Autenticación y autorización en aplicaciones multiusuario

José María Palazón Romero

PID_00158712

UOC
Universitat Oberta
de Catalunya

Los textos e imágenes publicados en esta obra están sujetos –excepto que se indique lo contrario– a una licencia de Reconocimiento-NoComercial-SinObraDerivada (BY-NC-ND) v.3.0 España de Creative Commons. Podéis copiarlos, distribuirlos y transmitirlos públicamente siempre que citéis el autor y la fuente (FUOC. Fundación para la Universitat Oberta de Catalunya), no hagáis de ellos un uso comercial y ni obra derivada. La licencia completa se puede consultar en http://creativecommons.org/licenses/by-nc-nd/3.0/es/legalcode.es

Índice

1. Autenticación y autorización en Java 5
1.1. Autenticación con JAAS 6
1.1.1. Desarrollando nuestro propio módulo de autenticación 10
1.2. Autorización con JAAS 14
1.3. Autenticación y autorización: ejecutando acciones privilegiadas 17
1.4. Autenticación web usando JAAS 19

2. Autenticación y Autorización en PHP 27
2.1. Autenticación en PHP 27
2.2. Autorización y autenticación en CakePHP 30
2.2.1. Autenticación en CakePHP 30
2.2.2. Autorización en CakePHP 35
2.3. Autorización y autenticación en Zend Framework 40
2.3.1. Autenticación en Zend Framework 41
2.3.2. Autorización en Zend Framework 46

3. Autorización y autenticación en aplicaciones .NET 53
3.1. Autenticación en ASP.NET 53
3.1.1. Autenticación basada en Windows 57
3.1.2. Autenticación basada en formularios 60
3.1.3. Autenticación contra un árbol LDAP 70
3.1.4. Autenticación contra otros repositorios de información 71
3.1.5. Controles ASP.NET para la gestión de usuarios 74
3.1.6. Autenticación Passport 75
3.1.7. Gestión de roles mediante Proveedores 76
3.2. Autorización en ASP.NET 79
3.3. Identidad del código y personalización 81
3.3.1. Modelo SSO en una intranet 83
3.4. Autenticación en aplicaciones Windows Forms 84
3.4.1. Aplicaciones *stand-alone* 84
3.4.2. Aplicaciones cliente-servidor 87

1. Autenticación y autorización en Java

El Java Authentication and Authorization Services o JAAS apareció como paquete opcional para la versión 1.3 de Java2 SDK. Más tarde, en la versión 1.4, pasó a ser integrado como parte principal del SDK. Pero, ¿qué es exactamente JAAS? Es el *framework* que Java proporciona, para integrar de una manera uniforme, a lo largo de nuestra aplicación, la autenticación y autorización de usuarios, y gestionar el acceso a los diferentes recursos que en ella manejamos. Por ejemplo, utilizando los servicios nos proporciona esta capa de abstracción, no necesitaremos alterar el código de nuestras aplicaciones para soportar o cambiar el método de autenticación, ni preocuparnos del método de almacenamiento de las credenciales de nuestros usuarios.

Además, JAAS nos permitirá utilizar diferentes métodos de autenticación del usuario, o incluso desarrollar los nuestros propios que, más tarde, podremos incluir. Para ello, Java implementa el estándar Pluggable Authentication Module (PAM) que permite emplear y combinar estos diferentes métodos en un único sistema de autenticación. Si estructuramos nuestra aplicación de la manera correcta, no tendremos la necesidad de cambiar ni una sola línea de ella para utilizar un sistema diferente. Un simple cambio de configuración hará que nuestra aplicación utilice un sistema de autenticación diferente, de una forma muy similar a la utilizada para configurar PAM en Linux.

JAAS no está limitado al lado del cliente o del servidor; simplemente, proporciona una serie de herramientas que podremos utilizar en cualquier desarrollo que realicemos. Con sus clases e interfaces, podemos desarrollar aplicaciones cliente que necesiten autenticación o servidores que las utilicen para integrar un sistema centralizado.

Incluyendo JAAS a partir de la versión 1.4, el *framework* de seguridad de Java está formado por tres componentes principales:

- Java Cryptography Architecture o JCA. Estas API proporcionan la funcionalidad de cifrado y descifrado y algunas otras relacionadas, como la gestión de certificados.

- Java Secure Socket Extension o JSSE. Toma como base la funcionalidad ofrecida por JCA y la utiliza, junto con los mecanismos de conexión de red de Java, para proporcionar conectividad segura.

- JAAS. Esta API proporciona la funcionalidad de autorización y autenticación. Sus clases e interfaces se encuentran bajo el paquete `javax.security.auth.`

JAAS, en sí mismo, no da toda la funcionalidad necesaria y se fundamenta en otras API ya existentes de Java, entre ellas, algunas del *framework* de seguridad.

Si bien la API que proporciona JAAS no se caracteriza por una excesiva complejidad, sí introduce una serie de conceptos que deberemos conocer con anterioridad, comunes a otros sistemas de autenticación y autorización en cualquier campo.

Para empezar, el concepto de *subject*. En general, podemos pensar en el *subject* como en el usuario que realiza una operación y, por tanto, require una autorización para ello. Aunque esta asociación entre *subject* y usuario es la más común, debemos tener en cuenta que no siempre tiene que ser así. El *subject* puede ser cualquier entidad que necesite de autorización para realizar una operación: por ejemplo, una tercera aplicación.

Un concepto muy relacionado con el *subject* son los *principals*. Un *principal* es una identidad del *subject* y, por lo tanto, forma parte de él. Intentando trasladar este concepto al mundo real para hacerlo más sencillo, entenderemos cada *principal* como cada una de nuestras identificaciones que podremos utilizar en diferentes situaciones según se requiera.

> El ejemplo más claro serían nuestro carné de identidad, nuestro pasaporte y nuestro carné de conducir. Si bien todos nos identifican, cada uno de ellos tiene unos usos concretos y no todos nos permiten identificarnos en todas las situaciones.

En el caso del software, un *principal* puede ser un número de identificación de usuario, su *login* o su correo electrónico.

Además de esto, un *principal* puede representarnos de forma que podamos separar nuestras identidades en roles.

> Puede identificarnos como administrador del sistema mientras que otro, asociado al mismo *subject*, nos identificaría como simple usuario del mismo.
>
> Otro ejemplo que nos ayudará a clarificar el concepto de *principal* es un usuario en un sistema Unix. Éste estará compuesto por un *principal* que será su nombre de usuario, y por uno o más *principals* representando sus grupos. Así que el *subject* puede actuar utilizando uno o más de sus *principals* para identificarse.

Los *subjects*, además de *principals*, contienen *credentials*. Cada *credential* representa una forma de autenticación, como puede ser un conjunto usuario y password, o un certificado digital.

1.1. Autenticación con JAAS

Veamos ahora, en más detalle, el funcionamiento del proceso de autenticación usando JAAS.

La clase principal que deberemos utilizar es `javax.security.auth.login.LoginContext`, la cual nos proporciona el punto de entrada a los diferentes sistemas de autenticación que hayamos configurado. Más adelante, veremos cómo realizar dicha configuración, pero comenzaremos escribiendo el esqueleto básico de nuestra pequeña aplicación:

```
public class Simple {
   public static void main(String[] args) {
      LoginContext lc = null;
      try {
           lc = new LoginContext("Simple",
              new CallbackHandler() {
                 public void handle(Callback[] callbacks) {
                 }
              }
          );
          lc.login();
          Subject subject = lc.getSubject();
          System.out.println("Logged in as "+subject.toString());
      } catch (LoginException e) {
         System.out.println("Imposible identificar al usuario: "+e.getLocalizedMessage());
   }
   }
}
```

Ahora ya podemos ejecutar nuestro código. Veremos que, al intentar ejecutarlo, se generará una `java.lang.SecurityException` que nos indica que no se puede localizar el fichero de configuración.

Esto es debido a que, para el funcionamiento de JAAS, se requiere un fichero que indicará la configuración básica y los módulos disponibles. Como ya hemos dicho anteriormente, el estándar PAM que JAAS implementa nos permitirá especificar, en dicho fichero, el o los módulos que se ejecutarán para autenticar al usuario y su configuración.

La especificación de dicho fichero para nuestra pequeña aplicación es como sigue:

```
Simple { com.sun.security.auth.module.UnixLoginModule required };
```

Sencillo, ¿verdad? Veamos ahora qué indica cada parte del fichero de configuración. Para comenzar, antes de la apertura de llaves, tenemos el identificador de contexto, el cual es necesario, ya que, en un único fichero, podremos especificar más configuraciones identificando cada una mediante este nombre al comienzo. Utilizaremos este identificador para indicar al `LoginContext` cuál de las diferentes configuraciones deseamos utilizar.

Inmediatamente después de la apertura de la llave, tenemos el nombre de `LoginModule` que vamos a utilizar. En este caso, emplearemos únicamente uno indicando, además, que es requerido. Esto hará que si la autenticación no es satisfactoria para dicho módulo, la autenticación fallará. El *LoginModule* que hemos decidido emplear es el `com.sun.security.auth.module.UnixLoginModule` que JAAS proporciona por defecto. Su funcionamiento es muy sencillo, pues se encarga de obtener las credenciales del usuario que ejecuta el programa, tanto su nombre de usuario como sus grupos, y utilizarlas automáticamente dando al usuario por autenticado. No requerirá, por lo tanto, ninguna interacción por nuestra parte. Como su nombre indica, este módulo está especialmente indicado para sistemas Unix, por lo que si nos encontramos en un sistema Windows, deberemos utilizar `com.sun.security.auth.module.NTLoginModule` en su lugar. Podemos encontrar los diferentes *LoginModules* que *JAAS* proporciona por defecto junto con una descripción de su utilidad en la referencia del SDK para el paquete `com.sun.security.auth.module`.

Por último, en nuestro recorrido por el fichero de configuración, tenemos la palabra *required*. Esta palabra es un *flag* que indica cuál será el comportamiento de esta línea en la pila de autenticación. Básicamente, podemos definir tantos módulos de autenticación como sean necesarios, que serán ejecutados en orden de aparición en nuestro fichero de configuración. El *flag* indicado después del módulo de autenticación puede tomar los siguientes valores:

- *required*. Para que la autenticación del usuario sea válida, el módulo que acompaña a este *flag* debe devolver cierto. Independientemente del resultado, el resto de módulos especificados en la pila serán ejecutados, aunque su resultado no será tenido en cuenta.

- *requisite*. Similar a *required*, pero en caso de que el módulo devuelva falso, ningún otro módulo será llamado.

- *sufficient*. Si la ejecución de este módulo devuelve cierto, no se ejecutará ningún otro módulo y se considerará la autenticación como válida. En caso de que falle, se seguirán ejecutando los módulos en orden y el resultado de la autenticación dependerá de ellos.

- *optional*. No importa si este módulo devuelve cierto o no; la ejecución de los siguientes módulos continuará igualmente.

El siguiente código muestra cómo se comportan los distintos *flags* en la evaluación y permite observar el resultado de la autenticación:

```
import java.util.Map;

import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
```

```
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;

/**
* LoginModule que siempre retorna falso como resultado de la autenticación.
*/
public class AlwaysFalseLoginModule implements LoginModule {
   public boolean abort() throws LoginException {
      return true;
   }

   public boolean commit() throws LoginException {
      return true;
   }

   public void initialize(Subject subject, CallbackHandler callbackHandler,
         Map<String, ?> sharedState, Map<String, ?> options) {
   }

   public boolean login() throws LoginException {
      return false;
   }

   public boolean logout() throws LoginException {
      return true;
   }
}
```

```
import java.util.Map;

import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;

/**
* LoginModule que siempre retorna cierto como resultado de la autenticación.
*/
public class AlwaysTrueLoginModule implements LoginModule {
   public boolean abort() throws LoginException {
   return true;
   }

   public boolean commit() throws LoginException {
      return true;
   }
```

```
    public void initialize(Subject subject, CallbackHandler callbackHandler,
          Map<String, ?> sharedState, Map<String, ?> options) {
    }

    public boolean login() throws LoginException {
       return true;
    }

    public boolean logout() throws LoginException {
       return true;
    }}
```

1.1.1. Desarrollando nuestro propio módulo de autenticación

Aunque de una forma básica, ya hemos visto cómo hacer uso de los diferentes servicios de autenticación que JAAS nos proporciona. Sin embargo, en muchas ocasiones, esto no será suficiente y, para exprimir verdaderamente el potencial y la flexibilidad de JAAS, nuestras aplicaciones harán uso de nuestros propios *LoginModule*. Esto es así, ya que, generalmente, nuestras aplicaciones deberán hacer uso de unos sistemas de autenticación previamente existentes, a los que deberemos amoldarnos.

Para ello, necesitaremos desarrollar un módulo de autenticación o `LoginModule`. Esta interfaz incluirá los métodos necesarios para cualquier sistema de autenticación que, una vez implementados, podremos incorporar en nuestro sistema.

El `LoginModule` que vamos a crear, si bien no será muy útil, nos servirá para realizar diferentes pruebas e investigar el modo en que funciona el API internamente, a la vez que nos guiará por los pasos necesarios.

La idea es crear un `LoginModule` que solicitará al usuario una clave de acceso. Nuestro módulo accederá a un fichero que contendrá pares con el nombre de usuario y el *hash* de la clave de acceso para dicho usuario, comprobará si coincide, y autenticará al usuario.

La creación de un `LoginModule` requerirá que implementemos los siguientes métodos de interfaz: `initialize`, `login`, `commit`, `abort` y `logout`. Estos métodos son los que darán comportamiento a nuestro módulo y autenticarán a los usuarios.

El método *initialize* será llamado durante la inicialización de nuestro módulo y nos permitirá, además de la configuración de éste, inicializar conexiones y obtener los datos que el usuario haya especificado en el fichero de configuración.

```
 public void initialize (Subject subject,
                            CallbackHandler handler,
```

```
                              Map<java.lang.String, ?> sharedState,
                              Map<ava.lang.String, ?> options);
```

El primer parámetro *Subject* será el sujeto que estamos autenticando y que deberemos modificar añadiendo los *principals* que correspondan.

El segundo, el *handler*, nos permite interactuar con el usuario de una forma que independiza nuestro *LoginModule* del método de entrada de datos. Esto permitirá que nuestro módulo sea utilizado para autenticar a un usuario, introduciendo su nombre y su password en la consola, de igual manera que se utilizaría para autenticar a un usuario que introduce sus credenciales en un formulario web.

Por último, recibimos dos objetos *Map*. El *sharedState*, como su nombre indica, nos permitirá mantener un estado común entre todos los `LoginModules` que se utilicen en la autenticación. Por otro lado, el parámetro *options* incluye las opciones que se hayan detallado en el fichero de configuración.

El método `login()` será llamado para autenticar al *Subject* que hemos recibido durante la inicialización. En nuestro caso, en este método solicitaremos el nombre de usuario y la clave de acceso, y procederemos a comprobar que se corresponda con los valores almacenados. El siguiente código realizará exactamente esa operación:

```
public bolean login() throws LoginException {
   try {
cbh.handle(cbs);
verifyCredentials(nameCb.getName(), new String(passwdCb.getPassword()));
subject.getPrincipals().add(new SimplePrincipal(nameCb.getName()));
   return true;
      } catch (IOException e) {
e.printStackTrace();
      } catch (UnsupportedCallbackException e) {
         e.printStackTrace();
      }
      throw new FailedLoginException();
```

Nuestra pequeña función de *login* realizará una primera llamada al `CallbackHandler` que hemos almacenado en la inicialización del `LoginModule`. Esta llamada recorrerá los `Callback` registrados por la aplicación que esté usando el `LoginModule` y obtendrá las credenciales del usuario.

El método *verifyCredentials* es una pequeña función de utilidad que hemos creado con el objeto de comprobar si el usuario y la clave coinciden. No devuelve ningún valor, ya que si la autenticación falla, emitirá una excepción `javax.security.auth.login. FailedLoginException`.

Si la autenticación tiene éxito, nuestro módulo deberá actualizar los *principals* del *Subject*, al que hemos guardado una referencia en la función de inicialización. Con esto, concedemos a nuestro *Subject* una identidad, con la que podrá ser posteriormente autenticado para comprobar cuándo dispone de permisos para ejecutar una acción.

Nuestro módulo utiliza también un *principal* desarrollado específicamente para él. Este *principal*, únicamente, se encargará de almacenar la identidad del usuario autenticado.

El *principal* no tiene por qué estar directamente relacionado con el módulo que lo utiliza, y diferentes módulos podrían compartir el mismo *principal*. Generalmente, esto no será así, puesto que las identidades generadas por diferentes sistemas de autenticación tienen distintos requisitos de almacenamiento. En nuestro caso, una única cadena alfanumérica será suficiente, quedando nuestra clase como sigue:

```
public class SimplePrincipal implements Principal {
   private final String name;
   public SimplePrincipal(String name) {
      this.name = name;
   }
   public String getName() {
   return name;
   }
   public String toString() {
      return getName();
   }
}
```

Los métodos `commit` y `abort` son utilizados en el caso de autenticaciones que requieran de dos fases para su funcionamiento. Esto permite que nuestro módulo libere recursos o modifique el comportamiento en función de si la autenticación ha sido correcta, en cuyo caso se llamará al método `commit`, o fallida, llamando al método `abort`. Si, por ejemplo, nuestro módulo reserva recursos en otro nodo o mantiene un estado de la autenticación mientras ésta se realiza, estos métodos son el lugar adecuado para liberar dicha sesión o informar al nodo remoto que la información no es necesaria.

Nuestro módulo de ejemplo es muy sencillo y no require de esta segunda fase de autenticación, por lo que los métodos `commit` y `abort` estarán vacíos y devolverán *true*.

El último paso es emplear nuestro recién creado módulo dentro de una autenticación. Aunque más adelante veremos su utilización en un ejemplo más completo, ahora simplemente nos quedaremos con un pequeño código de prueba. Basándonos en el ejemplo de autenticación, que se fundamentaba en

obtener las credenciales del usuario actualmente autenticado en el sistema, haremos unos pequeños cambios para proporcionar un usuario y una clave, que el `LoginModule` utilizará para comprobar las credenciales:

```
lc = new LoginContext("MyLoginModuleTest",
   new CallbackHandler() {
   public void handle(Callback[] callbacks)
      throws IOException, UnsupportedCallbackException {
         for(Callback cb : callbacks) {
            if (cb instanceof NameCallback) {
                   (NameCallback) cb).setName("usuario1");
            } else if (cb instanceof PasswordCallback) {
                   ((PasswordCallback) cb).setPassword("password1".toCharArray());
            } else if (cb instanceof TextOutputCallback) {
                  System.out.println(((TextOutputCallback) cb).getMessage());
            }
         }
      }
   }
   );
```

Vemos cómo, en esta ocasión, en lugar de no proveer de *Callbacks*, proporcionamos soporte para tres tipos diferentes. El primero, el `javax.security.auth.callback.NameCallback`, es incluido en la pila de llamadas cuando el `LoginModule` necesite del nombre de usuario. El segundo, el `javax.security.auth.callback.PasswordCallback`, de forma similar, se utilizará cuando se necesite de la *password* del usuario. Por último, el `javax.security.auth.callback.TextOuputCallback` es añadido por el `LoginModule` cuando necesita comunicar alguna información en forma de mensaje al usuario.

Para simplificar el ejemplo, hemos añadido la respuesta a los diferentes `Callback` de forma estática, pero suele ser habitual que estos `Callback` pidan la intervención del usuario, ya sea mediante un cuadro de diálogo o mediante la consola, pidiendo las credenciales.

Por último, indicaremos en la configuración de JAAS dentro del fichero *JAAS.conf*, que deseamos utilizar nuestro `LoginModule` para el *Context* "`MyLoginModuleTest`":

```
MyLoginModuleTest {
   ejemplos.login.modules.MyLoginModule required file="passwd";
};
```

1.2. Autorización con JAAS

La autenticación del usuario sólo es la primera parte del proceso que nos permite asociar un *subject* a un contexto de acceso. La segunda parte, la autorización, tiene únicamente sentido después de que el usuario ha sido autenticado.

La autorización con JAAS, como en cualquier otro sistema, consiste en determinar cuándo un *subject* tiene permiso para realizar una determinada acción; la única peculiaridad de JAAS es que nos permite añadir la dimensión del código que se está ejecutando.

Los permisos son clases que heredan de `Java.security.Permission` y representan la habilidad de realizar una o más acciones sobre un recurso determinado. Pueden representar desde acceso a ficheros como `Java.io.FilePermission`, hasta acceso a propiedades como `java.util.PropertyPermission`. Más adelante, veremos la manera de utilizar y configurar estos permisos para controlar el acceso sobre los recursos.

Además de su tipo, los permisos pueden opcionalmente contener una serie de acciones, que permitirán un control más granular de éstos. En el ejemplo del `java.util.PropertyPermission`, estas acciones serían "*read*" y "*write*".

Cuando comparamos dos permisos, más que comprobar que son exactamente iguales, comparamos si uno se incluye en el otro para determinar si la acción es válida. Supongamos que tenemos un permiso que nos permite leer y escribir ficheros que se encuentren bajo el directorio "/seguro", y queremos leer el fichero "/seguro/mucho". Si definimos los permisos como tuplas, obtendríamos algo como:

- Otorgado: ("/seguro", "leer y escribir")
- Requerido: ("/seguro/mucho", "leer")

Como vemos, una simple comprobación de igualdad no nos dirá si poseemos el permiso que requerimos. En cambio, gracias al método *implies*, podremos comprobar que ("/seguro", "leer y escribir") *implies* ("/seguro/mucho", "leer") y que, por lo tanto, estamos autorizados a realizar dicha operación.

Los permisos pueden ser otorgados a un código determinado mediante un archivo de políticas que, más adelante, veremos cómo definir. Dichos permisos otorgarán la capacidad de realizar una acción a un determinado código y, opcionalmente, a un *principal*.

La clase **`java.lang.SecurityManager`** es el punto de entrada a la API de autorización de JAAS y permite que nuestras aplicaciones implementen determinadas políticas de seguridad. Mediante esta clase, podremos comprobar cuándo una determinada acción está autorizada o no en un determinado contexto. Esta clase no es especialmente útil si nuestra única intención es comprobar

cuándo nuestro código tiene permisos sobre acciones ya existentes; sin embargo, será vital cuando deseemos saber qué partes determinadas de la ejecución se encuentran sujetas a restricciones.

Para obtener una instancia de `java.lang.SecurityManager` deberemos llamar al método `System.getSecurityManager()`. Este método nos devolverá la instancia actual que nos permitirá acceder a la funcionalidad de JAAS. Un detalle importante a tener en cuenta es que, por defecto, Java no proporciona un `java.lang.SecurityManager` por lo que la llamada a `System.getSecurityManager()` devolverá *null*. Para activar las capacidades de autorización, tendremos que activar la propiedad `-Djava.security.manager`, lo que inicializará el `SecurityManger`.

Aunque la clase `java.lang.SecurityManager` tiene muchos métodos, la mayor parte son heredados de versiones anteriores del *framework* de seguridad de Java y su uso está desaconsejado. Generalmente, utilizaremos el método `checkPermission` que determina, en el contexto actual, si un permiso ha sido concedido sobre un determinado recurso. La mayor parte de los métodos checkXXXX que encontraremos en esta clase hacen uso en algún momento del método `checkPermission`.

Las **políticas** son el punto de encuentro entre el *principal*, los permisos y las acciones. En ellas, se define cuándo un *principal* tiene permisos para ejecutar una determinada acción. Para ello, utilizará un fichero de texto que nos permitirá especificar dichas políticas, por ejemplo:

```
grant codebase "file:./SamplePermissions.jar",
      Principal login.MySampleLoginModule
            "usuario1" {
   permission java.util.PropertyPermission "java.home", "read";
};
```

Esta política indica que se concede (*grant*) el permiso `java.util.PropertyPermission` al código contenido en `SamplePermissions.jar` y al *principal* "usuario1" sobre la propiedad "`Java.home`". Además, en el caso de la `PropertyPermission`, hemos especificado también la acción (*read*) que deseamos permitir.

