Autenticació i autorització en aplicacions multiusuari

José María Palazón Romero

PID_00177496

Universitat Oberta de Catalunya

Índex

1. Autenticació i autorització en Java

El Java Authentication and Authorization Services (JAAS) va aparèixer com a paquet opcional per a la versió 1.3 de Java2 SDK. Més endavant, en la versió 1.4, va passar a ser integrat com a part principal de l'SDK. Però, què és exactament JAAS? És l'entorn de treball o *framework* que proporciona Java, per integrar d'una manera uniforme, al llarg de la nostra aplicació, l'autenticació i autorització d'usuaris, i gestionar l'accés als diferents recursos que hi manegem. Per exemple, utilitzant els serveis que ens proporciona aquesta capa d'abstracció, no necessitem alterar el codi de les nostres aplicacions per funcionar amb el mètode d'autenticació o canviar-lo, ni preocupar-nos del mètode d'emmagatzematge de les credencials dels nostres usuaris.

A més, JAAS ens permet utilitzar diferents mètodes d'autenticació de l'usuari, o fins i tot desenvolupar els nostres propis que, més endavant, hi podrem incloure. Per a això, Java implementa l'estàndard Pluggable Authentication Module (PAM), que permet emprar i combinar aquests mètodes en un únic sistema d'autenticació. Si estructurem la nostra aplicació de la manera correcta, no tindrem la necessitat de canviar-ne ni un sol indicador per utilitzar un sistema diferent. Un simple canvi de configuració farà que la nostra aplicació utilitzi un sistema d'autenticació diferent, d'una manera molt semblant a la utilitzada per a configurar PAM a Linux.

JAAS no està limitat al costat del client o del servidor; simplement, proporciona una sèrie d'eines que podrem utilitzar en qualsevol desenvolupament que fem. Amb les seves classes i interfícies, podem desenvolupar aplicacions client que necessitin autenticació o servidors que les utilitzin per a integrar un sistema centralitzat.

Havent-hi inclòs JAAS a partir de la versió 1.4, l'entorn de treball de seguretat de Java està format per tres components principals:

- Java Cryptography Architecture (JCA). Aquestes interfícies de programes d'aplicació (API) proporcionen la funcionalitat de xifratge i desxifratge i altres que hi estan relacionades, com la gestió de certificats.

- Java Secure Socket Extension (JSSE). Pren com a base la funcionalitat oferta per JCA i la utilitza, juntament amb els mecanismes de connexió de xarxa de Java, per a proporcionar connectivitat segura.

- JAAS. Aquesta API proporciona la funcionalitat d'autorització i autenticació. Les seves classes i interfícies es troben sota el paquet `javax.security.auth`.

JAAS, en si mateix, no dóna tota la funcionalitat necessària i es fonamenta en altres API de Java que ja hi ha, entre les quals, algunes de l'entorn de treball de seguretat.

Si bé l'API que proporciona JAAS no es caracteritza per una excessiva complexitat, sí que introdueix una sèrie de conceptes que hem de conèixer abans, comuns a altres sistemes d'autenticació i autorització en qualsevol camp.

Per començar, el concepte de *subject*. En general, podem pensar en el *subject* com en l'usuari que fa una operació i, per tant, requereix una autorització per a fer-la. Encara que aquesta associació entre *subject* i usuari és la més comuna, hem de tenir en compte que no ha de ser sempre així. El *subject* pot ser qualsevol entitat que necessiti autorització per a fer una operació; per exemple, una tercera aplicació.

Un concepte que està molt relacionat amb el *subject* són els *principals*. Un *principal* és una identitat del *subject* i, per tant, en forma part. Intentant traslladar aquest concepte al món real per fer-lo més senzill, entenem cada *principal* com cadascuna de les identificacions que podem utilitzar en diferents situacions segons que es requereixi.

> L'exemple més clar és el carnet d'identitat, el passaport i el permís de conduir. Si bé tots ens identifiquen, cadascun té uns usos concrets i no tots ens permeten identificar-nos en totes les situacions.

En el cas del programari, un *principal* pot ser un número d'identificació d'usuari, el seu inici de sessió o *login*, o el seu correu electrònic.

A més d'això, un *principal* ens pot representar de manera que puguem separar les nostres identitats en rols.

> Ens pot identificar com a administradors del sistema, mentre que un altre, associat al mateix *subject*, ens identificaria com a simples usuaris del sistema.
>
> Un altre exemple que ens ajudarà a aclarir el concepte de *principal* és un usuari en un sistema Unix. Aquest usuari està compost d'un *principal* que és el seu nom d'usuari, i d'un *principal* o més d'un que representen els seus grups. Per tant, el *subject* pot actuar utilitzant un dels seus *principals* per identificar-se o més d'un.

Els *subjects*, a més de *principals*, contenen *credentials*. Cada *credential* representa una forma d'autenticació, com ara un conjunt usuari i contrasenya, o un certificat digital.

1.1. Autenticació amb JAAS

Vegem ara, en més detall, el funcionament del procés d'autenticació usant JAAS.

La classe principal que hem d'utilitzar és *javax.security.auth.login.LoginContext*, la qual ens proporciona el punt d'entrada als diferents sistemes d'autenticació que hem configurat. Més endavant, veurem com s'ha de fer aquesta configuració, però ara comencem escrivint l'esquelet bàsic de la nostra petita aplicació:

```
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("Impossible identificar l'usuari: "+e.getLocalizedMessage());
   }
   }
}
```

Ara ja podem executar el nostre codi. Veiem que, en intentar executar-lo, es genera una *java.lang.SecurityException* que ens indica que no es pot localitzar el fitxer de configuració.

Això és perquè, per al funcionament de JAAS, es requereix un fitxer que indica la configuració bàsica i els mòduls disponibles. Com hem dit abans, l'estàndard PAM que implementa JAAS ens permet especificar, en aquest fitxer, el mòdul o els mòduls que s'executaran per a autenticar l'usuari i la seva configuració.

L'especificació d'aquest fitxer per a la nostra petita aplicació és com segueix:

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

Senzill, oi? Vegem ara què indica cada part del fitxer de configuració. Per començar, abans de l'obertura de claus, tenim l'identificador de context, que és necessari, ja que, en un únic fitxer, podrem especificar més configuracions identificant cadascuna d'aquestes configuracions mitjançant aquest nom al començament. Farem servir aquest identificador per a indicar al *LoginContext* quina de les configuracions volem utilitzar.

Immediatament després de l'obertura de la clau, tenim el nom de `LoginModule` que utilitzarem. En aquest cas, n'emprarem únicament un i indicarem, a més, que és requerit. Això farà que si l'autenticació no és satisfactòria per a aquest mòdul, l'autenticació fallarà. El *LoginModule* que hem decidit emprar és el `com.sun.security.auth.module.UnixLoginModule` que proporciona JAAS per defecte. El seu funcionament és molt senzill; s'encarrega d'obtenir les credencials de l'usuari que executa el programa, tant el seu nom d'usuari com els seus grups, i les utilitza automàticament, de manera que l'usuari queda autenticat. No requereix, per tant, cap interacció per la nostra banda. Com assenyala el nom, aquest mòdul està indicat especialment per a sistemes Unix, de manera que si ens trobem en un sistema Windows hem d'utilitzar `com.sun.security.auth.module.NTLoginModule` en lloc seu. Trobarem els diferents *LoginModules* que proporciona JAAS per defecte juntament amb una descripció de la utilitat que tenen a la referència de l'SDK per al paquet `com.sun.security.auth.module`.

Finalment, en el nostre recorregut pel fitxer de configuració, tenim la paraula *required*. Aquesta paraula és un indicador o *flag* que assenyala quin serà el comportament d'aquesta línia en la pila d'autenticació. Definim, bàsicament, tants mòduls d'autenticació com siguin necessaris, que són executats en ordre d'aparició en el nostre fitxer de configuració. L'indicador mostrat després del mòdul d'autenticació pot prendre els valors següents:

- *required*. Perquè l'autenticació de l'usuari sigui vàlida, el mòdul que acompanya aquest indicador ha de retornar *cert* (*true*). Independentment del resultat, s'executen la resta de mòduls especificats a la pila, encara que el resultat no es té en compte.

- *requisite*. Semblant a *required*, però si el mòdul retorna *fals* (*false*), no se'n crida a cap més.

- *sufficient*. Si l'execució d'aquest mòdul retorna *cert*, no s'executa cap més mòdul i es considera vàlida l'autenticació. Si falla, es continuen executant els mòduls en ordre i el resultat de l'autenticació depèn d'aquests mòduls.

- *optional*. No importa si aquest mòdul retorna *cert* o no; l'execució dels mòduls següents continua igualment.

El codi següent mostra com es comporten els diferents indicadors en l'avaluació i permet observar el resultat de l'autenticació:

```
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 sempre retorna fals com a resultat de l'autenticació.
*/
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 sempre retorna cert com a resultat de l'autenticació.
*/
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. Desenvolupar el nostre propi mòdul d'autenticació

Encara que d'una forma bàsica, ja hem vist com s'ha de fer ús dels diferents serveis d'autenticació que ens proporciona JAAS. No obstant això, moltes vegades, amb això no n'hi ha prou i, per a esprémer vertaderament el potencial i la flexibilitat de JAAS, les nostres aplicacions fan ús dels nostres propis *LoginModule*. Això és així perquè, generalment, les nostres aplicacions han de fer ús d'uns sistemes d'autenticació que ja hi havia, als quals hem d'emmotllar-nos.

Per a fer-ho, hem de desenvolupar un mòdul d'autenticació o `LoginModule`. Aquesta interfície inclou els mètodes necessaris per a qualsevol sistema d'autenticació que, una vegada implementats, podem incorporar al nostre sistema.

El `LoginModule` que crearem, si bé no serà gaire útil, ens servirà per a fer diferents proves i investigar la manera en què funciona internament l'API, i alhora ens guiarà en els passos necessaris.

La idea és crear un `LoginModule` que sol·licitarà a l'usuari una clau d'accés. El nostre mòdul accedirà a un fitxer que conté parelles amb el nom d'usuari i el *hash* de la clau d'accés per a aquest usuari, comprovarà si coincideix, i autenticarà l'usuari.

La creació d'un `LoginModule` requereix que implementem els mètodes d'interfície següents: `initialize`, `login`, `commit`, `abort` i `logout`. Aquests mètodes són els que donaran comportament al nostre mòdul i autenticaran els usuaris.

El mètode *initialize* és cridat durant la inicialització del nostre mòdul i ens permet configurar aquest mòdul, inicialitzar connexions i obtenir les dades que ha especificat l'usuari en el fitxer de configuració.

```
public void initialize (Subject subject,
                        CallbackHandler handler,
                        Map<java.lang.String, ?> sharedState,
                        Map<java.lang.String, ?> options);
```

El primer paràmetre *Subject* és el subjecte que autentiquem i que hem de modificar afegint-hi els *principals* que corresponguin.

El segon, el *handler*, ens permet interactuar amb l'usuari d'una manera que independitza el nostre *LoginModule* del mètode d'entrada de dades. Això permet que el nostre mòdul s'utilitzi per a autenticar un usuari, introduint-ne el nom i la contrasenya a la consola, de la mateixa manera que s'utilitzaria per a autenticar un usuari que introdueix les seves credencials en un formulari web.

Finalment, rebem dos objectes *Map*. El *sharedState*, com indica el nom, ens permet mantenir un estat comú entre tots els `LoginModules` que s'utilitzen en l'autenticació. D'altra banda, el paràmetre *options* inclou les opcions que s'han detallat en el fitxer de configuració.

El mètode `login()` és cridat per a autenticar el *Subject* que hem rebut durant la inicialització. En el nostre cas, en aquest mètode sol·licitem el nom d'usuari i la clau d'accés, i comprovem que es correspongui amb els valors emmagatzemats. El codi següent fa exactament aquesta operació:

```
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();
```

La nostra petita funció de *login* fa una primera crida al `CallbackHandler` que hem emmagatzemat en la inicialització del `LoginModule`. Aquesta crida recorre els `Callback` que ha registrat l'aplicació que usa el `LoginModule` i obté les credencials de l'usuari.

El mètode *verifyCredentials* és una petita funció d'utilitat que hem creat a fi de comprovar si l'usuari i la clau coincideixen. No retorna cap valor, ja que si l'autenticació falla emet una excepció `javax.security.auth.login. FailedLoginException`.

Si l'autenticació té èxit, el nostre mòdul ha d'actualitzar els *principals* del *Subject*, en el qual hem guardat una referència en la funció d'inicialització. Amb això, concedim al nostre *Subject* una identitat, amb la qual més endavant podrà ser autenticat per a comprovar quan té permisos per a executar una acció.

El nostre mòdul utilitza també un *principal* desenvolupat específicament per a aquest mòdul. Aquest *principal* s'encarrega, únicament, d'emmagatzemar la identitat de l'usuari autenticat.

El *principal* no ha pas d'estar relacionat directament amb el mòdul que l'utilitza, i hi ha diferents mòduls que poden compartir el mateix *principal*. Generalment, això no és així, ja que les identitats generades per diferents sistemes d'autenticació tenen diferents requisits d'emmagatzematge. En el nostre cas, n'hi ha prou amb una única cadena alfanumèrica, i la classe queda com segueix:

```
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();
   }
}
```

Els mètodes *commit* i *abort* s'utilitzen en el cas d'autenticacions que requereixen dues fases per a funcionar. Això permet que el nostre mòdul alliberi recursos o modifiqui el comportament depenent de si l'autenticació ha estat correcta, cas en què es crida al mètode *commit*, o ha estat fallida, cas en què es crida al mètode *abort*. Si, per exemple, el nostre mòdul reserva recursos en un altre node o manté un estat de l'autenticació mentre es porta a terme aquesta autenticació, aquests mètodes són el lloc adequat per a alliberar aquesta sessió o informar el node remot que la informació no és necessària.

El nostre mòdul d'exemple és molt senzill i no requereix aquesta segona fase d'autenticació, de manera que els mètodes *commit* i *abort* són buits i retornen *cert*.

L'últim pas és emprar el nostre mòdul acabat de crear dins d'una autenticació. Encara que més endavant en veurem la utilització en un exemple més complet, ara simplement ens quedarem amb un petit codi de prova. Basant-nos en l'exemple d'autenticació, que es fonamentava a obtenir les credencials de l'usuari autenticat actualment en el sistema, farem uns petits canvis per a proporcionar un usuari i una clau, que el *LoginModule* utilitzarà per a comprovar les credencials:

```
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("usuari1");
             } else if (cb instanceof PasswordCallback) {
                    ((PasswordCallback) cb).setPassword("password1".toCharArray());
             } else if (cb instanceof TextOutputCallback) {
                   System.out.println(((TextOutputCallback) cb).getMessage());
             }
          }
       }
    }
    );
```

Veiem que, aquesta vegada, en lloc de no proveir de *Callbacks*, proporcionem suport per a tres tipus diferents. El primer, el `javax.security.auth.callback.NameCallback`, és inclòs a la pila de crides quan el `LoginModule` necessiti del nom d'usuari. El segon, el `javax.security.auth.callback.PasswordCallback`, de manera semblant, s'utilitza quan es necessita la contrasenya de l'usuari. Finalment, el `javax.security.auth.callback.TextOuputCallback` és afegit pel `LoginModule` quan necessita comunicar alguna informació en forma de missatge a l'usuari.

Per simplificar l'exemple, hem afegit la resposta als diferents `Callback` de forma estàtica, però sol ser habitual que aquests `Callback` demanin la intervenció de l'usuari, sia mitjançant un quadre de diàleg o mitjançant la consola, i demanin les credencials.

Finalment, indiquem en la configuració de JAAS dins del fitxer *JAAS.conf* que volem utilitzar el nostre `LoginModule` per al *Context* "`MyLoginModuleTest`":

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

1.2. Autorització amb JAAS

L'autenticació de l'usuari només és la primera part del procés que ens permet associar un *subject* a un context d'accés. La segona part, l'autorització, té únicament sentit després que l'usuari ha estat autenticat.

L'autorització amb JAAS, com en qualsevol altre sistema, consisteix a determinar quan té permís un *subject* per a fer una determinada acció; l'única peculiaritat de JAAS és que ens permet afegir la dimensió del codi que s'executa.

Els permisos són classes que hereten de `Java.security.Permission` i representen l'habilitat de fer una acció o més d'una sobre un recurs determinat. Representen des d'accés a fitxers com `Java.io.FilePermission` fins a accés a propietats com `java.util.PropertyPermission`. Més endavant, veurem la manera d'utilitzar i configurar aquests permisos per controlar l'accés sobre els recursos.

A més del tipus a què pertanyen, els permisos poden contenir opcionalment una sèrie d'accions, que permeten un control més granular d'aquests permisos. En l'exemple del `java.util.PropertyPermission`, aquestes accions són "*read*" i "*write*".

Quan comparem dos permisos, més que comprovar que són exactament iguals, comparem si un s'inclou en l'altre per determinar si l'acció és vàlida. Suposem que tenim un permís que ens permet llegir i escriure fitxers que es troben sota el directori "/segur", i volem llegir el fitxer "/segur/molt". Si definim els permisos com a tuples, obtenim una cosa com:

- Atorgat: ("/segur", "llegir i escriure")
- Requerit: ("/segur/molt", "llegir")

Com veiem, una simple comprovació d'igualtat no ens diu si tenim el permís que requerim. En canvi, gràcies al mètode *implies*, podem comprovar que ("/segur", "llegir i escriure") *implies* ("/segur/molt", "llegir") i que, per tant, estem autoritzats a fer aquesta operació.

Els permisos es poden atorgar a un codi determinat mitjançant un arxiu de polítiques que, més endavant, veurem com s'ha de definir. Aquests permisos atorguen la capacitat de portar a terme una acció en un determinat codi i, opcionalment, en un *principal*.

La classe **`java.lang.SecurityManager`** és el punt d'entrada a l'API d'autorització de JAAS i permet que les nostres aplicacions implementin determinades polítiques de seguretat. Mitjançant aquesta classe, podem comprovar quan està autoritzada o no una determinada acció en un determinat context. Aquesta classe no és especialment útil si la nostra única intenció és comprovar quan té permisos el nostre codi sobre accions existents; no obstant això, és vital quan volem saber quines parts determinades de l'execució són subjectes a restriccions.

Per a obtenir una instància de `java.lang.SecurityManager` hem de cridar al mètode `System.getSecurityManager()`. Aquest mètode ens retorna la instància actual que ens permet accedir a la funcionalitat de JAAS. Un detall important que cal tenir en compte és que, per defecte, Java no proporciona un `java.lang.SecurityManager`, de manera que la crida a

`System.getSecurityManager()` retorna *nul* o *null*. Per a activar les capacitats d'autorització, hem d'activar la propietat `-Djava.security.manager`, cosa que inicialitza el `SecurityManager`.

Encara que la classe `java.lang.SecurityManager` té molts mètodes, la major part són heretats de versions anteriors de l'entorn de treball de seguretat de Java i es desaconsella usar-los. Generalment, hem d'utilitzar el mètode `checkPermission`, que determina, en el context actual, si s'ha concedit un permís a un determinat recurs. La major part dels mètodes checkXXXX que trobem en aquesta classe fan ús en algun moment del mètode `checkPermission`.

Les **polítiques** són el punt de trobada entre el *principal*, els permisos i les accions. En aquestes polítques, es defineix quan té permisos un *principal* per a executar una determinada acció. Per a fer-ho, ha d'utilitzar un fitxer de text que ens permet especificar aquestes polítiques; per exemple:

