no se puede considerar un mecanismo de autenticación, realmente es un complemento a un mecanismo de autenticación ya existente. Por lo que nuestro sistema será tan seguro como sea ese mecanismo de autenticación; y con este token de caducidad lo que hacemos es aumentar un poco su seguridad limitando el tiempo en el que permitimos acceder a la aplicación.

Como el mecanismo de autenticación más extendido es el de un formulario con usuario y contraseña, voy a tomar éste como mecanismo base. Extenderemos esta forma de autenticación añadiendo un token con caducidad que establezca un tiempo máximo para el proceso de login, desde que se inicia

Últimas Noticias

Autentia patrocina la CAS2011

Experiencia en la XPWeek.

Autentia patrocina la Apache Barcamp Spain

Autentia participa en el Día Mundial de la Enfermedad de Alzheimer

XPWeek en Madrid del 19 al 23 de septiembre.

Histórico de NOTICIAS

Últimos Tutoriales

Creación de un componente en JSF2: separando la renderización del propio componente

Gestión de eventos en el cliente con el soporte Ajax de PrimeFaces

Ejemplo de Viewpager para android

CAS REST: Cómo

pidiendo la autenticación al usuario, hasta que se recibe de este su usuario y contraseña.

2. Entorno

Este tutorial está escrito usando el siguiente entorno:

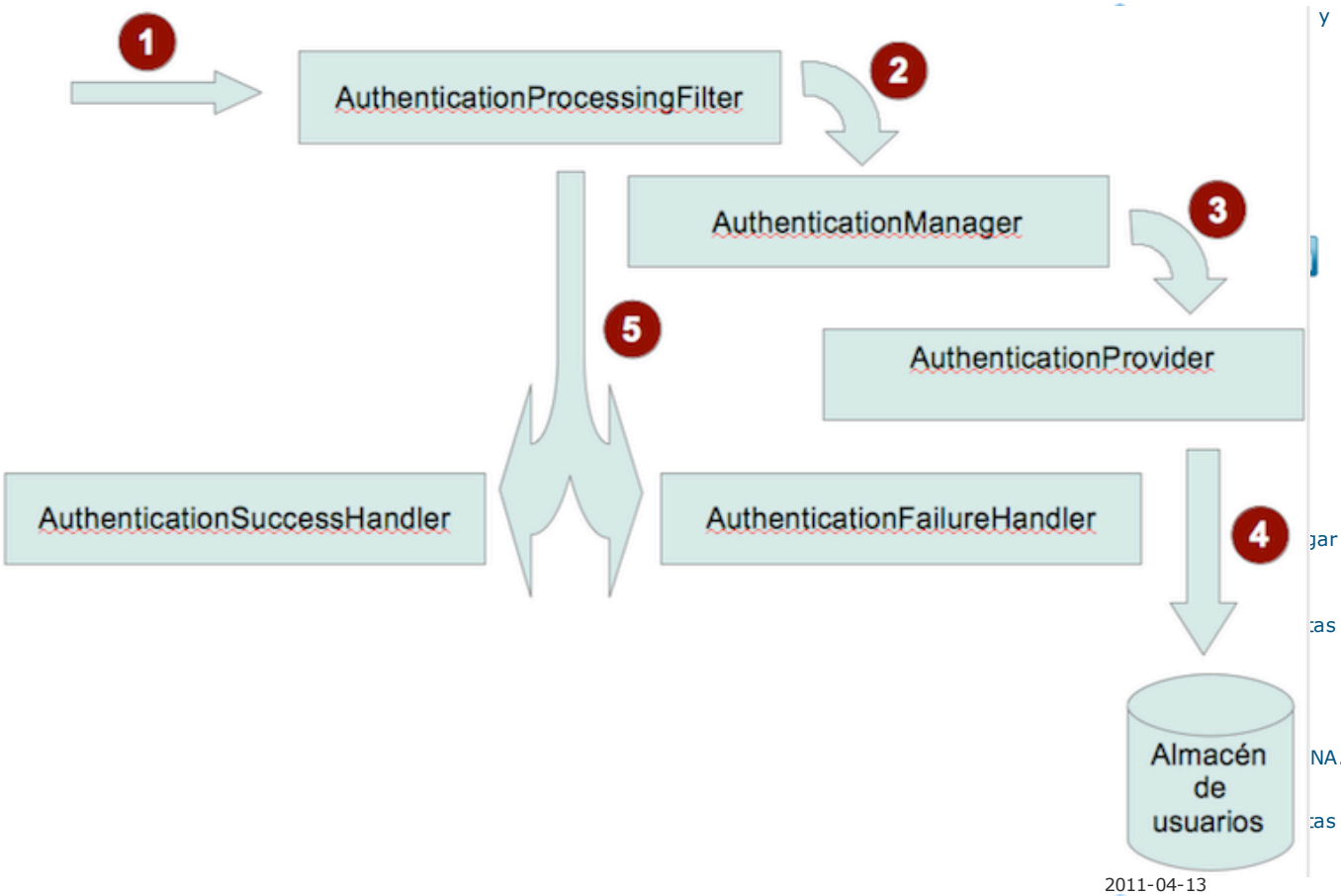
- Hardware: Portátil MacBook Pro 15' (2 GHz Intel Core i7, 8GB DDR3 SDRAM).
- Sistema Operativo: Mac OS X Lion 10.7.1
- Spring Security 3.1.0.RC2
- JSF 2.1.2