El formato completo de una política es el siguiente:

```
grant <signer(s) field>, <codeBase URL>
   Principal Principal_class "principal_name" {<br/>
   permission perm_class_name "target_name", "action";
   ....
   permission perm_class_name "target_name", "action";
   };
```

Veamos qué utilidad tiene cada campo de esta definición. Las políticas comienzan con la palabra clave "*grant*", que indica que se va a conceder permisos sobre un determinado recurso. Posteriormente, podemos indicar el firmado del código, de forma que aseguremos que éste no ha sido alterado.

Después, encontramos la URL que definirá a qué parte de nuestro código damos los permisos. En nuestro caso, en el que estamos ejecutando el código localmente, la URL tendrá un formato "`file://<fichero>`", donde fichero puede indicar una ruta y contener *wildcards* al final de la cadena. De la misma forma, fichero puede ser uno o más ficheros *class*, o incluso ficheros *jar*.

Algunos ejemplos de estas URL serían:

Patrón	Descripción
file://./MiClase.class	Concedería permisos a la clase MiClase en el directorio de ejecución.
file://./utilidades/MiPrograma.jar	Concedería permisos al código contenido en el fichero *jar* del directorio utilidades.
file://./librerias/red/*	Concedería permisos a todo el código contenido en directorio librerias/red, independientemente de si se trata de ficheros *jar* o ficheros *class*.

Ejemplos URL en políticas

Siguiendo en orden con la especificación de la política, encontraremos la palabra clave *Principal*, seguida de la clase que indica el tipo del mismo. A continuación, deberemos especificar el nombre del *Principal*, que dependerá de la clase *Principal* que estemos utilizando. Algunos ejemplos de definición de *Principals*son:

Definición	Módulo de autenticación
Principal com.sun.security.auth.UnixPrincipal "user"	`com.sun.security.auth.Module.UnixLoginModule`
Principal com.sun.security.auth.NTUserPrincipal "user"	`com.sun.security.auth.Module.NTLoginModule`

Ejemplos de *Principals* para autorizar al usuario con id "user"

En último lugar, encontramos la lista de permisos y acciones. Su definición incluirá la palabra clave "*permission*" junto con el nombre de la clase que lo implementa. El siguiente campo representa el "*target*" o recurso, y dependerá del tipo de permiso, al igual que las opciones.

Un ejemplo sería la siguiente definición:

```
permission java.util.PropertyPermission "java.home", "read";
```

Con ello, indicamos que se otorga permiso de lectura sobre la propiedad "*java.home*".

Una vez creado nuestro fichero de políticas, deberemos indicar que queremos utilizarlo mediante el uso de la propiedad `Java.security.policy`. Podemos indicarlo como parámetro al ejecutar nuestro código desde la línea de comandos utilizando el parámetro *-D*, al igual que hicimos con el fichero de configuración para la autenticación inicial.

> Es recomendable utilizar la opción `-Djava.security.debug=all` de la JVM que nos proporcionará información valiosa de depuración sobre el proceso de autenticación y autorización.

1.3. Autenticación y autorización: ejecutando acciones privilegiadas

Cuando ya hemos definido las políticas que se aplicarán a nuestro código, es el momento de comprobar su funcionamiento. Como hemos dicho anteriormente, la política definirá el momento en que nuestro código tiene permisos para ejecutar una determinada acción.

Vamos a crear un pequeño programa de ejemplo que nos permitirá combinar todo lo visto hasta ahora, desde la autenticación del usuario hasta la autorización de una determinada acción.

El esqueleto inicial de nuestra aplicación será el código y la configuración que utilizamos en el ejemplo de autenticación. Autenticaremos al usuario y, una vez obtenido el *subject*, lo utilizaremos para ejecutar una sencilla acción privilegiada, en este caso, el acceso a una propiedad del entorno.

Lo primero que tendremos que añadir será el mencionado fichero de políticas. La acción que queremos autorizar es la lectura de la propiedad del sistema "*Java.home*" que indica la ubicación del JRE. Para ello, utilizaremos la siguiente política:

```
grant codebase "file:./bin/*" {
   permission javax.security.auth.AuthPermission "createLoginContext.Simple";
   permission javax.security.auth.AuthPermission "doAsPrivileged";
};
grant codebase "file:./bin/*",
   Principal com.sun.security.auth.UnixPrincipal
            "usuario" {
   permission java.util.PropertyPermission "java.home", "read";
};
```

Veamos, paso a paso, cómo funciona cada parte de esta política. La primera parte crea una autorización que no depende de ningún *Principal* y, por tanto, será concedida a cualquier usuario de la aplicación sin necesidad de que éste se encuentre autenticado. Los dos `AuthPermission` los otorgamos sobre dos acciones "`createLoginContext.Simple`" y "`doAsPrivileged`".

En el primer caso, una vez activada la autenticación de Java, necesitamos permisos para la creación de nuevos `LoginContext` que nos permitan autenticar al usuario. Esta primera línea indica que queremos otorgar permisos a todos los usuarios para crear un nuevo `LoginContext` de tipo *Simple*.

La segunda línea nos permitirá llamar al método `doAsPrivileged` de la clase *Subject*. Dicho método nos permitirá ejecutar una acción privilegiada, en nuestro caso la lectura de la propiedad, asociando a esta acción un determinado *Subject* sobre el que se evaluarán las acciones a tomar. Dicho de otra manera, es la forma que tenemos de asociar un *Subject* y la acción que queremos ejecutar.

Generalmente, no encontraremos ambas políticas juntas, ya que el código que se encargará de la creación del contexto no se encontrará bajo el mismo dominio de seguridad que el resto de la aplicación. En nuestro ejemplo, a efectos de simplificar el código, utilizaremos una política que indicará los permisos tanto para la autenticación del usuario como para la acción concreta a ejecutar.

Una vez creada nuestra política, sólo nos queda añadir el código para ejecutar nuestra acción privilegiada. Dicho código será algo como:

```
Subject.doAsPrivileged(subject, new PrivilegedAction<Object>() {
   public Object run() {
            System.out.println(System.getProperty("java.home"));
            return null;
   }
}, null);
```

Este código, aunque no muy útil, nos permitirá comprobar cómo se ejecutan los controles de acceso sobre las acciones privilegiadas.

Si hemos realizado todo correctamente, al ejecutar nuestro programa deberíamos ver el valor de la propiedad `java.home`:

```
java.home=/System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Home
```

El resultado dependerá del sistema operativo en el que estemos ejecutando nuestro programa.

Si obtenemos una excepción de acceso, comprobará primero que los ficheros de políticas y de configuración se encuentran en el directorio actual de ejecución.

Si todo ha ido bien, el siguiente paso será comprobar que la política tiene algún efecto sobre nuestras acciones; para ello, simplemente cambiaremos el nombre del usuario por uno no existente en nuestro sistema y volveremos a ejecutar el programa. Esta vez, deberíamos obtener algo como:

```
Exception in thread "main" java.security.AccessControlException:
    access denied (java.util.PropertyPermission java.home read)
```

Con esto, hemos visto el modo en que se realiza todo el proceso de autenticación y autorización utilizando el *framework* de *JAAS*. Es extenso y complejo, pero ofrece infinidad de posibilidades.

1.4. Autenticación web usando JAAS

Un entorno muy apropiado, y que requiere una correcta gestión de la autorización y autenticación, son las aplicaciones web. El desarrollo de aplicaciones web en Java es un mundo muy extenso que daría para llenar estanterías de libros sobre el tema.

Nos vamos a centrar en cómo conseguir que una aplicación muy sencilla, mediante *Servlets*, nos permita autenticar a un usuario y utilizar esa autenticación para autorizarle a realizar diferentes acciones.

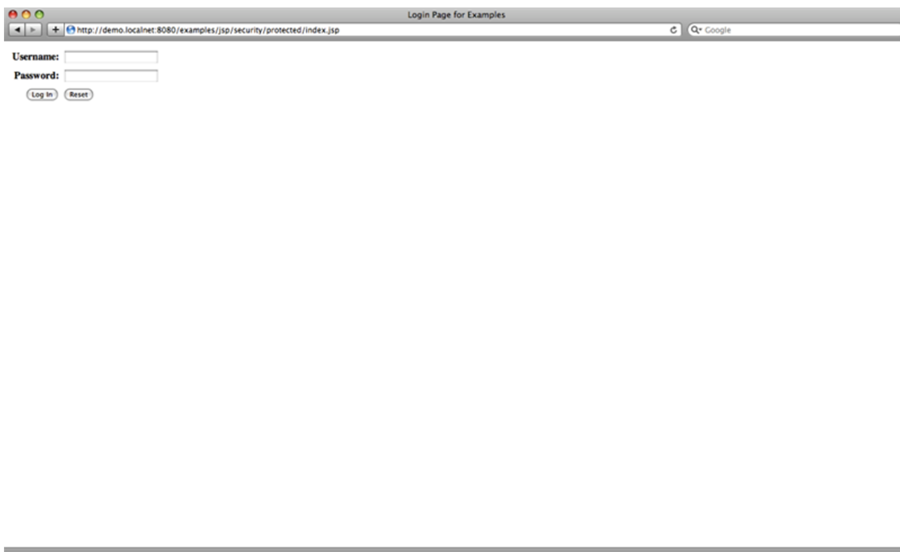
Desafortunadamente, la especificación de *Servlets* no habla de cómo la autenticación y autorización debe ser realizada, ni mucho menos cuál es el papel de JAAS en todo ello. Esto hace que, aunque los conceptos son similares entre los diferentes servidores de aplicaciones, cada uno implemente dichos conceptos de forma diferente, haciendo de la configuración de seguridad algo muy específico.

Para nuestro ejemplo, utilizaremos el servidor de aplicaciones Apache Tomcat, versión 6 por varias razones. La primera es que está disponible gratuitamente y su uso está muy extendido. La segunda es que la mayoría de los conceptos que apliquemos en Tomcat son fácilmente extrapolables a servidores de aplicaciones más complejos y potentes. Aprovecharemos para utilizar el *LoginModule* que creamos anteriormente con nuestra aplicación y que nos permitía almacenar las passwords en un archivo local.

Lo primero que debemos conocer, antes de lanzarnos a implementar seguridad en Tomcat, es el concepto de *Realm*. Tomcat define en su documentación un *Realm* como una base de datos de usuarios y passwords, unida a los *roles* que pueden desempeñar dichos usuarios. Mediante este concepto, Tomcat nos permite utilizar diferentes mecanismos o *backends* de autenticación como, por

ejemplo, *JDBC*, para mantener nuestros usuarios en una base de datos. También nos permite escribir nuestro propio *Realm* pero, en nuestro caso concreto, nos centraremos en el *JAASRealm*. Dicho *Realm* nos proporciona una interfaz para que Tomcat utilice los diferentes *LoginModules*, bien ya existentes, o como en este caso, uno construido a medida.

Tomcat 6 incluye por defecto, dentro de sus ejemplos, una aplicación que muestra cómo utilizar la autenticación en el servidor web, utilizando un formulario para solicitar las credenciales. Emplearemos dicha aplicación como base para, aplicando algunos cambios, hacer que utilice nuestro propio `LoginModule`. Una vez instalado Tomcat y los ejemplos, simplemente apuntaremos el navegador a http://localhost:8080/examples/jsp/security/protected y la siguiente pantalla aparecerá:



Login de la aplicación de ejemplo

> Modificaremos la configuración incluida con los ejemplos que acompañan a Tomcat. Si no se desea volver a instalarlos, hay que realizar una copia de seguridad de forma que se pueda devolver a su estado original.

Uno de los *Realms* que Tomcat incluye por defecto es el ***MemoryRealm***. Este *Realm* no está pensado para producción, pero es el que encontraremos por defecto para las aplicaciones de ejemplo y consola de administración de Tomcat, una vez lo hemos instalado.

Su funcionamiento es sencillo: lee el fichero *tomcat-users.xml* del directorio de configuración de Tomcat y carga los usuarios en memoria. Este fichero tiene el siguiente formato:

```
<tomcat-users>
   <user name="tomcat" password="tomcat" roles="tomcat" />
   <user name="role1"  password="tomcat" roles="role1" />
```

```
    <user name="both"   password="tomcat" roles="tomcat,role1" />
</tomcat-users>
```

El fichero se explica por sí mismo. Como ya hemos dicho, no está pensado para ser utilizado en un entorno "real", por varias razones. En primer lugar, las contraseñas están almacenadas en texto plano y, en segundo lugar, si tenemos que almacenar una base de usuarios medianamente grande, su gestión puede ser complicada.

Nuestro `LoginModule` viene a solucionar el primero de los dos problemas. Tiene una funcionalidad similar, pero no almacenamos las contraseñas, sino su *hash*. Incluso con esta pequeña mejora no se recomienda utilizar este sistema en entornos de producción, ya que, aunque mejora ligeramente la seguridad, sigue siendo vulnerable si se consigue el fichero; por otro lado, además, seguimos teniendo el problema de su administración.

Para **configurar nuestro *Realm***, lo primero que deberemos hacer es indicarle a Tomcat qué *Realm* queremos utilizar en lugar del que está configurado por defecto. Esta configuración se encuentra en el fichero `server.xml`. El tag *Realm* puede ser especificado en diferentes sitios en función del alcance que tenga nuestro sistema de autenticación; en nuestro caso, nos limitaremos a añadirlo dentro del tag *Context* de nuestra aplicación. La documentación de Tomcat describe cómo puede ser usado dentro de *Host* o *Engine* y sus implicaciones. Añadiremos el siguiente tag:

```
<Realm className="org.apache.catalina.realm.JAASRealm"
    appName="MyLoginModuleTest"
    userClassNames="login.principals.SimplePrincipal"
    roleClassNames="login.principals.SimpleGroupPrincipal" />
```

En primer lugar, el atributo *className* indica el *Realm* que queremos utilizar; en nuestro caso, `org.apache.catalina.realm.JAASRealm`, que es el encargado de enlazar el sistema de autenticación de Tomcat con JAAS.

`AppName` indica el nombre de la aplicación que estamos empleando y se utilizará para indicar cuál de las diferentes políticas de autorización aplica dentro de las que definamos en nuestro `JAAS.config`. Debemos emplear, por lo tanto, el mismo nombre de la política de autorización que queramos utilizar.

Por último, indicaremos `userClassNames` y `roleClassNames` para señalar qué clases almacenarán el *Principal* de usuario y de su rol. Veremos, más adelante, qué modificaciones debemos realizar a nuestro *LoginModule* para que soporte el concepto de *Rol*.

Una vez configurado el *Realm*, ya podemos ir al fichero `web.xml` de nuestra aplicación web para habilitar la autenticación.

Tomcat nos proporciona varios métodos de autenticación, como *BASIC, DIGEST* o *FORM*. En nuestro caso, queremos proporcionar nuestra propia página de autenticación, por lo que utilizaremos *FORM*. Para ello, habremos de especificar la configuración de *login* de nuestra aplicación:

Estas credenciales van a viajar en texto plano desde el navegador al servidor y, por tanto, como cualquier otro sistema de autenticación, deben ir protegidos por HTTPS.

```
<login-config>
   <auth-method>FORM</auth-method>
   <realm-name>Example Form-Based Authentication Area</realm-name>
   <form-login-config>
      <form-login-page>/jsp/security/protected/login.jsp</form-login-page>
      <form-error-page>/jsp/security/protected/error.jsp</form-error-page>
   </form-login-config>
</login-config>
```

Esta configuración se explica por sí misma. Nos permite cambiar el método de autenticación entre los antes mencionados y nos permite indicar cuáles son las páginas de autenticación y de error cuando el usuario no ha sido autenticado.

Los ***Roles* en Tomcat** son el mecanismo que nos permite realizar autorización de los usuarios para la ejecución de determinadas acciones. Una acción estará restringida a unos determinados *Roles*, negándose el acceso a aquellos que no dispongan de ese *Rol*.

Una vez hemos definido la estructura básica de nuestra aplicación, deberemos especificar qué roles podrán desempeñar los usuarios.

La configuración es muy sencilla, ya que se limita a una enumeración de los mismos con el siguiente formato:

```
<security-role>
   <role-name>role1</role-name>
</security-role>
<security-role>
   <role-name>tomcat</role-name>
</security-role>
```

Esta lista deberá contener todos los roles que más adelante queramos usar para gestionar la autorización en el acceso a determinadas URL.

Por último, deberemos especificar la *ACL* de nuestra aplicación utilizando los roles existentes. La aplicación de ejemplo que Tomcat nos proporciona contiene la siguiente definición:

```
<security-constraint>
   <display-name>Example Security Constraint</display-name>
   <web-resource-collection>
      <web-resource-name>Protected Area</web-resource-name>
      <url-pattern>/jsp/security/protected/*</url-pattern>
      <http-method>DELETE</http-method>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
      <http-method>PUT</http-method>
   </web-resource-collection>
   <auth-constraint>
      <role-name>tomcat</role-name>
      <role-name>role1</role-name>
   </auth-constraint>
</security-constraint>
```

Esta definición indica que todas aquellas páginas que se encuentren por debajo de */jsp/security/protected* deberán ser protegidas para cualquiera de los métodos especificados en la lista. Además, incluimos qué *Roles* tendrán acceso a esas páginas, negando el acceso a cualquier usuario que no disponga de ese *Rol*.

Nada impide utilizar diferentes niveles de autorización para diferentes métodos HTTP. Podremos definir varios *security-contraint* que indiquen la misma ruta, pero diferentes métodos o incluso *Roles*.

Debido a que el concepto de *Rol* no existe como tal en JAAS, utilizaremos una clave de configuración indicando **los grupos** a los que pertenecerá el usuario. Estos grupos identificarán a qué *Roles* pertenece el usuario de forma que, con sólo unos pequeños cambios, podamos hacer uso de los *security-constraint*.

Cuando inicialmente creamos el *LoginModule*, no tuvimos en cuenta el hecho de que el usuario pudiera pertenecer a grupos, pero con unas pequeñas modificaciones podremos obtener el mismo efecto. Crearemos una nueva clase *Principal*, que heredará de nuestro *SimplePrincipal*, a la que llamaremos *SimpleGroupPrincipal*, con el siguiente código:

```
public class SimpleGroupPrincipal extends SimplePrincipal {
   public SimpleGroupPrincipal(String name) {
      super(name);
   }
}
```

Ahora añadiremos el código para generar estos *Principal*. Lo haremos de una forma muy sencilla; cuando el usuario sea autenticado correctamente, comprobaremos si el fichero contiene una clave `usuario.groups=grupo1,grupo2,...,grupon` que nos indique a qué grupos pertenece. Añadiremos un `SimpleGroupPrincipa` por cada uno de los grupos.

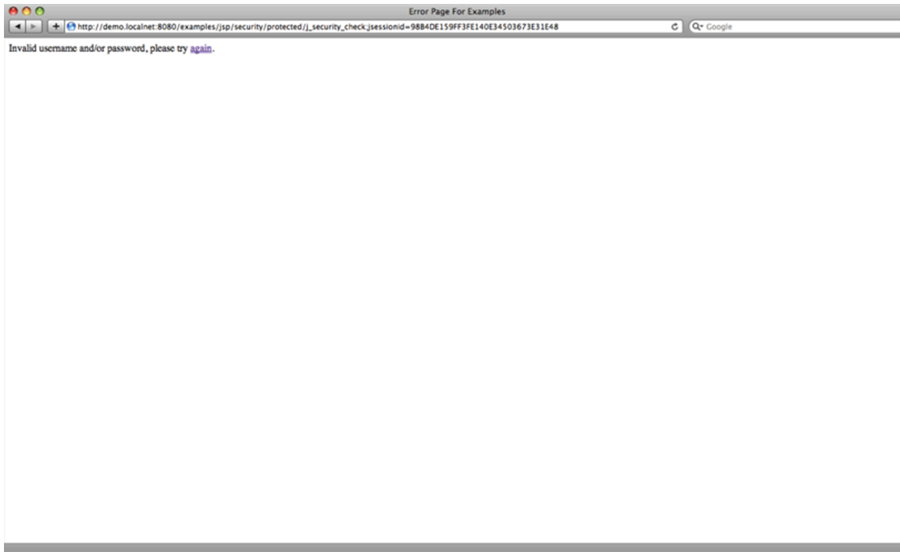
Por lo tanto, nuestro método `login()` quedará como sigue:

```
public boolean login() throws LoginException {
   try {
      cbh.handle(cbs);
      verifyCredentials(nameCb.getName(), new String(passwdCb.getPassword()));
      subject.getPrincipals().add(new SimplePrincipal(nameCb.getName()));
      String[] groups = getUserGroups(nameCb.getName());
      for(String group : groups) {
         subject.getPrincipals().add(new SimpleGroupPrincipal(group));
      }
      return true;
   } catch (IOException e) {
      e.printStackTrace();
   } catch (UnsupportedCallbackException e) {
      e.printStackTrace();
   }
   throw new FailedLoginException();
}
private String[] getUserGroups(String username) {
   String groups = users.getProperty(username+".groups");
   if (groups==null) {
      return new String[0];
   }
   return groups.split(",");
}
```

Debemos modificar también ahora el fichero de passwords que, inicialmente, creamos para añadir los grupos. El usuario1 pertenecerá a los grupos "tomcat" y "role1", mientras que el usuario2 no pertenecerá a ninguno. Esto nos permitirá comprobar cómo afectan los roles a los permisos del usuario autenticado.

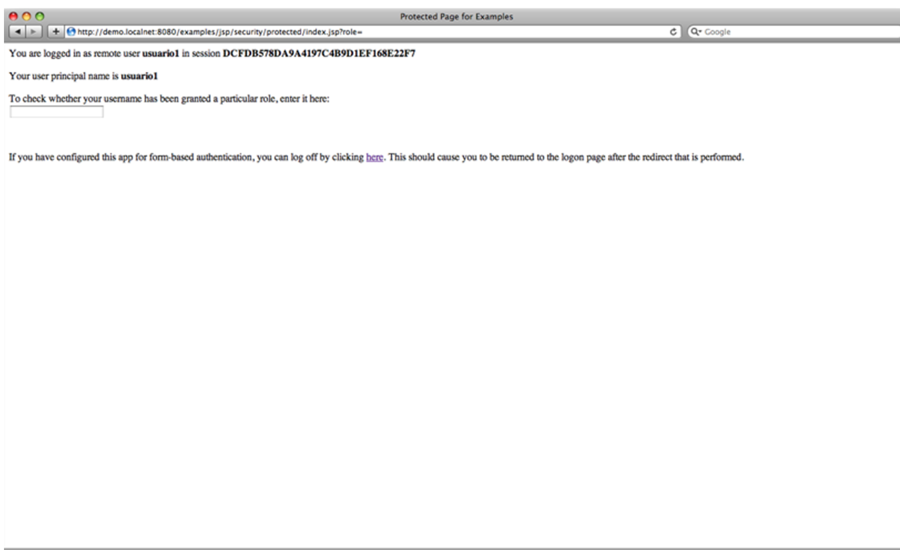
Todas las clases que creemos deberán estar disponibles en el *classpath* de Tomcat para que JAAS pueda acceder a ellas. Asimismo, hay que indicar la ruta en la que se encuentra nuestro fichero de configuración. Esto lo conseguiremos utilizando la opción `-Djava.security.auth.login.config=$CATALINA_HOME/conf/JAAS.config` en la inicialización, y sustituyendo la ruta por la adecuada en nuestro caso. Una vez hecho esto, deberemos reiniciar Tomcat para que cargue de nuevo nuestro fichero de configuración.