```
grant codebase "file:./SamplePermissions.jar",
      Principal login.MySampleLoginModule
            "usuari1" {
   permission java.util.PropertyPermission "java.home", "read";
};
```

Aquesta política indica que es concedeix (*grant*) el permís `java.util.PropertyPermission` al codi contingut a `SamplePermissions.jar` i al *principal* "usuari1" a la propietat "`Java.home`". A més, en el cas de la `PropertyPermission`, hem especificat també l'acció (*read*) que volem permetre que es porti a terme.

El format complet d'una política és el següent:

```
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";
   };
```

Vegem quina utilitat té cada camp d'aquesta definició. Les polítiques comencen amb la paraula clau "*grant*", que indica que es concediran permisos a un determinat recurs. Després, indiquem la signatura del codi, de manera que assegurem que aquest codi no ha estat alterat.

Tot seguit, trobem l'URL que defineix a quina part del nostre codi donem els permisos. En el nostre cas, en què executem el codi localment, l'URL té un format "`file://<fitxer>`", en què *fitxer* pot indicar una ruta i contenir comodins o *wild cards* al final de la cadena. De la mateixa manera, *fitxer* pot ser un fitxer *class* o més d'un, o fins i tot fitxers *jar*.

Vegem alguns exemples d'aquests URL:

Patró	Descripció
file://./LaMevaClasse.class	Concediria permisos a la classe LaMevaClasse en el directori d'execució.
file://./utilitats/ElMeuPrograma.jar	Concediria permisos al codi contingut en el fitxer *jar* del directori utilitats.
file://./biblioteques/xarxa/*	Concediria permisos a tot el codi contingut en el directori biblioteques/xarxa, independentment de si es tracta de fitxers *jar* o fitxers *class*.

Exemples d'URL en polítiques

Continuant d'acord amb l'especificació de la política, trobem la paraula clau *Principal*, seguida de la classe que indica el tipus d'aquest *Principal*. A continuació, hem d'especificar el nom del *Principal*, que depèn de la classe *Principal* que utilitzem. Vegem alguns exemples de definició de *Principals*:

Definició	Mòdul d'autenticació
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`

Exemples de *Principals* per a autoritzar l'usuari amb ID "user"

Finalment, trobem la llista de permisos i accions. La definició d'aquests permisos i accions inclou la paraula clau "*permission*" juntament amb el nom de la classe que els implementa. El camp següent representa el recurs o "*target*", i depèn del tipus de permís, igual que les opcions.

Un exemple d'això és la definició següent:

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

Així indiquem que s'atorga permís de lectura a la propietat "*java.home*".

Una vegada creat el nostre fitxer de polítiques, hem d'indicar que el volem utilitzar mitjançant l'ús de la propietat `Java.security.policy`. Ho podem indicar com a paràmetre en executar el nostre codi des de l'indicador d'ordres utilitzant el paràmetre *-D*, com hem fet amb el fitxer de configuració per a l'autenticació inicial.

És recomanable utilitzar l'opció `-Djava.security.debug=all` de la JVM, que ens proporcionarà informació valuosa de depuració sobre el procés d'autenticació i autorització.

1.3. Autenticació i autorització: executar accions privilegiades

Quan ja hem definit les polítiques que s'aplicaran al nostre codi, és hora de comprovar-ne el funcionament. Com hem dit abans, la política defineix el moment en què el nostre codi té permisos per a executar una determinada acció.

Crearem un petit programa d'exemple que ens permetrà combinar tot el que hem vist fins ara, des de l'autenticació de l'usuari fins a l'autorització d'una determinada acció.

L'esquelet inicial de la nostra aplicació és el codi i la configuració que utilitzem en l'exemple d'autenticació. Autentiquem l'usuari i, una vegada obtingut el *subject*, l'utilitzem per a executar una senzilla acció privilegiada, en aquest cas, l'accés a una propietat de l'entorn.

El primer que hi hem d'afegir és el fitxer de polítiques esmentat. L'acció que volem autoritzar és la lectura de la propietat del sistema "*Java.home*" que indica la ubicació del JRE. Per a fer-ho, utilitzem la política següent:

```
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
            "usuari" {
   permission java.util.PropertyPermission "java.home", "read";
};
```

Vegem, pas a pas, com funciona cada part d'aquesta política. La primera part crea una autorització que no depèn de cap *Principal* i, per tant, es concedeix a qualsevol usuari de l'aplicació sense que aquest usuari hagi d'estar autenticat. Els dos `AuthPermission` els atorguem a dues accions, "`createLoginContext.Simple`" i "`doAsPrivileged`".

En el primer cas, una vegada activada l'autenticació de Java, necessitem permisos per a crear nous `LoginContext` que ens permetin autenticar l'usuari. Aquest primer indicador assenyala que volem atorgar permisos a tots els usuaris per a crear un nou `LoginContext` de tipus *Simple*.

El segon indicador ens permet cridar al mètode *doAsPrivileged* de la classe *Subject*. Aquest mètode ens permet executar una acció privilegiada, en el nostre cas la lectura de la propietat, associant a aquesta acció un determinat *Subject* sobre el qual s'avaluen les accions que cal portar a terme. Altrament dit, és la manera que tenim d'associar un *Subject* i l'acció que volem executar.

Generalment, no trobarem aquestes dues polítiques juntes, ja que el codi que s'encarrega de crear el context no es troba sota el mateix domini de seguretat que la resta de l'aplicació. En el nostre exemple, a l'efecte de simplificar el codi, utilitzem una política que indica els permisos tant per a l'autenticació de l'usuari com per a l'acció concreta que cal executar.

Una vegada creada la nostra política, només ens falta afegir el codi per a executar la nostra acció privilegiada. Aquest codi és una cosa com:

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

Aquest codi, encara que no és gaire útil, ens permet comprovar com s'executen els controls d'accés sobre les accions privilegiades.

Si ho hem fet tot correctament, en executar el nostre programa hem de veure el valor de la propietat *java.home*:

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

El resultat depèn del sistema operatiu en què executem el nostre programa.

Si obtenim una excepció d'accés, comprova primer que els fitxers de polítiques i de configuració es troben en el directori actual d'execució.

Si ha anat tot bé, el pas següent és comprovar que la política té algun efecte sobre les nostres accions; per a fer-ho, simplement canviem el nom de l'usuari per un de no existent en el nostre sistema i tornem a executar el programa. Aquesta vegada, hem d'obtenir una cosa com:

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

Amb això, hem vist la manera en què es porta a terme tot el procés d'autenticació i autorització utilitzant l'entorn de treball de JAAS. És extens i complex, però ofereix una infinitat de possibilitats.

1.4. Autenticació web usant JAAS

Un entorn molt apropiat, i que requereix una gestió correcta de l'autorització i autenticació, són les aplicacions web. El desenvolupament d'aplicacions web en Java és un món molt extens que donaria per omplir prestatgeries de llibres sobre el tema.

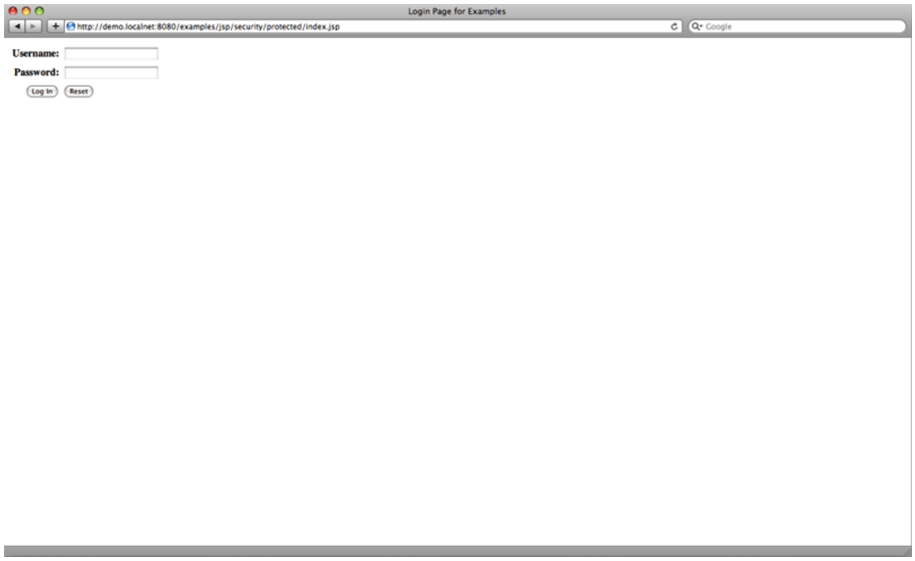
Ens centrarem en la manera d'aconseguir que una aplicació molt senzilla, mitjançant *Servlets*, ens permeti autenticar un usuari i utilitzar aquesta autenticació per a autoritzar-lo a fer diferents accions.

Malauradament, l'especificació de *Servlets* no diu res de com s'ha de fer l'autenticació i autorització, i encara menys de quin és el paper de JAAS en tot plegat. Això fa que, encara que els conceptes són semblants entre els diferents servidors d'aplicacions, cadascun implementi aquests conceptes de manera diferent, i així, doncs, fan de la configuració de seguretat una cosa molt específica.

Per al nostre exemple, utilitzem el servidor d'aplicacions Apache Tomcat, versió 6, per diverses raons. La primera és que és disponible gratuïtament i té un ús molt estès. La segona és que la majoria dels conceptes que apliquem en el Tomcat són fàcils d'extrapolar a servidors d'aplicacions més complexos i potents. Ho aprofitarem per a utilitzar el *LoginModule* que hem creat abans amb la nostra aplicació i que ens permetia emmagatzemar les contrasenyes en un arxiu local.

El primer que hem de conèixer, abans de llançar-nos a implementar seguretat en el Tomcat, és el concepte de *Realm*. El Tomcat defineix en la seva documentació un *Realm* com una base de dades d'usuaris i contrasenyes, unida als rols que poden exercir aquests usuaris. Mitjançant aquest concepte, el Tomcat ens permet utilitzar diferents mecanismes o *backends* d'autenticació com, per exemple, JDBC, per a mantenir els nostres usuaris en una base de dades. També ens permet escriure el nostre propi *Realm*, però, en el nostre cas concret, ens centrarem en el *JAASRealm*, que ens proporciona una interfície perquè el Tomcat utilitzi els diferents *LoginModules*, sia els que ja hi ha o, com en aquest cas, un de construït a mida.

El Tomcat 6 inclou per defecte, dins dels seus exemples, una aplicació que mostra com s'ha d'utilitzar l'autenticació en el servidor web, utilitzant un formulari per a sol·licitar les credencials. Emprarem aquesta aplicació com a base per a fer que, aplicant-hi alguns canvis, utilitzi el nostre propi `LoginModule`. Una vegada instal·lat el Tomcat i els exemples, simplement apuntem el navegador a http://localhost:8080/examples/jsp/security/protected i ens sortirà la pantalla següent:



Inici de sessió de l'aplicació d'exemple

Modifiquem la configuració que s'hi inclou amb els exemples que acompanyen el Tomcat. Si no es vol tornar a instal·lar-los, cal fer una còpia de seguretat de manera que es pugui retornar a l'estat original.

Un dels *Realms* que inclou per defecte el Tomcat és el ***MemoryRealm***. Aquest *Realm* no està pensat per a producció, però és el que trobarem per defecte per a les aplicacions d'exemple i consola d'administració del Tomcat, una vegada l'hem instal·lat.

El funcionament és senzill: llegeix el fitxer *tomcat-users.xml* del directori de configuració del Tomcat i carrega els usuaris en memòria. Aquest fitxer té el format següent:

```
<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 fitxer s'explica per si mateix. Com ja hem dit, no està pensat per a utilitzar-lo en un entorn "real", per diverses raons. En primer lloc, les contrasenyes estan emmagatzemades en text pla i, en segon lloc, si hem d'emmagatzemar una base d'usuaris més o menys gran, la gestió d'aquesta base pot ser complicada.

El nostre `LoginModule` soluciona el primer d'aquests dos problemes. Té una funcionalitat semblant, però no emmagatzemem les contrasenyes, sinó el *hash* d'aquestes contrasenyes. Fins i tot amb aquesta petita millora no es recomana

utilitzar aquest sistema en entorns de producció, ja que, encara que millora lleugerament la seguretat, continua essent vulnerable si s'aconsegueix el fitxer; d'altra banda, continuem tenint el problema d'administrar-lo.

Per a **configurar el nostre *Realm***, el primer que hem de fer és indicar al Tomcat quin *Realm* volem utilitzar en lloc del que hi ha configurat per defecte. Aquesta configuració es troba al fitxer `server.xml`. L'etiqueta (*tag*) *Realm* es pot especificar en diferents llocs segons l'abast que tingui el nostre sistema d'autenticació; en el nostre cas, ens limitem a afegir-la a l'etiqueta *Context* de la nostra aplicació. La documentació del Tomcat descriu com es pot usar dins de *Host* o *Engine* i les implicacions que té. Hi afegim l'etiqueta següent:

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

En primer lloc, l'atribut *className* indica el *Realm* que volem utilitzar; en el nostre cas, `org.apache.catalina.realm.JAASRealm`, que és l'encarregat d'enllaçar el sistema d'autenticació del Tomcat amb JAAS.

`AppName` indica el nom de l'aplicació que emprem i s'utilitza per a indicar quina de les diferents polítiques d'autorització aplica dins de les que definim en el nostre `JAAS.config`. Hem d'emprar, per tant, el mateix nom de la política d'autorització que volem utilitzar.

Finalment, hi indiquem `userClassNames` i `roleClassNames` per assenyalar quines classes emmagatzemaran el *Principal* d'usuari i del seu rol. Veurem, més endavant, quines modificacions hem de fer al nostre *LoginModule* perquè funcioni amb el concepte de *Rol*.

Una vegada configurat el *Realm*, ja podem anar al fitxer `web.xml` de la nostra aplicació web per habilitar l'autenticació.

El Tomcat ens proporciona diversos mètodes d'autenticació, com *BASIC, DIGEST* i *FORM*. En el nostre cas, volem proporcionar la nostra pròpia pàgina d'autenticació, de manera que utilitzem *FORM*. Per a fer-ho, hem d'especificar la configuració d'inici de sessió de la nostra aplicació:

Aquestes credencials viatjaran en text pla des del navegador fins al servidor i, per tant, com qualsevol altre sistema d'autenticació, han de ser protegides per 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>
```

Aquesta configuració s'explica per si mateixa. Ens permet canviar el mètode d'autenticació entre els esmentats abans i ens permet indicar quines són les pàgines d'autenticació i d'error quan l'usuari no ha estat autenticat.

Els ***Rols* en el Tomcat** són el mecanisme que ens permet fer l'autorització dels usuaris per a executar determinades accions. Una acció està restringida a uns determinats *Rols*, de manera que es nega l'accés als que no tenen aquest *Rol*.

Una vegada hem definit l'estructura bàsica de la nostra aplicació, hem d'especificar quins rols poden exercir els usuaris.

La configuració és molt senzilla, ja que es limita a una enumeració d'aquests rols amb el format següent:

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

Aquesta llista ha de contenir tots els rols que volem usar més endavant per a gestionar l'autorització en l'accés a determinats URL.

Finalment, hem d'especificar l'*ACL* de la nostra aplicació utilitzant els rols que ja hi ha. L'aplicació d'exemple que ens proporciona el Tomcat conté la definició següent:

```
<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></http-method>
   </web-resource-collection>
   <auth-constraint>
      <role-name>PUT tomcat</role-name>
```