3. Proceso de autenticación en Spring Security.

Antes de nada, voy a explicar brévemente cómo funciona el proceso de autenticación en Spring Security. Este proceso se basa en un conjunto de interfaces y clases que interactúan conjuntamente en un orden establecido para terminar decidiendo si una petición tiene o no acceso. Spring realiza esto mediante una cadena de filtros en los que se irán haciendo las distintas acciones y comprobaciones necesarias hasta decidir si la autenticación es correcta o no. En este proceso podemos identificar básicamente a 3 actores principales:

- **AuthenticationFilter**: Es el responsable de crear una instancia concreta del usuario y sus credenciales de autenticación.
- **AuthenticationManager**: Responsable de la validación del usuario y credenciales y rellenar los permisos/roles que tiene el usuario o de lanzar los distintos tipos de excepciones en caso de fallo en la autenticación. Esto lo hace apoyándose en uno o varios proveedores de autenticación.
- **AuthenticationProvider**: Es en quien se delega la correcta validación del usuario y recuperación de roles.

Este proceso a alto nivel sería:



- 1.- El filtro de seguridad intercepta la petición de autenticación. Crea una instancia de Authentication para su validación.
- 2.- El filtro de seguridad pasa al AuthenticationManager el objeto creado para que realice su validación. El AuthenticationManager delegará esta validación al proveedor o proveedores de autenticación.
- 3.- El proveedor de autenticación comprobará si los datos de autenticación son correctos. En caso de haber algún tipo de error lanzará una excepción.
- 4.- El filtro de seguridad comprueba el resultado de la autenticación solicitada al AuthenticationManager. Si se produjo una excepción redirigirá el flujo al AuthenticationFailureHandler y si hubo éxito al AuthenticationSuccessHanlder.

4. Configuración de nuestro mecanismo de autenticación.

Como sabéis, en Spring utilizamos un fichero de configuración. Así que en el caso de Spring Security no iba a ser distinto. Aquí os pongo el fichero de configuración que vamos a utilizar en nuestro ejemplo, luego pasaré a contar cuál es cada uno de los elementos del fichero relacionados con el proceso de autenticación.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <beans:beans xmlns="http://www.springframework.org/schema/security"
03     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