Una vez reiniciado, nos dirigiremos a la ruta donde se encuentra el ejemplo de autenticación que se entrega con Tomcat en la URL http://localhost:8080/examples/jsp/security/protected donde encontraremos la página de *login* que vimos anteriormente. Introduciremos uno de los usuarios predefinidos de Tomcat para comprobar que la autenticación no se hace contra la configuración por defecto; tomcat/tomcat servirá. Comprobamos que nuestro *login* fallará:



Usuario/password incorrecto

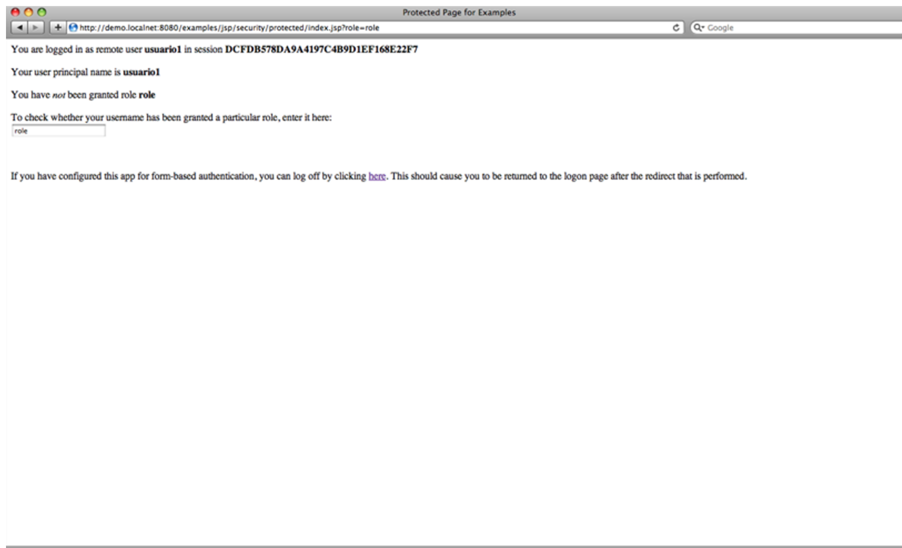
Sin embargo, si ahora pasamos a utilizar uno de nuestros usuarios, como por ejemplo, usuario1/password1, comprobaremos que el *login* se realiza satisfactoriamente.



Usuario/password correcto

Si introducimos uno de los roles que se encuentran configurados en nuestro fichero `passwd`, como por ejemplo Tomcat, comprobaremos que, efectivamente, nuestro módulo ha añadido también los *Principal* correspondientes a los grupos.

Si ahora hacemos *logout* y utilizamos usuario2 en su lugar, obtendremos la siguiente página de error:



Usuario no pertenece al *Rol*

Esto es debido a que, en esta ocasión, nuestro usuario2 no pertenece a ninguno de los roles que anteriormente dimos de alta en el *security-constraint* que protege esta página.

2. Autenticación y Autorización en PHP

A diferencia de Java o ASP.NET, desgraciadamente, en PHP no existe un estándar tan claro de autenticación y autorización que nos permita implementar, de una forma transparente y única, los elementos de seguridad adecuados. PHP nos da por defecto unos simples mecanismos para utilizar autenticación HTTP; sin embargo, no trata en ningún momento la autorización. Por este motivo, y por el hecho de que, salvo para pequeñas aplicaciones, no suelen ser usados directamente, no nos centraremos en ellos. Son sencillos de utilizar y podremos encontrarlos documentados y con ejemplos en la referencia de PHP.

Debido a esto, diferentes *frameworks* en PHP proporcionan diferentes soluciones para el mismo problema, cada uno con sus pros y sus contras. Aunque cada uno mantiene una filosofía diferente, los conceptos son iguales para todos ellos; únicamente, la asociación de dichos conceptos a los mecanismos que los implementan varía de un *framework* a otro. Si tenemos los conceptos de autenticación y autorización claros, el entender la funcionalidad en cada *framework* se reducirá a entender dónde están situados estos mecanismos.

En este capítulo, veremos cómo se manejan estos conceptos en dos de los *frameworks* más conocidos de PHP, tanto la autenticación de usuarios remotos como la posterior autorización de los mismos para realizar diferentes acciones.

La funcionalidad de los *frameworks* es, principalmente, la de facilitarnos el diseño de tareas repetitivas o muy comunes mediante el uso de patrones y estructuras de datos que, siguiendo una estructura lógica, nos lleven a realizar dichas tareas.

No nos centraremos en la instalación de los diferentes *frameworks*, ya que se encuentra perfectamente documentada en los sitios web al ser muy específica del entorno en el que realicemos las pruebas.

2.1. Autenticación en PHP

Antes de empezar a ver el funcionamiento de los diferentes *frameworks*, una aclaración: PHP nos da los mecanismos necesarios para acceder a los elementos de autenticación que HTTP proporciona. Podremos por tanto, sin necesidad de ningún *framework*, utilizar la autenticación básica o *digest* de HTTP. El problema principal de este sistema de autenticación es que, sin un sistema de autorización adecuado, no sirve de mucho y deja enteramente la responsabilidad de determinar si una acción puede realizarse al código de nuestro clien-

te. Esto provoca que muchas aplicaciones escritas en PHP integren la autorización de usuarios en la lógica de negocio, lo que complica enormemente su funcionamiento y mantenimiento.

Este capítulo no pretende ser una referencia sobre funcionamiento y filosofía de cada uno de los *frameworks*. La elección del *framework* depende de muchos factores que escapan al alcance de este texto. Durante este capítulo, nos centraremos únicamente en las capacidades de autenticación y autorización de los *frameworks*, sin entrar en detalles en el resto de componentes.

Dicho esto, vamos a explicar por encima el funcionamiento de la autenticación en PHP. Básicamente, podemos encontrar dos modos de autenticación que, como hemos dicho antes, se corresponden con los métodos de autenticación de HTTP y variarán el canal de transmisión de las credenciales.

La mejor manera de demostrar el funcionamiento de las diferentes API **es con ejemplos**. Nos centraremos, por tanto, en el desarrollo de una pequeña aplicación, muy simple, en la que implementaremos el control de usuarios y el acceso a algún recurso. Nuestro objetivo no es mostrar el uso de todas las posibilidades que los *frameworks* proporcionan, por lo que nos limitaremos a una sencilla aplicación que nos sirva de entorno de pruebas con el que poder jugar y ver los cambios.

La aplicación hará las veces de un pequeño libro de visitas. Contemplaremos los siguientes casos:

- Los usuarios anónimos podrán ver las notas dejadas por todos.
- Los usuarios autenticados podrán crear notas que irán acompañadas de su nombre.
- Los usuarios con permisos de administración podrán borrar las notas.

Estos tres casos nos permitirán explorar los conceptos de autenticación, para controlar qué usuarios añaden comentarios, y de autorización al gestionar cuáles de dichos usuarios tienen permisos para eliminar comentarios.

Aunque trabajaremos con dos *frameworks* diferentes, la estructura de la aplicación será muy similar en ambos casos, con variaciones muy específicas de cada implementación.

Siguiendo el patrón modelo vista controlador (MVC), nuestra aplicación proporcionará un controlador con las acciones de consultar, añadir y eliminar notas. Consultar y añadir tendrán una vista asociada que proporcionará la interfaz de usuario necesaria para cada acción. Eliminar no dará ninguna vista, ya que será utilizada directamente desde la acción de consultar y nos devolverá a la vista principal una vez se haya completado la operación.

En el caso de la consulta de notas, proporcionaremos una lista con cada una de las notas dejadas por los usuarios. Cuando el usuario disponga de permisos para ello, añadiremos un pequeño enlace que nos permitirá eliminar cualquiera de las notas.

La acción de añadir ofrecerá una interfaz con un pequeño cuadro de texto que nos permitirá añadir el comentario que dejaremos para futuras visitas.

Por último, el modelo será, al igual que lo visto anteriormente, extremadamente sencillo. Dispondremos de una serie de elementos *Visita* que almacenarán tres campos, uno de ellos entero para identificar la nota y los otros dos, de texto, con el nombre del usuario y el comentario dejado por él.

Este será el esqueleto básico de nuestra aplicación, aunque, con el objeto de soportar la autenticación y autorización en ella, nos veremos obligados a añadir algún controlador y vista más que nos permita esta gestión.

En nuestros ejemplos, vamos a utilizar como motor de **base de datos SQLite** debido a las siguientes razones:

- Multiplataforma. Podemos encontrar SQLite ejecutándose en multitud de sistemas, y existen multitud de *bindings* para diferentes lenguajes que nos permiten utilizar este motor de BBDD en un rango de entornos muy amplio.

- Pequeño y sencillo de configurar. Para nuestro ejemplo, utilizaremos una o dos tablas que podremos definir en un pequeño número de sentencias.

- Conocimiento extrapolable. Aunque pequeño, SQLite proporciona una completa base de datos SQL, lo que nos permite que el trabajo invertido en ésta sea extrapolable a otra base de datos más potente.

Al centrarnos únicamente en los mecanismos de seguridad, no entraremos en las complejidades de la definición de modelos y su completa utilización desde los *frameworks*. Por otro lado, debido a que sí emplearemos usuarios, veremos cómo éstos son generalmente almacenados utilizando los mecanismos de acceso a datos dando coherencia a la utilización general. Por este motivo, sí que utilizaremos lo que necesitemos para ello.

Nuestra base de datos será la encargada de mantener el modelo del que antes hablamos. La idea será compartir la misma base de datos desde ambas implementaciones, ya que los requisitos de la aplicación son los mismos.

```
sqlite> CREATE TABLE visitas (
   ...> id INTEGER PRIMARY KEY,
   ...> usuario CHAR(50),
```

```
    ...> nota CHAR(140));
```

A efectos de facilitarnos, más tarde, el hecho de comprobar que nuestra aplicación funciona correctamente, añadiremos un par de notas de ejemplo inicialmente. Esto nos permitirá hacer pruebas sin esperar a tener la capacidad de añadir nuevas notas en su lugar.

```
sqlite> INSERT INTO visitas VALUES (null, "anónimo", "ejemplo de nota 1");
sqlite> INSERT INTO visitas VALUES (null, "anónimo", "ejemplo de nota 2");
```

2.2. Autorización y autenticación en CakePHP

En el caso de CakePHP, encontraremos la funcionalidad de autorización y autenticación en los componentes `Acl`, `Auth` y `Security`. Los componentes en CakePHP son clases que insertan un cierto comportamiento dentro de la lógica de nuestra aplicación. Dicho de otra manera, estos componentes se encargan de funcionalidades horizontales que son comunes a toda o gran parte de nuestra aplicación. Lógicamente, las funcionalidades de seguridad, entre ellas la autenticación y la autorización, se encuentran incluidas.

2.2.1. Autenticación en CakePHP

La autenticación es una de esas tareas vitales en prácticamente toda aplicación y, por supuesto, CakePHP se encarga de facilitarnos su realización.

Para la autenticación de usuarios, utilizaremos el componente *Auth* de CakePHP. Este componente, extremadamente configurable, nos permite realizar diferentes tipos de autenticación.

Si queremos utilizar las facilidades que nos proporciona CakePHP para autenticar usuarios, se nos exige cumplir algunas convenciones.

El primer requisito es crear una tabla que deberá contener los campos "*username*" y "password". En nuestro SQLite, la descripción de la tabla sería:

```
sqlite> CREATE TABLE users (
    ...>   id INTEGER PRIMARY KEY,
    ...>   username CHAR(50),
    ...>   password CHAR(40),
    ...>   groupname CHAR(20));
```

En caso de que ya tuviéramos una tabla de usuarios existente y los campos no coincidieran con los que CakePHP utiliza por defecto, existe la posibilidad de cambiarlos para emplear otros. El *groupname* no lo utilizaremos hasta más adelante, en la autorización de usuarios.

De igual forma, utilizaremos la tabla creada anteriormente "visitas" para el modelo de nuestra aplicación.

Ahora simplemente crearemos un *Controller* que se encargará de gestionar la autenticación de nuestros usuarios. Para seguir los estándares de nomenclatura de CakePHP, llamaremos a nuestro *Controller*`UsersController`, ya que el modelo que maneja es el de *Users*. Crearemos dicho *Controller* en el fichero `controller/users_controller.php` de nuestra aplicación. El código de nuestro `UsersController` es muy sencillo y podremos encontrarlo detallado en la documentación de CakePHP:

```
class UsersController extends AppController {
   var $name = 'Users';
   var $components = array('Auth');
   function login() {
   }
   function logout() {
      $this->redirect($this->Auth->logout());
   }
}
```

Eso es todo lo que necesitamos. La acción *login* vendrá definida con el objeto de poder utilizar la vista *login* que nos proporcionará el formulario de autenticación para los usuarios.

La acción *logout*, por otro lado, simplemente se encargará de llamar al método `AuthComponent::logout()` que a su vez limpiará las credenciales del usuario de la sesión. Si el usuario necesita volver a acceder a alguna función que requiera que esté autenticado, deberá volver a pasar por la acción de *login* previamente.

La parte más sencilla es la de añadir la autenticación a nuestra aplicación.

Nuestro *Controller* se compone de tres acciones, dos de las cuales deseamos que requiera que el usuario esté autenticado y otra que no. Su código en CakePHP será algo como:

```
class VisitasController extends AppController {
   var $name = "Visitas";
   var $uses = array('Visita');
   function view() {
      $this->set('visitas', $this->Visita->find('all'));
   }
   function add() {
      if (!empty($this->data)) {
         $this->data['Visita']['id'] = null;
         if ($this->Visita->save($this->data)) {
```

```
                $this->Session->setFlash('Tu nota ha sido añadida.');
            }
            $this->redirect(array('action' => 'view'));
        }
    }
    function remove($id) {
        if ($this->Visita->remove($id)) {
                $this->Session->setFlash('Nota eliminada.');
        }
        $this->redirect(array('action' => 'view'));
    }
    function beforeFilter(){
        parent::beforeFilter();
    }
}
```

Este *Controller* proporciona toda la funcionalidad que necesitamos para nuestro libro de visitas y, además de él, dispondremos de dos vistas de CakePHP, una para la acción de añadir, que nos mostrará el formulario que necesitamos, y otra para ver nuestras acciones.

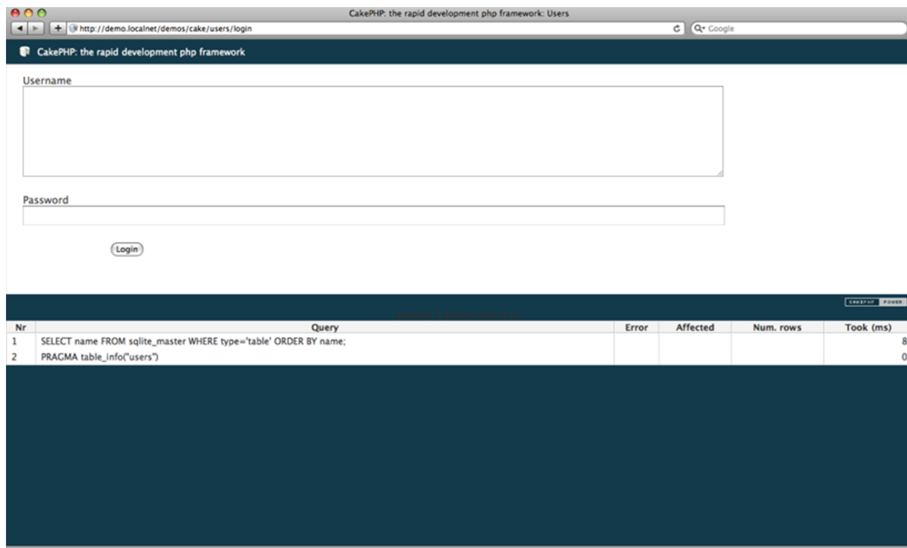
Vista anónima de visitas

Hasta ahora, únicamente hemos añadido la funcionalidad de nuestra aplicación y creado la infraestructura necesaria para autenticar a los usuarios, pero como ya hemos comprobado, aún no la estamos utilizando. Implementar el soporte de autenticación en CakePHP es muy sencillo: simplemente, necesitaremos añadir el componente `AuthComponent` a nuestro controlador.

Aunque no nos detuvimos en ello, es algo que ya hicimos para el `UsersController` y que para realizarlo en nuestro `VisitasController` requerirá de la siguiente línea:

```
var $components = array('Auth');
```

Sólo con esa línea, nuestra aplicación ha añadido autenticación de usuarios a sus funciones.



Pantalla de *login* de visitas

Para hacer esto funcionar, debemos primero crear un usuario de prueba. Para centrarnos en la autenticación y autorización, no vamos a complicar nuestra aplicación de ejemplo con la gestion de usuarios, sino que vamos a insertarlos directamente en la base de datos como hicimos con las primeras visitas. La contraseña, por supuesto, no se almacena en claro en la base de datos, sino que se almacena su *hash* SHA1.

```
INSERT INTO "Users" VALUES (null, "usuario",
   "874d0b1e41c13d9b457a0975a0323e13e711b8c8", "Usuarios");
```

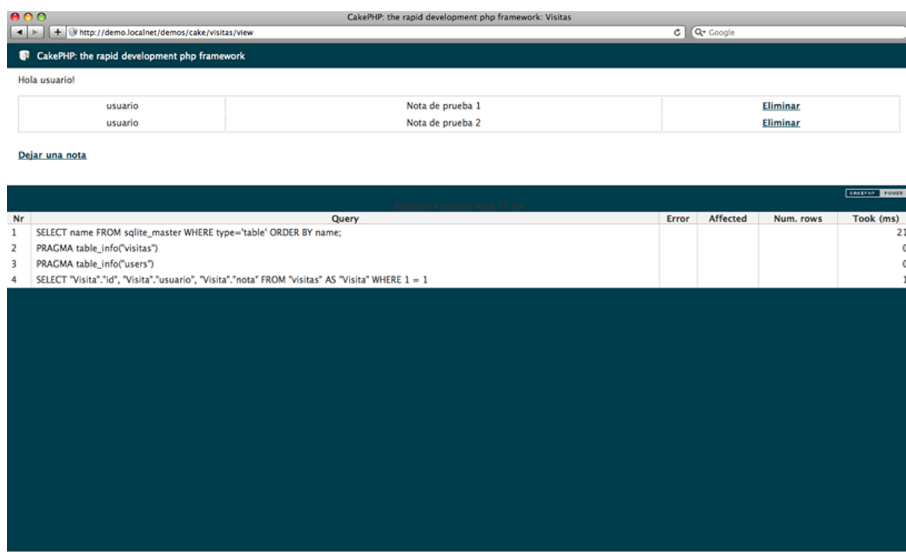
Hemos añadido el usuario al grupo "Usuarios". Debemos también crear un nuevo usuario *admin*, que no utilizaremos hasta más adelante. La sentencia para el usuario "admin" con clave "password" será:

```
INSERT INTO "Users" VALUES (null, "admin",
   "874d0b1e41c13d9b457a0975a0323e13e711b8c8", "Administradores");
```

Como ya hemos dicho, CakePHP utiliza SHA1 para realizar un *hash* del password. Además de esto, para añadir algo más de seguridad, emplea un *salt* que encontraremos en `conf/core.php`. Dicho *salt* se utiliza para que, en el caso de que nuestra base de datos sea comprometida pero nuestro código no, al atacante le sea más complicado obtener las claves por fuerza bruta.

Debido a que CakePHP gestiona todo este proceso automáticamente, esto no tendrá más importancia salvo que deseemos compartir nuestra tabla de usuarios entre varios sistemas. Más adelante, cuando hablemos de la autenticación en *Zend Framework*, veremos la utilidad de ello.

También se nos da la posibilidad de cambiar el algoritmo de *hashing* utilizado mediante la llamada `Security::setHash($hashType)`. Entre los algoritmos soportados, encontramos *SHA1, SHA256* y *MD5*.



Vista de visitas para un usuario

En el estado actual de la aplicación, la vista que nos muestra todas las firmas dejadas por los usuarios nos solicita autenticación. Si bien este es el comportamiento que deseamos para las acciones de añadir y eliminar, queremos que la acción de *consulta* pueda ser realizada sin autenticación. Para permitir esto, simplemente deberemos añadir en el método `UserController::beforeFilter()` de nuestro *Controller* una llamada al método *allow* del componente *Auth*. Este método controla el *Access Control Object* del *Controller* y nos permite alterarlo dinámicamente. Lo utilizaremos de la siguiente forma para permitir directamente el acceso a ver nuestras visitas:

```
$this->Auth->allow('view');
```

El método *allow* nos deja, además de acciones individuales, especificar un *array* de acciones si queremos que varias de ellas no sean autenticadas, o incluso especificar "*" si queremos que ninguna acción de nuestro *Controller* requiera que el usuario esté previamente autenticado.

Con esto ya hemos recorrido las posibilidades básicas de autenticación de CakePHP. Como vemos, toda su utilización es muy intuitiva y sencilla, lo que permite que con pocos cambios tengamos nuestra aplicación funcionando.

2.2.2. Autorización en CakePHP

La autorización en CakePHP se basa en la utilización de dos conceptos básicos:

- ACO (Access Control Object). Son aquellos recursos a los que queremos permitir o denegar el acceso en función del usuario.
- ARO (Access Request Object).Son aquellos objetos a los que deseamos restringir el acceso sobre los ACO.

Podemos entender los ARO como información para un contexto de autorización sobre nuestros usuarios. La utilización de ARO nos evitará tener que almacenar, en la tabla de usuarios, información que no está directamente relacionada con esa entidad, sino que tiene que ver con la autorización. Además, los ARO nos permiten establecer jerarquías en los usuarios de una forma muchos más potente y flexible que lo que supondría implementar la misma funcionalidad en la tabla de usuarios.

Podremos, por lo tanto, crear un ARO asociado a cada uno de nuestros usuarios y que este ARO tenga un determinado padre. Más adelante, podremos asignar los ACO a los padres en lugar de los hijos, lo que proporciona un control mucho más potente. Podremos seguir asignando ACO directamente a los ARO de usuario si quisiéramos para un control más granular.

Al igual que ya hicimos para `AuthComponente`, añadir autorización requerirá que incluyamos el componente de control de listas de acceso `AclComponent` en nuestro *Controller*. Para ello, añadiremos la siguiente línea:

```
var $components = array('Auth', 'Acl');
```

Con esto, ya podremos emplear nuestros dos componentes en nuestro *Controller*. Ya casi estamos listos para comenzar a utilizar la autorización.

CakePHP nos proporciona dos formas de almacenar las listas de acceso: en base de datos o en ficheros de configuración. Nos centraremos en la primera por ser la más flexible.

Para poder utilizar nuestra base de datos para almacenar ACO y ARO, necesitaremos primero crear el *schema*. Aunque podemos hacerlo manualmente, en este caso recurriremos al *shell cake* que lo hará por nosotros. Para ello, ejecutaremos el siguiente comando:

```
cake schema run create DbAcl
```

Al ejecutarlo, se nos avisará en primer lugar de que las definiciones y datos existentes serán eliminadas. Una vez aceptado, un mensaje nos informará de que las nuevas definiciones de las tablas van a ser creadas. Después de esto, se habrán generado tres nuevas tablas:

- **acos**. Contiene el árbol de ACO.
- **aros**. Al igual que el anterior, pero para el árbol de ARO.
- **aros_acos**. Contiene el mapeo entre ARO y ACO.

Para nuestra aplicación, las acciones o ACO a las que deseamos controlar el acceso son básicamente dos:

- Añadir Comentarios. Permitir añadir visitas a nuestro libro. Este permiso será común a todos los usuarios autenticados.
- Eliminar Comentarios. Permitir eliminar visitas. Este permiso sólo lo obtendrán los usuarios administradores.

Hora de comenzar a asignar permisos. Para ello, volveremos a utilizar la *shell* de CakePHP aunque, en esta ocasión, podríamos optar por acceder directamente al modelo de ACO y ARO utilizando el componente `Acl`. Veremos un pequeño ejemplo de cómo hacerlo de las dos formas, para luego centrarlos en la *shell* por ser más práctica.