```
      <role-name>role1</role-name>
   </auth-constraint>
</security-constraint>
```

Aquesta definició indica que totes les pàgines que es troben per sota de */jsp/security/protected* s'han de protegir per a qualsevol dels mètodes especificats a la llista. A més, hi incloem quins *Rols* tenen accés a aquestes pàgines i quins el neguen a qualsevol usuari que no tingui aquest *Rol*.

No hi ha res que impedeixi utilitzar diferents nivells d'autorització per a diferents mètodes HTTP. Podem definir diversos *security-constraint* que indiquin la mateixa ruta, però diferents mètodes o fins i tot *Rols*.

Com que el concepte de *Rol* no existeix com a tal en JAAS, hem d'utilitzar una clau de configuració que indiqui **els grups** a què pertanyerà l'usuari. Aquests grups identifiquen a quins *Rols* pertany l'usuari, de manera que, amb només uns petits canvis, puguem fer ús dels *security-constraint*.

Quan al començament vam crear el *LoginModule*, no vam tenir en compte el fet que l'usuari pogués pertànyer a grups, però amb unes petites modificacions podrem obtenir el mateix efecte. Creem una nova classe *Principal*, que hereta del nostre *SimplePrincipal*, a la qual anomenem *SimpleGroupPrincipal*, amb el codi següent:

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

Ara hi afegim el codi per a generar aquests *Principal*. Ho fem d'una manera molt senzilla: quan l'usuari sigui autenticat correctament, comprovem si el fitxer conté una clau `usuari.groups=grup1,grup2,...,grupn` que ens indiqui a quins grups pertany. Hi afegim un `SimpleGroupPrincipal` per cadascun dels grups.

Per tant, el nostre mètode `login()` queda com segueix:

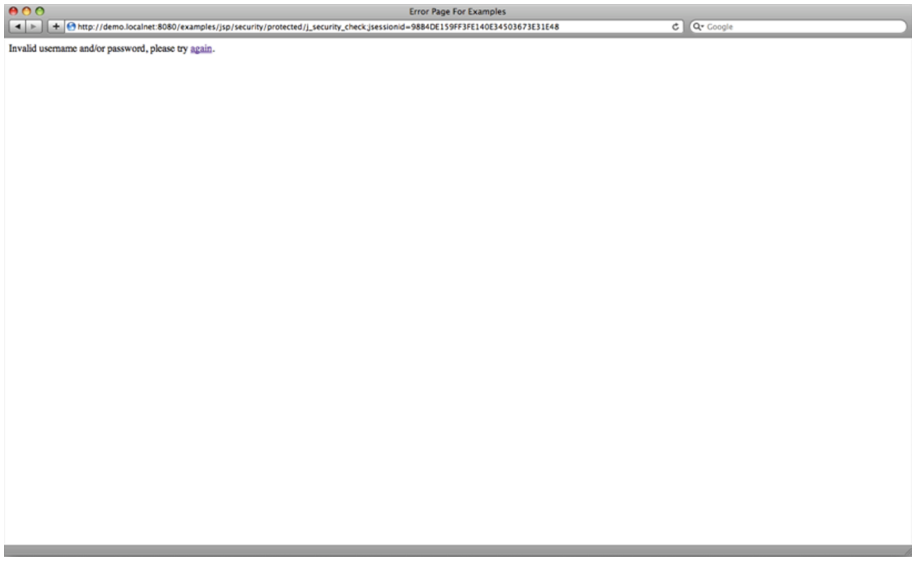
```
public boolean login() throws LoginException {
   try {
      cbh.handle(cbs);
      verifyCredentials(nameCb.getName(), new String(passwdCb.getPassword()));
      subject.getPrincipals().add(new SimplePrincipal(nameCb.getName()));
      String[] groups = getUserGroups(nameCb.getName());
      for(String 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(",");
    }
```

Hem de modificar també ara el fitxer de contrasenyes que, al començament, vam crear per afegir els grups. L'usuari1 pertany als grups "tomcat" i "role1", mentre que l'usuari2 no en pertany a cap. Això ens permet comprovar com afecten els rols als permisos de l'usuari autenticat.

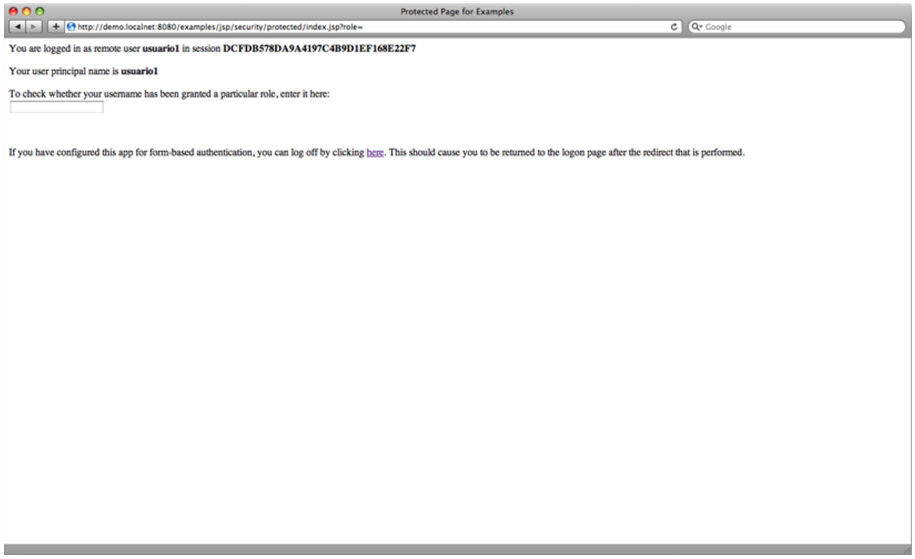
Totes les classes que creem han de ser disponibles al *classpath* del Tomcat perquè JAAS hi pugui accedir. Així mateix, cal indicar la ruta en què hi ha el nostre fitxer de configuració. Això ho aconseguim utilitzant l'opció *-Djava.security.auth.login.config=$CATALINA_HOME/conf/JAAS.config* en la inicialització, i substituint la ruta per l'adequada en el nostre cas. Una vegada fet això, hem de reiniciar el Tomcat perquè torni a carregar el nostre fitxer de configuració.

Una vegada reiniciat, ens dirigim a la ruta on hi ha l'exemple d'autenticació que es lliura amb el Tomcat en l'URL http://localhost:8080/examples/jsp/security/protected, on trobem la pàgina d'inici de sessió que hem vist abans. Hi introduïm un dels usuaris predefinits del Tomcat per comprovar que l'autenticació no es fa contra la configuració per defecte; tomcat/tomcat hi va bé. Comprovem que el nostre inici de sessió fallarà:



Usuari/contrasenya incorrecte

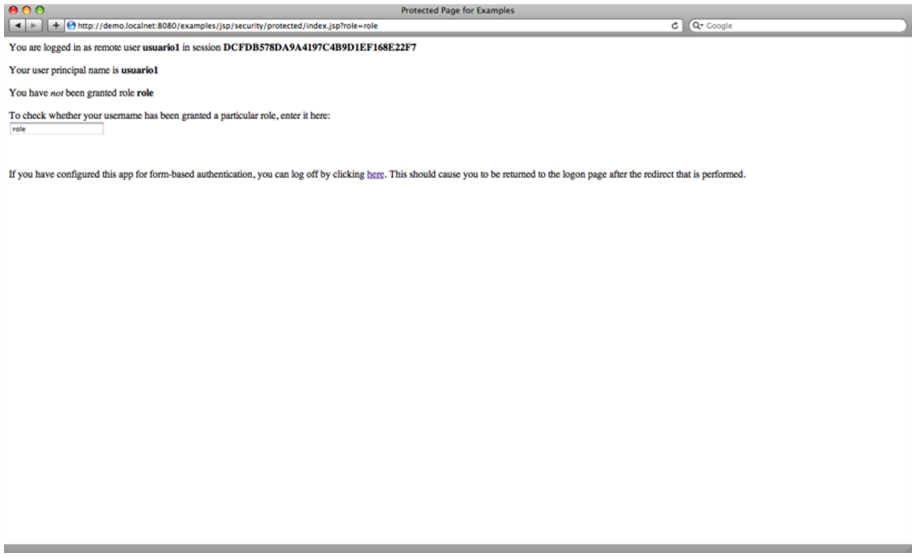
No obstant això, si ara utilitzem un dels nostres usuaris, com per exemple usuari1/contrasenya1, comprovem que l'inici de sessió es fa satisfactòriament.



Usuari/contrasenya correcte

Si introduïm un dels rols que hi ha configurats en el nostre fitxer `passwd`, com per exemple Tomcat, comprovem que, efectivament, el nostre mòdul ha afegit també els *Principal* corresponents als grups.

Si ara fem final de sessió o *logout* i utilitzem usuari2 en lloc seu, obtenim la pàgina d'error següent:



Usuari no pertany al *Rol*

Això és perquè, aquesta vegada, el nostre usuari2 no pertany a cap dels rols que abans hem donat d'alta en el *security-constraint* que protegeix aquesta pàgina.

2. Autenticació i autorització en PHP

A diferència de Java o ASP.NET, en PHP, desgraciadament, no hi ha un estàndard tan clar d'autenticació i autorització que ens permeti implementar, d'una manera transparent i única, els elements de seguretat adequats. PHP ens dóna per defecte uns simples mecanismes per a utilitzar autenticació HTTP; no obstant això, no tracta en cap moment l'autorització. Per aquest motiu, i pel fet que, excepte per a petites aplicacions, no se solen usar directament, no ens hi centrarem. Són senzills d'utilitzar i els trobarem documentats i amb exemples a la referència de PHP.

Per això, hi ha diferents entorns de treball a PHP que proporcionen diferents solucions per al mateix problema, cadascun amb pros i contres. Encara que cadascun manté una filosofia diferent, els conceptes són iguals per a tots; únicament, l'associació d'aquests conceptes als mecanismes que els implementen varia d'un entorn de treball a un altre. Si tenim clars els conceptes d'*autenticació* i *autorització*, entendre la funcionalitat en cada entorn de treball es redueix a entendre on hi ha aquests mecanismes.

En aquest capítol, veurem com es manegen aquests conceptes en dos dels entorns de treball més coneguts de PHP, tant l'autenticació d'usuaris remots com l'autorització posterior d'aquests usuaris per a portar a terme diferents accions.

La funcionalitat dels entorns de treball és, principalment, la de facilitar-nos el disseny de tasques repetitives o molt comunes mitjançant l'ús de patrons i estructures de dades que, seguint una estructura lògica, ens portin a fer aquestes tasques.

No ens centrarem en la instal·lació dels diferents entorns de treball, ja que està perfectament documentada en els llocs web perquè és molt específica de l'entorn en què fem les proves.

2.1. Autenticació en PHP

Abans de començar a veure el funcionament dels diferents entorns de treball, cal fer un aclariment: PHP ens dóna els mecanismes necessaris per a accedir als elements d'autenticació que proporciona HTTP. Per tant, sense haver de tenir cap entorn de treball, podem utilitzar l'autenticació bàsica o *digest* d'HTTP. El problema principal d'aquest sistema d'autenticació és que, sense un sistema d'autorització adequat, no serveix de gaire i passa al codi del nostre client tota la responsabilitat de determinar si es pot portar a terme una acció. Això pro-

voca que hi hagi moltes aplicacions escrites en PHP que integrin l'autorització d'usuaris en la lògica de negoci, cosa que en complica enormement el funcionament i manteniment.

Aquest capítol no pretén ser una referència sobre funcionament i filosofia de cadascun dels entorns de treball. L'elecció de l'entorn de treball depèn de molts factors que escapen a l'abast d'aquest text. Durant aquest capítol, ens centrarem únicament en les capacitats d'autenticació i autorització dels entorns de treball, sense entrar en els detalls en la resta de components.

Dit això, comencem a explicar per sobre el funcionament de l'autenticació en PHP. Hi trobem, bàsicament, dues maneres d'autenticació, que, com hem dit abans, es corresponen amb els mètodes d'autenticació d'HTTP i varien el canal de transmissió de les credencials.

La millor manera de demostrar el funcionament de les diferents API **és amb exemples**. Ens centrarem, per tant, en el desenvolupament d'una petita aplicació, molt simple, en què implementarem el control d'usuaris i l'accés a algun recurs. El nostre objectiu no és mostrar l'ús de totes les possibilitats que proporcionen els entorns de treball, de manera que ens limitarem a una aplicació senzilla que ens serveixi d'entorn de proves per a jugar i veure els canvis.

L'aplicació farà d'un petit llibre de visites. Preveurem els casos següents:

- Els usuaris anònims podran veure les notes que hi deixi tothom.
- Els usuaris autenticats podran crear notes que aniran acompanyades del seu nom.
- Els usuaris amb permisos d'administració podran esborrar les notes.

Aquests tres casos ens permetran explorar els conceptes d'*autenticació*, per controlar quins usuaris afegeixen comentaris, i d'*autorització* en gestionar quins d'aquests usuaris tenen permisos per a eliminar comentaris.

Encara que treballarem amb dos entorns de treball diferents, l'estructura de l'aplicació serà molt semblant en tots dos casos, amb variacions molt específiques de cada implementació.

Seguint el patró model vista controlador (MVC), la nostra aplicació proporcionarà un controlador amb les accions de consultar, afegir i eliminar notes. Consultar i afegir tenen una vista associada que proporcionarà la interfície d'usuari necessària per a cada acció. Eliminar no donarà cap vista, ja que s'utilitzarà directament des de l'acció de consultar i ens retornarà a la vista principal una vegada s'haurà completat l'operació.

En el cas de la consulta de notes, proporcionarem una llista amb cadascuna de les notes que hi han deixat els usuaris. Quan l'usuari disposi de permisos per a això, hi afegirem un petit enllaç que ens permetrà eliminar qualsevol de les notes.

L'acció d'afegir oferirà una interfície amb un petit quadre de text que ens permetrà afegir-hi el comentari que deixarem per a futures visites.

Finalment, el model serà, com el que hem vist abans, extremament senzill. Disposarem d'una sèrie d'elements *Visita* que emmagatzemaran tres camps, un dels quals serà sencer per a identificar la nota, i els altres dos, de text, amb el nom de l'usuari i el comentari que hi ha fet.

Aquest serà l'esquelet bàsic de la nostra aplicació, encara que, amb l'objectiu de fer-hi compatible l'autenticació i autorització, ens veurem obligats a afegir-hi algun controlador i vista més que ens permeti aquesta gestió.

En els nostres exemples, utilitzarem com a motor de **base de dades SQLite** per les raons següents:

- Multiplataforma. Trobem SQLite executant-se en molts sistemes, i hi ha molts *bindings* per a diferents llenguatges que ens permeten utilitzar aquest motor de base de dades en un rang d'entorns molt ampli.

- Petit i senzill de configurar. Per al nostre exemple, utilitzarem una taula o dues que podrem definir en un petit nombre de sentències.

- Coneixement extrapolable. Encara que petita, SQLite proporciona una completa base de dades SQL, cosa que ens permet que el treball invertit en aquesta base de dades sigui extrapolable a una altra de més potent.

Com que ens centrarem únicament en els mecanismes de seguretat, no entrarem en les complexitats de la definició de models i la completa utilització d'aquests models des dels entorns de treball. D'altra banda, com que sí que emprarem usuaris, veurem com s'emmagatzemen generalment utilitzant els mecanismes d'accés a dades, cosa que dóna coherència a la utilització general. Per aquest motiu, sí que utilitzarem el que necessitem per a fer-ho.

La nostra base de dades serà l'encarregada de mantenir el model de què parlàvem abans. La idea és compartir la mateixa base de dades des de totes dues implementacions, ja que els requisits de l'aplicació són els mateixos.

```
sqlite> CREATE TABLE visites (
   ...> id INTEGER PRIMARY KEY,
   ...> usuari CHAR(50),
   ...> nota CHAR(140));
```

A l'efecte de facilitar-nos, més endavant, el fet de comprovar que la nostra aplicació funciona correctament, afegirem un parell de notes d'exemple al començament. Això ens permetrà fer proves sense esperar a tenir la capacitat d'afegir noves notes en lloc seu.

```
sqlite> INSERT INTO visites VALUES (null, "anònim", "exemple de nota 1");
sqlite> INSERT INTO visites VALUES (null, "anònim", "exemple de nota 2");
```

2.2. Autorització i autenticació en CakePHP

En el cas de CakePHP, trobarem la funcionalitat d'autorització i autenticació en els components *Acl*, *Auth* i *Security*. Els components en CakePHP són classes que insereixen un cert comportament dins de la lògica de la nostra aplicació. Dit d'una altra manera, aquests components s'encarreguen de funcionalitats horitzontals que són comunes a tota la nostra aplicació o a una bona part. Lògicament, les funcionalitats de seguretat, entre les quals hi ha l'autenticació i l'autorització, hi són incloses.

2.2.1. Autenticació en CakePHP

L'autenticació és una d'aquestes tasques vitals a pràcticament tota aplicació i, per descomptat, CakePHP s'encarrega de facilitar-nos-en la realització.

Per a autenticar usuaris, utilitzarem el component *Auth* de CakePHP. Aquest component, extremament configurable, ens permet fer diferents tipus d'autenticació.

Si volem utilitzar les facilitats que ens proporciona CakePHP per autenticar usuaris, se'ns exigeix complir algunes convencions.

El primer requisit és crear una taula que ha de contenir els camps "nom d'usuari" (*username*) i "contrasenya" (*password*). En el nostre SQLite, la descripció de la taula és:

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

Si ja teníem una taula d'usuaris i els camps no coincideixen amb els que utilitza CakePHP per defecte, hi ha la possibilitat de canviar-los per emprar-ne uns altres. El nom de grup o *groupname* no l'utilitzarem fins més endavant, en l'autorització d'usuaris.

De la mateixa manera, utilitzarem per al model de la nostra aplicació la taula "visites" creada abans.

Ara simplement crearem un *Controller* que s'encarregarà de gestionar l'autenticació dels nostres usuaris. Per seguir els estàndards de nomenclatura de CakePHP, nomenarem el nostre *Controller*`UsersController`, ja que el model que maneja és el de *Users*. Crearem aquest *Controller* en el fitxer `controller/users_controller.php` de la nostra aplicació. El codi del nostre `UsersController` és molt senzill i el trobarem detallat en la documentació de CakePHP:

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

Això és tot el que necessitem. L'acció *login* està definida a fi d'utilitzar la vista *login* que ens proporcionarà el formulari d'autenticació per als usuaris.

L'acció *logout*, d'altra banda, s'encarregarà simplement de cridar al mètode `AuthComponent::logout()`, que al seu torn netejarà les credencials de l'usuari de la sessió. Si l'usuari necessita tornar a accedir a alguna funció que requereixi que estigui autenticat, ha de tornar a passar abans per l'acció de *login*.

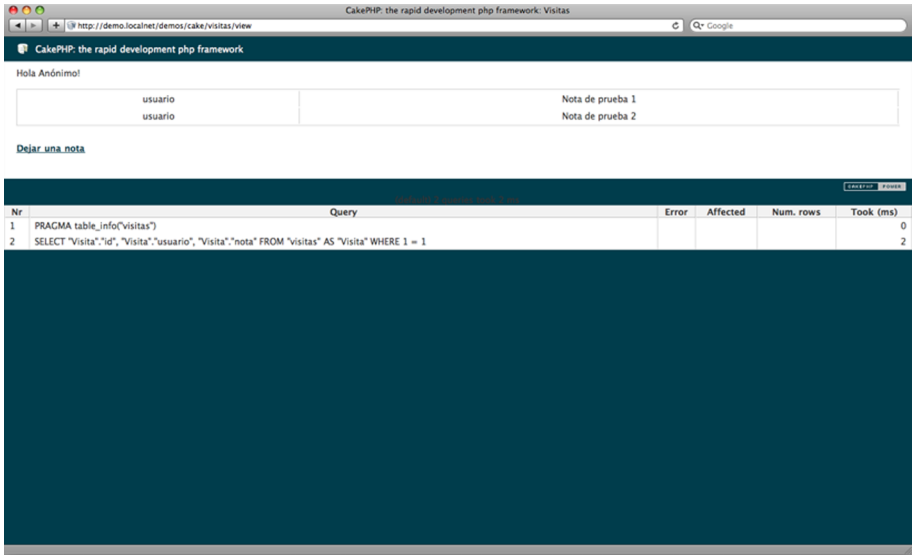
La part més senzilla és la d'afegir l'autenticació a la nostra aplicació.

El nostre *Controller* es compon de tres accions, dues de les quals volem que requereixin que l'usuari estigui autenticat, i l'altra que no. El seu codi en CakePHP és una cosa com:

```
class VisitesController extends AppController {
   var $name = "Visites";
   var $uses = array('Visita');
   function view() {
      $this->set('visites', $this->Visita->find('all'));
   }
   function add() {
      if (!empty($this->data)) {
         $this->data['Visita']['id'] = null;
         if ($this->Visita->save($this->data)) {
            $this->Session->setFlash('La teva nota ha estat afegida.');
         }
```

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

Aquest *Controller* proporciona tota la funcionalitat que necessitem per al nostre llibre de visites i, a més d'aquest *Controller*, disposarem de dues vistes de CakePHP, una per a l'acció d'afegir, que ens mostrarà el formulari que necessitem, i una altra per a veure les nostres accions.



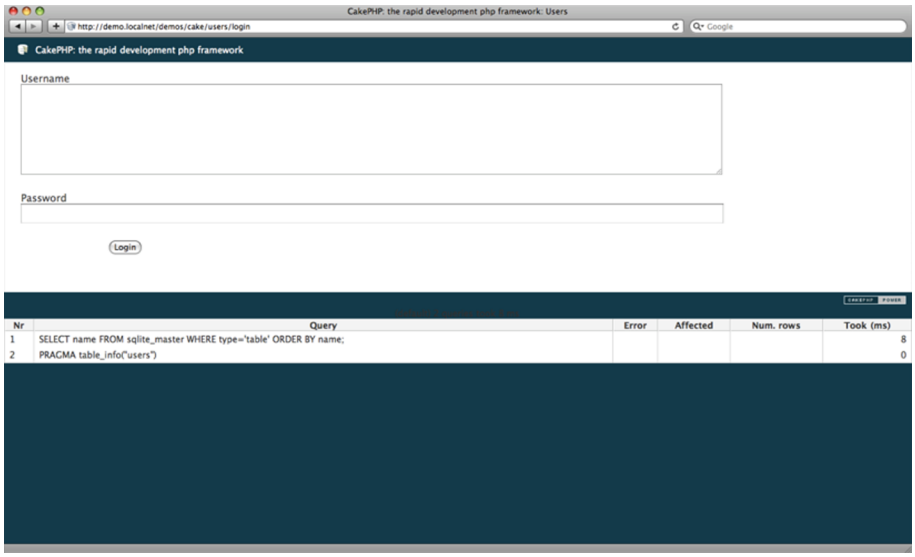
Vista anònima de visites

Fins ara, únicament hem afegit la funcionalitat de la nostra aplicació i creat la infraestructura necessària per a autenticar els usuaris, però com ja hem comprovat, encara no l'utilitzem. Implementar el suport d'autenticació en CakePHP és molt senzill: simplement, hem d'afegir el component *AuthComponent* al nostre controlador.

Encara que no ens hi vam aturar, és una cosa que ja vam fer per al *UsersController* i que per a fer-la en el nostre *VisitesController* requereix l'indicador següent:

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

Només amb aquest indicador, la nostra aplicació ha afegit autenticació d'usuaris a les seves funcions.



Pantalla d'inici de sessió de visites

Per a fer funcionar això, hem de crear primer un usuari de prova. Per a centrar-nos en l'autenticació i autorització, no complicarem la nostra aplicació d'exemple amb la gestió d'usuaris, sinó que els inserirem directament a la base de dades com hem fet amb les primeres visites. La contrasenya, per descomptat, no s'emmagatzema en blanc a la base de dades, sinó que s'hi emmagatzema el seu *hash* SHA1.

```
INSERT INTO "Users" VALUES (null, "usuari",
   "874d0b1i41c13d9b457a0975a0323i13i711b8c8", "Usuaris");
```

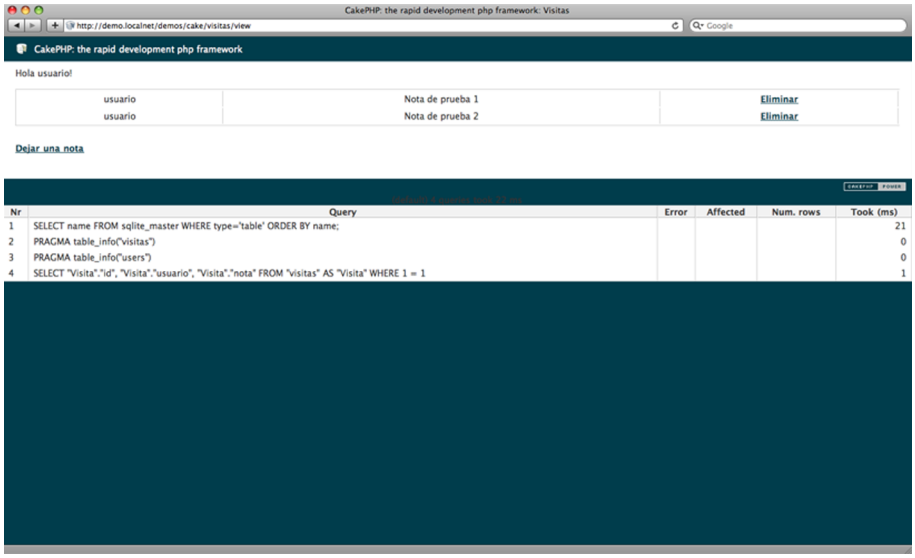
Hem afegit l'usuari al grup "Usuaris". Hem de crear també un nou usuari *admin*, que no utilitzarem fins més endavant. La sentència per a l'usuari "admin" amb clau "password" és:

```
INSERT INTO "Users" VALUES (null, "admin",
   "874d0b1i41c13d9b457a0975a0323i13i711b8c8", "Administradors");
```

Com ja hem dit, CakePHP utilitza SHA1 per a fer un *hash* de la contrasenya. A més d'això, per a afegir-hi una mica més de seguretat, empra un *salt* que trobarem a `conf/core.php`. Aquest *salt* s'utilitza perquè, si la nostra base de dades és compromesa però el nostre codi no, a l'atacant li sigui més complicat obtenir les claus per força bruta.

Com que CakePHP gestiona tot aquest procés automàticament, això no té més importància tret que vulguem compartir la nostra taula d'usuaris entre diversos sistemes. Més endavant, quan parlem de l'autenticació en *Zend Framework*, veurem la utilitat d'això.

També se'ns dóna la possibilitat de canviar l'algorisme de *hashing* utilitzat mitjançant la crida a `Security::setHash($hashType)`. Entre els algorismes que hi són compatibles, hi ha *SHA1*, *SHA256* i *MD5*.



Vista de visites per a un usuari

En l'estat actual de l'aplicació, la vista que ens mostra totes les signatures que hi han deixat els usuaris ens sol·licita autenticació. Si bé aquest és el comportament que volem per a les accions d'afegir i eliminar, volem que l'acció de *consulta* es pugui fer sense autenticació. Per a permetre-ho, simplement hem d'afegir en el mètode `UserController::beforeFilter()` del nostre *Controller* una crida al mètode *allow* del component *Auth*. Aquest mètode controla l'*Access Control Object* del *Controller* i ens permet alterar-lo dinàmicament. L'utilitzarem de la manera següent per a permetre directament l'accés a veure les nostres visites:

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

El mètode *allow* ens deixa especificar, a més d'accions individuals, una matriu o *array* d'accions si volem que no siguin autenticades diverses d'aquestes accions, o fins i tot especificar "*" si volem que no hi hagi cap acció del nostre *Controller* que requereixi que l'usuari estigui autenticat prèviament.

Amb això ja hem recorregut les possibilitats bàsiques d'autenticació de CakePHP. Com veiem, tota la utilització és molt intuïtiva i senzilla, cosa que permet que amb pocs canvis tinguem funcionant la nostra aplicació.

2.2.2. Autorització en CakePHP

L'autorització en CakePHP es basa a utilitzar dos conceptes bàsics:

- Access Control Object (ACO). Són els recursos als quals volem permetre o denegar l'accés depenent de l'usuari.

- Access Request Object (ARO). Són els objectes als quals volem restringir l'accés sobre els ACO.

Podem entendre els ARO com a informació per a un context d'autorització sobre els nostres usuaris. La utilització d'ARO ens estalvia haver d'emmagatzemar, a la taula d'usuaris, informació que no està relacionada directament amb aquesta entitat, sinó que té a veure amb l'autorització. A més, els ARO ens permeten establir jerarquies en els usuaris d'una manera molt més potent i flexible que el que comportaria implementar la mateixa funcionalitat a la taula d'usuaris.

Per tant, podem crear un ARO associat a cadascun dels nostres usuaris i que aquest ARO tingui un determinat pare. Més endavant, podrem assignar els ACO als pares en lloc de fer-ho als fills, cosa que proporciona un control molt més potent. Si volem, podrem continuar assignant ACO directament als ARO d'usuari per a un control més granular.

Com ja vam fer per a `AuthComponent`, afegir autorització requereix que incloguem el component de control de llistes d'accés `AclComponent` al nostre *Controller*. Per a fer-ho, hi afegim l'indicador següent:

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

Amb això, ja podem emprar els nostres dos components en el nostre *Controller*. Gairebé estem a punt per a començar a utilitzar l'autorització.

CakePHP ens proporciona dues maneres d'emmagatzemar les llistes d'accés: en base de dades o en fitxers de configuració. Ens centrarem en la primera perquè és la més flexible.

Per a utilitzar la nostra base de dades per a emmagatzemar ACO i ARO, necessitem crear primer l'esquema o *schema*. Encara que ho podem fer manualment, en aquest cas recorrerem a l'intèrpret d'ordres de Cake o *shell cake*, que ho farà per nosaltres. Per a portar-ho a terme, executem l'ordre següent:

```
cake schema run create DbAcl
```

En executar-la, ens avisa en primer lloc que s'eliminaran les definicions i dades que hi ha. Una vegada acceptat, un missatge ens informa que es crearan les noves definicions de les taules. Després d'això, s'han generat tres noves taules:

- **acos**. Conté l'arbre d'ACO.
- **aros**. Com l'anterior, però per a l'arbre d'ARO.
- **aros_acos**. Conté el mapatge entre ARO i ACO.

Per a la nostra aplicació, les accions o ACO a les quals volem controlar l'accés són bàsicament dues:

- Afegir comentaris. Permetre afegir visites al nostre llibre. Aquest permís és comú a tots els usuaris autenticats.
- Eliminar comentaris. Permetre eliminar visites. Aquest permís només l'obtenen els usuaris administradors.

És hora de començar a assignar permisos. Per a fer-ho, tornem a utilitzar l'intèrpret d'ordres de CakePHP, encara que, aquesta vegada, podem optar per accedir directament al model d'ACO i ARO utilitzant el component *Acl*. Veurem un petit exemple de com fer-ho de les dues maneres, per a centrar-nos després en l'intèrpret d'ordres perquè és més pràctic.

El nostre arbre d'usuaris és molt senzill. Disposem de dos grups a l'arrel, anomenats *Usuaris* i *Administradors*. Només permetrem eliminar comentaris als usuaris que es trobin en el segon grup. Per a crear aquests dos grups, executarem les següents ordres de l'intèrpret d'ordres de CakePHP:

```
cake acl create aro / Usuaris
cake acl create aro / Administradors
```

Això indica que es creen dos ARO directament a l'arrel "/", cosa que ens permet establir una estructura jeràrquica per als nostres ARO. Els podem crear executant un petit *script* en una de les accions de la nostra aplicació utilitzant el codi següent:

```
function crearArosAction()
{
   $groups = array(
      0 => array(
         'alias' => 'Usuaris'
      ),
      1 => array(
         'alias' => 'Administradors'
      )
   );
   foreach($groups as $group)
   {
      $this->Acl->Aro->create();
      $this->Acl->Aro->save($group);
   }
}
```

Ara ja podem incloure en aquests grups algun dels usuaris que ja tenim. Afegim el nostre usuari "usuari" al grup "Usuaris" i l'usuari administrador al grup "Administradors", mostrant un complet menyspreu per l'originalitat. Per a fer-ho, utilitzem les ordres següents:

```
cake acl create aro Usuaris User.1
cake acl create aro Administradors User.2
```

Això fa que l'usuari identificat com a "1" dins del model "User" pertanyi al grup "Usuaris". Passa el mateix per a l'usuari "2", en aquest cas al grup "Administradors". Més endavant, utilitzarem la mateixa nomenclatura "User." més l'identificador de l'usuari per a referir-nos a un usuari del nostre model.

Vegem l'estructura d'usuaris que acabem de crear utilitzant l'ordre:

```
cake acl view aro
Aro tree:
---------------------------------------------------------------
   [1]Usuaris
      [3]
   [2]Administradors
      [4]
---------------------------------------------------------------
```

El número entre claudàtors és l'identificador de l'ACO que haurem d'utilitzar per a esborrar-los. El nom que hi ha a continuació és l'àlies i no és obligatori, sempre que l'ACO es refereixi a un model existent com en el cas dels usuaris. El mateix aplicarem als grups si, en lloc de manejar-los com a cadenes o *strings*, haguéssim triat de tractar-los com un model *Group*. No obstant això, no cal fer-ho i obtindrem el mateix efecte sense un model que doni suport al nostre model.

A diferència de l'autenticació, que únicament utilitza un component per a proporcionar el servei, les ACL hi involucren no solament un component, sinó també un comportament.

A continuació crearem els ACO, que representaran les accions que volem protegir. L'organització dels ACO és idèntica a la que hem vist per als ARO, de manera que, intuïtivament, els podrem començar a crear.

També tenim la possibilitat de crear jerarquies, cosa que, en cas de tenir una estructura de recursos complexa, ens permet facilitar després la creació de llistes.

En el nostre cas, l'estructura és senzilla i no fa falta establir una jerarquia; no obstant això, a l'efecte de demostrar-ne el funcionament, en crearem una de molt simple:

- accions
  - crear
  - eliminar

Els "Usuaris" tindran dret de crear, mentre que els "Administradors" tindran accés a totes les accions i no a cadascuna individualment. Encara que aquí l'avantatge de fer l'assignació és mínim, hem de tenir en compte que aquestes llistes d'accés poden escalar perquè tenen centenars d'accions individuals, cosa que fa immanejable mantenir-les cadascuna individualment.

Executarem, per tant, les ordres següents a l'intèrpret d'ordres de Cake per obtenir aquesta estructura:

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

Finalment, ens falta establir la relació entre els recursos que volem protegir (ACO) i els rols que volem utilitzar (ARO).

```
cake acl grant Usuaris crear all
cake acl grant Administradors accions all
```

Amb això, hem atorgat el permís crear a tots els ACO que es troben sota "Usuaris", mentre que permetem totes les accions als "Administradors". L'últim paràmetre és l'acció que volem permetre. Se sol utilitzar per a especificar l'acció concreta sobre un recurs. Les accions ens permeten definir l'operació que protegim sobre un recurs. Encara que nosaltres hem utilitzat directament el nom de l'ACO per a fer-ho, podríem haver usat una estructura ben diferent com, per exemple, utilitzant el nom d'ACO "visites" i les accions crear i eliminar com a accions.

En general, en aplicacions més complexes, una organització relativament lògica és utilitzar el controlador com a nom de l'ACO, i l'acció com a nom de recurs que cal autoritzar.

Igual que hem utilitzat *grant*, podem usar *deny* amb el significat oposat per a denegar una acció explícitament. Si no ho hem permès abans, està denegada; no obstant això, resulta útil per a quan volem controlar amb detall l'accés a un recurs. Per exemple, permetre a tots els Administradors totes les accions, però revocar a un en concret la capacitat d'afegir noves visites.

Ha arribat el moment de verificar si el nostre usuari pot accedir a les dades. El format d'una comprovació d'una ACL és el següent:

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

En el nostre cas, el paràmetre *$aco* es correspon amb l'acció que executem, sia crear o eliminar. Com a *$aro* utilitzarem l'identificador d'usuari i l'acció és "*". La taula de comprovacions queda com segueix:

ARO	ACO	Action	Permès?
User.1	crear	*	Sí (heretat d'Usuaris)
User.1	eliminar	*	No
User.1	accions	*	No
User.2	crear	*	Sí (heretat d'Administradors)
User.2	eliminar	*	Sí (heretat d'Administradors)
User.2	accions	*	Sí (heretat d'Administradors)
Usuaris	crear	*	Sí
Usuaris	eliminar	*	No
Usuaris	accions	*	No
Administradors	crear	*	Sí (heretat d'accions)
Administradors	eliminar	*	Sí (heretat d'accions)
Administradors	accions	*	Sí

Combinacions per a l'exemple d'autorització

Com veiem, el fet de permetre una acció específica en propaga el valor a tots els fills. Això ens dóna flexibilitat i facilita administrar els permisos quan tenim molts ARO i ACO.

Afegirem, per tant, les condicions necessàries en les nostres accions, utilitzant el codi següent:

```
if (!$this->Acl->check($this->Auth->user('groupname'),"crear","*")) {
   $this->Session->setFlash('No teniu prou permisos per a fer aquesta acció.');
   $this->redirect(array('action' => 'view'));
}
```

Aquest és tot el codi que necessitarem per a autoritzar les nostres accions. El mètode `Acl::check()` retorna *cert* si el grup de l'usuari té permisos per a fer una acció. Aquest acostament, però, utilitzant l'àlies de l'ACO, té la limitació que no podem eliminar permisos a un usuari concret, ja que, únicament, comprovem si el seu grup té permisos o no.

A l'hora de solucionar això, encara que els usuaris no tinguin un àlies, poden ser referenciats perquè la funció *check* els utilitzi. Per a fer-ho, en lloc d'utilitzar l'àlies, hem d'utilitzar una matriu i indicar a quin model pertany l'ACO i quina clau té en aquest model. Amb aquest petit canvi, el nostre codi queda així:

```
$aco = array(
   'model' => 'User',
   'foreign_key' => $this->Auth->user('id')
);
if (!$this->Acl->check($aco),"eliminar","*")) {
   $this->Session->setFlash('No teniu prou permisos per a fer aquesta acció.');
   $this->redirect(array('action' => 'view'));
}
```

En aquest cas, obtenim tots els avantatges del maneig jeràrquic d'ACO i, alhora, la possibilitat d'eliminar accions per a un usuari concret.

2.3. Autorització i autenticació en Zend Framework

Zend Framework, d'ara endavant *ZF*, és un altre dels coneguts entorns de treball per a PHP. En molts aspectes és semblant a CakePHP, cosa que ens permet associar molts dels conceptes que hem vist abans als nous utilitzats en Zend.

Com CakePHP, ZF permet desenvolupar aplicacions web seguint un patró MVC, encara que utilitzant Zend no estem obligats a seguir-lo, com passava amb CakePHP. Això, per al cas que ens ocupa, implica que podem utilitzar les classes d'autorització i autenticació de Zend sense que la nostra aplicació hagi de tenir un controlador, un model i una vista.

Ens tornarem a basar en la nostra petita aplicació de visites per a seguir el mateix procés que hem fet abans amb CakePHP. El nostre objectiu és implementar la mateixa funcionalitat en aquest entorn de treball.

Aquesta vegada, ZF no s'encarrega automàticament de generar per nosaltres el model, de manera que hem de crear unes quantes classes que ens permetin accedir a les visites emmagatzemades a la base de dades.

Necessitem tres classes diferents per a gestionar el model. La primera, a la qual anomenem `Default_Model_Visites`, és el model pròpiament dit una vegada es troba a la memòria, sia abans de ser serialitzat o una vegada recuperat de la base de dades.

La segona classe representa la nostra taula visites d'*SQLite*. Aquesta classe, anomenada `Default_Model_DbTable_Visites`, conté simplement un atribut *$name* que indica el nom de la taula.

Finalment, el *Mapper* fa la unió entre els dos mons. El *Mapper*, que en el nostre cas es diu `Default_Model_VisitesMapper`, s'encarrega de serialitzar i recuperar de la taula física els nostres objectes model. No hi dedicarem més estona, a això, ja que el nostre model és molt senzill; per a obtenir més detalls sobre aquest tipus d'implementació estàndard, aneu a la documentació de ZF.

El controlador de "Visites" en ZF ha de ser una classe que hereti de `Zend_Controller_Action` i que ha de tenir un mètode amb el nom `actionnameAction`. Per exemple, en el nostre cas té tres mètodes diferents: `addAction`, `viewAction` i `removeAction`. Com hem vist abans, en aquesta classe se centra principalment tot el control d'accés i l'autorització de la nostra aplicació.

2.3.1. Autenticació en Zend Framework

Ara centralitzarem les funcionalitats bàsiques d'autenticació, és a dir, farem *login* i *logout* en un *Controller* com hem fet en el cas de CakePHP. Aquest *Controller*, anomenat *UsersController*, disposa de dues accions que representen les dues funcions esmentades abans i que duran a terme el procés.

D'altra banda, tots els nostres *Controller* han d'implementar la verificació de la identitat de l'usuari abans de permetre-li continuar endavant. Quan no hagi estat autenticat i l'acció ho requereixi, redirigirem l'usuari a l'acció de *login* del nostre `UsersController`.

Com que la comprovació de si l'usuari està autenticat o no és comuna a totes les nostres accions, la implementarem en el mètode `preDispatch()` del nostre `VisitaController`. Aquest mètode és cridat abans de l'execució de l'acció i ens permet decidir si volem continuar el flux d'execució o redirigir-lo cap a un altre `Controller` i `Action`. Això ens farà molt servei, ja que si comprovem que l'usuari no està autenticat el redirigirem a l'acció *login* del nostre controlador Users. El codi quedarà com segueix:

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

```
    }

    if (!$this->_auth->hasIdentity()) {
       $this->view->message = 'Autentiqui's abans de continuar, si us plau.';
       $this->_forward('login', 'users');
    }
}
```

No ens hem de descuidar d'inicialitzar la variable *$_auth* en el nostre mètode *init*:

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

En el nostre mètode *preDispatch()*, el primer que fem és comprovar l'acció que executa l'usuari. Si l'acció és "view", no hem de verificar que estigui autenticat prèviament, ja que permetem que qualsevol pugui veure el nostre llibre de visites.

En el cas de qualsevol de les altres dues accions, obtenim una instància de *Zend_Auth* i utilitzem el mètode *Zend_Auth::hasIdentity()*, que ens indica si l'usuari ha estat autenticat o no. Si no ho ha estat, el redirigim cap a la pàgina d'inici de sessió. La classe *Zend_Auth* és un *singleton* que serveix d'entrada per a tota la funcionalitat d'autenticació de ZF. A més de comprovar si l'usuari està autenticat, podem obtenir la identitat mitjançant el mètode *Zend_Auth::getIdentity()* o fins i tot eliminar-la, cosa que ens serà útil en l'acció *logout* del nostre *UsersController*, mitjançant el mètode *Zend_Auth::clearIdentity()*.

Una vegada afegit el codi anterior en el nostre mètode *preDispatch()*, les nostres accions *add* i *remove* ja estan protegides i no són disponibles per a usuaris no autenticats. Ara, per tant, crearem el nostre *UsersController* que s'encarregarà de l'autenticació de l'usuari.

ZF proporciona un sistema d'autenticació molt flexible gràcies als diferents adaptadors que proveeix. Cada adaptador permet que l'usuari sigui autenticat d'una manera o utilitzant un mecanisme diferent.

Vegem-ne alguns exemples:

Classe	Descripció
Zend_Auth_Adapter_DbTable	Permet autenticar els usuaris contra una taula en una base de dades.
Zend_Auth_Adapter_Digest	Autentica els usuaris utilitzant l'autenticació *HTTP digest*.
Zend_Auth_Adapter_Infocard	Permet utilitzar InfoCard com a credencial en l'autenticació.

Adaptadors d'autenticació en Zend Framework

Classe	Descripció
Zend_Auth_Adapter_Ldap	Ens permet autenticar l'usuari utilitzant per a fer-ho un directori *LDAP*.
Zend_Auth_Adapter_OpenID	L'autenticació es fa utilitzant credencials *OpenID*.

Adaptadors d'autenticació en Zend Framework

Per descomptat, podem implementar el nostre mecanisme heretant de la interfície `Zend_Auth_Adapter_Interface`, implementant el mètode `authenticate()` d'aquesta interfície i retornant un `Zend_Auth_Result` d'acord amb el resultat de l'autenticació.

Nosaltres ens centrarem en el `Zend_Auth_Adapter_DbTable`, amb el qual podem accedir a la nostra taula *user* d'*SQLite*. Per tant, hem d'inicialitzar una instància d'aquest adaptador i configurar les dades necessàries, com el nom de la taula i el dels camps que contenen el nom d'usuari i la nostra contrasenya. El `Zend_Auth_Adapter_DbTable` rep tots aquests paràmetres en el seu 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'
);
```

El primer que hem fet ha estat crear un adaptador de base de dades. Com amb els d'autenticació, ZF ens proporciona diferents adaptadors que permeten que les nostres dades estiguin guardades en diferents tipus de repositori; aquí ens centrarem en el `Zend_Db_Adapter_Pdo_Sqlite`. Aquest adaptador és el que utilitzarem per a accedir a la nostra base de dades *SQLite*. L'inicialitzarem simplement passant una matriu amb les opcions, entre les quals, el fitxer on hi ha la nostra base de dades i un segon paràmetre *sqlite2* que indica que té un format *SQLite* v2.

Aquest adaptador de base de dades és el primer paràmetre que rebrà el nostre adaptador d'autenticació. El segon paràmetre indica el nom de la taula en la qual s'emmagatzemen les credencials, en el nostre cas `'users'`. El tercer i el quart indiquen, respectivament, l'identificador de la columna que conté els noms d'usuari i la clau.

Hi ha un cinquè paràmetre que no utilitzarem en aquest moment, ja que *SQLite* no ens en deixa treure partit. Aquest últim paràmetre permet especificar modificacions que volem fer a la contrasenya abans que sigui comprovada. Això és útil quan les contrasenyes no estan emmagatzemades, sinó que s'utilitza algun algorisme de *hashing*. Aquest paràmetre ens especifica la transformació, utilitzant el símbol (?) com a substitut de la contrasenya.

Per exemple, per a utilitzar un *hash MD5* per a la contrasenya, la nostra creació de l'adaptador queda d'aquesta manera:

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

Podem entendre aquest últim paràmetre com l'última clàusula de la sentència SQL que s'executarà en comprovar les credencials de l'usuari. Això permet que fem algunes verificacions addicionals amb molta comoditat. Per exemple:

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

Que comprovaria que el compte de l'usuari no ha caducat. Perquè ho faci, per descomptat, hem d'afegir el camp `expiryDate` a l'esquema de la nostra base de dades, però és una cosa relativament senzilla.

SQLite no pot utilitzar funcions com MD5 o SHA1 en les seves consultes o *queries*, cosa que provoca que la capacitat de modificar la consulta en el *Zend_Auth_Adapter_DbTable* no ens sigui útil. Hem de tenir en compte, a més, que utilitzem les contrasenyes generades per CakePHP que inclouen un *salt* que està emmagatzemat a *conf/core.php*. Per a utilitzar la mateixa taula, hem de concatenar la nostra contrasenya amb aquest *salt*.

Una vegada configurat l'adaptador d'autenticació, solament ens falta establir quines credencials volem comprovar mitjançant *Zend_Auth_Adapter_DbTable::setIdentity()* i *Zend_Auth_Adapter_DbTable::setCredential()*.

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

La classe *Zend_Auth* és l'encarregada de fer l'autenticació pròpiament dita una vegada li hem proporcionat l'accés a les dades que requereix. Passem com a paràmetre l'adaptador que necessitem segons el tipus d'autenticació que utilitzem, en el nostre cas, la base de dades:

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

Una vegada ha estat autenticat l'usuari, hem de comprovar sempre si aquesta autenticació ha estat satisfactòria per mitjà del mètode *Zend_Auth_Result::isValid()*.

La nostra acció *login* queda, per tant, com segueix:

```
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('usuari'))
               ->setCredential(sha1('3447a33b473cdde3f399c8dc3cd2ea573ecc4868'.
      $form->getValue('password')));
            $auth= Zend_Auth::getInstance();
            $result=$auth->authenticate($authAdapter);
            if ($result->isValid()) {
                     return $this->_helper->redirector('view','visites');
            }
```

```
            }
        $this->view->message = 'Usuari/contrasenya incorrecte. Torneu-ho a provar.';
    }
    $this->view->form = $form;
}
```

Perquè el nostre codi d'autenticació funcioni correctament amb les dades que ja hi ha de CakePHP, hem de fer el *hash* SHA1 del *salt*, que hem obtingut de la configuració de la nostra aplicació CakePHP, juntament amb la contrasenya.

Després de comprovar que la contrasenya és correcta, tan sols hem de redirigir l'usuari a la pàgina principal.

2.3.2. Autorització en Zend Framework

ZF segueix un model semblant al que hem vist en CakePHP i que trobem en molts altres entorns de treball dins i fora de PHP. Per tant, tornem a centrar-nos a protegir l'accés a recursos, tant si són accions com si són controladors o arxius, utilitzant una sèrie de rols als quals assignem dret d'accés.

En el cas de ZF, trobarem que els ACO es corresponen amb els *Rols*, mentre que els ARO es corresponen amb els *Resources*.

Fins ara, a més de l'autenticació, tenim una autorització molt bàsica, ja que qualsevol usuari que no estigui autoritzat no pot portar a terme les accions d'afegir o eliminar visites. No obstant això, com hem vist abans, necessitem un control més granular dels usuaris autoritzats, perquè tots tenen accés als recursos.

En el nostre cas, com ja hem vist, el recurs que hem optat per protegir en el nostre controlador és la vista. Autoritzarem, per tant, amb diferents privilegis, els *Rols* que creeem a ZF sobre les vistes.

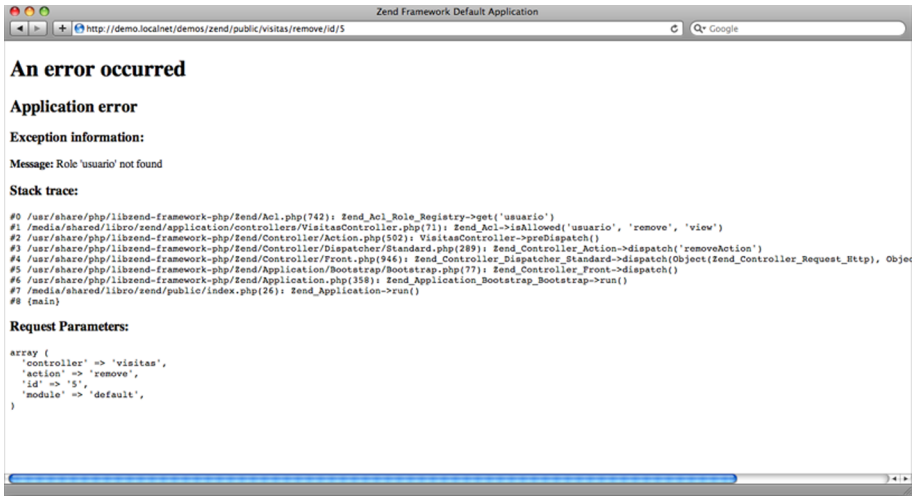
Si l'usuari no està autoritzat a fer una determinada acció, el redirigim amb un missatge que indiqui l'error.

El codi que cal afegir és, per tant, el següent:

```
$resource = $this->_actionsNames[$request->getActionName()];
$role = $this->_auth->getIdentity();
if(!$this->_acl->isAllowed($role, $resource, null)) {
    $this->view->message = 'No esteu autoritzats a fer l'acció.';
    $this->_forward('login', 'users', array('continue' => $request->getActionName()));
}
```

La matriu `_actionsNames` conté únicament el mapatge entre els noms de les accions i el seu nom com a *Resource* (add > crear, remove > eliminar).

Si intentem accedir ara, en la nostra aplicació, a alguna acció que no sigui *view*, veurem que es produeix un error:



Excepció usuari no autoritzat

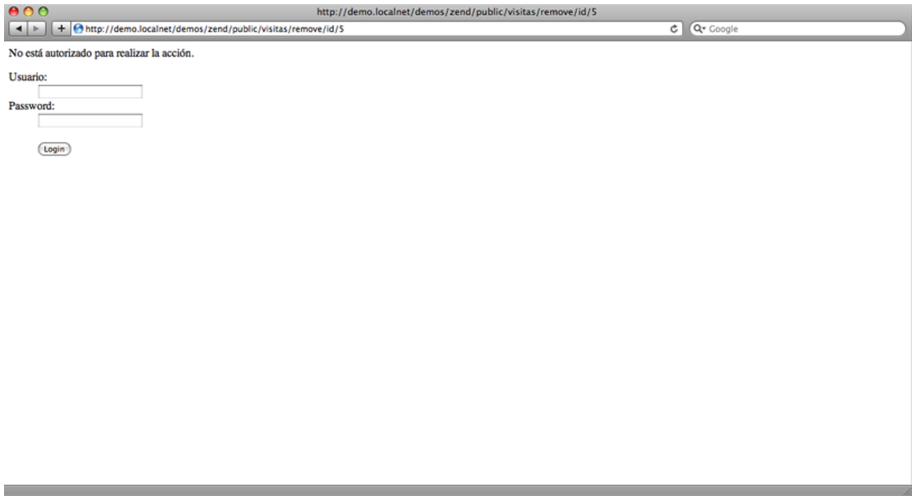
L'error és perquè *Zend* requereix que els *Rols* i *Resources* hagin estat afegits prèviament per a fer l'autorització. Això no és gaire pràctic, però més endavant veurem com solucionar-ho. De moment, posarem un exemple de la manera en què funciona la creació dinàmica de *Rols* i *Resources*, cosa que ens permetrà veure el nostre exemple funcionant. En aquest cas, hem d'afegir, just a sota la creació de l'objecte `Zend_Acl` en el mètode `init` del nostre `VisitesController`, la inicialització següent:

```
$this->_acl->addRole("Usuaris");
$this->_acl->addRole("usuari", "Usuaris");
$this->_acl->addResource("accions");
$this->_acl->addResource("eliminar", "accions");
$this->_acl->addResource("crear", "accions");
```

Això crea dos *Rols*: un d'anomenat *"Usuaris"*, que fa de grup, i un d'anomenat *"Administrador"*. El nostre "Usuari" hereta els permisos que atorguem al pare "Usuaris". La crida `addRole` rep, com a primer paràmetre, una cadena amb el nom del *Rol* o un objecte del tipus `Zend_Acl_Role_Interface`. Aquesta interfície, simplement, ens permet encapsular la funcionalitat d'obtenir el *Rol* en altres objectes com, per exemple, el model.

De manera anàloga, tenim la funció `Zend_Acl::addResource()` que ens permet afegir els recursos.

Si ara, després d'autenticar-nos com a "usuari", intentem accedir a una de les accions, com ara afegir o eliminar, obtenim un missatge diferent:

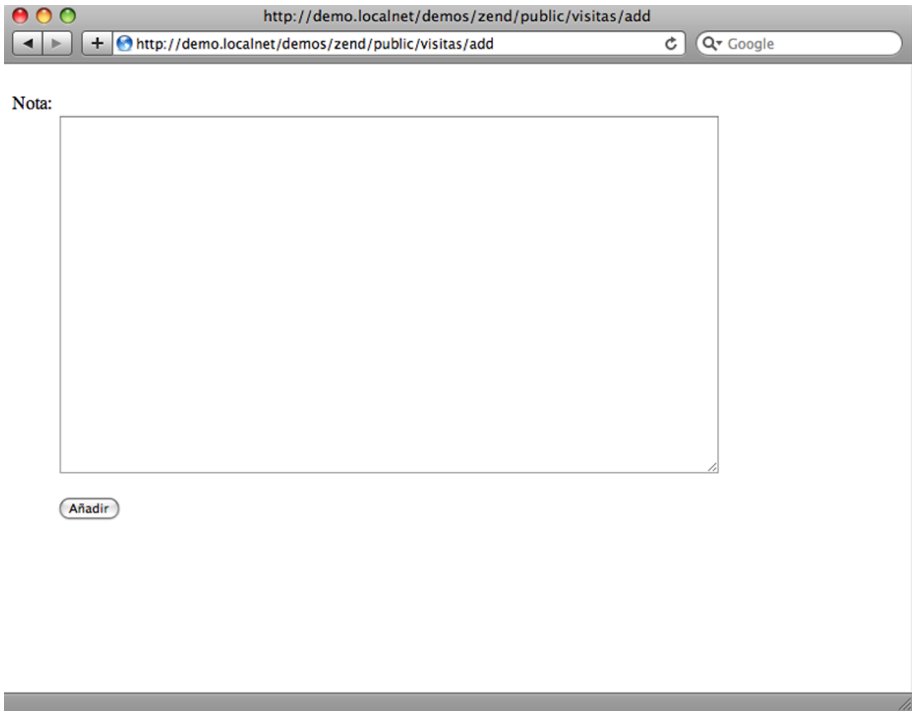


Pantalla d'inici de sessió en ZF

Ara només ens falta autoritzar els usuaris a fer una de les accions. En el nostre cas, permetrem a tots els "Usuaris" que accedeixin a l'acció d'afegir. Per a fer-ho, utilitzem el mètode *Zend_Acl::allow()*:

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

Si aquesta vegada intentem eliminar una visita, rebrem el mateix missatge que abans, ja que el nostre usuari no està autoritzat. No obstant això, comprovem que podem afegir noves visites.