logarnos en CAS sin ir a la pantalla de login por defecto

Usando el componente PickList de Primefaces

Últimos Tutoriales del Autor

Gestión de eventos en el cliente con el soporte Ajax de PrimeFaces

Usando el componente PickList de Primefaces

Release Bugzilla Maven Plugin

Enlazar Bugzilla con MavenChangesPlugin

```
xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-3.1.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">

    <context:annotation-config />

    <aop:aspectj-autoproxy />

    <context:component-scan base-package="com.autentia.tutoriales" />

    <http access-denied-page="/_403.xhtml" auto-config="false"
        entry-point-ref="LoginUrlAuthenticationEntryPoint">
        <custom-filter ref="authenticationFilter"
position="FORM_LOGIN_FILTER"></custom-filter>
        <intercept-url pattern="/*" access="IS_AUTHENTICATED_FULLY" />
    </http>

    <beans:bean id="LoginUrlAuthenticationEntryPoint"
        class="org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoint">
        <beans:property name="loginFormUrl" value="/login.xhtml" />
    </beans:bean>

    <beans:bean id="authenticationFilter"
        class="com.autentia.tutoriales.UsernamePasswordWithTimeoutAuthenticationFilter">
        <beans:property name="authenticationManager"
ref="authenticationManager" />
        <beans:property name="authenticationSuccessHandler"
ref="authenticationSuccessHandler" />
        <beans:property name="authenticationFailureHandler"
ref="authenticationFailureHandler" />
        <beans:property name="filterProcessesUrl" value="/login" />
        <beans:property name="usernameParameter" value="username" />
        <beans:property name="passwordParameter" value="passwword" />
        <beans:property name="timeoutParameter" value="timeout" />
    </beans:bean>

    <beans:bean id="authenticationFailureHandler"
        class="com.autentia.tutoriales.AutentiaAuthenticationFailureHandler">
        <beans:property name="defaultFailureUrl"
value="/bad_credentials.html"/>
        <beans:property name="expiredUrl" value="/login_timeout.html"/>
    </beans:bean>

    <beans:bean id="authenticationSuccessHandler"
        class="com.autentia.tutoriales.AutentiaAuthenticationSuccessHandler">
    </beans:bean>


    <beans:bean id="authenticationProvider"
        class="com.autentia.tutoriales.AutentiaAuthenticationProvider">
        <beans:property name="nonceValiditySeconds" value="10"/>
        <beans:property name="key" value="KEY"/>
    </beans:bean>

    <authentication-manager alias="authenticationManager">
        <authentication-provider ref="authenticationProvider"/>
    </authentication-manager>
</beans:beans>
```

Respecto al mecanismo de seguridad en el fichero podemos ver:

- Línea 16: Elemento que nos define cómo va a ser la autenticación de nuestra aplicación web por HTTP.
- Línea 18: Establece la referencia al filtro de autenticación.
- Línea 19: Se definen cuales són los recursos protegidos y quiénes tendrán acceso a ellos. En este caso se protege toda la aplicación y basta con que el usuario esté autenticado para que pueda acceder a los mismos.
- Línea 22: Define cuál va a ser el punto de entrada para la autenticación. En este caso se redirige a la página /login.xhtml
- Línea 27: Aquí definimos cual va a ser el filtro de autenticación responsable de formar un objeto con los parámetos recibidos por el usuario (username,password,timeout), pasarlo al AuthenticationManager para su validación y finalmente delegar el final del proceso dependiendo de si ha habido o no error de autenticación (authenticationSuccessHandler y authenticationFailureHandler).
- Línea 38: Bean responsable de tratar los intentos de autenticación que han fallado y redirigir a la página de error.
- Línea 44: Define cual va a ser el proveedor de autenticación responsable de comprobar que el usuario es válido y recuperar sus roles. En nuestro caso indicamos además el tiempo máximo para el proceso de login (nonceValiditySeconds) y la clave utilizada para generar una firma y evitar que el token pueda ser manipulado.
- Línea 50: Indica que vamos a utilizar el AuthenticationManager por defecto de Spring y que este va a utilizar como proveedor de autenticación al que hemos definido en la línea 44.

5. Implementación del mecanismo de autenticación.

Pués como podéis ver en el fichero de configuración, tenemos alguna que otra clase que hay que implementar. El "EntryPoint" que utilizamos es el propio de Spring, así que lo primero que tenemos