Nuestro árbol de usuarios será muy sencillo. Dispondremos de dos grupos en la raíz, llamados Usuarios y Administradores. Sólo a aquellos usuarios que se encuentren en el segundo grupo, les permitiremos eliminar comentarios. Para crear estos dos grupos, ejecutaremos los siguientes comandos de la *shell* de CakePHP:

```
cake acl create aro / Usuarios
cake acl create aro / Administradores
```

Esto indica que se creen dos ARO directamente en la raíz "/", lo que nos permite establecer una estructura jerárquica para nuestros ARO. Podríamos crearlos ejecutando un pequeño *script* en una de las acciones de nuestra aplicación utilizando el siguiente código:

```
function crearArosAction()
{
   $groups = array(
      0 => array(
         'alias' => 'Usuarios'
      ),
      1 => array(
         'alias' => 'Administradores'
      )
   );
```

```
    foreach($groups as $group)
    {
        $this->Acl->Aro->create();
        $this->Acl->Aro->save($group);
    }
}
```

Ahora ya podemos incluir alguno de nuestros usuarios existentes en estos grupos. Añadiremos nuestro usuario "usuario" al grupo de Usuarios y el usuario administrador al grupo "Administradores", mostrando un total desprecio por la originalidad. Para ello, utilizaremos los siguientes comandos:

```
cake acl create aro Usuarios User.1
cake acl create aro Administradores User.2
```

Esto hace que el usuario identificado como "1" dentro del modelo "User" pertenezca al grupo "Usuarios". Lo mismo ocurre para el usuario "2", en este caso al grupo "Administradores". Más adelante, podremos utilizar la misma nomenclatura "User." más el identificador del usuario para referirnos a un usuario de nuestro modelo.

Podemos ver la estructura de usuarios que acabamos de crear utilizando el comando:

```
cake acl view aro
Aro tree:
---------------------------------------------------------------
    [1]Usuarios
        [3]
    [2]Administradores
        [4]
---------------------------------------------------------------
```

El número entre corchetes es el identificador del ACO que deberemos utilizar para borrarlos. El nombre que aparece a continuación es el alias y no es obligatorio, siempre y cuando el ACO se refiera a un modelo existente como en el caso de los usuarios. Lo mismo aplicaría a los grupos si, en lugar de manejarlos como *strings*, los hubiéramos elegido tratar como un modelo *Group*. Sin embargo, no es necesario y podemos obtener el mismo efecto sin un modelo que respalde nuestro modelo.

A diferencia de la autenticación, que únicamente utiliza un componente para proporcionar el servicio, las ACL involucran no sólo un componente, sino también un comportamiento.

Pasaremos a continuación a crear los ACO. Estos representarán las acciones que queremos proteger. La organización de los ACO es idéntica a la ya vista para los ARO, por lo que, de forma intuitiva, podremos comenzar a crearlos.

También disponemos de la posibilidad de crear jerarquías, lo que nos permite, en el caso de tener una estructura de recursos compleja, facilitar la posterior creación de listas.

En nuestro caso, la estructura es sencilla y no sería necesario establecer una jerarquía; sin embargo, a efectos de demostrar su funcionamiento, crearemos una muy simple:

- acciones
  - crear
  - eliminar

Los "Usuarios" tendrán derecho a crear, mientras que los "Administradores" tendrán acceso a todas las acciones y no sobre cada una individualmente. Aunque aquí la ventaja de realizar la asignación sea mínima, deberemos tener en cuenta que estas listas de acceso pueden escalar al tener cientos de acciones individuales, lo que haría inmanejable el mantener cada una individualmente.

Ejecutaremos, por lo tanto, los siguientes comandos en la *Cake Shell* para obtener esta estructura:

```
cake acl create aco / acciones
cake acl create aco acciones crear
cake acl create aco acciones eliminar
cake acl view aco
Aco tree:
---------------------------------------------------------------
   [1]acciones
      [2]crear
      [3]eliminar
---------------------------------------------------------------
```

En último lugar, queda establecer la relación entre los recursos que queremos proteger (ACO) y los roles que queremos utilizar (ARO).

```
cake acl grant Usuarios crear all
cake acl grant Administradores acciones all
```

Con esto, habremos otorgado el permiso crear a todos los ACO que se encuentren debajo de "Usuarios", mientras que permitimos todas las acciones a los "Administradores". El último parámetro es la acción que queremos permitir. Suele utilizarse para especificar la acción concreta sobre un recurso. Las accio-

nes nos permiten definir la operación que estamos protegiendo sobre un recurso. Aunque nosotros hemos utilizado directamente el nombre del ACO para ello, podríamos haber usado una estructura bien diferente como, por ejemplo, utilizando el nombre de ACO "visitas" y las acciones crear y eliminar como acciones.

En general, en aplicaciones más complejas, una organización relativamente lógica sería utilizar como nombre del ACO el controlador, y la acción como nombre de recurso a autorizar.

Al igual que hemos utilizado `grant`, podemos usar `deny` con el significado opuesto para denegar una acción explícitamente. Si no la hemos permitido anteriormente, estará denegada; sin embargo, resulta útil para cuando queremos controlar con detalle el acceso a un recurso. Por ejemplo, permitir a todos los Administradores todas las acciones, pero revocar a uno en concreto la capacidad de añadir nuevas visitas.

Ha llegado el momento de verificar si nuestro usuario puede acceder a los datos. El formato de una comprobación de una ACL sería el siguiente:

```
$this->Acl->check($aro, $aco, $action);
```

En nuestro caso, el parámetro `$aco` se corresponderá con la acción que estamos ejecutando, sea crear o eliminar. Como *$aro* utilizaremos el identificador de usuario y la acción será "*". La tabla de comprobaciones quedaría como sigue:

ARO	ACO	Action	¿Permitido?
User.1	crear	*	Sí (heredado de Usuarios)
User.1	eliminar	*	No
User.1	acciones	*	No
User.2	crear	*	Sí (heredado de Administradores)
User.2	eliminar	*	Sí (heredado de Administradores)
User.2	acciones	*	Sí (heredado de Administradores)
Usuarios	crear	*	Sí
Usuarios	eliminar	*	No
Usuarios	acciones	*	No
Administradores	crear	*	Sí (heredado de acciones)
Administradores	eliminar	*	Sí (heredado de acciones)
Administradores	acciones	*	Sí

Combinaciones para el ejemplo de autorización

Como podemos ver, el hecho de permitir una acción específica propaga su valor a todos los hijos. Esto nos da flexibilidad y facilita administrar los permisos cuando tenemos muchos ARO y ACO.

Añadiremos, por consiguiente, las condiciones necesarias en nuestras acciones, utilizando el siguiente código:

```
if (!$this->Acl->check($this->Auth->user('groupname'),"crear","*")) {
   $this->Session->setFlash('No tienes permisos suficientes para realizas esta acción.');
   $this->redirect(array('action' => 'view'));
}
```

Este es todo el código que necesitaremos para autorizar nuestras acciones. El método `Acl::check()` devolverá *true* si el grupo del usuario tiene permisos para realizar una acción. Pero esta aproximación, utilizando el alias del ACO, tiene la limitación de que no podríamos eliminar permisos a un usuario concreto, ya que, únicamente, estamos comprobando si su grupo tiene permisos o no.

A la hora de solucionar tal cosa, a pesar de que los usuarios no dispongan de un alias, pueden ser referenciados para que la función *check* los utilice. Para ello, en lugar de utilizar el alias, utilizaremos un *array* indicando a qué modelo pertenece el ACO y cuál es su clave en ese modelo. Con ese pequeño cambio, nuestro código quedaría:

```
$aco = array(
   'model' => 'User',
   'foreign_key' => $this->Auth->user('id')
);
if (!$this->Acl->check($aco),"eliminar","*")) {
   $this->Session->setFlash('No tienes permisos suficientes para realizas esta acción.');
   $this->redirect(array('action' => 'view'));
}
```

En este caso, obtendremos todas las ventajas del manejo jerárquico de ACO y, a la vez, la posibilidad de eliminar acciones para un usuario concreto.

2.3. Autorización y autenticación en Zend Framework

Zend Framework, en adelante ZF, es otro de los conocidos *frameworks* para PHP. En muchos aspectos es similar a CakePHP, lo que nos permitirá asociar muchos de los conceptos ya vistos anteriormente a los nuevos utilizados en Zend.

Como CakePHP, ZF permite desarrollar aplicaciones web siguiendo un patrón MVC, aunque utilizando Zend no estamos obligados a ello, como sucedía con CakePHP. Esto, para el caso que nos ocupa, implica que podremos utilizar las clases de autorización y autenticación de Zend sin necesidad de que nuestra aplicación disponga de un controlador, un modelo y una vista.

Volveremos a basarnos en nuestra pequeña aplicación de visitas para seguir el mismo proceso que realizamos previamente con CakePHP. Nuestro objetivo es implementar la misma funcionalidad en este *framework*.

En esta ocasión, ZF no se encargará automáticamente de generar por nosotros el modelo, así que tendremos que crear unas cuantas clases que nos permitan acceder a las visitas almacenadas en la base de datos.

Necesitaremos tres clases diferentes para gestionar el modelo. La primera, a la que llamaremos `Default_Model_Visitas`, es el modelo propiamente dicho una vez se encuentra en memoria, ya sea antes de ser serializado o una vez recuperado de la base de datos.

La segunda clase representa nuestra tabla visitas de *SQLite*. Esta clase, llamada `Default_Model_DbTable_Visitas` simplemente contendrá un atributo *$name* que indicará el nombre de la tabla.

Por último, el *Mapper* realiza la unión entre los dos mundos. El *Mapper*, que en nuestro caso se llamará `Default_Model_VisitasMapper`, se encarga de serializar y recuperar de la tabla física nuestros objetos modelo. No dedicaremos más tiempo a esto, ya que nuestro modelo es muy sencillo; para obtener más detalles sobre este tipo de implementación estándar, dirígete a la documentación de ZF.

El controlador de "Visitas" en ZF deberá ser una clase que herede de `Zend_Controller_Action` y que deberá tener un método con el nombre `actionnameAction`. Por ejemplo, en nuestro caso dispondrá de tres métodos diferentes: `addAction`, `viewAction` y `removeAction`. Como hemos visto antes, en esta clase se centrará principalmente todo el control de acceso y la autorización de nuestra aplicación.

2.3.1. Autenticación en Zend Framework

Vamos a centralizar las funcionalidades básicas de autenticación, es decir, hacer *login* y *logout* en un *Controller* al igual que hicimos en el caso de CakePHP. Este *Controller*, llamado *UsersController*, dispondrá de dos acciones que representan las dos funciones antes mencionadas y que llevarán a cabo el proceso.

Por otro lado, todos nuestros *Controller* deberán implementar la verificación de la identidad del usuario antes de permitirle seguir adelante. Cuando el usuario no haya sido autenticado y la acción lo requiera, lo redirigiremos a la acción de *login* de nuestro `UsersController`.

Ya que la comprobación de si el usuario está autenticado o no es común a todas nuestras acciones, la implementaremos en el método `preDispatch()` de nuestro `VisitaController`. Este método es llamado antes de la ejecución de la acción y nos permite decidir si queremos continuar el flujo de ejecución o redirigir hacia otro `Controller` y `Action`. Esto nos será muy útil, pues si comprobamos que el usuario no está autenticado, procederemos a redirigirle a la acción *login* de nuestro controlador Users. El código quedaría como sigue:

```
public function preDispatch() {
   parent::preDispatch();
    $request = Zend_Controller_Front::getInstance()->getRequest();
   if ($request->getActionName()=='view') {
      return;
   }

   if (!$this->_auth->hasIdentity()) {
      $this->view->message = 'Por favor, autentíquese antes de continuar.';
      $this->_forward('login', 'users');
   }
}
```

No hay que olvidar inicializar la variable `$_auth` en nuestro método `init`:

```
$this->_auth = Zend_Auth::getInstance();
```

En nuestro método `preDispatch()`, lo primero que hacemos es comprobar la acción que está ejecutando el usuario. Si la acción es "view", no verificaremos que esté previamente autenticado, ya que permitimos que cualquiera pueda ver nuestro libro de visitas.

En el caso de cualquiera de las otras dos acciones, obtenemos una instancia de `Zend_Auth` y utilizamos el método `Zend_Auth::hasIdentity()` que nos indica si el usuario ha sido autenticado o no. En el caso de que no lo haya sido, procedemos a redirigirle hacia la página de *login*. La clase `Zend_Auth` es un *singleton* que sirve de entrada para toda la funcionalidad de autenticación de ZF. Además de comprobar si el usuario está autenticado, podremos obtener la identidad mediante el método `Zend_Auth::getIdentity()` o incluso eliminarla, lo que nos será útil en la acción *logout* de nuestro *UsersController*, mediante el método `Zend_Auth::clearIdentity()`.

Una vez añadido el código anterior en nuestro método `preDispatch()`, nuestras acciones *add* y *remove* ya están protegidas y no estarán disponibles para usuarios no autenticados. Pasaremos ahora, por tanto, a crear nuestro `UsersController` que se encargará de la autenticación del usuario.

ZF proporciona un sistema de autenticación muy flexible gracias a los diferentes adaptadores que provee. Cada adaptador permite que el usuario sea autenticado de una forma o utilizando un *backend* diferente.

Algunos ejemplos:

Clase	Descripción
Zend_Auth_Adapter_DbTable	Permite autenticar a los usuarios contra una tabla en una base de datos.
Zend_Auth_Adapter_Digest	Autentica a los usuarios utilizando la autenticación *HTTP digest*.
Zend_Auth_Adapter_Infocard	Permite utilizar InfoCard como credencial en la autenticación.
Zend_Auth_Adapter_Ldap	Nos permite autenticar al usuario utilizando un directorio *LDAP* para ello.
Zend_Auth_Adapter_OpenID	La autenticación se realizará utilizando credenciales *OpenID*.

Adaptadores de autenticación en Zend Framework

Por supuesto, podremos implementar nuestro mecanismo heredando de la interfaz `Zend_Auth_Adapter_Interface`, implementando su método `autenticate()` y devolviendo un `Zend_Auth_Result` acorde con el resultado de la autenticación.

Nosotros nos centraremos en el `Zend_Auth_Adapter_DbTable` con el que podremos acceder a nuestra tabla *user* de *SQLite*. Necesitaremos, por lo tanto, inicializar una instancia de este adaptador y configurar los datos necesarios como el nombre de la tabla, y el de los campos que contienen el nombre de usuario y nuestra password. El `Zend_Auth_Adapter_DbTable` recibe todos estos parámetros en su constructor.

```
$dbAdapter = new Zend_Db_Adapter_Pdo_Sqlite(array(
   'dbname' => '/var/www/demos/db/demos.db',
   'sqlite2' => true
));
$authAdapter = new Zend_Auth_Adapter_DbTable(
   $dbAdapter,
   'users',
   'username',
   'password'
);
```

Lo primero que hemos hecho ha sido crear un adaptador de base de datos. Al igual que con los de autenticación, ZF nos proporciona diferentes adaptadores que permiten que nuestros datos estén guardados en diferentes tipos de repositorio; aquí nos centraremos en el `Zend_Db_Adapter_Pdo_Sqlite`. Este adaptador es el que utilizaremos para acceder a nuestra base de datos *SQLite*. Simplemente, lo inicializaremos pasando un *array* con las opciones, entre ellas, el fichero donde se encuentra nuestra base de datos y un segundo parámetro *sqlite2* que indica que tiene un formato *SQLite* v2.

Este adaptador de base de datos es el primer parámetro que recibirá nuestro adaptador de autenticación. El segundo parámetro indica el nombre de la tabla en la que se almacenan las credenciales, en nuestro caso `'users'`. El tercer y cuarto indican, respectivamente, el identificador de la columna que contiene los nombres de usuario y la clave.

Existe un quinto parámetro que nosotros no utilizaremos en este momento, ya que *SQLite* no nos deja sacar partido de él. Este último parámetro permite especificar modificaciones que queremos realizar a la password antes de que sea comprobada. Esto es útil cuando las password no se encuentran almacenadas, sino que se utiliza algún algoritmo de *hashing*. Este parámetro nos especifica la transformación, utilizando el símbolo ? como sustituto de la password.

Por ejemplo, para utilizar un *hash MD5* para nuestra password, nuestra creación del adaptador quedaría como:

```
$authAdapter = new Zend_Auth_Adapter_DbTable(
    $dbAdapter,
    'users',
    'username',
    'password',
    'MD5(?)'
);
```

Podemos entender este último parámetro como la última cláusula de la sentencia SQL que se ejecutará al comprobar las credenciales del usuario. Esto permite que realicemos algunas verificaciones adicionales con mucha comodidad, por ejemplo:

```
$authAdapter = new Zend_Auth_Adapter_DbTable(
    $dbAdapter,
    'users',
    'username',
    'password',
    'date("now")<expiryDate'
);
```

Que comprobaría que la cuenta del usuario no ha caducado. Para ello, por supuesto, deberíamos añadir el campo `expiryDate` al esquema de nuestra base de datos, pero es algo relativamente sencillo.

> *SQLite* no soporta utilizar funciones como MD5 o SHA1 en sus *queries*, lo que provoca que la capacidad de modificar la *query* en el *Zend_Auth_Adapter_DbTable* no nos sea útil. Deberemos tener en cuenta, además, que estamos utilizando las passwords generadas por CakePHP que incluyen un *salt* que se encuentra almacenado en *conf/core.php*. Para utilizar la misma tabla, concatenaremos nuestra password con ese *salt*.

Una vez configurado el adaptador de autenticación, solamente nos queda establecer qué credenciales queremos comprobar mediante `Zend_Auth_Adapter_DbTable::setIdentity()` y `Zend_Auth_Adapter_DbTable::setCredential()`.

```
$authAdapter
   ->setIdentity($usuario)
   ->setCredential($password);
```

La clase `Zend_Auth` es la encargada de realizar la autenticación propiamente dicha una vez que le proporcionemos el acceso a los datos que requiere. Pasaremos como parámetro el adaptador que necesitemos en función del tipo de autenticación que estemos utilizando, en nuestro caso, la base de datos:

```
   $result=Zend_Auth::getInstance()->authenticate($authAdapter);
```

Una vez que el usuario ha sido autenticado, debemos siempre comprobar si esta autenticación ha sido satisfactoria por medio del método `Zend_Auth_Result::isValid()`.

Nuestra acción *login* quedaría por lo tanto como sigue:

```
public function loginAction()
{
      $request = $this->getRequest();
      $form = new Default_Form_Login();
      if ($request->isPost()) {
         if ($form->isValid($request->getPost())) {
            $dbAdapter = new Zend_Db_Adapter_Pdo_Sqlite(array(
                  'dbname' => '/var/www/demos/db/demos.db',
                  'sqlite2' => true
            ));
            $authAdapter = new Zend_Auth_Adapter_DbTable(
                  $dbAdapter,
```

```
                    'users',
                    'username',
                    'password'
              );
              $authAdapter
                 ->setIdentity($form->getValue('usuario'))
                 ->setCredential(sha1('3447a33b473cdde3f399c8dc3cd2ea573ecc4868'.
        $form->getValue('password')));
              $auth= Zend_Auth::getInstance();
              $result=$auth->authenticate($authAdapter);
              if ($result->isValid()) {
                       return $this->_helper->redirector('view','visitas');
              }
           }
        $this->view->message = 'Usuario/Password incorrecto. Inténtelo de nuevo.';
     }
     $this->view->form = $form;
  }
```

Para que nuestro código de autenticación funcione correctamente con los datos ya existentes de CakePHP, deberemos hacer el *hash SHA1* del *salt*, que hemos obtenido de la configuración de nuestra aplicación CakePHP, junto con la password.

Tras comprobar que la password es correcta, simplemente redirigimos al usuario a la página principal.

2.3.2. Autorización en Zend Framework

ZF sigue un modelo similar al que ya vimos en CakePHP y que podemos encontrar en muchos otros *frameworks* dentro y fuera de PHP. Volvemos, por tanto, a centrarnos en proteger el acceso a recursos, sean éstos acciones, controladores, o archivos, utilizando una serie de roles a los que asignaremos derecho de acceso sobre ellos.

En el caso de ZF, encontraremos que los ACO se corresponden con los *Roles*, mientras que los ARO se corresponderán con los *Resources*.

Hasta ahora, además de la autenticación, disponemos de una autorización muy básica, ya que cualquier usuario que no esté autorizado no podrá realizar las acciones de añadir o eliminar visitas. Sin embargo, como ya vimos anteriormente, necesitaremos un control más granular de los usuarios autorizados, pues todos disponen de acceso a los recursos.

En nuestro caso, como ya vimos, el recurso que hemos optado proteger en nuestro controlador es la vista. Autorizaremos, por lo tanto, con diferentes privilegios, a los *Roles* que creemos en ZF sobre las vistas.

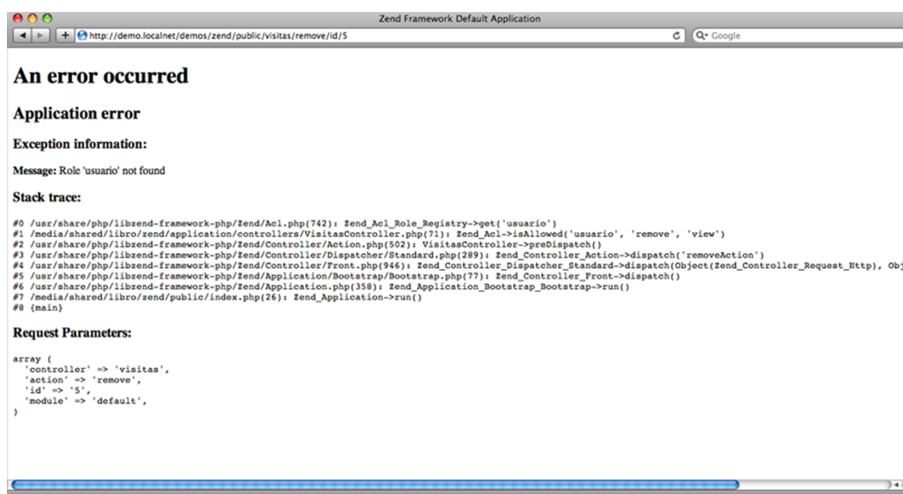
En el caso de que el usuario no esté autorizado para realizar una determinada acción, redirigimos al usuario con un mensaje indicando el error.

El código a añadir sería, por lo tanto, el siguiente:

```
$resource = $this->_actionsNames[$request->getActionName()];
$role = $this->_auth->getIdentity();
if(!$this->_acl->isAllowed($role, $resource, null)) {
   $this->view->message = 'No está autorizado para realizar la acción.';
   $this->_forward('login', 'users', array('continue' => $request->getActionName()));
}
```

El *array* `_actionsNames` únicamente contiene el mapeo entre los nombres de las acciones y su nombre como *Resource* (add > crear, remove > eliminar).

Si intentamos acceder ahora, en nuestra aplicación, a alguna acción que no sea *view*, veremos que se produce un error:



Excepción usuario no autorizado

El error es debido a que *Zend* requiere que los *Roles* y *Resources* hayan sido añadidos previamente para realizar la autorización. Esto no es muy práctico, pero más adelante veremos cómo solucionarlo. Por ahora, vamos a realizar un ejemplo de la forma en que funciona la creación dinámica de *Roles* y *Resources*, lo cual nos permitirá ver nuestro ejemplo funcionando. En este caso, deberemos añadir, justo debajo de la creación del objeto `Zend_Acl` en el método `init` de nuestro `VisitasController`, la inicialización siguiente:

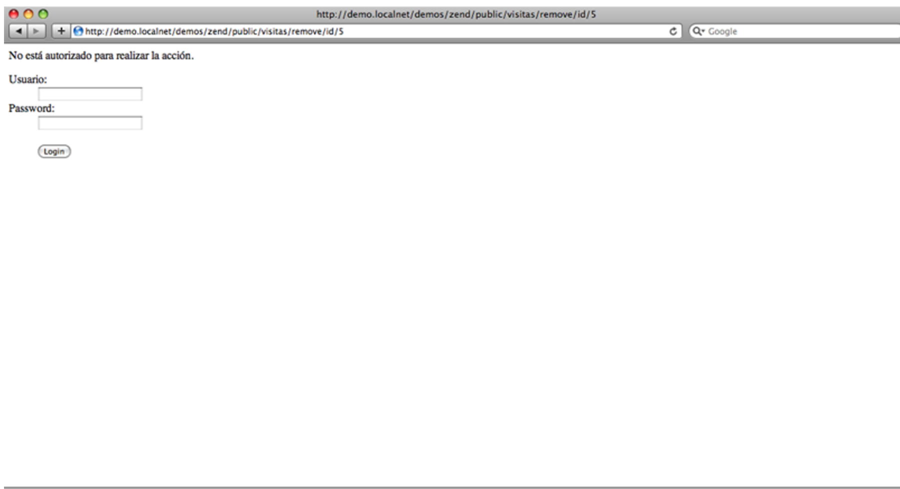
```
$this->_acl->addRole("Usuarios");
$this->_acl->addRole("usuario", "Usuarios");
$this->_acl->addResource("acciones");
```

```
$this->_acl->addResource("eliminar", "acciones");
$this->_acl->addResource("crear", "acciones");
```

Esto creará dos *Roles*, uno llamado "Usuarios", que hace las veces de grupo, y otro "Administrador". Nuestro "Usuario" heredará los permisos que otorguemos al padre "Usuarios". La llamada `addRole` recibe, como primer parámetro, una cadena con el nombre del *Rol* o un objeto del tipo `Zend_Acl_Role_Interface`. Esta interfaz, simplemente, nos permite encapsular la funcionalidad de obtener el *Rol* en otros objetos como, por ejemplo, el modelo.