Afegir una visita

Tant el mètode *Zend_Acl::allow()* com *Zend_Acl::isAllowed()* poden rebre, a més, un tercer paràmetre anomenat *priviledge*. Aquest paràmetre, generalment, representa l'acció que autoritzem a l'objecte ("*read*", "*write*", "*delete*", etc). Nosaltres, no obstant això, hem optat per utilitzar les accions com

a recursos per a protegir-les. Aquesta decisió és de disseny i prou ja que els recursos ens permeten utilitzar herència, mentre que els privilegis no. Utilitzar les accions com a privilegi és un canvi trivial en el codi.

Finalment, el mètode `Zend_Acl::allow()` pot rebre, com a últim paràmetre, un objecte que implementi la interfície `Zend_Acl_Assert_Interface`. Les ACL, encara que les puguem comprovar i fins i tot afegir de manera dinàmica, tenen una sèrie de limitacions.

> Implementar comprovacions que, a més de l'accés al recurs, depenguin de dades dinàmiques com la data o que vinguin en la petició com la IP ens obligaria a afegir una altra capa d'autenticació diferent. No obstant això, implementant aquesta interfície, podem crear condicions que es comprovin en el moment de la verificació de credencials.

Podem utilitzar aquestes condicions en combinació amb l'autorització dels *Rols* i *Resources* o, fins i tot, de manera independent, passant simplement *null* en els tres primers paràmetres.

Si bé és senzill, aquest mètode de mantenir les llistes és poc pràctic, encara més quan abans hem estat utilitzant les llistes d'accés des de la base de dades. El fet d'inicialitzar les vistes, com hem fet fins ara, requereix afegir o eliminar codi segons els diferents *Rols* que proporcionem.

No hi ha un mecanisme per defecte per a serialitzar les ACL del ZF. Per això, no se'ns proporcionen les facilitats que teníem en CakePHP per a emmagatzemar les nostres ACL a la base de dades. Això no vol dir que no es pugui fer; de fet, la classe `Zend_Acl` és serialitzable, cosa que en permet l'emmagatzematge.

Nosaltres optarem per utilitzar una classe `Zend_Acl` modificada lleugerament, a la qual anomenarem *DbAcl*, que afegirà suport dels ACO i ARO que hem definit per a la nostra aplicació en CakePHP.

En primer lloc, ens hem de fixar en l'estructura de les taules que l'ordre `cake schema run create DbAcl` genera. Les taules d'ACO i ARO comparteixen un model comú:

Camp	Descripció
id	Identificador de l'ACO/ARO.
parent_id	Identificador del pare de l'ACO/ARO o *NULL* si es troba a l'arrel.
model	Si l'ACO/ARO té alguna representació a la base de dades, a més del mateix ACO/ARO, aquest camp indicarà el model.
foreign_key	Si l'ACO/ARO té alguna representació a la base de dades, a més del mateix ACO/ARO, aquest camp indicarà la clau de l'objecte que representa en el model.
alias	Nom de l'ACO/ARO. Si l'ACO/ARO està relacionat amb un model, no hi fa falta.

Esquema de les taules aco i aro

Camp	Descripció
lft	Anterior ACO/ARO a la llista.
rght	Següent ACO/ARO a la llista.

Esquema de les taules aco i aro

La taula *aros_acos* s'encarrega de relacionar els dos models, i és la que emmagatzema les llistes d'accés. El seu esquema és el següent:

Camp	Descripció
id	Identificador de l'ACL.
aro_id	Identificador de l'ARO o *Rol* que disposa del permís.
aco_id	Identificador de l'ACO o *Resource* a què s'atorga el permís.

Esquema de la taula *aro_acos*

Cridem a la nostra classe `DbAcl`, que accedirà a l'estructura representada a les taules acos i aros, i *aros_acos*, que recull el nom i jerarquia dels *Rols* i *Resources* que hem creat utilitzant l'intèrpret d'ordres de CakePHP. Aquesta classe, en la seva inicialització, recorre la taula i, utilitzant els mètodes *`Zend_Acl::addRole()`* i *`Zend_Acl::add()`* que hem vist abans, recrea en memòria la llista d'accés que hi havia emmagatzemada. No solament afegim els *Rols* i *Resources* que correspongui, sinó que també recreem l'estructura jeràrquica. A més, quan l'ACO està associat a un *User*, utilitzem directament el nom del *user* com a identificador del *Rol*.

L'avantatge d'utilitzar aquesta classe és no solament que no necessitem incloure en el codi la inicialització de les nostres llistes d'accés, sinó que, a més, podem utilitzar l'intèrpret d'ordres de CakePHP per a manipular-les com hem fet abans.

```
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()
   {
```

```
        $rols = $this->_db->fetchAll(
           $this->_db->select()
              ->from('aros')
              ->order(array('id ASC')));
        foreach ($rols 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'];
              /* Afegir àlies */
              $this->add(
                 new Zend_Acl_Resource($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']]);
              }
```

```
            }
        }
    }
```

Quan la classe `DbAcl` és inicialitzada, el mètode `DbAcl::roleResources()` s'encarrega de carregar els *Rols* i *Resources* de les taules i crear l'estructura en memòria. Per a utilitzar-la, únicament hem de reemplaçar l'ús que hem fet de `Zend_Acl` per la nostra nova classe `DbAcl`. El funcionament d'aquesta classe és completament transparent a la resta de codi.

Ara ja podem comprovar el funcionament de la nostra aplicació usant, exactament, els mateixos usuaris i llistes d'autorització que hem utilitzat abans amb CakePHP.

3. Autorització i autenticació en aplicacions .NET

A l'hora de desenvolupar una aplicació, una de les parts més crítiques, i per tant, una de les més importants, és la relacionada amb l'autenticació i autorització d'usuaris. Dit d'una altra manera, el desenvolupament del mecanisme que utilitzaran les aplicacions a l'hora de gestionar els usuaris i els recursos a què tindrà accés cada usuari o grup d'usuaris.

Mitjançant el procés d'autenticació, es determina la identitat d'un usuari. Després d'identificar-lo, mitjançant el procés d'autorització, es determina a quins recursos pot accedir aquest usuari. Amb aquest procés, és possible personalitzar les aplicacions basant-se en els tipus o preferències dels usuaris.

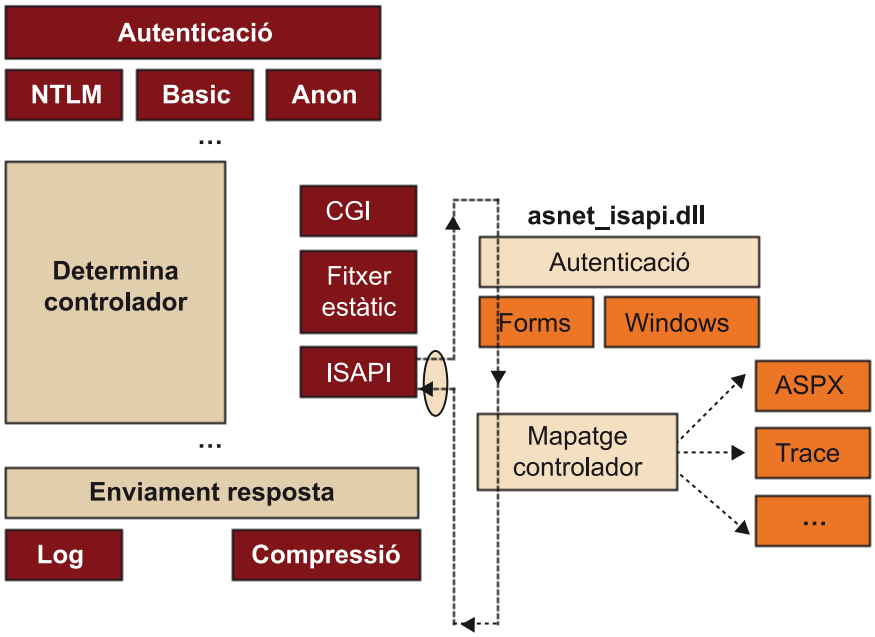
Al llarg d'aquest capítol, es presenten els mecanismes que s'incorporen en la plataforma .NET Framework, des de la versió 2.0, que permeten a un desenvolupador crear aplicacions integrant-hi mecanismes d'autenticació i autorització d'usuaris.

3.1. Autenticació en ASP.NET

La plataforma ASP.NET proporciona diferents tipus d'autenticació d'usuaris: l'autenticació bàsica, l'autenticació de text implícita, l'autenticació de formularis, l'autenticació Passport i l'autenticació de Windows Integrada. De manera complementària, és possible proporcionar un mecanisme propi per a l'autenticació d'usuaris.

Com que ASP.NET s'executa amb el servidor d'aplicacions Internet Information Server (IIS), cal comprendre com s'hi integra ASP.NET, per a entendre la configuració necessària que s'ha d'aplicar per a una configuració correcta del mecanisme d'autenticació.

En les versions anteriors a IIS7, el codi ASP.NET s'executa amb un filtre ISAPI; per tant, en una aplicació ASP.NET es té un doble procés d'autenticació: d'una banda, el que fa el mateix servidor IIS, i de l'altra, el que fa l'aplicació ASP.NET, com es mostra en la imatge següent:

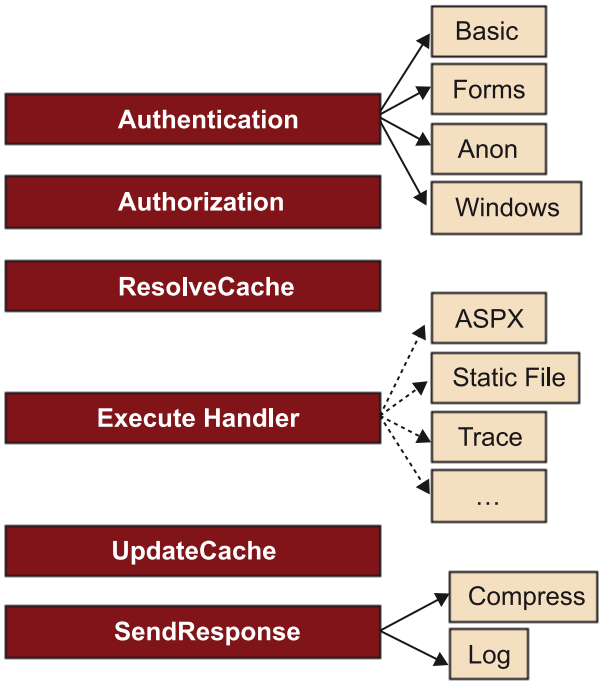


Autenticació en IIS6 i ASP.NET

En aquesta imatge s'aprecia que, en fer una petició a una aplicació ASP.NET, en primer lloc es porta a terme el procés d'autenticació del servidor d'aplicacions i, en segon lloc, una vegada superada aquesta autenticació, es passa la petició al filtre ISAPI, que executa el codi ASP.NET, el qual implementa el seu propi mecanisme d'autenticació.

Si es vol configurar una autenticació mitjançant formularis a ASP.NET, cal configurar l'autenticació anònima en IIS, per a cedir tot el mecanisme d'autenticació a ASP.NET. Si, en canvi, es vol una autenticació de Windows en l'aplicació ASP.NET, cal configurar l'IIS amb l'autenticació volguda, ja que és aquest IIS mateix qui s'encarregarà del mecanisme d'autenticació.

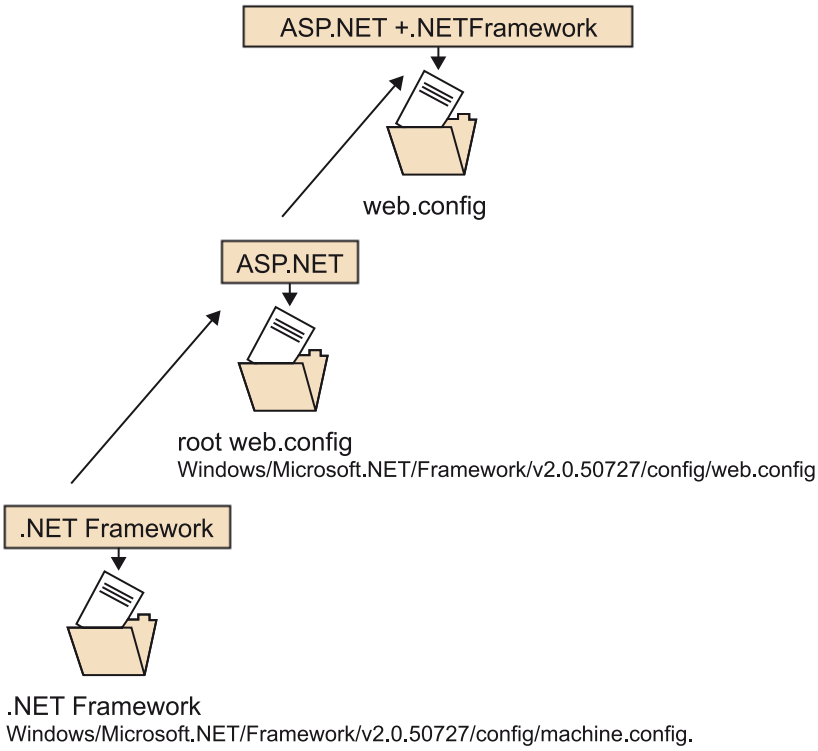
En canvi, amb el servidor d'aplicacions IIS7, treballant amb el *pipeline* en mode administrat (en mode clàssic treballa de manera semblant a IIS6), l'autenticació d'ASP.NET s'integra al nivell del servidor d'aplicacions, i no hi ha, en cap cas, un doble procés d'autenticació, com es veu en la imatge següent:



Autenticació en IIS7 i ASP.NET

A la figura, s'aprecia la manera en què, durant l'execució d'una petició al servidor web, el procés d'autenticació es porta a terme una única vegada segons el procés d'autenticació triat.

Per a configurar el mètode d'autenticació en una aplicació ASP.NET, s'utilitzen els fitxers de configuració *.config*. Aquests fitxers són documents XML que s'organitzen jeràrquicament, de manera que els fitxers en el servidor especifiquen les opcions generals de configuració, les quals se sobreescriuen en els fitxers de configuració a escala de lloc web, directori virtual o carpeta.



Herència en els fitxers de configuració

La configuració de cada lloc web s'especifica en el fitxer *web.config* del lloc, sense modificar, excepte casos molt puntuals, els fitxers en la màquina *machine.config* i *web.config*. En aquest fitxer de configuració, mitjançant l'element *authentication* es configura l'esquema d'autenticació que ha d'utilitzar l'aplicació ASP.NET.

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

Per mitjà de l'atribut *mode* de l'element *authentication* s'especifiquen els possibles valors que cal utilitzar: Windows, Forms, Passport i None. El significat d'aquests valors s'indica en la taula següent:

Valor	Descripció
Windows	Especifica l'autenticació de Windows com a mode d'autenticació predeterminat. S'ha d'usar amb qualsevol forma d'autenticació de serveis de Microsoft Internet Information Server (IIS): bàsica, implícita, integrada de Windows (NTLM o Kerberos) o certificats. En aquest cas, la seva aplicació delega la responsabilitat de l'autenticació al servidor IIS subjacent.
Forms	Especifica l'autenticació ASP.NET basada en formularis com a mode d'autenticació predeterminat.
Passport	Especifica l'autenticació de xarxa de Microsoft Passport com a mode d'autenticació predeterminat.

Mètodes d'autenticació en ASP.NET

Valor	Descripció
None	No especifica cap autenticació. L'aplicació espera només usuaris anònims o proporciona la seva pròpia autenticació.

Mètodes d'autenticació en ASP.NET

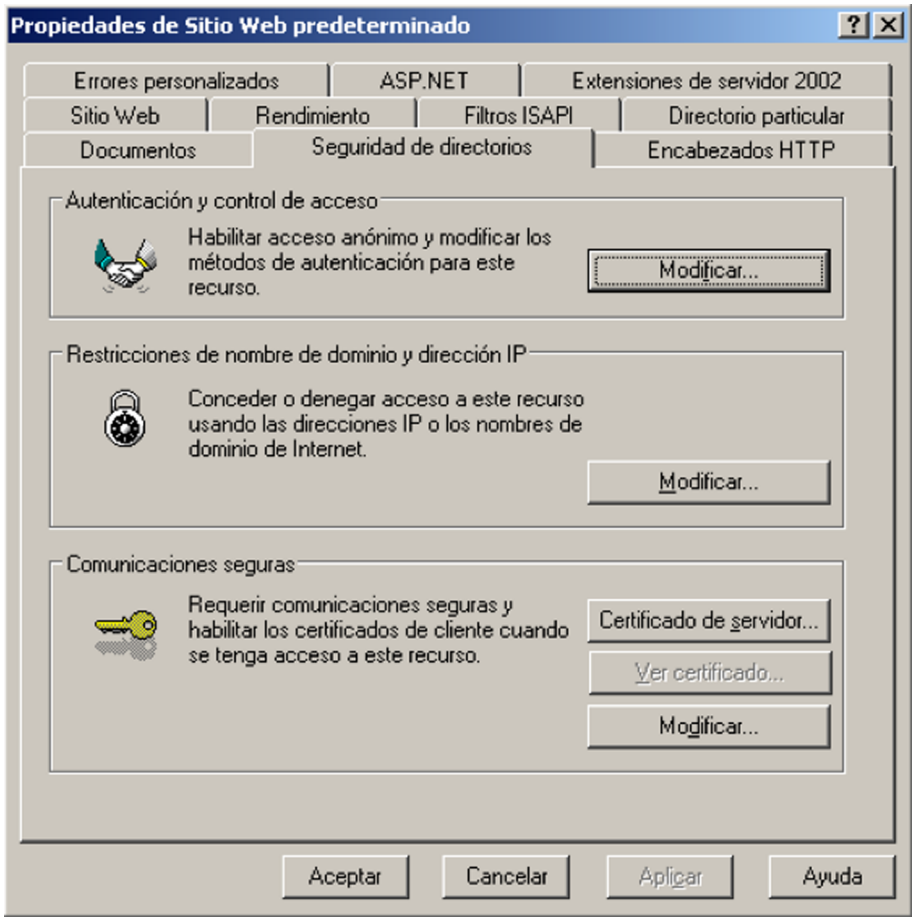
El valor per defecte que s'indica en els fitxers de configuració en màquina és *Windows*. En els apartats següents, s'especifica com s'ha de treballar amb els diferents mecanismes d'autenticació que són compatibles amb ASP.NET.