que implementar es el filtro. Como hemos dicho, vamos a hacer una extensión de la autenticación por usuario y contraseña, así que como Spring ya nos ofrece un filtro para este tipo de autenticación, lo que hacemos es directamente heredar de él y sobreescribir aquello que nos haga falta. Básicamente lo realmente importante en el filtro es el método "attempAuthentication(..)". Éste será casi identico al de Spring salvo por un par de detalles, además de recuperar el usuario y la contraseña, se debe recuperar el timeout y con esto formará un objeto de autenticación que ya lleve informado el timeout. Así que el filtro quedará:

```
01 package com.autentia.tutoriales;
02
03 import javax.servlet.http.HttpServletRequest;
04 import javax.servlet.http.HttpServletResponse;
05
06 import
   org.springframework.security.authentication.AuthenticationServiceException;
07 import org.springframework.security.core.Authentication;
08 import org.springframework.security.core.AuthenticationException;
09 import
   org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
10
11 public class UsernamePasswordWithTimeoutAuthenticationFilter extends
12         UsernamePasswordAuthenticationFilter {
13
14     private String timeoutParameter = "timeout";
15     private boolean postOnly;
16
17     @Override
18     public void setPostOnly(boolean postOnly) {
19         super.setPostOnly(postOnly);
20         this.postOnly = postOnly;
21     }
22
23     @Override
24     public Authentication attemptAuthentication(HttpServletRequest request,
25             HttpServletResponse response) throws AuthenticationException {
26         if (postOnly && !"POST".equals(request.getMethod())) {
27             throw new AuthenticationServiceException(
28                     "Authentication method not supported: "
29                             + request.getMethod());
30         }
31
32         String username = obtainUsername(request);
33         String password = obtainPassword(request);
34         final String timeout = obtainTimeout(request);
35
36         if (username == null) {
37             username = "";
38         }
39
40         if (password == null) {
41             password = "";
42         }
43
44         username = username.trim();
45
46         final UsernamePasswordWithTimeoutAuthenticationToken authRequest =
   new UsernamePasswordWithTimeoutAuthenticationToken(
47                 username, password, timeout);
48
49         setDetails(request, authRequest);
50
51         return this.getAuthenticationManager().authenticate(authRequest);
52     }
53
54     protected String obtainTimeout(HttpServletRequest request) {
55         return request.getParameter(timeoutParameter);
56     }
57
58     public void setTimeoutParameter(String timeoutParameter) {
59         this.timeoutParameter = timeoutParameter;
60     }
61 }
```

Como véis el filtro crea una instancia de un token de autenticación que además de usuario y contraseña tiene un timeout. Este token lo implementamos también heredando del token que ofrece Spring para usuario y contraseña, de forma que lo único que hacemos es extenderlo para que tenga el timeout.

```
01 package com.autentia.tutoriales;
02
03 import
   org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
04
05 public class UsernamePasswordWithTimeoutAuthenticationToken extends
06         UsernamePasswordAuthenticationToken {
07
08     private String timeout;
09
10     public UsernamePasswordWithTimeoutAuthenticationToken(Object principal,
11             Object credentials) {
12         super(principal, credentials);
13         this.timeout=null;
14     }
15
16     public UsernamePasswordWithTimeoutAuthenticationToken(Object principal,
17             Object credentials, String timeout) {
```

```
        this.timeout=timeout;
    }

    public String getTimeout() {
        return timeout;
    }
}
```

Ahora nos vamos a encargar de los dos responsables del post-proceso de una autenticación con éxito o con error. Para esto también nos apoyamos en las clases de Spring de las cuáles extendemos nuestras propias clases.

En caso de éxito: (Se realizaría la lógica del intento de autenticación correcta y luego se redirige a una página de bienvenida)

```
package com.autentia.tutoriales;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.security.core.Authentication;
import
org.springframework.security.web.authentication.AuthenticationSuccessHandler;
import org.springframework.stereotype.Service;

@Service
public class AutentiaAuthenticationSuccessHandler implements
AuthenticationSuccessHandler {

    @Override
    public final void onAuthenticationSuccess(HttpServletRequest request,
HttpServletResponse response,
            Authentication authentication) throws IOException,
ServletException {
        //lógica de tratamiento de autenticación correcta
        response.sendRedirect(response.encodeRedirectURL("welcome.xhtml"));
    }
}
```