De forma análoga, tenemos la función `Zend_Acl::addResource()` que nos permite añadir los recursos.

Si ahora, después de autenticarnos como "usuario", intentamos acceder a una de las acciones como añadir o eliminar, obtendremos un mensaje diferente:

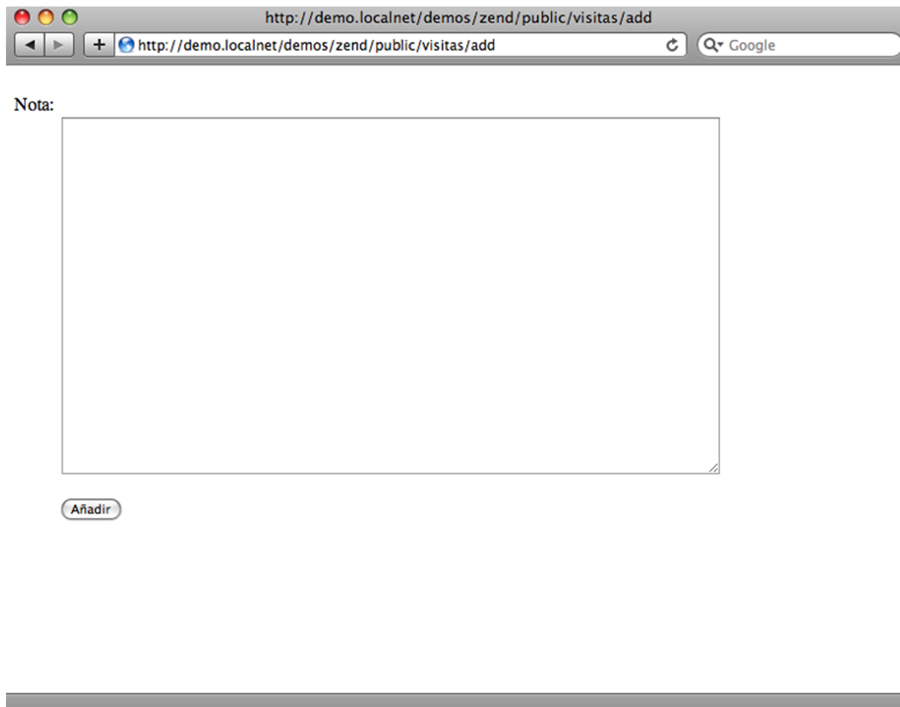


Pantalla de *login* en ZF

Ahora sólo queda autorizar a los usuarios a realizar una de las acciones. En nuestro caso, permitiremos a todos los "Usuarios" que accedan a la acción de añadir. Para ello, utilizaremos el método `Zend_Acl::allow()`

```
$this->_acl->allow("Usuarios", "crear");
```

Si en esta ocasión intentamos eliminar una visita, recibiremos el mismo mensaje que anteriormente, ya que nuestro usuario no está autorizado. Sin embargo, comprobaremos que ya podemos añadir nuevas visitas.



Añadir una visita

Tanto el método `Zend_Acl::allow()` como `Zend_Acl::isAllowed()` pueden recibir, además, un tercer parámetro llamado *priviledge*. Este parámetro, generalmente, representa la acción que estamos autorizando sobre el objeto ("*read*", "*write*", "*delete*", etc). Nosotros, sin embargo, hemos optado por utilizar las acciones como recursos para protegerlas. Esta es una mera decisión de diseño debido a que los recursos nos permiten utilizar herencia, mientras que los privilegios no. Utilizar las acciones como privilegio es un cambio trivial en el código.

Finalmente, el método `Zend_Acl::allow()` puede recibir, como último parámetro, un objeto que implemente la interfaz `Zend_Acl_Assert_Interface`. Las ACL, aunque podamos comprobarlas e incluso añadirlas de forma dinámica, tienen una serie de limitaciones.

> Implementar comprobaciones que, además del acceso al recurso, dependan de datos dinámicos como la fecha o que vengan en la petición como la IP, nos obligaría a añadir otra capa de autenticación diferente. Sin embargo, implementando esta interfaz, podemos crear condiciones que se comprueben en el momento de la verificación de credenciales.

Podemos utilizar estas condiciones en combinación con la autorización de los *Roles* y *Resources* o, incluso, de modo independiente, simplemente pasando *null* en los tres primeros parámetros.

Aunque sencillo, este método de mantener las listas es poco práctico, más aún cuando anteriormente hemos estado utilizando las listas de acceso desde la base de datos. El hecho de inicializar las vistas, como hemos hecho hasta ahora, requeriría añadir o eliminar código en función de los diferentes *Roles* que estuviéramos proporcionando.

No existe un mecanismo por defecto para serializar las ACL del ZF. Debido a esto, no se nos proporcionan las facilidades de las que disponíamos en CakePHP para almacenar nuestras ACL en la base de datos. Tal cosa no significa que no pueda hacerse; de hecho, la clase `Zend_Acl` es serializable, lo que permite su almacenaje.

Nosotros vamos a optar por utilizar una clase `Zend_Acl` ligeramente modificada, a la que llamaremos *DbAcl*, que añadirá soporte de los ACO y ARO que hemos definido para nuestra aplicación en CakePHP.

En primer lugar, debemos fijarnos en la estructura de las tablas que el comando `cake schema run create DbAcl` genera. Las tablas de ACO y ARO comparten un modelo común:

Campo	Descripción
id	Identificador del ACO/ARO.
parent_id	Identificador del padre del ACO/ARO o *NULL* si se encuentra en la raíz.
model	Si el ACO/ARO posee alguna representación en la base de datos, además del propio ACO/ARO, este campo indicará el modelo.
foreign_key	Si el ACO/ARO posee alguna representación en la base de datos, además del propio ACO/ARO, este campo indicará la clave del objeto que representa en el modelo.
alias	Nombre del ACO/ARO. Si el ACO/ARO está relacionado con un modelo, no es necesario.
lft	Anterior ACO/ARO en la lista.
rght	Siguiente ACO/ARO en la lista.

Esquema de las tablas aco y aro

La tabla *aros_acos* se encarga de relacionar los dos modelos, y es la que almacena las listas de acceso. Su esquema es el siguiente:

Campo	Descripción
id	Identificador de la ACL.
aro_id	Identificador del ARO o *Rol* que dispone del permiso.
aco_id	Identificador del ACO o *Resource* al que se otorga el permiso.

Esquema de la tabla aros_acos

Llamaremos a nuestra clase `DbAcl`, y accederá a la estructura representada en las tablas *acos, aros* y *aros_acos*, que recoge el nombre y jerarquía de los *Roles* y Recursos que creamos utilizando la *shell* de CakePHP. Esta clase, en su inicialización, recorre las tabla y, utilizando los métodos `Zend_Acl::addRole()` y `Zend_Acl::add()` que hemos visto anteriormente, recrea en memoria la lista de acceso que se encontraba almacenada. No sólo añadiremos los *Roles* y

Resources que corresponda, sino que también recrearemos la estructura jerárquica. Además, cuando el ACO esté asociado a un *User*, utilizaremos directamente el nombre del *user* como identificador del *Rol*.

La ventaja de utilizar esta clase es, no sólo que no necesitaremos incluir en el código la inicialización de nuestras listas de acceso, sino que además, podemos utilizar la *shell* de CakePHP para manipularlas como hicimos antes.

```
class DbAcl extends Zend_Acl {
   private $_db;
   public $roleIds = array();
   public $resourceIds = array();
   public function __construct()
   {
      $params = array(
         'dbname' => APPLICATION_PATH."/../../db/demos.db",
         'sqlite2' => true
         );
      $this->_db = Zend_Db::factory('PDO_SQLITE', $params);
      self::roleResource();
   }
   private function initRoles()
   {
      $roles = $this->_db->fetchAll(
         $this->_db->select()
            ->from('aros')
            ->order(array('id ASC')));
      foreach ($roles as $role) {
            if (!empty($role['foreign_key'])) {
               $user = $this->_db->fetchRow(
                  $this->_db->select()
                     ->from('Users')
                     ->where('Users.id='.$role['foreign_key']));
               $this->_roleIds[$role['id']] = $user['username'];
               $roleId = $user['username'];
            } else {
               $this->_roleIds[$role['id']] = $role['alias'];
               $roleId = $role['alias'];
            }
            $this->addRole(
               new Zend_Acl_Role($roleId),
               $this->_roleIds[$role['parent_id']]);
      }
   }
   private function initResources()
   {
      self::initRoles();
```

```
        $resources = $this->_db->fetchAll(
        $this->_db->select()
              ->from('acos')
              ->order(array('id ASC')));
        foreach ($resources as $resource) {
              $this->_resourceIds[$resource['id']] = $resource['alias'];
              /* Añadir alias */
              $this->add(
                 new Zend_Acl_Resource($resource['alias']),
                 $this->_resourceIds[$resource['parent_id']]?$this->
        _resourceIds[$resource['parent_id']]:null);
        }
     }
     private function roleResource()
     {
        self::initResources();
        $acls = $this->_db->fetchAll(
        $this->_db->select()
              ->from('aros_acos'));
        foreach ($acls as $acl) {
              if ($this->_resourceIds[$acl['aco_id']] && $this->_roleIds[$acl['aro_id']]) {
                 $this->allow($this->_roleIds[$acl['aro_id']], $this->_resourceIds[$acl['aco_id']]);
              }
        }
     }
  }
```

Cuando la clase `DbAcl` es inicializada, el método `DbAcl::roleResources()` se encargará de cargar los *Roles* y *Resources* de las tablas y crear la estructura en memoria. Para utilizarla deberemos únicamente reemplazar el uso que hemos hecho de `Zend_Acl` por nuestra nueva clase `DbAcl`. El funcionamiento de dicha clase será completamente transparente al resto de código.

Ahora, ya podremos comprobar el funcionamiento de nuestra aplicación usando, exactamente, los mismos usuarios y listas de autorización que utilizamos anteriormente con CakePHP.

3. Autorización y autenticación en aplicaciones .NET

A la hora de desarrollar una aplicación, una de las partes más críticas, y por tanto, una de las más importantes, es la relacionada con la autenticación y autorización de usuarios. Dicho de otro modo, el desarrollo del mecanismo que van a utilizar las aplicaciones a la hora de gestionar los usuarios y los recursos a los que cada usuario o grupo de usuarios va a tener acceso.

Mediante el proceso de autenticación, se determina la identidad de un usuario. Tras su identificación, mediante el proceso de autorización, se determina a qué recursos puede acceder dicho usuario. Con dicho proceso, es posible personalizar las aplicaciones basándose en los tipos y/o preferencias de los usuarios.

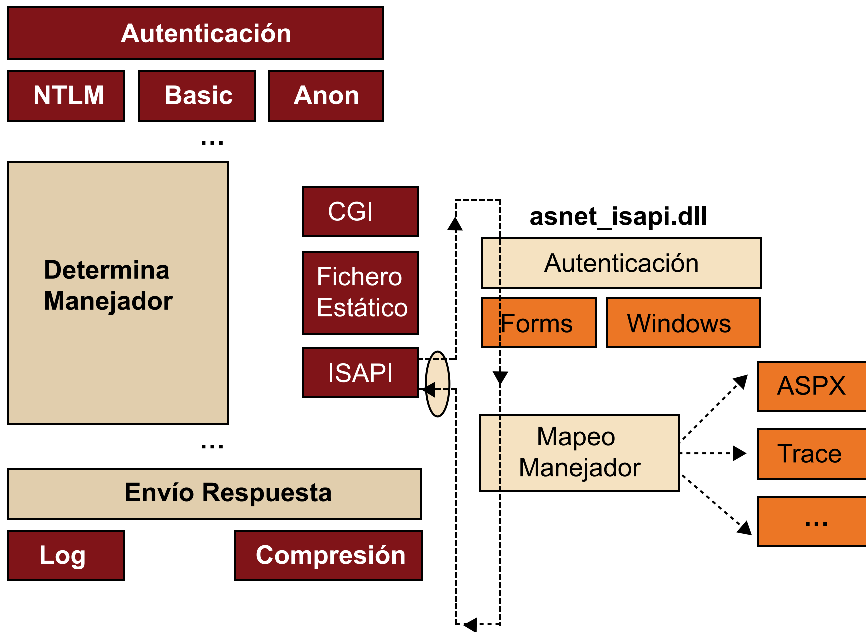
A lo largo de este capítulo, se presentan los mecanismos que se incorporan en la plataforma .NET Framework, desde la versión 2.0, que permiten a un desarrollador crear aplicaciones integrando en ellas mecanismos de autenticación y autorización de usuarios.

3.1. Autenticación en ASP.NET

La plataforma ASP.NET proporciona diferentes tipos de autenticación de usuarios: la autenticación básica, la autenticación de texto implícita, la autenticación de formularios, la autenticación Passport y la autenticación de Windows Integrada. De forma complementaria, es posible proporcionar un mecanismo propio para la autenticación de Usuarios.

Debido a que ASP.NET se ejecuta bajo el servidor de aplicaciones Internet Information Server (IIS), es necesario comprender cómo se integra ASP.NET en el mismo, para poder entender la configuración necesaria a aplicar para una correcta configuración del mecanismo de autenticación.

En las versiones anteriores a IIS7, el código ASP.NET se ejecuta a través de un filtro ISAPI; por lo tanto, en una aplicación ASP.NET se tiene un doble proceso de autenticación: por un lado, el que realiza el propio servidor IIS, y por el otro, el que hace la aplicación ASP.NET, como se muestra en la siguiente imagen.

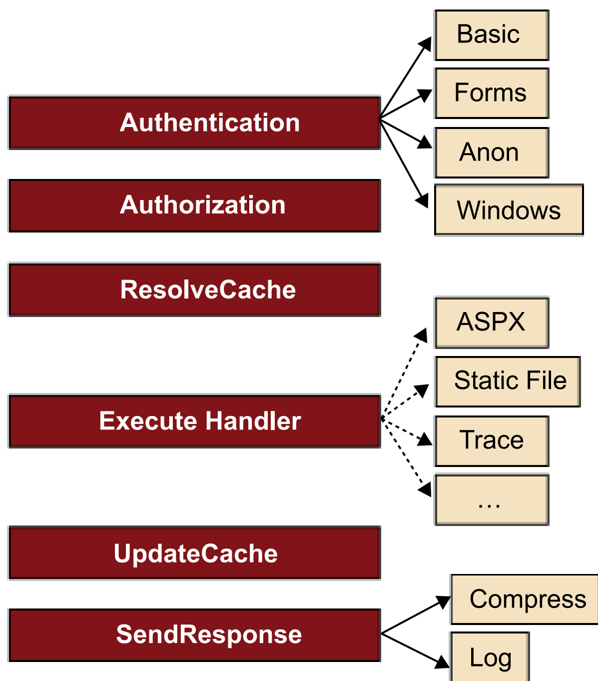


Autenticación en IIS6 y ASP.NET

En la imagen anterior se aprecia cómo, al realizar una petición a una aplicación ASP.NET, en primer lugar se realiza el proceso de autenticación del servidor de aplicaciones y, una vez superada ésta, se pasa la petición al filtro ISAPI, que ejecuta el código ASP.NET, el cual implementa su propio mecanismo de autenticación.

Si se desea configurar una autenticación mediante formularios en ASP.NET, será necesario configurar la autenticación anónima en IIS, para ceder todo el mecanismo de autenticación a ASP.NET. Si, por el contrario, se desea una autenticación de Windows en la aplicación ASP.NET, será necesario configurar el IIS con la autenticación deseada, ya que será éste quien se encargue del mecanismo de autenticación.

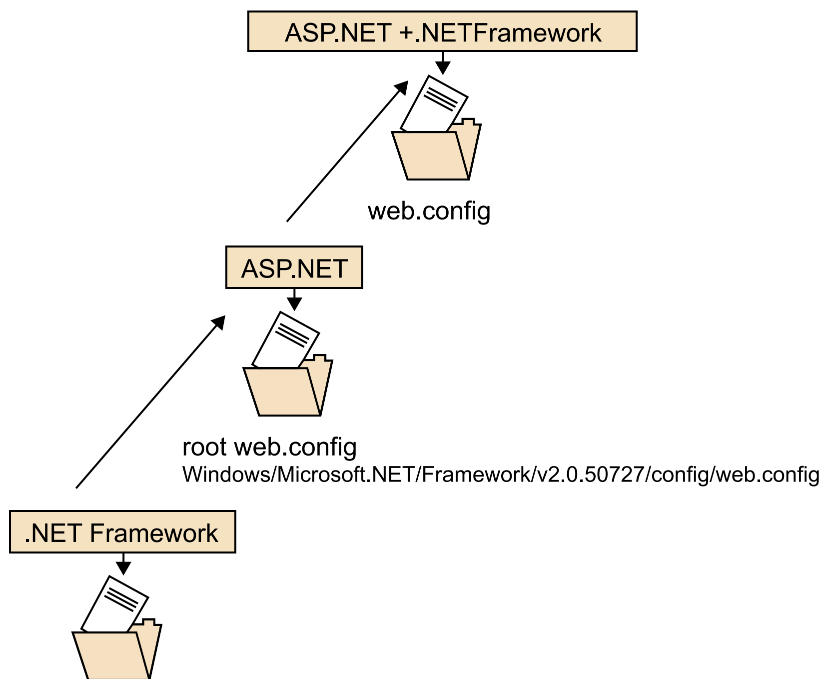
Por el contrario, con el servidor de aplicaciones IIS7, trabajando con la *pipeline* en modo administrado (en modo clásico trabaja de forma semejante a IIS6), la autenticación de ASP.NET se integra al nivel del servidor de aplicaciones, no existiendo en ningún caso, un doble proceso de autenticación, como se puede ver en la siguiente imagen:



Autenticación en IIS7 y ASP.NET

En la figura, se aprecia la forma en que, durante la ejecución de una petición al servidor web, el proceso de autenticación se realiza una única vez según el proceso de autenticación elegido.

Para configurar el método de autenticación en una aplicación ASP.NET, se utilizan los ficheros de configuración *.config*. Estos ficheros son documentos XML que se organizan de forma jerárquica, de modo que los ficheros en el servidor especifican las opciones generales de configuración, las cuales se sobrescriben en los ficheros de configuración a nivel de sitio web, directorio virtual o carpeta.



Herencia en los ficheros de configuración

La configuración de cada sitio web se especifica en el fichero *web.config* del sitio, sin modificar, salvo casos muy puntuales, los ficheros en la máquina *machine.config* y *web.config*. En este fichero de configuración, mediante el elemento *authentication* se configura el esquema de autenticación a utilizar por la aplicación ASP.NET.

```
<configuration>
  <system.web>
    <authentication mode="[Windows|Forms|Passpor|None]">
    </authentication>
  </system.web>
</configuration>
```

Por medio del atributo *mode* del elemento *autentication* se especifican los posibles valores a utilizar: Windows, Forms, Passport y None. El significado de estos valores se indica en la siguiente tabla.

Valor	Descripción
Windows	Especifica la autenticación de Windows como modo de autenticación predeterminado. Se debe usar con cualquier forma de autenticación de servicios de Microsoft Internet Information Server (IIS): básica, implícita, integrada de Windows (NTLM o Kerberos) o certificados. En este caso, su aplicación delega la responsabilidad de la autenticación al servidor IIS subyacente.
Forms	Especifica la autenticación ASP.NET basada en formularios como modo de autenticación predeterminado.
Passport	Especifica la autenticación de red de Microsoft Passport como modo de autenticación predeterminado.

Métodos de autenticación en ASP.NET

Valor	Descripción
None	No especifica ninguna autenticación. La aplicación espera sólo usuarios anónimos o proporciona su propia autenticación.

Métodos de autenticación en ASP.NET

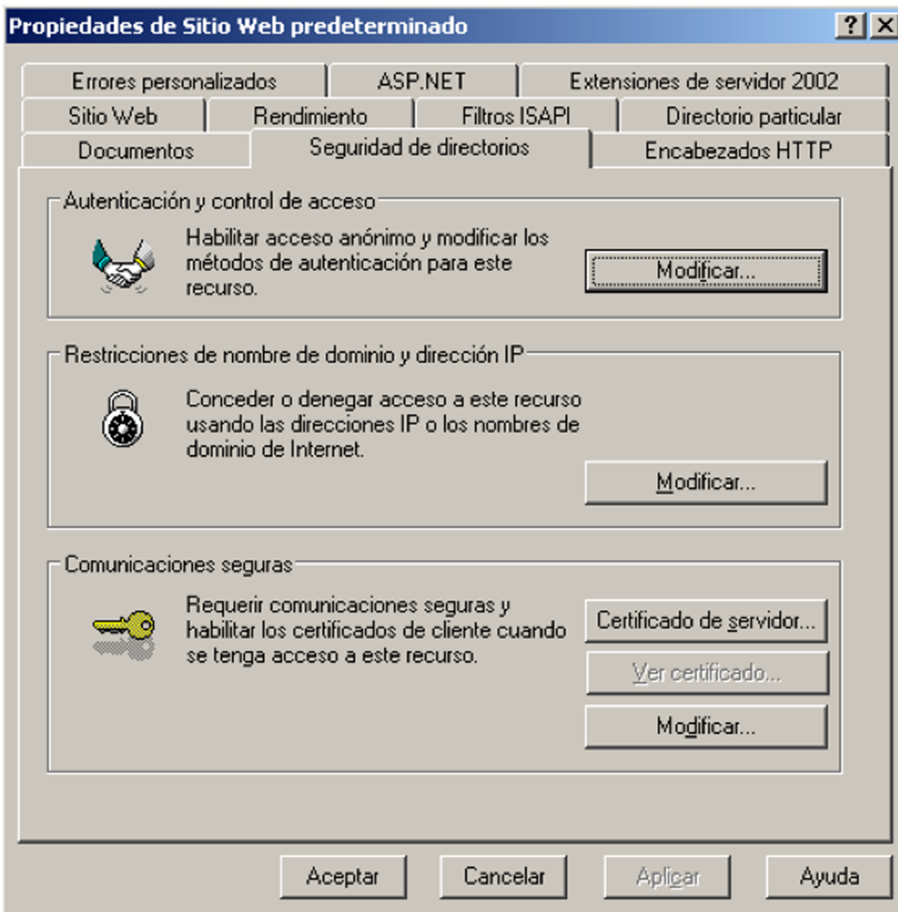
El valor por defecto que se indica en los ficheros de configuración en maquina es *Windows*. En los siguientes apartados, se especifica cómo trabajar con los diferentes mecanismos de autenticación soportados en ASP.NET.