3.1.1. Autenticació basada en Windows

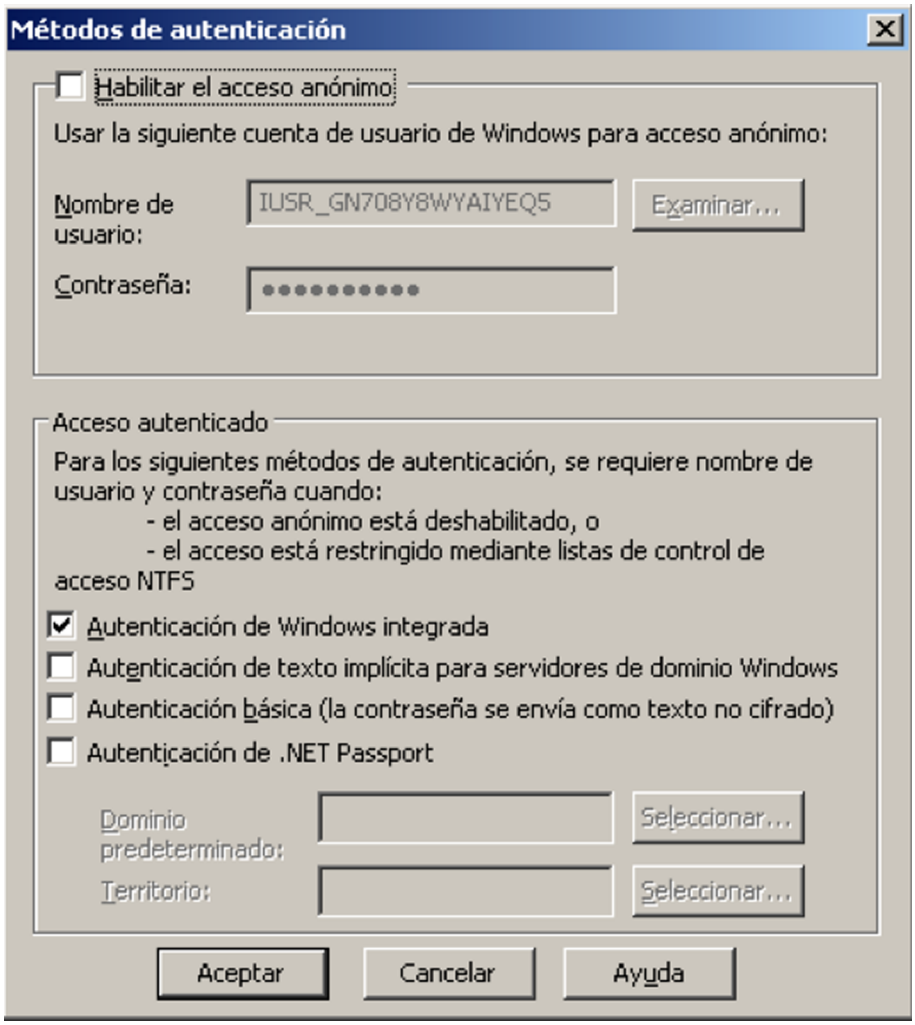
L'esquema d'autenticació basat en Windows és l'utilitzat per defecte en les aplicacions ASP.NET. Utilitzant aquest esquema, l'aplicació ASP.NET delega el procés d'autenticació al servidor Internet Information Server subjacent.

Per a treballar així cal especificar en el node *authentication* del fitxer *web.config* el mode Windows, i a continuació especificar en la consola d'administració del servidor Internet Information Server la manera d'autenticar que s'ha de seguir.

Per a configurar el mètode d'autenticació en IIS6, cal anar a l'administrador d'Internet Information Service a les *Eines administratives* de l'equip, desplegar el node de l'equip i, a la carpeta *Llocs web*, fer clic amb el botó dret del ratolí sobre el lloc web que s'ha d'administrar i seleccionar el menú Propietats. A la finestra, cal seleccionar la pestanya *Seguretat de directoris* i prémer el botó *Modificar* dins de la secció *Autenticació i control i accés*, com s'observa en les imatges següents:

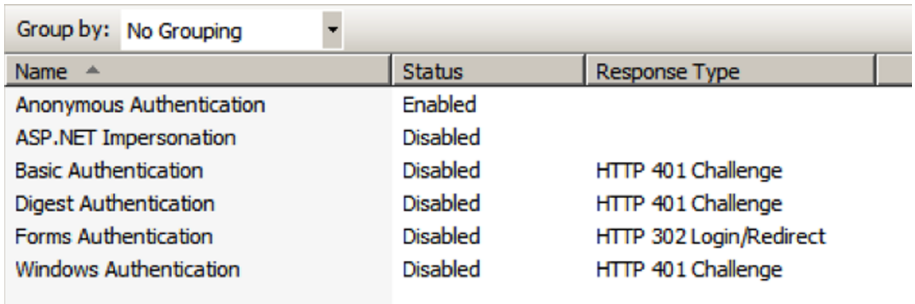


Propietats d'un lloc web en IIS6



Mètodes d'autenticació en IIS6

En IIS7, la consola d'administració d'Internet Information Server és diferent, però de la mateixa manera, per mitjà de la icona d'autenticació, situada al panell d'administració d'un lloc web, és possible configurar l'esquema d'autenticació que s'ha de seguir, com es veu en la imatge següent:



Autenticació en IIS7

En tots dos casos, és possible triar entre quatre tipus d'autenticació (deixant de banda les maneres d'autenticar de formularis i .NET Passport, que les veurem més endavant), que són els següents: accés anònim, autenticació de Windows integrada, autenticació de text i autenticació bàsica. En la taula següent, s'especifiquen aquests tipus d'autenticació:

Manera	Descripció
Anònima	En aquesta manera, no es fa cap tasca d'autenticació, sinó que IIS proporciona l'usuari que té configurat, i les credencials d'aquest usuari són les utilitzades per a accedir als diferents recursos. En IIS6, si es vol utilitzar autenticació de formularis en ASP.NET, cal configurar amb autenticació anònima el servidor d'aplicacions, per a delegar a ASP.NET el procés d'autenticació.
Bàsica	En aquesta manera, IIS implementa l'autenticació bàsica especificada en HTTP 1.0 utilitzant els comptes d'usuaris de Windows. Les credencials es transmeten codificades en base64, de manera que és molt fàcil descodificar-les, i això fa que sigui un mètode insegur. A l'hora de protegir aquest mètode d'autenticació, cal utilitzar-lo juntament amb SSL/TLS (Secure Socket Layer / Transport Layer Security) per a xifrar la connexió HTTP (HTTPS). A més, és imprescindible que tots els usuaris tinguin un compte Windows donat d'alta en el servidor.
Implícita	Corregeix les principals deficiències de l'autenticació bàsica, ja que es crea un procés de desafiament-resposta. En contrapartida, requereix un directori actiu on hi hagi donats d'alta els usuaris i només funciona amb el navegador Internet Explorer.
Windows integrada	És el millor esquema d'autenticació per a utilitzar en una intranet en què tots els usuaris treballen amb Internet Explorer i tenen un usuari del domini. Com en la implícita, no s'envien les credencials per la Xarxa. A diferència de l'autenticació implícita, amb aquest esquema es permet la delegació de credencials.

Maneres d'autenticació en IIS

Com s'observa, aquests mètodes d'autenticació són molt vàlids per a desenvolupar intranets corporatives, en què es controlen els navegadors client i tots els usuaris estan donats d'alta en un directori actiu. En combinació amb altres característiques, com la personificació, que veurem més endavant, és possible crear un entorn SSO en què, quan un usuari inicia sessió en el seu equip mitjançant el domini, és possible accedir a la intranet i els recursos que té sense tornar-li a sol·licitar credencials.

3.1.2. Autenticació basada en formularis

En l'autenticació basada en formularis, és ASP.NET qui s'encarrega de gestionar els usuaris i els recursos de l'aplicació a què tenen accés aquests usuaris. En utilitzar aquest mecanisme, l'aplicació està proveïda d'un formulari d'inici de sessió en què s'introdueixen les credencials d'accés per a accedir a l'aplicació web i els recursos disponibles.

Cal destacar que, si no s'utilitza SSL, les dades d'usuari i la clau s'envien sense xifrar per la Xarxa, cosa que implica un inconvenient de seguretat.

Com s'ha indicat abans, per a configurar una aplicació ASP.NET amb seguretat basada en formularis cal seleccionar aquest tipus d'autenticació a la consola d'administració d'IIS7, i, en cas de treballar amb IIS6, seleccionar l'accés anònim com a mètode d'autenticació.

Una vegada configurat adequadament el servidor IIS, el fitxer *web.config* de l'aplicació web ASP.NET ha d'especificar el mode d'autenticació *Forms*, com s'observa en el següent fragment de codi.

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

Com s'aprecia en el fragment de codi anterior, l'atribut *mode* de l'element *authentication* pren el valor *Forms*; apareix un nou element *forms* en què es configuren diferents propietats de l'esquema d'autenticació, fent ús de diversos atributs. A continuació, s'especifiquen els possibles atributs que cal utilitzar, els possibles valors i significat que tenen.

<table>
<tr><th>Atribut</th><th>Descripció</th></tr>
<tr><td>Cookieless</td><td>Defineix si s'utilitzen galetes i el comportament que tenen. El valor predeterminat és UseDeviceProfile.

Aquest atribut pot tenir un dels valors següents:

<table>
<tr><th>Atribut</th><th>Descripció</th></tr>
<tr><td>UseCookies</td><td>S'utilitzen sempre galetes, independentment del dispositiu.</td></tr>
<tr><td>UseUriEspecifica</td><td>No s'utilitzen mai galetes.</td></tr>
<tr><td>AutoDetect</td><td>S'utilitzen galetes si el perfil del dispositiu n'admet; si no n'admet, no s'utilitzen.</td></tr>
<tr><td>UseDeviceProfile</td><td>S'utilitzen galetes si l'explorador n'admet; si no n'admet, no s'utilitzen. En el cas dels dispositius que admeten galetes, no s'intenta determinar si està habilitat l'ús de galetes.</td></tr>
</table></td></tr>
<tr><td>defaultUrl</td><td>Defineix l'adreça URL per defecte que s'utilitza per al redireccionament després de l'autenticació. El valor predeterminat és "default.aspx".</td></tr>
<tr><td>Domain</td><td>Especifica un domini opcional que s'estableix en les galetes d'autenticació dels formularis sortints. El valor predeterminat és una cadena buida ("").</td></tr>
<tr><td>enableCrossAppRedirects</td><td>Indica si els usuaris autenticats es redirigeixen a adreces URL en altres aplicacions web. El valor predeterminat és "false".</td></tr>
<tr><td>loginUrl</td><td>Especifica l'adreça URL a la qual s'ha de redirigir la sol·licitud d'inici de sessió si no es troba cap galeta d'autenticació vàlida. El valor predeterminat és "login.aspx".</td></tr>
<tr><td>Name</td><td>Especifica la galeta HTTP utilitzada per a l'autenticació. Si s'executen diverses aplicacions en un mateix servidor i cadascuna requereix una galeta única, s'ha de configurar el nom de la galeta en cada arxiu Web.config de cada aplicació. El valor predeterminat és ".ASPXAUTH".</td></tr>
<tr><td>Path</td><td>Especifica la ruta d'accés de les galetes emeses per l'aplicació. El valor predeterminat és una barra inclinada (/).</td></tr>
<tr><td>Protection</td><td>Si s'ha utilitzat algun tipus de xifratge, especifica el que s'ha fet servir per a les galetes. El valor predeterminat és "All".

Aquest atribut pot tenir un dels valors següents:

<table>
<tr><th>Atribut</th><th>Descripció</th></tr>
<tr><td>All</td><td>L'aplicació utilitza validació de dades i xifratge per a ajudar a protegir la galeta.</td></tr>
<tr><td>None</td><td>Tant el xifratge com la validació estan deshabilitats.</td></tr>
<tr><td>Encryption</td><td>Les galetes es xifren amb 3DES o DES, però no es validen les dades en la galeta. Les galetes utilitzades d'aquesta manera poden tenir atacs en el text sense format.</td></tr>
<tr><td>Validation</td><td>Es porta a terme la validació de dades, però la galeta no es xifra.</td></tr>
</table></td></tr>
<tr><td>requireSSL</td><td>Especifica si es requereix una connexió SSL per a transmetre la galeta d'autenticació. El valor predeterminat és "false".</td></tr>
<tr><td>slidingExpiration</td><td>Especifica si el termini de caducitat està habilitat. Aquest termini restableix el temps de què disposa una galeta d'autenticació activa fins que caduca en cada sol·licitud feta durant una sessió. El valor predeterminat és "true".</td></tr>
<tr><td>timeout</td><td>Especifica l'interval de temps, en minuts sencers, passat el qual la galeta caduca. Si el valor de l'atribut SlidingExpiration és true, l'atribut timeout és un valor variable. El valor predeterminat és "30" (30 minuts).</td></tr>
</table>

Atributs de l'element *forms* del fitxer *web.config*

Una vegada configurada l'aplicació ASP.NET per a utilitzar l'autenticació basada en formularis, cal determinar contra quina informació es validaran els usuaris. A ASP.NET és possible validar els usuaris en el mateix fitxer *web.config* o utilitzar un repositori extern, com un servidor de bases de dades, un arbre LDAP o fitxers XML. Per a aquests casos, l'opció més bona és recórrer al model de proveïdors d'ASP.NET.

Cal indicar aquí la manera en què el model que segueix ASP.Net per a autenticar usuaris mitjançant repositoris externs, com són les bases de dades o directoris, implica l'existència d'una única cadena de connexió per a connectar-se al repositori on hi ha emmagatzemats els usuaris i la informació que hi està associada.

Un altre possible model d'autenticació és aquell en què la cadena de connexió al repositori es construeix d'acord amb l'usuari i la clau que ha introduït l'usuari de l'aplicació, de manera que l'autenticació es basa en el mecanisme d'autenticació del repositori. Per exemple, en una base de dades SQL Server, caldria donar d'alta com a usuaris SQL a tots els usuaris del portal.

Aquest últim mecanisme d'autenticació l'utilitzen les aplicacions encarregades de gestionar els mateixos motors de bases de dades i pateixen de vulnerabilitats CSPP (Connection String Parameter Polution), en què injectant sobre l'usuari i la contrasenya es manipula la cadena de connexió que ha creat el programador, i es pot arribar a saltar el procés d'autenticació.

Autenticació mitjançant usuaris del fitxer *web.config*

En aplicacions petites, en què el nombre d'usuaris que cal gestionar és molt limitat, és possible donar d'alta els usuaris en el mateix fitxer *web.config*, de manera que no cal recórrer a un repositori extern ni implementar mecanismes propis.

Per a utilitzar aquesta característica, cal fer ús del subelement *credentials* de l'element *forms*. Mitjançant aquest element, s'indica el format de les contrasenyes que s'utilitzaran, i mitjançant els subelements *user* s'afegeixen usuaris a l'aplicació web, com es mostra en el quadre següent:

```
<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>
```

Els possibles valors que pren l'atribut *passwordFormat* de l'element *credentials* i que indiquen el format de xifratge per a emmagatzemar les contrasenyes són els següents:

Valor	Descripció
Clear	Especifica que les contrasenyes no es xifren.
MD5	Especifica que les contrasenyes es xifren amb l'algorisme *hash* MD5.

Possibles valors de l'atribut *passwordFormat* de l'element *credentials*

Valor	Descripció
SHA1	Especifica que les contrasenyes es xifren amb l'algorisme *hash* SHA1. Aquest és el valor per defecte.

Possibles valors de l'atribut *passwordFormat* de l'element *credentials*

Una vegada definida la forma en què s'han de xifrar les contrasenyes dels usuaris, es poden afegir nous usuaris al fitxer *web.config*; per a fer-ho, i com s'ha mostrat, s'utilitza l'element *user* i s'indica el nom d'usuari amb l'atribut *name* i la clau, xifrada correctament, amb el camp *password*.

A continuació es mostra un petit exemple, amb una aplicació web amb un fitxer *Default.aspx* al qual no s'ha de poder accedir, fins que l'usuari iniciï la sessió mitjançant el formulari web disponible a la pàgina *Login.aspx*. Es mostra també la configuració del fitxer *web.config*.

El fitxer *Default.aspx* consisteix, simplement, en una pàgina de benvinguda 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>Benvingut</title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
      Benvingut al portal web
   </div>
   </form>
</body>
</html>
```

En el fitxer *Login.aspx* es crea un formulari simple en què l'usuari ha d'introduir les seves credencials d'accés 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>Inici de sessió</title>
</head>
<body>
   <form id="Form1" runat="server">
   <div>
```

```
      Usuari: <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>
```

Al fitxer *Login.aspx.cs* s'hi introdueix el codi necessari per a l'autenticació:

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

En aquest codi es veu com s'utilitza la classe *FormsAuthentication*, que s'encarrega d'administrar els serveis d'autenticació de formularis en aplicacions ASP.NET. El mètode *Authenticate* s'usa per a validar el nom d'usuari i contrasenya passats per mitjà del formulari web contra les credencials emmagatzemades al fitxer *web.config*, mentre que el mètode *RedirectFromLoginPage* redirigeix l'usuari autenticat cap a l'adreça URL sol·licitada originalment o l'adreça URL predeterminada. El segon argument del mètode indica si s'ha de crear una galeta o *cookie* permanent.