Para el caso de un intento de autenticación errónea: (Se redirecciona a una página de error de autenticación o de timeout dependiendo del tipo de error)

```
package com.autentia.tutoriales;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.security.core.AuthenticationException;
import
org.springframework.security.web.authentication.SimpleUrlAuthenticationFailureHandler;
import
org.springframework.security.web.authentication.www.NonceExpiredException;

public class AutentiaAuthenticationFailureHandler extends
        SimpleUrlAuthenticationFailureHandler {

    private String defaultFailureUrl;
    private String expiredUrl;

    @Override
    public void onAuthenticationFailure(HttpServletRequest request,
HttpServletResponse response,
            AuthenticationException exception) throws IOException,
ServletException {
        final String failureUrl=getFailureUrl(exception);
        if (failureUrl == null) {
            logger.debug("No failure URL set, sending 401 Unauthorized
error");

            response.sendError(HttpServletResponse.SC_UNAUTHORIZED,
"Authentication Failed: " + exception.getMessage());
        } else {
            saveException(request, exception);

            if (isUseForward()) {
                logger.debug("Forwarding to " + failureUrl);

                request.getRequestDispatcher(failureUrl).forward(request,
response);
            } else {
                logger.debug("Redirecting to " + failureUrl);
                getRedirectStrategy().sendRedirect(request, response,
failureUrl);
            }
        }
    }

    @Override
    public void setDefaultFailureUrl(String defaultFailureUrl) {
       super.setDefaultFailureUrl(defaultFailureUrl);
        this.defaultFailureUrl = defaultFailureUrl;
    }

    private String getFailureUrl(AuthenticationException exception) {
        if(exception instanceof NonceExpiredException){
            return expiredUrl;
```

```
        }
        return defaultFailureUrl;
    }

    public void setExpiredUrl(String expiredUrl) {
        this.expiredUrl = expiredUrl;
    }
}
```

Ahora ya sólo nos queda la implementación del proveedor de autenticación, que será el responsable de controlar que el token de caducidad sea correcto y que no se ha superado el timeout establecido. Para la generación y comprobación de este token, nos hemos basado en como Spring hace lo mismo para el tipo de autenticación DigestAuthentication, pero que no lo implementa en otros tipos de autenticación, por eso nos lo estamos teniendo que implementar nosotros.

La idea es generar un token de caducidad que no haga falta almacenar en el servidor para su posterior comprobación. Es decir, se meterá en un campo oculto del formulario y cuando éste se devuelva al servidor se comprobará su validez en el servidor. Para hacer esto tiene que ser autodefinido, es defir, debe contener la fecha de caducidad, y hay que protegerlo a posibles manipulaciones.