3.1.1. Autenticación basada en Windows

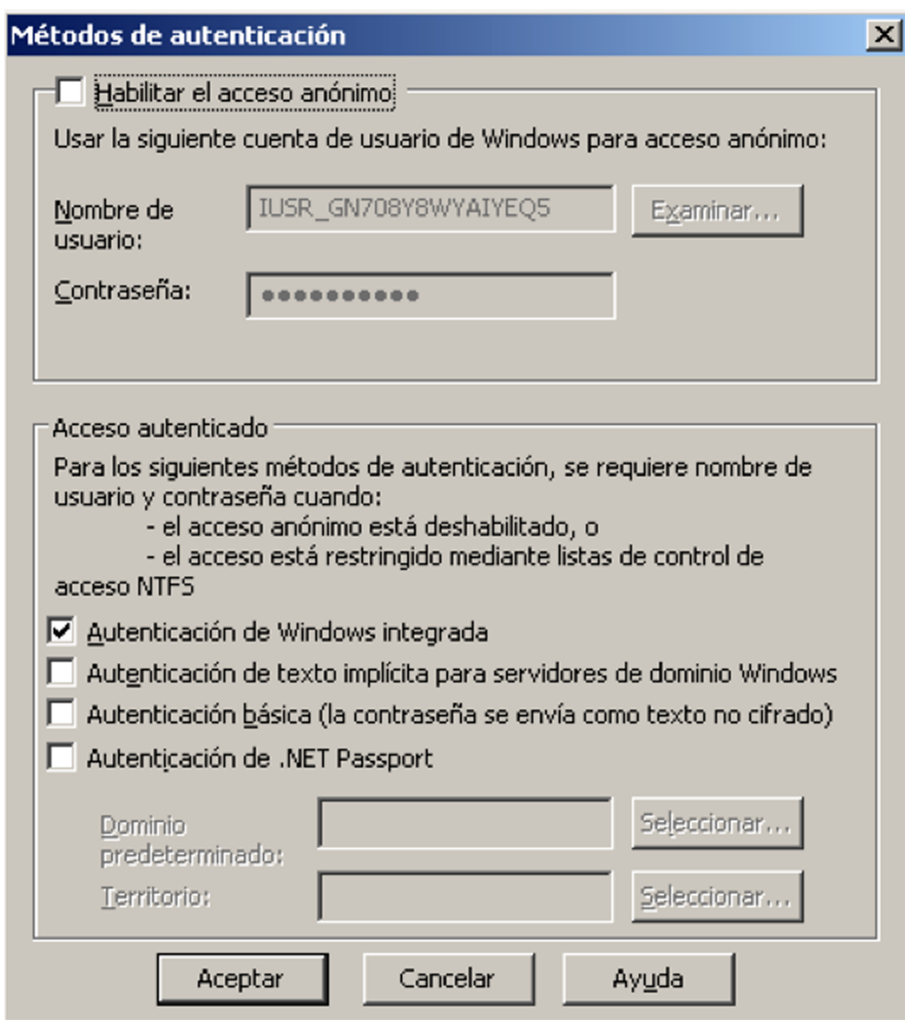
El esquema de autenticación basado en Windows es el utilizado por defecto en las aplicaciones ASP.NET. Utilizando este esquema, la aplicación ASP.NET delega el proceso de autenticación al servidor Internet Information Server subyacente.

Para trabajar así es necesario especificar en el nodo *authentication* del fichero *web.config* el modo Windows, y a continuación especificar en la consola de administración del servidor Internet Information Server la manera de autenticación a seguir.

Para configurar el método de autenticación en IIS 6, hay que ir al *Administrador de Internet Information Service* en las *Herramientas Administrativas* del equipo, desplegar el nodo del equipo y, en la carpeta *Sitios Web*, hacer clic con el botón derecho del ratón sobre el sitio web a administrar, seleccionando el menú Propiedades. En la ventana, hay que seleccionar la pestaña *Seguridad de Directorios* y pulsar sobre el botón *Modificar* dentro de la sección *Autenticación y control e acceso*, como se observa en las siguientes imágenes.



Propiedades de un sitio web en IIS6



Métodos de autenticación en IIS6

En IIS7, la consola de *Administración de Internet Information Server* es diferente, pero de igual modo, por medio del icono de *Autenticación*, situado en el panel de administración de un sitio web, es posible configurar el esquema de autenticación a seguir, como se puede ver en la siguiente imagen.

Group by: No Grouping

Name	Status	Response Type
Anonymous Authentication	Enabled	
ASP.NET Impersonation	Disabled	
Basic Authentication	Disabled	HTTP 401 Challenge
Digest Authentication	Disabled	HTTP 401 Challenge
Forms Authentication	Disabled	HTTP 302 Login/Redirect
Windows Authentication	Disabled	HTTP 401 Challenge

Autentificación en IIS7

En los dos casos, es posible elegir entre cuatro tipos de autenticación (dejando a un lado los modos de autenticación de formularios y .NET Passport, que se verán más adelante), que son: acceso anónimo, autenticación de Windows integrada, autenticación de texto y autenticación básica. En la siguiente tabla, se especifican estos tipos de autenticación:

Modo	Descripción
Anónima	En este modo, no se realiza ninguna tarea de autenticación, sino que IIS proporciona el usuario que tenga configurado, siendo las credenciales de éste las utilizadas para acceder a los diferentes recursos. En IIS6, si se quiere utilizar autenticación de formularios en ASP.NET, es necesario configurar con autenticación anónima el servidor de aplicaciones, para delegar en ASP.NET el proceso de autenticación.
Básica	En este modo, IIS implementa la autenticación básica especificada en HTTP 1.0 utilizando las cuentas de usuarios de Windows. Las credenciales se transmiten codificadas en base64, con lo que es muy fácil su decodificación haciendo de él un método inseguro. A la hora de securizar este método de autenticación, es necesario utilizarlo junto con SSL/TLS (Secure Socket Layer/Transport Layer Security) para cifrar la conexión HTTP (HTTPS). Además, resulta imprescindible que todos los usuarios tengan una cuenta Windows dada de alta en el servidor.
Implícita	Corrige las principales deficiencias de la autenticación básica, ya que se crea un proceso de desafío-respuesta. Como contrapartida, requiere de un directorio activo donde estén dados de alta los usuarios y sólo funciona con los navegadores Internet Explorer.
Windows integrada	Es el mejor esquema de autenticación a utilizar en una intranet donde todos los usuarios trabajan con Internet Explorer y cuentan con un usuario del dominio. Al igual que la implícita, no se envían las credenciales por la red. A diferencia de la autenticación implícita, con este esquema se permite la delegación de credenciales.

Modos de autenticación en IIS

Como se puede observar, estos métodos de autenticación son muy válidos para el desarrollo de intranets corporativas, donde se controlan los navegadores clientes y todos los usuarios están dados de alta en un directorio activo. En combinación con otras características, como la personificación, que se verán más adelante, es posible crear un entorno SSO donde, una vez que un usuario inicia sesión en su equipo a través del dominio, es posible acceder a la intranet y sus recursos sin volver a solicitarle credenciales.

3.1.2. Autenticación basada en formularios

En la autenticación basada en formularios, es ASP.NET quien se encarga de gestionar los usuarios y los recursos de la aplicación a los que éstos tienen acceso. Al utilizar este mecanismo, la aplicación estará provista de un formulario de *login* donde se introducirán las credenciales de acceso para poder acceder a la aplicación web y recursos disponibles.

Es necesario destacar que, si no se utiliza SSL, los datos de usuario y clave se enviarán sin cifrar por la red, suponiendo esto un inconveniente de seguridad.

Como se ha indicado anteriormente, para configurar una aplicación ASP.NET con seguridad basada en formularios es necesario seleccionar dicho tipo de autenticación en la consola de administración de IIS7, y, en caso de trabajar con IIS6, seleccionando el acceso anónimo como método de autenticación.

Una vez configurado adecuadamente el servidor IIS, el fichero *web.config* de la aplicación web ASP.NET debe especificar el modo de autenticación *Forms*, como se puede observar en el siguiente fragmento de código.

```
<configuration>
   <system.web>
      <authentication mode="Forms">
         <forms loginUrl="Login.aspx" name="demoWeb" />
      </authentication>
   </system.web>
</configuration>
```

Como se aprecia en el fragmento de código anterior, el atributo *mode* del elemento *authentication* toma el valor *Forms;* aparece un nuevo elemento *forms* donde se configuran diferentes propiedades del esquema de autenticación, haciendo uso de varios atributos. A continuación, se especifican los posibles atributos a utilizar, sus posibles valores y significado.

<table>
<tr><th>Atributo</th><th>Descripción</th></tr>
<tr><td>Cookieless</td><td>Define si se utilizan cookies y su comportamiento. El valor predeterminado es UseDeviceProfile.

Este atributo puede tener uno de los siguientes valores:
<table>
<tr><th>Atributo</th><th>Descripción</th></tr>
<tr><td>UseCookies</td><td>Siempre se utilizarán cookies, independientemente del dispositivo.</td></tr>
<tr><td>UseUriEspecifica</td><td>Nunca se utilizarán cookies.</td></tr>
<tr><td>AutoDetect</td><td>Se utilizan cookies si el perfil del dispositivo las admite; en caso contrario, no se utilizan cookies.</td></tr>
<tr><td>UseDeviceProfile</td><td>Se utilizan cookies si el explorador las admite; en caso contrario, no se utilizan cookies. En el caso de los dispositivos que admiten cookies, no se intenta determinar si está habilitado el uso de cookies.</td></tr>
</table></td></tr>
<tr><td>defaultUrl</td><td>Define la dirección URL predeterminada que se utiliza para el redireccionamiento después de la autenticación. El valor predeterminado es "default.aspx".</td></tr>
<tr><td>Domain</td><td>Especifica un dominio opcional que se va a establecer en las cookies de autenticación de los formularios salientes. El valor predeterminado es una cadena vacía ("").</td></tr>
<tr><td>enableCrossAppRedirects</td><td>Indica si los usuarios autenticados se redirigen a direcciones URL en otras aplicaciones web. El valor predeterminado es “false”.</td></tr>
<tr><td>loginUrl</td><td>Especifica la dirección URL a la que debe redirigirse la solicitud de inicio de sesión si no se encuentra ninguna cookie de autenticación válida. El valor predeterminado es “login.aspx”.</td></tr>
<tr><td>Name</td><td>Especifica la cookie HTTP utilizada para la autenticación. Si se ejecutan varias aplicaciones en un mismo servidor y cada una requiere una cookie única, se debe configurar el nombre de la cookie en cada archivo Web.config de cada aplicación. El valor predeterminado es ".ASPXAUTH".</td></tr>
<tr><td>Path</td><td>Especifica la ruta de acceso de las cookies emitidas por la aplicación. El valor predeterminado es una barra diagonal (/).</td></tr>
<tr><td>Protection</td><td>Si se ha utilizado algún tipo de cifrado, especifica el empleado para las cookies. El valor predeterminado es “All”.

Este atributo puede tener uno de los siguientes valores:
<table>
<tr><th>Atributo</th><th>Descripción</th></tr>
<tr><td>All</td><td>La aplicación utiliza validación de datos y cifrado para ayudar a proteger la cookie.</td></tr>
<tr><td>None</td><td>Tanto el cifrado como la validación están deshabilitados.</td></tr>
<tr><td>Encryption</td><td>Las cookies se cifran con 3DES o DES, pero no se validan los datos en la cookie. Las cookies empleadas de esta manera pueden sufrir ataques en el texto sin formato.</td></tr>
<tr><td>Validation</td><td>Se lleva a cabo la validación de datos, pero la cookie no se cifra.</td></tr>
</table></td></tr>
<tr><td>requireSSL</td><td>Especifica si se requiere una conexión SSL para transmitir la cookie de autenticación. El valor predeterminado es “false”.</td></tr>
<tr><td>slidingExpiration</td><td>Especifica si el plazo de caducidad está habilitado. Este plazo restablece el tiempo del que dispone una cookie de autenticación activa hasta su caducidad en cada solicitud realizada durante una sesión. El valor predeterminado es “trae”.</td></tr>
<tr><td>timeout</td><td>Especifica el intervalo de tiempo, en minutos enteros, transcurrido el cual la cookie caduca. Si el valor del atributo SlidingExpiration es true, el atributo timeout es un valor variable. El valor predeterminado es "30" (30 minutos).</td></tr>
</table>

Atributos de elemento *forms* del fichero web.config

Una vez configurada la aplicación ASP.NET para utilizar la autenticación basada en formularios, es necesario determinar contra qué información se van a validar los usuarios. En ASP.NET es posible validar los usuarios en el propio fichero *web.config* o utilizar un repositorio externo, como un servidor de bases de datos, un árbol LDAP o ficheros XML. Para estos casos, la mejor opción es recurrir al modelo de proveedores de ASP.NET.

Es necesario indicar aquí la forma en que el modelo que sigue ASP.Net para la autenticación de usuarios mediante repositorios externos, como son las bases de datos o directorios, implica la existencia de una única cadena de conexión para conectarse al repositorio donde se encuentran almacenados los usuarios y su información asociada.

Otro posible modelo de autenticación es aquel en el que la cadena de conexión al repositorio se construye de acuerdo al usuario y clave introducidas por el usuario de la aplicación, de modo que la autenticación se basa en el mecanismo de autenticación del repositorio. Por ejemplo, en una base de datos SQL Server, sería necesario dar de alta como usuarios SQL a todos los usuarios del portal.

Este último mecanismo de autenticación es utilizado por las aplicaciones encargadas de gestionar los propios motores de bases de datos y adolecen de vulnerabilidades CSPP (Connection String Parameter Polution), donde inyectando sobre el usuario y password se manipula la cadena de conexión creada por el programador, pudiendo llegar a saltarse el proceso de autenticación.

Autenticación mediante usuarios del fichero web.config

En aplicaciones pequeñas, donde el número de usuarios a gestionar es muy limitado, es posible dar de alta los usuarios en el propio fichero web.config, de modo que no es necesario recurrir a un repositorio externo ni implementar mecanismos propios.

Para utilizar esta característica, hay que hacer uso del subelemento *credentials* del elemento *forms*. Mediante este elemento, se indica el formato de las contraseñas que se van a utilizar, y mediante los subelementos *user*, se añaden usuarios a la aplicación web, como se muestra en el siguiente cuadro:

```
<configuration>
   <system.web>
      <authentication mode="Forms">
         <forms loginUrl="Login.aspx" name="demoWeb">
            <credentials passwordFormat="SHA1">
               <user name="User1" password="F0578F1E7174B1A41C4EA8C6E17F7A8A3B88C92A"/>
               <user name="User2" password="8BE52126A6FDE450A7162A3651D589BB51E9579D"/>
            </credentials>
         </forms>
```

```
      </authentication>
   </system.web>
</configuration>
```

Los posibles valores que toma el atributo *passwordFormat* del elemento *credentials* y que indican el formato de cifrado para almacenar las contraseñas son los siguientes:

Valor	Descripción
Clear	Especifica que las contraseñas no se cifran.
MD5	Especifica que las contraseñas se cifran con el algoritmo *hash* MD5.
SHA1	Especifica que las contraseñas se cifran con el algoritmo hash SHA1. Este es el valor por defecto.

Posibles valores del atributo *passwordFormat* del elemento *credentials*

Una vez definida la forma en que se van a cifrar las contraseñas de los usuarios, se pueden añadir nuevos usuarios al fichero *web.config;* para ello, y como se ha mostrado, se utiliza el elemento *user* indicando el nombre de usuario con el atributo *name* y la clave, correctamente cifrada, con el campo password.

A continuación se muestra un pequeño ejemplo, con una aplicación web con un fichero *Default.aspx* al que no se debe poder acceder, hasta que el usuario inicie sesión mediante el formulario web disponible en la página *Login.aspx*. Se muestra también la configuración del fichero *web.config*.

El fichero *Default.aspx* consiste, simplemente, en una página de bienvenida al portal:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="_Default" %>
<!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">
<head runat="server">
<title>Bienvenido</title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
      Bienvenido al portal Web
   </div>
   </form>
</body>
</html>
```

En el fichero *Login.aspx* se crea un formulario simple donde el usuario deberá introducir sus credenciales de acceso al portal:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="login.aspx.cs" Inherits="login" %>
<!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">
<head runat="server">
<title>Inicio de Sesión</title>
</head>
<body>
   <form id="Form1" runat="server">
   <div>
      Usuario: <asp:TextBox ID="txtUser" runat="server"></asp:TextBox>
      Password: <asp:TextBox ID="txtPass" runat="server"></asp:TextBox>
      <asp:Button ID="btLogin" runat="server" Text="Entrar" onclick="btLogin_Click" />
   </div>
   </form>
</body>
</html>
```

En el fichero *Login.aspx.cs* se introduce el código necesario para la autenticación:

```
protected void btLogin_Click(object sender, EventArgs e)
{
   if(FormsAuthentication.Authenticate(txtUser.Text, txtPass.Text))
      FormsAuthentication.RedirectFromLoginPage(txtUser.Text, true);
   else
      Response.Write("Credenciales inválidas");
}
```

En este código se puede ver cómo se utiliza la clase *FormsAuthentication*, la cual se encarga de administrar los servicios de autenticación de formularios en aplicaciones ASP.NET. El método *Authenticate* se usa para validar el nombre de usuario y contraseña pasados por medio del formulario web contra las credenciales almacenadas en el fichero *web.config*, mientras que el método *RedirectFromLoginPage* redirige al usuario autenticado hacia la dirección URL originalmente solicitada o la dirección URL predeterminada. El segundo argumento del método indica si se debe crear una cookie permanente.

El fichero de configuración *web.config* deberá tener un aspecto como el siguiente, omitiendo el resto de configuraciones contenidas en la sección *configuration/system.web*.

```
<authentication mode="Forms">
   <forms loginUrl="Login.aspx" name="demoWeb">
```

```
        <credentials passwordFormat="SHA1">
           <user name="User1" password="F0578F1E7174B1A41C4EA8C6E17F7A8A3B88C92A"/>
        </credentials>
     </forms>
  </authentication>
  <authorization>
     <deny users="?" />
  </authorization>
```

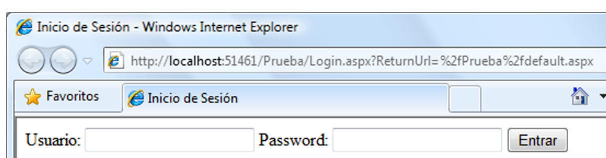
Se ha dado de alta un usuario indicando su nombre "User1" y su password cifrada "pass1". Las passwords se almacenan utilizando las funciones *hash* MD5 o SHA1, que no son reversibles y, por lo tanto, incluso para el programador no es posible recuperar la password original.

Dado que MD5 es una función de *hash* que presenta muchas más colisiones que SHA1, es decir, es más fácil encontrar passwords que generen el mismo *hash* en MD5 que en SHA1, se recomienda utilizar este último como sistema de cifrado para almacenar las contraseñas.

El cambio de sistema de cifrado sólo afecta al fichero *web.config*, no siendo necesario modificar el código de autenticación, se utilice o no el almacenamiento cifrado de claves. El programador puede almacenar claves cifradas (para dar de alta usuarios) haciendo uso del método HashPasswordForStoringInConfigFile, de la clase *FormsAuthentication*, el cual permite generar una contraseña con el algoritmo *hash* apropiado para almacenar en un fichero de configuración pasándosele la contraseña y algoritmo *hash* a utilizar (MD5 o SHA1).

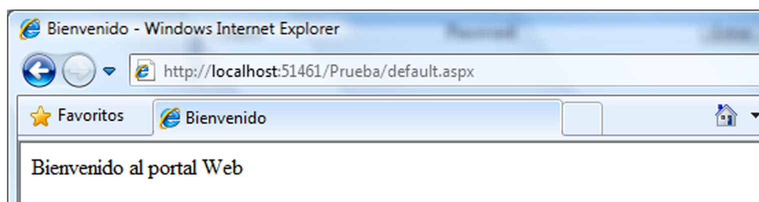
Al final del fichero de configuración se observa el elemento *authorization;* se introduce aquí para especificar que solamente los usuarios autenticados pueden acceder a las páginas del portal. Este elemento se ve en detalle en el apartado de "Autorización en ASP.NET".

En la siguiente imagen se observa cómo, al intentar acceder a la página *Default.aspx*, ASP.NET verifica si el usuario tiene la cookie que lo identifica como usuario autenticado en la aplicación; como no es así, redirige al usuario a la página de *login*, pasándole el parámetro *ReturnUrl*, cuyo valor es la página aspx a la cual va a ser redirigido el usuario en caso de tener éxito en el proceso de autenticación.



Acceso al sitio web con autenticación mediante formularios

Cuando el usuario introduce sus credenciales de acceso, la función *Authenticate* va a codificar la password con la función *hash* SHA1 y compararla con la almacenada en el fichero *web.config* para el usuario pasado. Si el usuario existe, y la password coincide, se autenticará al usuario redirigiéndolo a la página de bienvenida, como se puede observar en la siguiente imagen.



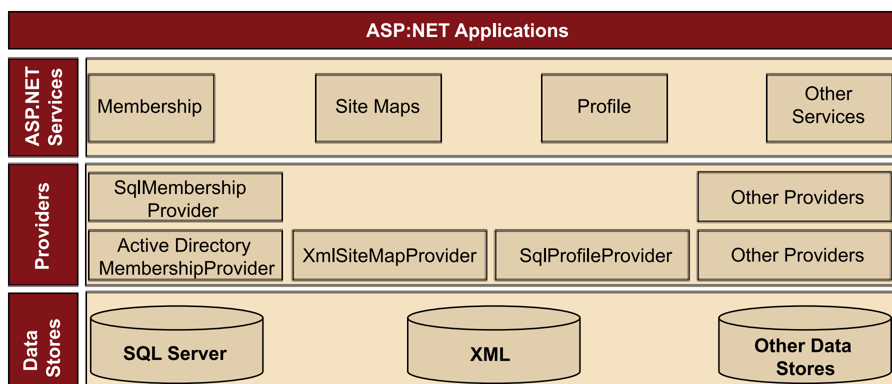
Usuario autenticado mediante formularios

Mediante estas configuraciones es posible crear portales con autenticación mediante formularios, que requieran un conjunto reducido de usuarios. Cuando el portal web se complica y se requiere una gestión más compleja de usuarios, basados en roles, es necesario recurrir a un repositorio externo, haciendo uso del modelo de proveedores de ASP.NET, que se explica en el siguiente apartado.

Modelo de proveedores

Con ASP.NET 2.0 nació el modelo de proveedores; gracias a este modelo ASP.NET, incluye una serie de servicios que almacenan información en bases de datos u otros sistemas de almacenamiento. Así, es posible configurar una aplicación ASP.NET para que el servicio encargado de gestionar usuario y roles utilice un SQL Server o el directorio Activo, mientras que el servicio encargado del mapa y navegación del sitio web utilice ficheros XML.

De este modo, y como se puede ver en la siguiente imagen, es posible separar el código de almacenamiento del código de negocio, utilizando siempre los mismos componentes en las aplicaciones ASP.NET, pero haciendo uso de distintos proveedores, o desarrollando proveedores personalizados según las necesidades.



Modelo de proveedores en ASP.NET

Autenticación contra una base de datos SQL Server

Para la gestión de usuarios en las aplicaciones ASP.NET se utiliza el proveedor *MemberShipProvider*. Por defecto, el proveedor está configurado para hacer uso del gestor de bases de datos SQL Server; dicha configuración se puede ver en el archivo de configuración a nivel de máquina *web.config* que se muestra a continuación:

```
<connectionStrings>
   <add name="LocalSqlServer" connectionString="data source=
      .\SQLEXPRESS;Integrated Security=SSPI;AttachDBFilename=
   |DataDirectory|aspnetdb.mdf;User Instance=
      true" providerName="System.Data.SqlClient"/>
</connectionStrings>
<system.web>
   <membership>
      <providers>
         <add name="AspNetSqlMembershipProvider" type=
"System.Web.Security.SqlMembershipProvider, System.Web, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a" connectionStringName="LocalSqlServer" enablePasswordRetrieval=
"false" enablePasswordReset="true" requiresQuestionAndAnswer=
"true" applicationName="/" requiresUniqueEmail="false" passwordFormat=
"Hashed" maxInvalidPasswordAttempts="5" minRequiredPasswordLength=
"7" minRequiredNonalphanumericCharacters="1" passwordAttemptWindow=
"10" passwordStrengthRegularExpression=""/>
      </providers>
   </membership>
</system.web>
```

En este fichero de configuración, se observa cómo se añade a las aplicaciones ASP.NET el proveedor *AspNetSqlMembershipProvider*, que utiliza la cadena de conexión *LocalSqlServer*, que apunta a un fichero *aspnetdb.mdf* que se crea en el directorio AppData de la aplicación y se adjunta a una instancia de SQL Server Express.