Así que nuestro proveedor de autenticación quedará de la siguiente forma:

```
001 package com.autentia.tutoriales;
002
003 import java.security.MessageDigest;
004 import java.security.NoSuchAlgorithmException;
005 import java.util.HashSet;
006 import java.util.Set;
007
008 import org.apache.commons.lang.StringUtils;
009 import org.slf4j.Logger;
010 import org.slf4j.LoggerFactory;
011 import org.springframework.context.support.MessageSourceAccessor;
012 import org.springframework.security.authentication.AuthenticationProvider;
013 import org.springframework.security.authentication.BadCredentialsException;
014 import
    org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
015 import org.springframework.security.core.Authentication;
016 import org.springframework.security.core.GrantedAuthority;
017 import org.springframework.security.core.SpringSecurityMessageSource;
018 import org.springframework.security.crypto.codec.Base64;
019 import org.springframework.security.crypto.codec.Hex;
020 import
    org.springframework.security.web.authentication.www.NonceExpiredException;
021 import org.springframework.stereotype.Service;
022
023 @Service
024 public class AutentiaAuthenticationProvider implements
    AuthenticationProvider {
025
026     private static final Logger Log =
    LoggerFactory.getLogger(AutentiaAuthenticationProvider.class);
027
028     private static final String NONCE_FIELD_SEPARATOR = ":";
029
030     private String key = "KEY";
031
032     private long nonceValiditySeconds=10;
033
034
035     protected final MessageSourceAccessor messages =
    SpringSecurityMessageSource.getAccessor();
036
037     @Override
038     public final Authentication authenticate(Authentication authentication)
    {
039         final UsernamePasswordWithTimeoutAuthenticationToken
    authenticationToken =
    (UsernamePasswordWithTimeoutAuthenticationToken)authentication;
040         validateTimeout(authenticationToken);
041         //lógica de comprobación de usuario y contraseña
042         return createSuccessAuthentication(authenticationToken);
043     }
044
045     @Override
046     public final boolean supports(Class<?> authentication) {
047         return
    UsernamePasswordWithTimeoutAuthenticationToken.class.isAssignableFrom(authentication);
048     }
049
050     public long getNonceValiditySeconds() {
051         return nonceValiditySeconds;
052     }
053
054     public void setNonceValiditySeconds(long nonceValiditySeconds) {
055         this.nonceValiditySeconds = nonceValiditySeconds;
056     }
057

058     public String getKey() {
059         return key;
060     }
061
062     public void setKey(String key) {
063         this.key = key;
064     }
```

```
066     private void validateTimeout(
067             UsernamePasswordWithTimeoutAuthenticationToken
    authenticationToken) {
068         if(StringUtils.isEmpty( authenticationToken.getTimeout())){
069             final String msg="Timeout signature not present.";
070             Log.error(msg);
071             throw new BadCredentialsException(msg);
072         }
073         final long
    timeOutTime=extractNonceValue(authenticationToken.getTimeout());
074 
075         if (isNonceExpired(timeOutTime)){
076             final String msg="Login timeout";
077             Log.error(msg);
078             throw new NonceExpiredException(msg);
079         }
080     }
081 
082     boolean isNonceExpired(final long timeoutTime) {
083         final long now = System.currentTimeMillis();
084         return timeoutTime < now;
085     }
086 
087     private long extractNonceValue(final String nonce) {
088         // Check nonce was Base64 encoded (as sent by the filter)
089         if (!Base64.isBase64(nonce.getBytes())) {
090             throw new
    BadCredentialsException(messages.getMessage("DigestAuthenticationFilter.nonceEncoding",
091                     new Object[]{nonce}, "Nonce is not encoded in Base64;
    received nonce {0}"));
092         }
093 
094         // Decode nonce from Base64
095         // format of nonce is:
096         // base64(expirationTime + ":" + md5Hex(expirationTime + ":" + key))
097         final String nonceAsPlainText = new
    String(Base64.decode(nonce.getBytes()));
098         final String[] nonceTokens =
    org.springframework.util.StringUtils.delimitedListToStringArray(nonceAsPlainText,
    NONCE_FIELD_SEPARATOR);
099 
100         if (nonceTokens.length != 2) {
101             throw new
    BadCredentialsException(messages.getMessage("DigestAuthenticationFilter.nonceNotTwoTokens",
102                     new Object[]{nonceAsPlainText}, "Nonce should
    have yielded two tokens but was {0}"));
103         }
104 
105         // Extract expiry time from nonce
106         long nonceExpiryTime;
107         try {
108             nonceExpiryTime = Long.valueOf(nonceTokens[0]);
109         } catch (NumberFormatException nfe) {
110             throw new
    BadCredentialsException(messages.getMessage("DigestAuthenticationFilter.nonceNotNumeric",
111                     new Object[]{nonceAsPlainText},
112                     "Nonce token should have yielded a numeric first
    token, but was {0}"),nfe);
113         }
114 
115         // Check signature of nonce matches this expiry time
116         final String expectedNonceSignature = md5Hex(nonceExpiryTime +
    NONCE_FIELD_SEPARATOR + key);
117 
118         if (!expectedNonceSignature.equals(nonceTokens[1])) {
119            throw new
    BadCredentialsException(messages.getMessage("DigestAuthenticationFilter.nonceCompromised",
120                     new Object[]{nonceAsPlainText}, "Nonce token
    compromised {0}"));
121         }
122 
123         return nonceExpiryTime;
124     }
125 
126     private Authentication
    createSuccessAuthentication(UsernamePasswordAuthenticationToken
    authenticationToken) {
127         final Set<GrantedAuthority> authorities = new
    HashSet<GrantedAuthority>();
128         //lógica de asignación de roles en authorities
129         return new
    UsernamePasswordAuthenticationToken(authenticationToken.getPrincipal(),
130                 authenticationToken.getCredentials(), authorities);
131     }
132 
133     public String calculateNonce() {
134         final long expiryTime = System.currentTimeMillis()
135                 + (nonceValiditySeconds * 1000);
136         final String signatureValue = md5Hex(new
    StringBuilder().append(expiryTime).append(NONCE_FIELD_SEPARATOR).append(key).toString());
137         final String nonceValue = new
    StringBuilder().append(expiryTime).append(NONCE_FIELD_SEPARATOR).append(signatureValue).toString();
138         return new String(Base64.encode(nonceValue.getBytes()));
139     }
140 
141     public static String md5Hex(String data) {
142         try {
143             MessageDigest digest = MessageDigest.getInstance("MD5");
```

```
            MessageDigest digest = MessageDigest.getInstance("MD5");
            return new String(Hex.encode(digest.digest(data.getBytes())));
        } catch (NoSuchAlgorithmException e) {
            throw new IllegalStateException("No MD5 algorithm available!");
        }

    }
}
```

Como se puede ver, el método "calculateNonce()" es el que calcula el valor del token de seguridad que se enviará al formulario Web del cliente. Este valor se calcula por composición de dos partes y se codifica en Base64. Las dos partes del token son:

- Fecha máxima para el login: Cálculo con el tiempo actual en milisegundos más el valor de los segundos definidos como periodo de login
- Firma del servidor para evitar manipulación: Se compone de un String con la fecha anteriormente calculada en milisegundos, más una clave propia del servidor sobre la que se hace un digest en MD5 en formato hexadecimal.

A la hora de recibir el formulario con el intento de autenticación, lo primero que se hace es validar el timeout con el método "validateTimeout()", donde se recuperará el valor de la fecha máxima para el proceso de login y luego que aÃºn no se ha superado. En el método "extractNonceValue(...)" se comprueba que éste parámetro no ha sido manipulado y que está correctamente informado, en caso contrario se lanza la excepción "BadCredentialsException". Por último, ya solo basta comprobar que no se ha superado el tiempo para el proceso de login, método "isNonceExpired(...)", que en caso de haberse superado se lanza la excepción "NonceExpiredException".

Ya sólo nos falta crear el formulario de autenticación al que le pasaremos el cálculo de este token de caducidad. Primero creamos la página del formulario:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html">
<h:head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
    <title><h:outputText value="Token con caducidad en Spring Security" />
</title>
</h:head>
<h:body>
        <div id="login">
            <form id="loginForm" action="login" method="post">
                <input type="hidden" name="username" value="" />
                <input type="hidden" name="password" value="" />
                <h:inputHidden id="timeout" value="#
{loginTimeOutView.nonce}"/>
                <input type="submit" value="Entrar"/>
            </form>
        </div>
</h:body>
</html>
```

Como podéis ver en el campo oculto "timeout" hemos metido el valor que nos devuelve el controlador "LoginTimeout" en su método "getNonce(...)". Este será de la siguiente forma:

```
package com.autentia.tutoriales;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.ManagedProperty;
import javax.faces.bean.RequestScoped;

@ManagedBean
@RequestScoped
public class LoginTimeOutView {

    @ManagedProperty("#{autentiaAuthenticationProvider}")
    private transient AutentiaAuthenticationProvider authenticationProvider;

    public void setAuthenticationProvider(
            AutentiaAuthenticationProvider authenticationProvider) {
        this.authenticationProvider = authenticationProvider;
    }

    public String getNonce(){
        return authenticationProvider.calculateNonce();
    }
}
```

Aquí podéis ver cómo en este controlador se inyecta el proveedor de autenticación (AutentiaAuthenticationProvider) para pedirle que genere el token de caducidad que luego él mismo deberá conprobar cuando se le envíe el formulario.

6. Conclusiones

Bueno, pues con este ejemplo se puede comprobar que gracias a la arquitectura de autenticación que tiene Spring Security, y a la implementación por defecto que ya trae, podemos realmente hacer muchas cosas en apenas unos pocos pasos. Como en el ejemplo propuesto, añadiendo un token de caducidad generado en el servidor para el proceso de login.

Ya sabeís que esto es un ejemplo básico de lo que podemos hacer, pero espero que os sirva a alguno en vuestros casos particulares.

Saludos.

Anímate y coméntanos lo que pienses sobre este **TUTORIAL:**

Puedes opinar o comentar cualquier sugerencia que quieras comunicarnos sobre este tutorial; con tu ayuda, podemos ofrecerte un mejor servicio.

Enviar comentario

(Sólo para usuarios registrados)

» **Registrate** y accede a esta y otras ventajas «

COMENTARIOS