En la configuración del proveedor, se fijan características tales como el formato de almacenamiento de las claves, el número de reintentos máximos antes de bloquear la cuenta de usuario, o la complejidad de la clave requerida para los usuarios.

Si se quiere modificar esta configuración en las diferentes aplicaciones ASP.NET, es necesario modificarla en los ficheros *web.config* de cada aplicación; a continuación, se muestra la configuración necesaria para un sitio web que va a utilizar autenticación basada en formularios haciendo uso de una base de datos de un servidor SQL Server.

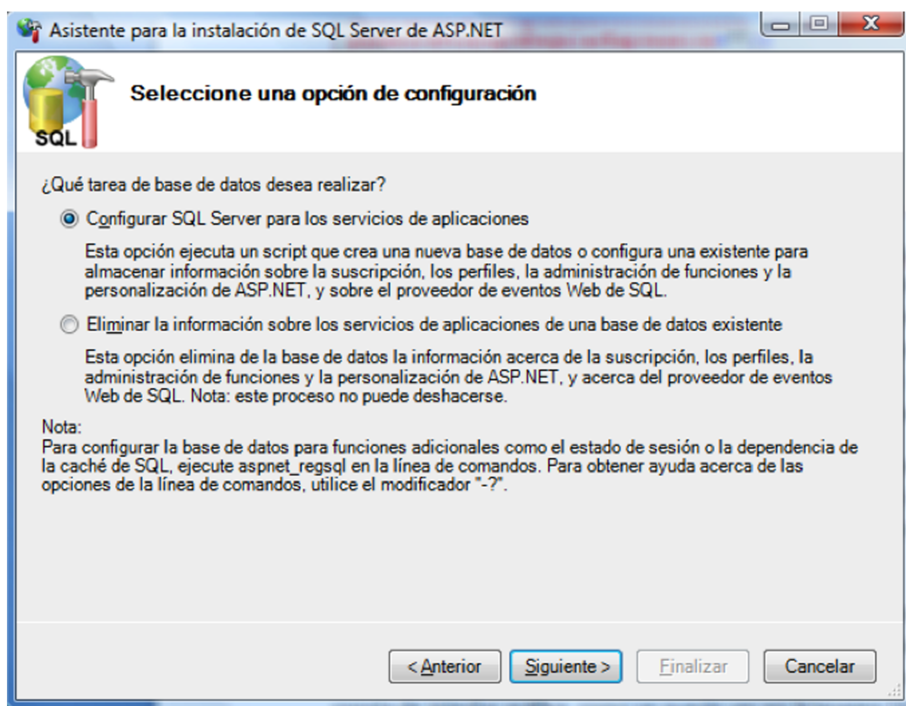
```
<connectionStrings>
```

```
      <add name="miCadenaDeConexion" connectionString="Data Source=SERVIDOR;Initial Catalog=
         WEBDEMO;User ID=sa;Password=123abc." providerName="System.Data.SqlClient"/>
   </connectionStrings>
   <system.web>
      <authentication mode="Forms">
         <forms loginUrl="Login.aspx" name="demoWeb" />
      </authentication>
      <membership>
         <providers>
            <clear/>
            <add name="AspNetSqlMembershipProvider" type="System.Web.Security.SqlMembershipProvider,
   System.Web, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
   connectionStringName="miCadenaDeConexion" enablePasswordRetrieval=
   "false" enablePasswordReset="true" requiresQuestionAndAnswer="false" applicationName=
   "/" requiresUniqueEmail="false" passwordFormat="Hashed" maxInvalidPasswordAttempts=
   "5" minRequiredPasswordLength="7" minRequiredNonalphanumericCharacters=
   "1" passwordAttemptWindow="10" passwordStrengthRegularExpression=""/>
         </providers>
      </membership>
      <authorization>
         <deny users="?" />
      </authorization>
   </system.web>
```

Como se observa, esta aplicación va a utilizar el proveedor *AspNetSqlMembe-rShipProvider;* se han eliminado todos los proveedores heredados para así poder darle el mismo nombre por defecto, y se ha cambiado su configuración, de modo que no se requiere pregunta y respuesta (el atributo *requiresQuestio-nAndAnswer* se ha colocado al valor *false)* y se utiliza una cadena de conexión distinta, de modo que los usuarios se van a almacenar en la base de datos *WEBDEMO* del equipo *SERVIDOR* al cual se va a conectar por medio de un usuario SQL Server.

Lógicamente, el proveedor *AspNetSqlMembershipProvider* necesita que la base de datos SQL Server que utiliza tenga una estructura de tablas, funciones y procedimientos almacenados específica, la cual se puede generar haciendo uso de la herramienta *aspnet_regsql* que se incluye en la instalación del .NET Framework.

Esta herramienta permite configurar una base de datos SQL Server para almacenar información de los diferentes servicios proporcionados por los proveedores de ASP.NET, pudiendo almacenar usuarios, roles, perfiles, etc. La herramienta es muy sencilla de manejar y consta de interfaz gráfica, como se puede ver en la siguiente imagen.



Herramienta *aspnet_regsql* para configurar bases de datos SQL Server

Una vez configurado el proveedor de usuarios, es posible utilizar los controles que proporciona ASP.NET para el inicio de sesión, creación de usuarios y demás tareas relacionadas con la gestión y autenticación de usuario, que junto con la gestión de roles y la autorización, que se ven más adelante, permiten la creación de portales web seguros de una forma rápida y sencilla.

3.1.3. Autenticación contra un árbol LDAP

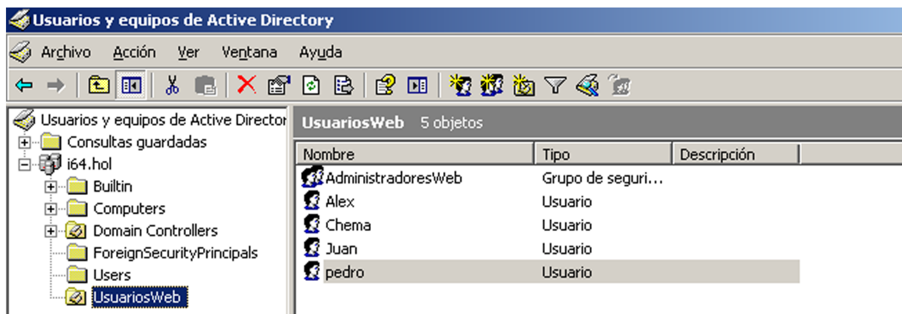
En un entorno empresarial, es muy común disponer de un directorio donde están dados de alta los diferentes usuarios de la corporación. En estos casos, el sitio web de la empresa hará uso de estos usuarios, y no de usuarios de bases de datos.

Para ello, es necesario configurar la aplicación web para que haga uso de un proveedor distinto al *AspNetSqlMemberShipProvider;* en este caso, para conectarse al directorio activo de Microsoft se hará uso del proveedor *AspNetActiveDirectoryMemberShipProvider*, como se puede ver en el siguiente fragmento de un fichero de *web.config* de una aplicación web.

En este caso, se marca el proveedor *AspNetActiveDirectoryMembershipProvider* como el proveedor por defecto; se configura de modo que se conecta al directorio activo mediante la cadena LDAP *ADConnString* y se especifica el usuario y password utilizado para la conexión y el atributo del directorio que se utiliza para mapear usuarios; en caso del directorio activo, el atributo utilizado es *sAMAccountName*.

Tal y como se especifica en la cadena de conexión LDAP, en el directorio activo existe una unidad organizativa llamada *UsuariosWeb*, donde están dados de alta los usuarios que acceden al portal, como se puede ver en la imagen.

```
<connectionStrings>
<add
connectionString="LDAP://server.i64.hol/OU=UsuariosWeb,DC=i64,DC=hol" name="ADConnString"/>
</connectionStrings>
<system.web>
   <membership defaultProvider="AspNetActiveDirectoryMembershipProvider">
      <providers>
            <clear/>
            <add name="AspNetActiveDirectoryMembershipProvider"
type="System.Web.Security.ActiveDirectoryMembershipProvider,System.Web, Version=
2.0.3600.0, Culture=neutral,PublicKeyToken=b03f5f7f11d50a3a"
attributeMapUsername="sAMAccountName"
enableSearchMethods="true"
connectionStringName="ADConnString"
connectionUsername="server\Administrador"
connectionPassword="admin" />
      </providers>
   </membership>
</system.web>
```



Configuración del directorio activo

De este modo, es posible autenticar usuarios contra un directorio sin necesidad de implementar código adicional por parte del programador. Como sucedía en la autenticación contra usuarios de SQL Server, es posible utilizar aquí los controles ASP.NET sin mayor inconveniente.

3.1.4. Autenticación contra otros repositorios de información

Siguiendo el modelo de proveedores propuesto por ASP.NET, es posible utilizar otros repositorios de información para el almacenamiento de usuarios.

> En el portal www.codeproject.com existen proveedores para trabajar con otros motores de bases de datos como MySQL o SQLite, o para trabajar con ficheros XML como repositorio de información.

Pese a todos los proveedores existentes, tanto los proporcionados por Microsoft como los proporcionados por terceros, éstos no son suficientes para cubrir todas las necesidades, bien porque el repositorio de información a utilizar no cuenta con un proveedor ya desarrollado, o bien porque, pese a utilizar un repositorio con soporte, como SQL Server, se cuenta ya con tablas de usuario y roles que no se adaptan a las necesidades del proveedor.

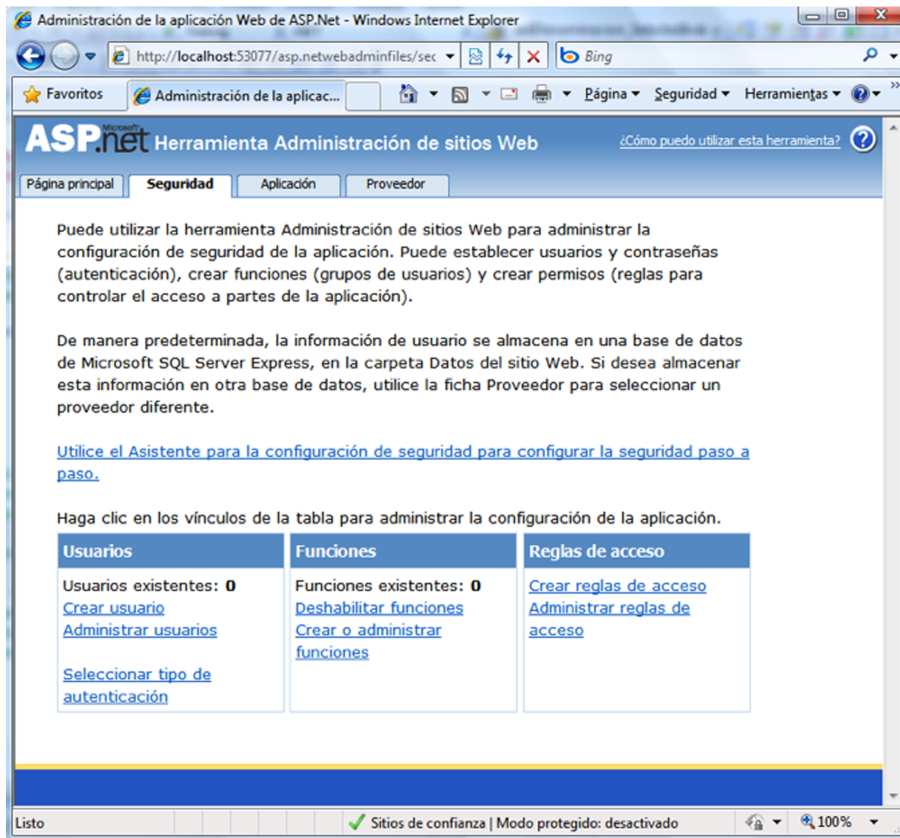
Aquí, es necesario el desarrollo de un proveedor personalizado, que permita el uso de los controles y características proporcionados por ASP.NET, pero haciendo uso de un repositorio a medida. Para ello, utilizando las características de la programación orientada a objetos, se desarrolla una clase que herede de *MembershipProvider* y proporcione implementación a todos los métodos necesarios:

```
public class MiProveedorDeUsuarios : MembershipProvider
{
      public MiProveedorDeUsuarios()
{


      }

   public override string ApplicationName
   {
         get { throw new NotImplementedException(); }
         set { throw new NotImplementedException(); }
   }

   public override bool ChangePassword(string username, string oldPassword, string newPassword)
   {
         throw new NotImplementedException();
   }

   public override bool ChangePasswordQuestionAndAnswer(string username, string password,
      string newPasswordQuestion, string newPasswordAnswer)
   {
         throw new NotImplementedException();
   }
   public override MembershipUser CreateUser(string username, string password,
      string email, string passwordQuestion, string passwordAnswer, bool isApproved,
   object providerUserKey, out MembershipCreateStatus status)
   {
         throw new NotImplementedException();
   }

   public override bool DeleteUser(string username, bool deleteAllRelatedData)
   {
         throw new NotImplementedException();
```

```
    }
...
```

Como se puede ver en este codigo, al crear una clase que herede de *MembershipProvider*, el programador está obligado a dar implementación a una serie de métodos que son los encargados de crear un usuario, borrar un usuario, cambiar la password, etc. De este modo, indicando en el fichero *web.config* que se va a utilizar este proveedor, es posible gestionar y controlar la autenticación de usuarios en ASP.NET siempre del mismo modo, sea cual sea el repositorio donde se van a almacenar los usuarios.

Es posible crear y gestionar los usuarios mediante la Herramienta de Administración de Sitios Web, la cual es accesible desde Visual Studio por medio del icono correspondiente en el Explorador de Soluciones. La herramienta permite la gestión de usuario, roles y reglas de autorización desde la pestaña *Seguridad*, como se observa en la siguiente imagen.



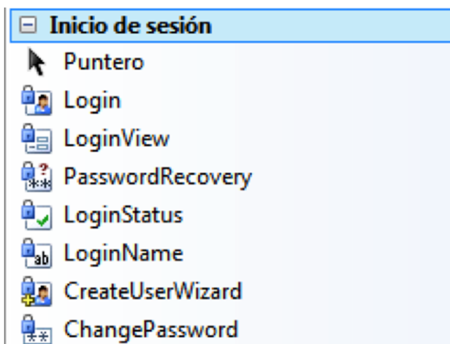
Herramienta Administración de Sitios Web

Mediante esta herramienta es posible seleccionar el tipo de autenticación, el proveedor de usuarios y roles, gestionar dichos usuarios y roles, y administrar las reglas de acceso. Todo lo que se haga gracias a esta herramienta quedará reflejado en el fichero *web.config* de la aplicación tal y como se ha mostrado en este apartado y los apartados que hablan de roles y autorización.

Por otro lado, ASP.Net proporciona una seria de controles para que los programadores los incluyan en sus aplicaciones. Estos controles permiten iniciar sesión, crear usuarios, cambiar passwords y otra serie de acciones, haciendo uso del proveedor de usuarios que se utilice. Estos controles se explicarán de forma resumida en el siguiente apartado.

3.1.5. Controles ASP.NET para la gestión de usuarios

Al utilizar la autenticación mediante formularios haciendo uso del modelo de proveedores, los desarrolladores tienen a su disposición una serie de controles relacionados con el inicio de sesión y gestión de usuarios. Estos controles están situados en la barra de herramientas de Visual Studio, en la pestaña *Inicio de Sesión*, como se puede ver en la siguiente imagen.



Controles de inicio de sesión

En la siguiente tabla, se resumen las funcionalidades de cada uno de estos controles:

Ved también

Para conocer un mayor detalle de la configuración y uso, visitad la página de MSDN en msdn.microsoft.com.

Control	Descripción
Login	Muestra una interfaz de usuario para la autenticación de usuario. El control Login contiene cuadros de texto para el nombre de usuario y la contraseña, y una casilla de verificación que permite a los usuarios indicar si quieren que el servidor almacene su identidad utilizando los proveedores de ASP.NET y que los autentique automáticamente la próxima vez que visiten el sitio. El control Login tiene propiedades para una presentación personalizada, para mensajes personalizados y para vínculos a otras páginas en las que los usuarios pueden cambiar su contraseña o recuperarla si la han olvidado.
LoginView	El control LoginView permite mostrar información diferente a los usuarios anónimos y a los que han iniciado una sesión. El control muestra una de las dos plantillas: AnonymousTemplate o LoggedInTemplate. En estas plantillas, se pueden añadir controles que muestren información apropiada para usuarios anónimos y usuarios autenticados, respectivamente. El control LoginView también incluye eventos para ViewChanging y ViewChanged, que permiten escribir controladores para cuando el usuario inicie una sesión y cambie el estado.

Controles de inicio de sesión

Control	Descripción
LoginStatus	El control LoginStatus muestra un vínculo de inicio de sesión para los usuarios que no están autenticados y un vínculo de cierre de sesión para los que están autenticados. El vínculo de inicio de sesión lleva al usuario a una página de inicio de sesión. El vínculo de cierre de sesión restablece la identidad del usuario actual para que sea un usuario anónimo. Se puede personalizar el aspecto del control LoginStatus estableciendo las propiedades LoginText y LoginImageUrl.
LoginName	El control LoginName muestra el nombre de inicio de sesión de un usuario si el usuario ha iniciado la sesión mediante la autenticación mediante formularios de ASP.NET. De forma alternativa, si el sitio utiliza Autenticación de Windows integrada, el control muestra el nombre de cuenta de Windows del usuario.
PasswordRecovery	El control PasswordRecovery permite recuperar las contraseñas del usuario basándose en la dirección de correo electrónico que se utilizó cuando se creó la cuenta. El control PasswordRecovery envía un mensaje de correo electrónico con la contraseña al usuario. Este control requiere que se haya configurado un servidor SMTP, y su comportamiento varía según las propiedades establecidas en el proveedor de usuarios.
CreateUserWizard	El control CreateUserWizard recoge información de los posibles usuarios. De forma predeterminada, el control CreateUserWizard agrega el nuevo usuario al sistema de proveedores de ASP.NET. Por defecto, la información solicitada para un usuario consiste en nombre de usuario, contraseña, confirmación de contraseña, dirección de correo electrónico, pregunta de seguridad y respuesta de seguridad; estas dos últimas dependen de la configuración establecida en el proveedor de usuarios.
ChangePassword	El control ChangePassword permite a los usuarios cambiar su contraseña. El usuario debe proporcionar primero la contraseña original y, a continuación, crear y confirmar la nueva contraseña. Si la contraseña original es correcta, la contraseña del usuario se cambia a la nueva contraseña. El control también se encarga de enviar un mensaje de correo electrónico sobre la nueva contraseña. El control ChangePassword funciona con usuarios autenticados y no autenticados. Si el usuario no se ha autenticado, el control solicita al usuario que escriba un nombre de inicio de sesión. Si el usuario se ha autenticado, el control rellena el cuadro de texto con el nombre de inicio de sesión del usuario.

Controles de inicio de sesión

Mediante el uso de estos controles, aplicando la configuración necesaria a cada uno de ellos, y extendiéndolos según las necesidades del portal, el programador puede implementar en sus sitios las tareas habituales relacionadas con el inicio de sesión y gestión de usuarios, utilizando por debajo el proveedor de usuarios adecuado.

3.1.6. Autenticación Passport

Otro método implementado en ASP.Net para la autenticación de usuarios es utilizar el sistema de identidad de Passport de Microsoft. Si todos los usuarios disponen de una cuenta Passport es posible desarrollar una solución de una sola firma, lo que significa que con dichas credenciales pueden acceder al sitio web y a todos los sitios y aplicaciones de Internet que tengan habilitado dicho tipo de autenticación.

Salvo las propias aplicaciones y sitios web de Microsoft (Technet, msdn, live, etc.), son muy pocos los sitios en Internet que dispongan de la autenticación Passport, recurriendo la mayoría de ellos a otras soluciones de autenticacion delegada.

3.1.7. Gestión de roles mediante Proveedores

En una aplicación ASP.Net es posible activar el uso de roles, esto es, la posibilidad de agrupar usuarios por sus funciones, de modo que se pueda crear diferentes roles a los que van a pertenecer los usuarios. Ejemplos típicos de roles pueden ser el rol de administrador o el rol de gestor, de modo que los recursos a los que puede acceder un usuario del portal web dependen del rol a que pertenezcan.

Para activar el uso de roles en una aplicación web, es necesario activar esta característica en el fichero *web.config*. La gestión de roles se realiza mediante el elemento *roleManager*, indicando el proveedor de roles que se va a utilizar. El sistema de roles se puede usar, utilizando tanto la autenticación basada en Windows como la autenticación basada en formularios, siendo necesario en cada caso especificar el proveedor a utilizar.

Al igual que sucede con el proveedor de usuarios, en el fichero de configuración a nivel de máquina *machine.config* están dados de alta, por defecto, el proveedor *AspNetSqlRoleProvider* para almacenar roles en una base de datos SQL Server convenientemente preparada, como se indicaba al explicar el proveedor *AspNetMembershipProvider*, y el proveedor *AspNetWindowsTokenRoleProvider*utilizado en la autenticación de Windows, de manera que los roles representan los grupos de usuarios de Windows.

En una aplicación con autenticación basada en Windows, la configuración del fichero *web.config* de la aplicación sería la siguiente:

```
<authentication mode="Windows" />
<roleManager enabled="true" defaultProvider="AspNetWindowsTokenRoleProvider">
   <providers>
      <clear />
      <add name="AspNetWindowsTokenRoleProvider"
         applicationName="/"
         type="System.Web.Security.WindowsTokenRoleProvider, System.Web, Version=2.0.0.0,
      Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"/>
   </providers>
</roleManager>
```

Una aplicación así configurada está delegando la autenticación de usuarios al servidor web subyacente, y por otro lado tiene activado el uso de roles en la aplicación, utilizando el proveedor *AspNetWindowsTokenRoleProvider*, de modo que los roles de los usuarios están representados por los grupos de usuarios de Windows.

Una aplicación que utilice la autenticación mediante formularios también puede trabajar con roles, usando el proveedor adecuado, en caso de utilizar el repositorio SQL Server con el proveedor de autenticación *SqlMembershipProvider*, la aplicación puede utilizar el proveedor *AspNetSqlRoleProvider*, configurándolo adecuadamente en el fichero *web.config*, como se observa en el siguiente cuadro:

```
<connectionStrings>
   <add name="miCadenaDeConexion" connectionString="Data Source=SERVIDOR;Initial
      Catalog=WEBDEMO;User ID=sa;Password=123abc."
   providerName="System.Data.SqlClient"/>
</connectionStrings>
<system.web>
   <authentication mode="Forms">
      <forms loginUrl="Login.aspx" name="demoWeb" />
   </authentication>
   <membership>
      <providers>
         <clear/>
         <add name="AspNetSqlMembershipProvider" type="System.Web.Security.
SqlMembershipProvider, System.Web, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a" connectionStringName="miCadenaDeConexion"
enablePasswordRetrieval="false" enablePasswordReset="true" requiresQuestionAndAnswer=
"false" applicationName="/" requiresUniqueEmail="false" passwordFormat="Hashed"
maxInvalidPasswordAttempts="5" minRequiredPasswordLength="7"
minRequiredNonalphanumericCharacters="1" passwordAttemptWindow="10"
passwordStrengthRegularExpression=""/>
      </providers>
   </membership>
   <roleManager enabled="true" defaultProvider="AspNetSqlRoleProvider">
      <providers>
         <clear />
         add name="AspNetSqlRoleProvider"
            connectionStringName="miCadenaDeConexion"
            applicationName="/"
            type="System.Web.Security.SqlRoleProvider, System.Web, Version=2.0.0.0, Culture=
   neutral, PublicKeyToken=b03f5f7f11d50a3a"/>
      </providers>
   </roleManager>
   <authorization>
      <deny users="?" />
```

```
    </authorization>
</system.web>
```

En esta configuración, se muestra una aplicación con autenticación basada en formularios que utiliza el proveedor *AspNetSqlMembershipProvider* para la gestión de usuarios y el proveedor *AspNetSqlRoleProvider* para la gestión de roles. En ambos casos, la información se almacena en la misma base de datos, que es la apuntada por la cadena de conexión *miCadenaDeConexión*, la cual se ha preparado previamente con la herramienta *aspnet_regsql*.

Los proveedores ASP.NET proporcionados por terceros para otros repositorios, como MySQL, SQLite, o ficheros XML, también incluyen soporte de roles. En caso de haber desarrollado un proveedor de usuarios propio, será necesario desarrollar también el proveedor para el soporte a roles en caso de querer utilizar esta característica.

En este caso, es necesario desarrollar una clase que extienda a *RoleProvider* proporcionando la implementación a los métodos necesarios, tales como la creación de un rol, su eliminación, la búsqueda de usuario por rol, etc. Tal y como se observa en el siguiente cuadro:

```
public class miProveedorDeRoles : RoleProvider
{
    public miProveedorDeRoles()
    {
    }

    public override void AddUsersToRoles(string[] usernames, string[] roleNames)
    {
        throw new NotImplementedException();
    }

    public override string ApplicationName
    {
        get { throw new NotImplementedException(); }
        set { throw new NotImplementedException(); }
    }

    public override void CreateRole(string roleName)
    {
        throw new NotImplementedException();
    }

    public override bool DeleteRole(string roleName, bool throwOnPopulatedRole)
    {
        throw new NotImplementedException();
    }
```

```
    public override string[] FindUsersInRole(string roleName, string usernameToMatch)
    {
       throw new NotImplementedException();
    }
...
```

Al igual que en la gestión de usuarios, por medio de la herramienta de *Administración de Sitios Web* es posible gestionar los roles o funciones y asignarlas a los diferentes usuarios según las necesidades del sitio web.

3.2. Autorización en ASP.NET

Hasta ahora se ha hablado de los diferentes mecanismos que proporción ASP.Net para la autenticación de usuarios, bien a través del servidor web subyacente o a través de la autenticación mediante formularios haciendo uso de usuarios del fichero *web.config* o haciendo uso del modelo de proveedores.

Mediante la autorización, lo que se pretende es controlar el acceso de los usuarios a los diferentes recursos de la aplicación web, es decir, controlar a qué páginas se puede acceder dependiendo del usuario de que se trate o del rol al que pertenezca.

La autorización se configura con el elemento *authorization* del fichero *web.config* de la aplicación web. Dado que los ficheros de configuración siguen una estructura jerárquica, lo más habitual es que, en cada carpeta de la aplicación web, exista un fichero *web.config* donde se especifican las reglas de autorización para esa carpeta. La sintaxis del elemento *authorization* es la siguiente:

```
<authorization>
   <allow ... />
   <deny ... />
</authorization>
```

Con los elementos *allow* y *deny* se añaden las reglas de autorización; así, cuando un usuario trata de acceder a una página, el módulo de autorización recorre en iteración los elementos *allow* y *deny*, empezando por el archivo de configuración más local hasta que encuentra la primera regla de acceso adecuada para la cuenta de usuario específica. Después, el módulo de autorización concede o niega el acceso a un recurso de direcciones URL dependiendo de si la primera regla de acceso encontrada es una regla *allow* o *deny*.

Por defecto, la regla especificada en el fichero de configuración a nivel de maquina *machine.config* es *<allow users="*" />*, con lo que el acceso a todos los recursos URL de la aplicación están permitidos.

Si se quiere denegar el acceso a todos los usuarios que no estén autenticados, se utilizaría la siguiente configuración:

```
<authorization>
   <deny users="?" />
</authorization>
```

Con esta configuración los usuarios anónimos tendrían denegado el acceso, mientras que al resto de usuarios (usuarios autenticados) se les aplicaría la regla por defecto y, por lo tanto, se les concedería acceso al recurso solicitado.

Es posible configurar reglas de autorización tanto por usuarios como por roles; así, por ejemplo, si se quiere permitir el acceso a los usuarios con role *Administrador* y denegar el acceso al resto de usuarios la configuración a aplicar sería la siguiente:

```
<authorization>
   <allow roles="Administrador" />
   <deny users="*" />
</authorization>
```

En los ejemplos vistos, se está configurando el acceso a los recursos URL de toda la carpeta donde se encuentra el fichero *web.config;* sin embargo, también es posible realizar configuraciones personalizadas para un recurso URL concreto. Para ello, es necesario utilizar el elemento *location*, para indicar el recurso URL concreto al que aplicar la regla de autorización, tal y como se ve a continuación:

```
<configuration>
   <location path="Administracion.aspx">
      <system.web>
         <authorization>
            <allow roles="Administrador" />
            <deny users="*" />
         </authorization>
      </system.web>
   </location>
</configuration>
```

En este caso, se aplica una regla de autorización al recurso *Administracion.aspx* de modo que sólo los usuarios con el role *Administrador* pueden acceder a él; el resto de usuarios no tienen autorización para acceder a ese recurso.

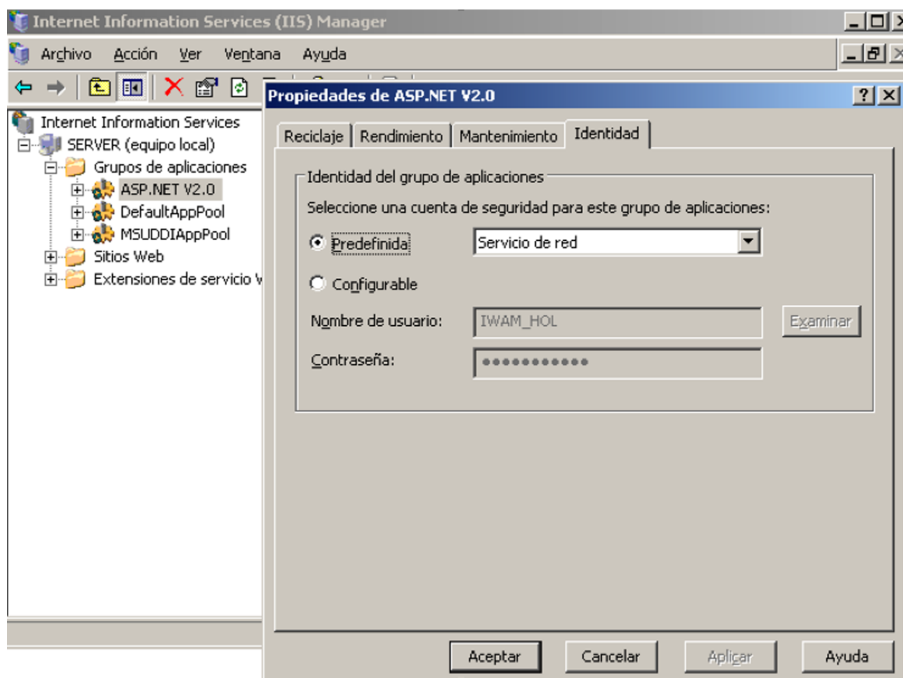
Es importante destacar que en IIS 6 o inferior las reglas de autorización sólo afectan a recursos procesados por el *framework* .NET, de modo que no se pueden utilizar para proteger el acceso a otro tipo de ficheros, como ficheros zip

o doc. En IIS7 es posible utilizar estas reglas de autorización para proteger todo el contenido, gracias a la integración total de la plataforma ASP.NET en el servidor de aplicaciones.

3.3. Identidad del código y personalización

En una aplicación web es muy importante conocer las credenciales con las que se ejecuta el proceso ASP.NET. En un IIS5 por defecto se utiliza la cuenta ASP.NET, mientras que en un IIS6 o IIS7 se utilizan las credenciales bajo las que corre el grupo de aplicaciones al cual pertenece la aplicación web.

Por defecto, el *pool* de aplicaciones utiliza al usuario *Network Service* o *Servicio de red*, estas cuentas deben ser lo menos privilegiadas posibles concediendo acceso únicamente a los recursos que necesite la aplicación web. Con la *Herramienta de Administración de IIS* es posible configurar el grupo de aplicaciones a utilizar por una aplicación web, así como el usuario bajo el que corre la aplicación, como se observa en la siguiente imagen.

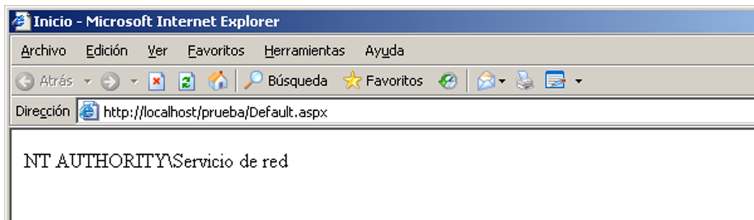


Configuración del grupo de aplicaciones

Utilizando el objeto *WindowsIdentity* de la clase *Security.Principal* es posible, mediante programación, determinar el usuario bajo se está ejecutando la aplicación. Para mostrar por pantalla el nombre de usuario que se está utilizando, basta con utilizar el siguiente fragmento de código:

```
WindowsIdentity User = WindowsIdentity.GetCurrent();
Response.Write(User.Name);
```

En una aplicación web, cuyo grupo de aplicaciones está configurado como se muestra en la imagen anterior, el resultado de la ejecución de este código sería el mostrado en la siguiente imagen, independientemente de que el tipo de autenticación esté basado en formularios o en Windows.



Usuario utilizado por el grupo de aplicaciones

Sin embargo, por medio del elemento *identity* del fichero de configuración *web.config* es posible activar la personificación. Esto permite a ASP.NET ejecutarse como un proceso que utiliza los privilegios de otro usuario.

Si en el fichero de configuración se activa el atributo *impersonate* del elemento *indentity*, el proceso ASP.NET pasa a ejecutarse con los privilegios del usuario que realiza la petición; por lo tanto, si se coloca la siguiente configuración en el fichero *web.config*:

```
<configuration>
   <system.web>
      <identity impersonate="true" />
   </system.web>
</configuration>
```

Independientemente del tipo de autenticación, si el servidor IIS está configurado con acceso anónimo, las credenciales utilizadas por el proceso ASP.NET para ejecutar la aplicación serán las configuradas en el acceso anónimo.

En caso de utilizar la autenticación básica o integrada, las credenciales utilizadas serán las del usuario que realice la petición. También es posible, por medio del elemento *identity*, configurar la aplicación para utilizar una cuenta específica, como se puede ver en el siguiente cuadro:

```
<configuration>
   <system.web>
      <identity impersonate="true"
            userName="Usuario"
            password="Clave" />
   </system.web>
</configuration>
```

3.3.1. Modelo SSO en una intranet

Aprovechando las capacidades de personificación de ASP.NET, en una red corporativa donde existe un controlador de dominio al que se encuentran suscritos los equipos de los diferentes usuarios, es posible desplegar un sistema único de autenticación que permita a un usuario, una vez iniciada su sesión con la cuenta de dominio, acceder a los diferentes recursos de la intranet sin necesidad de volver a autenticarse.

En la siguiente imagen, se muestra un pequeño diagrama de red donde se representa una infraestructura en la cual existe un controlador de dominio y un servidor web; todos los equipos cliente, bien sean ordenadores de sobremesa, portátiles, PDA u otros dispositivos, se encuentran suscritos al dominio, de modo que los usuarios inician sesión con una cuenta del dominio.

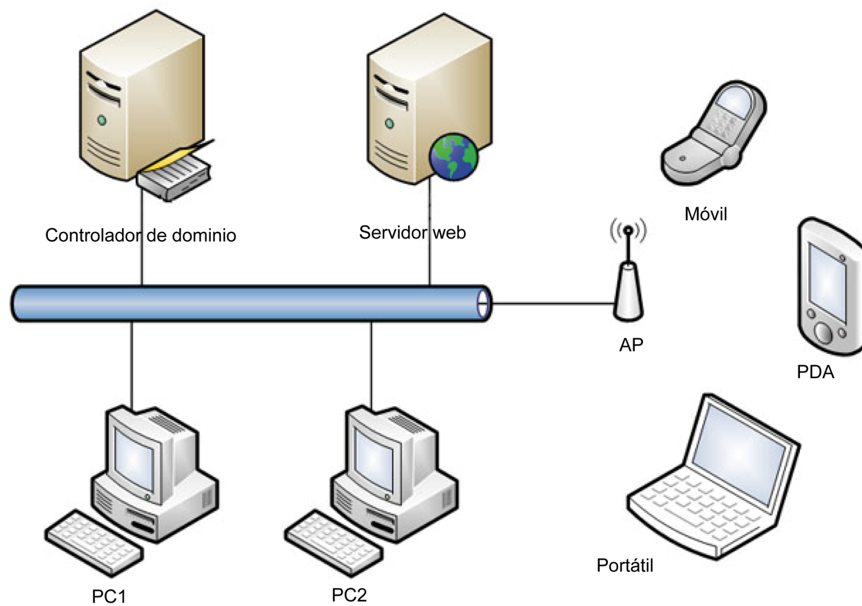
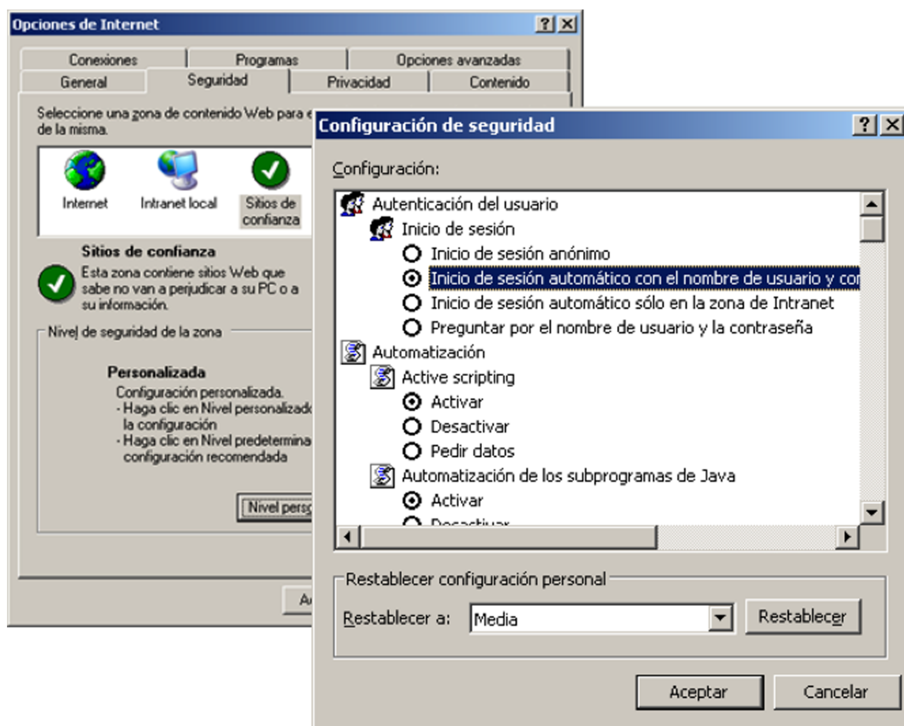


Diagrama de red corporativo

La aplicación ASP.NET de la intranet se configura utilizando el modelo de autenticación Windows y se activa la personalización; en el servidor web, se especifica la autenticación de Windows integrada y como proveedor de roles en la aplicación ASP.NET se indica que se va a utilizar el proveedor *AspNetWindowsTokenRoleProvider*. De este modo, es posible utilizar los grupos de usuarios del dominio en las reglas de autorización de la aplicación ASP.NET. Estas configuraciones se han visto a lo largo del manual.

En los navegadores de los equipos clientes, es necesario configurar el envío de credenciales de modo automático; para ello, si se añade la dirección web de la aplicación de la intranet a los sitios de confianza, se puede cambiar las opciones de configuración para enviar, de forma automática, las credenciales con las que se ha iniciado sesión, tal y como se muestra en la siguiente imagen.



Configuración de Internet Explorer

De esta forma, cuando el usuario accede a la intranet, se van a transmitir sus credenciales, de modo que el proceso ASP.NET se ejecutará con sus privilegios, accediendo únicamente a los sitios donde tenga autorización su usuario según los grupos a los que pertenezca; si la aplicación se conecta a la base de datos SQL Server mediante la autenticación integrada, sus credenciales serán transmitidas al servidor de bases de datos, controlándose también a este nivel los recursos a los que tiene acceso el usuario.

3.4. Autenticación en aplicaciones Windows Forms

Los conceptos de autenticación y autorización están presentes no sólo en las aplicaciones ASP.NET, sino también en el resto de aplicaciones .NET, incluidas las de escritorio. En una organización con directorio activo, es posible utilizar las clases *WindowsIdentity* y *WindowsPrincipal* para determinar la identidad de los usuarios y proteger las aplicaciones de un uso no autorizado, incluyendo aquí ensamblados completos, métodos o fragmentos de código.

Implementando clases que hereden de las clases *GenericIdentity* y *GenericPrincipal*, es posible controlar la autenticación y autorización en entornos no Microsoft, como son aquellos en donde los usuarios y las reglas de acceso residen en bases de datos.

3.4.1. Aplicaciones *stand-alone*

Con el término *stand alone* se define a aquellas aplicaciones que pueden ejecutarse y controlarse como entidades independientes.

Un ejemplo de estas aplicaciones sería un procesador de texto; esta aplicación es completamente autónoma y no necesita estar conectada a ningún servidor para su funcionamiento.

En este tipo de aplicaciones, la autenticación y autorización de acceso a los diferentes recursos y funcionalidades deben implementarse en el código de la propia aplicación. En este apartado, se presentan las clases principales que un programador puede utilizar para identificar al usuario que ejecuta la aplicación, el grupo al que pertenece y, en función de ello, permitir el acceso o la ejecución de determinadas funcionalidades.

La clase *System.Security.Principal.WindowsIdentity* representa la cuenta del usuario Windows, incluyendo datos como el nombre de usuario, el tipo de autenticación y el *token* de autenticación; mediante el código que se presenta a continuación, es posible determinar el nombre del usuario que está ejecutando la aplicación:

```
static void Main(string[] args)
{
   WindowsIdentity user = WindowsIdentity.GetCurrent();
   Console.WriteLine(user.Name);
}
```

El objeto *WindowsIdentity* cuenta con otra serie de propiedades y métodos para recuperar más información acerca del usuario que ha iniciado sesión.

Con la clase *WindowsPrincipal* es posible acceder al grupo de usuarios al cual pertenece el usuario, de modo que es posible realizar comprobaciones para autorizar el acceso a determinados recursos o funcionalidades. El siguiente cuadro muestra el código necesario para comprobar si el usuario pertenece al grupo administrador o un grupo específico de usuarios:

```
static void Main(string[] args)
{
   WindowsIdentity user = WindowsIdentity.GetCurrent();
   WindowsPrincipal current = new WindowsPrincipal(user);

   if (current.IsInRole(WindowsBuiltInRole.Administrator))
      Console.WriteLine("El Usuario pertenece administradores");

   if(current.IsInRole(@"DOMINIO\GRUPO"))
      Console.WriteLine("El Usuario pertenece a GRUPO");
}
```

Mediante la clase *PrincipalPermission* es posible demandar, de forma declarativa o imperativa (mediante programación), que el usuario que ejecuta la aplicación pertenezca a un grupo de usuarios para poder ejecutar un clase, un método o un fragmento de código.

En el siguiente cuadro, se muestra de forma declarativa que para ejecutar la aplicación es necesario un usuario autenticado que pertenezca al grupo de administradores locales de la máquina.

```
static void Main(string[] args)
{
      AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
   try
   {
            imprimirMensaje();
   }
   catch (Exception exc)
   {
         Console.WriteLine(exc.Message);
   }
   Console.ReadLine();
}

[PrincipalPermission(SecurityAction.Demand, Authenticated=true, Role="Administradores")]
static void imprimirMensaje()
{
   Console.WriteLine("Hola Mundo");
}
```

En el método *Main* de la aplicación, se especifica la política de seguridad que se va a seguir en los hilos de esta aplicación, mientras que en el método *imprimirMensaje* se solicita la necesidad de ser un usuario autenticado y pertenecer al grupo Administradores de la máquina local.

Si la aplicación se ejecuta con un usuario que no pertenece al grupo de administradores de la máquina o, aun perteneciendo a este grupo en un sistema Windows Vista o Windows 7, se ejecuta sin elevar privilegios, el resultado de la ejecución sería un error de permisos. Por el contrario, si el usuario pertenece al grupo Administradores, o se realiza la elevación de privilegios en un sistema Windows Vista o Windows 7, el código podrá ejecutarse correctamente.

De este modo es posible, mediante el código de la aplicación, identificar al usuario que está ejecutando la aplicación y proteger el acceso a los recursos de la aplicación según el perfil del mismo.

3.4.2. Aplicaciones cliente-servidor

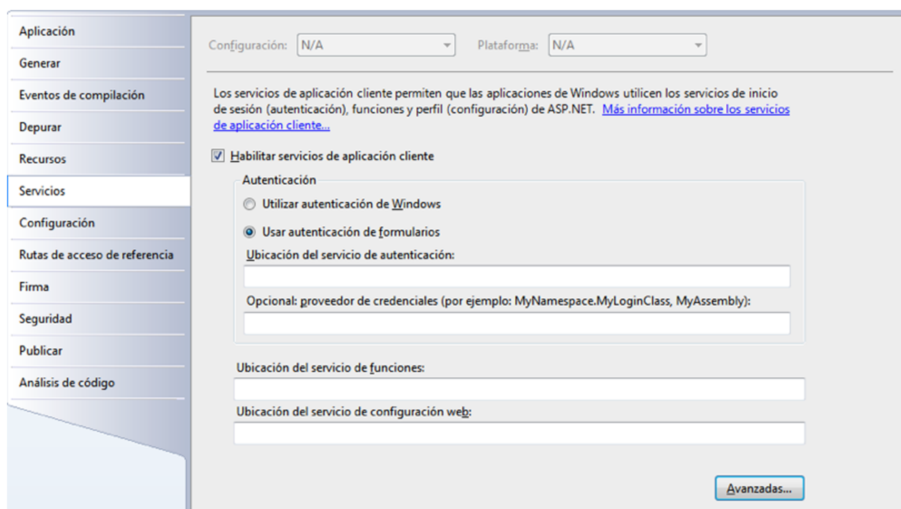
Al contrario que las aplicaciones *stand alone*, existen multitud de aplicaciones que necesitan conexión con un servidor para realizar diferentes tareas, caso de aplicaciones tan conocidas como Windows Live Messenger o Yahoo Messenger, donde el proceso de autenticación del usuario no se realiza en la máquina local, sino en los servidores dedicados a tal efecto.

Para este tipo de aplicaciones es necesario que, desde la aplicación cliente, se recoja el nombre de usuario y clave y se envíe al servidor para su verificación. El envío de credenciales se puede realizar mediante un servicio web, que se encargará de validar al usuario.

Con el *framework* .NET 3.5 Microsoft proporciona una novedad para las aplicaciones Windows Forms y WPF (Windows Presentation Foundation). Se trata de los servicios de aplicación cliente que proporcionan, a este tipo de aplicaciones, acceso simplificado al inicio de sesión, roles y servicios de perfil disponible en ASP.

Con los servicios de aplicación de cliente es posible utilizar un servidor centralizado para autenticar a los usuarios, determinar las funciones asignadas a cada usuario y almacenar información de configuración de aplicaciones por usuario que puede compartir en la red.

Si se cuenta con una aplicación que se ejecuta bajo el *framework* 3.5, accediendo a las propiedades del proyecto en Visual Studio, con la pestaña *Servicios* se puede configurar esta característica, como se puede ver en la siguiente imagen.



Servicios de aplicación cliente

Mediante la configuración adecuada en cliente y en servidor, es posible lograr que las aplicaciones se validen en el servidor haciendo uso de los mecanismos ya vistos, que permitirían autentificar usuarios contra diferentes repositorios de almacenamiento como SQL Server o directorios LDAP, y gestionar roles para las reglas de autorización.

El uso de clientes que utilicen versiones anteriores del *framework* .NET hace necesario que el programador implemente su propio mecanismo para la autenticación de usuario y gestión de roles. Con las últimas versiones del *framework* .NET se logra seguir la misma dinámica de trabajo, se trate de aplicaciones ASP.NET o aplicaciones Clientes.