El fitxer de configuració *web.config* ha de tenir un aspecte com el següent, ometent la resta de configuracions contingudes en la secció *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>
```

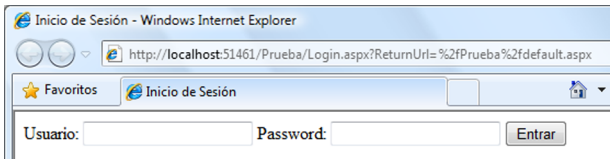
S'ha donat d'alta un usuari indicant el nom "User1" i la contrasenya xifrada "pass1". Les contrasenyes s'emmagatzemen utilitzant les funcions de *hash* MD5 o SHA1, que no són reversibles i, per tant, fins i tot per al programador no és possible recuperar la contrasenya original.

Com que MD5 és una funció de *hash* que presenta moltes més col·lisions que SHA1, és a dir, és més fàcil trobar contrasenyes que generin el mateix *hash* en MD5 que en SHA1, es recomana utilitzar aquest últim com a sistema de xifratge per a emmagatzemar les contrasenyes.

El canvi de sistema de xifratge només afecta el fitxer *web.config*, i no cal modificar el codi d'autenticació, tant si s'utilitza l'emmagatzematge xifrat de claus com si no. El programador pot emmagatzemar claus xifrades (per donar d'alta usuaris) fent ús del mètode HashPasswordForStoringInConfigFile, de la classe *FormsAuthentication*, el qual permet generar una contrasenya amb l'algorisme *hash* apropiat per a emmagatzemar en un fitxer de configuració i hi passa la contrasenya i algorisme *hash* que cal utilitzar (MD5 o SHA1).

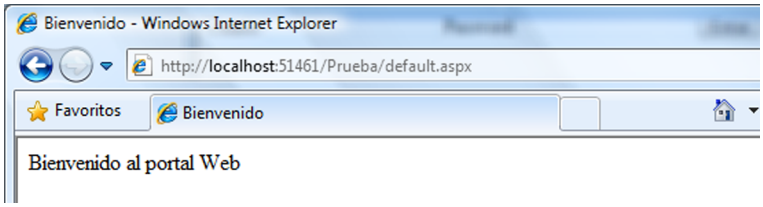
Al final del fitxer de configuració hi ha l'element *authorization*; s'introdueix aquí per a especificar que solament poden accedir a les pàgines del portal els usuaris autenticats. Aquest element es veu detalladament en l'apartat "Autorització en ASP.NET".

En la imatge següent s'observa que, en intentar accedir a la pàgina *Default.aspx*, ASP.NET verifica si l'usuari té la galeta que l'identifica com a usuari autenticat en l'aplicació; com que no és així, redirigeix l'usuari a la pàgina d'inici de sessió, passant-li el paràmetre *ReturnUrl*, el valor del qual és la pàgina aspx a la qual es redirigirà l'usuari si té èxit en el procés d'autenticació.



Accés al lloc web amb autenticació mitjançant formularis

Quan l'usuari introdueix les credencials d'accés, la funció *Authenticate* codifica la contrasenya amb la funció *hash* SHA1 i la compara amb l'emmagatzemada en el fitxer *web.config* per a l'usuari passat. Si l'usuari existeix, i la contrasenya coincideix, s'autentica l'usuari redirigint-lo a la pàgina de benvinguda, com es pot observar en la imatge següent:



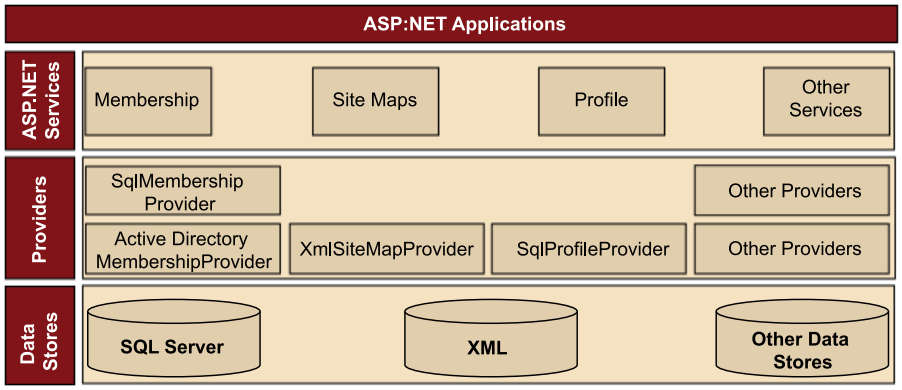
Usuari autenticat mitjançant formularis

Mitjançant aquestes configuracions és possible crear portals amb autenticació mitjançant formularis, que requereixin un conjunt reduït d'usuaris. Quan el portal web es complica i es requereix una gestió més complexa d'usuaris, basats en rols, s'ha de recórrer a un repositori extern, fent ús del model de proveïdors d'ASP.Net, que s'explica en l'apartat següent.

Model de proveïdors

Amb ASP.Net 2.0 va néixer el model de proveïdors; gràcies a aquest model, ASP.Net inclou una sèrie de serveis que emmagatzemen informació en bases de dades o altres sistemes d'emmagatzematge. Així, és possible configurar una aplicació ASP.Net perquè el servei encarregat de gestionar usuari i rols utilitzi un SQL Server o el directori Actiu, i el servei encarregat del mapa i la navegació del lloc web utilitzi fitxers XML.

D'aquesta manera, com es veu en la imatge següent, és possible separar el codi d'emmagatzematge del codi de negoci, utilitzant sempre els mateixos components en les aplicacions ASP.Net, però fent ús de diferents proveïdors, o desenvolupant proveïdors personalitzats segons les necessitats:



Model de proveïdors a ASP.Net

Autenticació contra una base de dades SQL Server

Per a la gestió d'usuaris en les aplicacions ASP.NET s'utilitza el proveïdor *MemberShipProvider*. Per defecte, el proveïdor està configurat per a fer ús del gestor de bases de dades SQL Server; aquesta configuració es veu en l'arxiu de configuració a escala de màquina *web.config* que es mostra a continuació:

```
<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 aquest fitxer de configuració, s'observa com s'afegeix a les aplicacions ASP.Net el proveïdor *AspNetSqlMembershipProvider*, que utilitza la cadena de connexió *LocalSqlServer*, que apunta a un fitxer *aspnetdb.mdf* que es crea en el directori AppData de l'aplicació i s'adjunta a una instància d'SQL Server Express.

En la configuració del proveïdor, es fixen característiques com el format d'emmagatzematge de les claus, el nombre de reintents màxims abans de bloquejar el compte d'usuari, o la complexitat de la clau requerida per als usuaris.

Si es vol modificar aquesta configuració en les diferents aplicacions ASP.Net, cal fer-ho en els fitxers *web.config* de cada aplicació; a continuació, es mostra la configuració necessària per a un lloc web que utilitzarà autenticació basada en formularis fent ús d'una base de dades d'un servidor SQL Server.

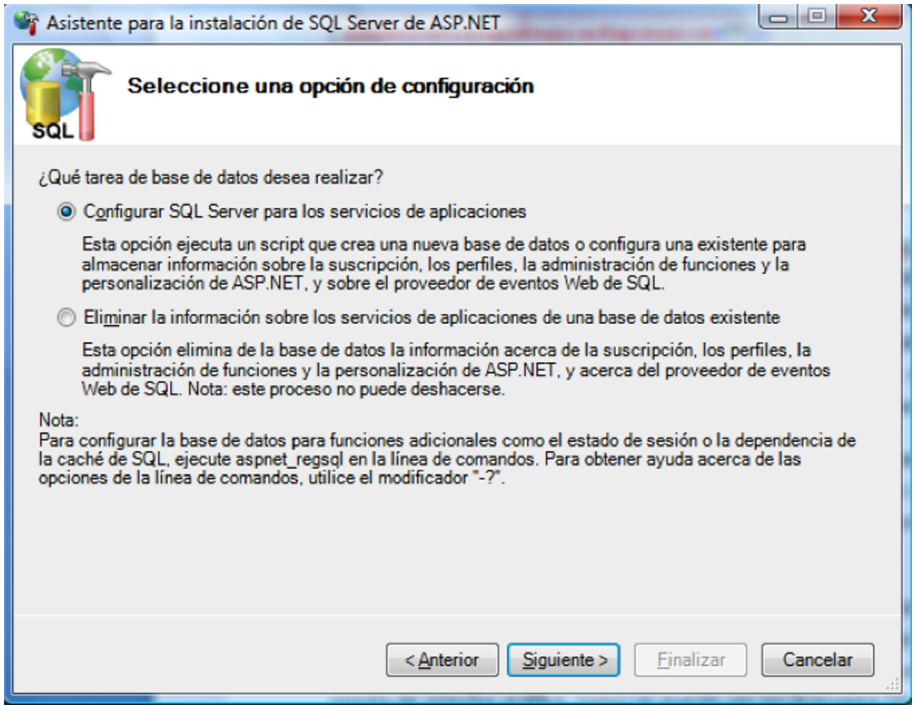
```
<connectionStrings>
   <add name="lamevaCadenaDeConnexio" connectionString="Data Source=SERVIDOR;Initial Catalog=
      WEBDEMO;User ID=sa;Password=123abc." providerName="System.Data.SqlClient"/>
</connectionStrings>
<system.web>
   <authentication mode="Forms">
      <forms loginUrl="Login.aspx" 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="lamevaCadenaDeConnexio" 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>
```

Com s'observa, aquesta aplicació utilitzarà el proveïdor *AspNetSqlMemberShipProvider*; s'han eliminat tots els proveïdors heretats per a donar-hi així el mateix nom per defecte, se n'ha canviat la configuració, de manera que no es requereix pregunta i resposta (l'atribut *requiresQuestionAndAnswer* s'ha col·locat en el valor *false*), i s'utilitza una cadena de connexió diferent, de manera que els usuaris s'emmagatzemaran en la base de dades *WEBDEMO* de l'equip *SERVIDOR* al qual es connectarà per mitjà d'un usuari SQL Server.

Lògicament, el proveïdor *AspNetSqlMembershipProvider* necessita que la base de dades SQL Server que utilitza tingui una estructura de taules, funcions i procediments emmagatzemats específica, la qual es pot generar fent ús de l'eina *aspnet_regsql* que s'inclou en la instal·lació del .NET Framework.

Aquesta eina permet configurar una base de dades SQL Server per a emmagatzemar informació dels diferents serveis proporcionats pels proveïdors d'ASP.NET, i pot emmagatzemar usuaris, rols, perfils, etc. L'eina és molt senzilla de manejar i consta d'interfície gràfica, com es veu en la imatge següent:



Eina *aspnet_regsql* per a configurar bases de dades SQL Server

Una vegada configurat el proveïdor d'usuaris, és possible utilitzar els controls que proporciona ASP.NET per a l'inici de sessió, creació d'usuaris i altres tasques relacionades amb la gestió i autenticació d'usuari, que juntament amb la gestió de rols i l'autorització, que es veuen més endavant, permeten crear portals web segurs d'una manera ràpida i senzilla.

3.1.3. Autenticació contra un arbre LDAP

En un entorn empresarial, és molt comú tenir un directori en què hi ha donats d'alta els diferents usuaris de la corporació. En aquests casos, el lloc web de l'empresa fa ús d'aquests usuaris, i no d'usuaris de bases de dades.

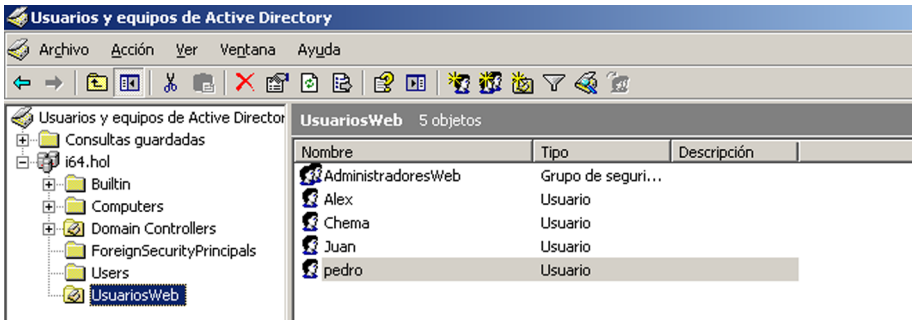
Per a fer-ho, cal configurar l'aplicació web perquè faci ús d'un proveïdor diferent de l'*AspNetSqlMemberShipProvider*; en aquest cas, per a connectar-se al directori actiu de Microsoft es fa ús del proveïdor *AspNetActiveDirectoryMemberShipProvider*, com es veu en el següent fragment d'un fitxer de *web.config* d'una aplicació web.

En aquest cas, es marca el proveïdor *AspNetActiveDirectoryMembershipProvider* com el proveïdor per defecte; es configura de manera que es connecta al directori actiu mitjançant la cadena LDAP *ADConnString* i s'especifica l'usuari i la contrasenya utilitzats per a la connexió i l'atribut del directori que s'utilitza per a mapar usuaris; en el cas del directori actiu, l'atribut utilitzat és *sAMAccountName*.

Tal com s'especifica en la cadena de connexió LDAP, en el directori actiu hi ha una unitat organitzativa anomenada *UsuarisWeb*, en què hi ha donats d'alta els usuaris que accedeixen al portal, com es veu en la imatge:

```
<connectionStrings>
<add
connectionString="LDAP://server.i64.hol/OU=UsuarisWeb,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>
```



Configuració del directori actiu

D'aquesta manera, és possible autenticar usuaris contra un directori sense que el programador hagi d'implementar codi addicional. Com passava en l'autenticació contra usuaris d'SQL Server, és possible utilitzar aquí els controls ASP.NET sense cap inconvenient.

3.1.4. Autenticació contra altres repositoris d'informació

Seguint el model de proveïdors proposat per ASP.NET, és possible utilitzar altres repositoris d'informació per a emmagatzemar usuaris.

> Al portal www.codeproject.com hi ha proveïdors per a treballar amb altres motors de bases de dades com MySQL o SQLite, o per a treballar amb fitxers XML com a repositori d'informació.

Malgrat tots els proveïdors que hi ha, tant els que proporciona Microsoft com els que proporcionen tercers, no són suficients per a cobrir totes les necessitats, sia perquè el repositori d'informació que s'ha d'utilitzar no té un proveïdor que ja està desenvolupat, sia perquè, malgrat utilitzar un repositori amb suport, com SQL Server, ja s'explica amb taules d'usuari i rols que no s'adapten a les necessitats del proveïdor.

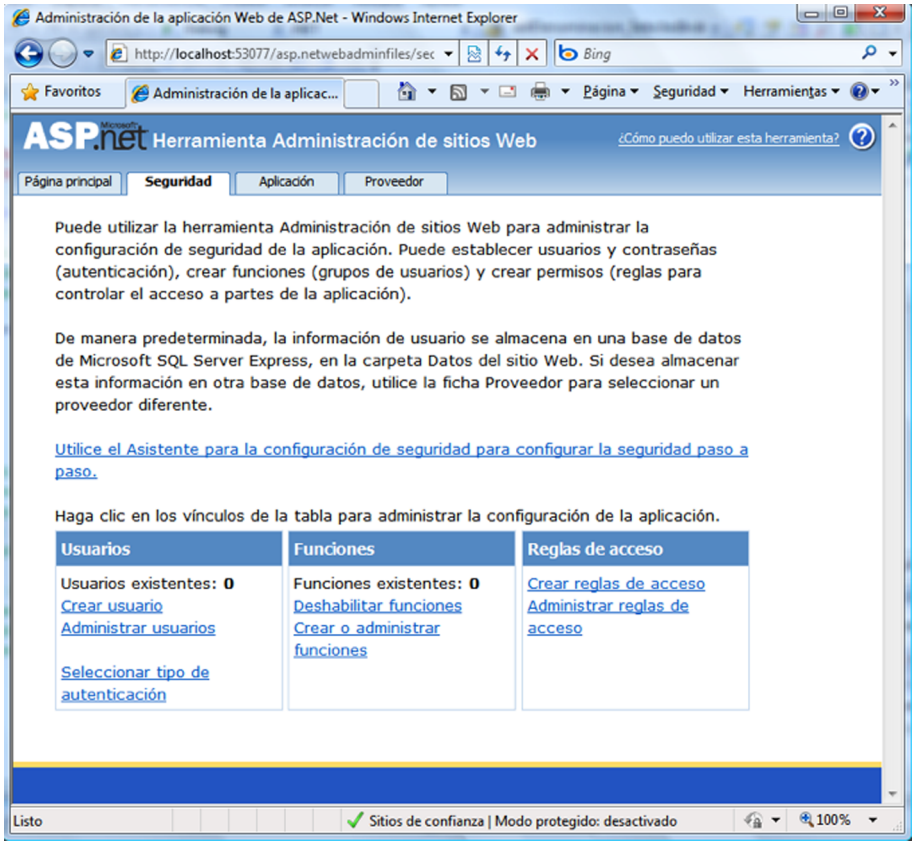
Aquí, és necessari el desenvolupament d'un proveïdor personalitzat, que permeti l'ús dels controls i les característiques que proporciona ASP.NET, però fent ús d'un repositori a mida. Per a això, utilitzant les característiques de la programació orientada a objectes, es desenvolupa una classe que hereti de *MembershipProvider* i proporcioni implementació a tots els mètodes necessaris:

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

      }

   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();
      }
...
```

Com es veu en aquest codi, en crear una classe que hereti de *MembershipProvider*, el programador està obligat a donar implementació a una sèrie de mètodes que són els encarregats de crear un usuari, esborrar un usuari, canviar la contrasenya, etc. D'aquesta manera, indicant en el fitxer *web.config* que s'utilitzarà aquest proveïdor, és possible gestionar i controlar l'autenticació d'usuaris a ASP.NET sempre de la mateixa manera, amb independència del repositori en què s'emmagatzemaran els usuaris.

És possible crear i gestionar els usuaris mitjançant l'eina d'administració de llocs web, que és accessible des de Visual Studio per mitjà de la icona corresponent en l'explorador de solucions. L'eina permet la gestió d'usuari, rols i regles d'autorització des de la pestanya *Seguretat*, com s'observa en la imatge següent:



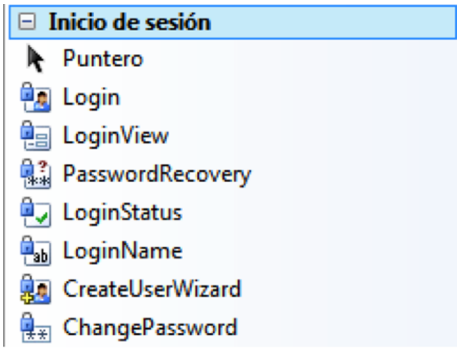
Eina d'administració de llocs web

Mitjançant aquesta eina és possible seleccionar el tipus d'autenticació i el proveïdor d'usuaris i rols, gestionar aquests usuaris i rols, i administrar les regles d'accés. Tot el que es fa gràcies a aquesta eina queda reflectit en el fitxer *web.config* de l'aplicació tal com s'ha mostrat en aquest apartat i els apartats que parlen de rols i autorització.

D'altra banda, ASP.NET proporciona una sèrie de controls perquè els programadors els incloguin en les seves aplicacions. Aquests controls permeten iniciar sessió, crear usuaris, canviar contrasenyes i una altra sèrie d'accions, fent ús del proveïdor d'usuaris que s'utilitzi. Aquests controls s'explicaran de manera resumida en l'apartat següent.

3.1.5. Controls ASP.NET per a la gestió d'usuaris

En utilitzar l'autenticació mitjançant formularis fent ús del model de proveïdors, els desenvolupadors tenen a la seva disposició una sèrie de controls relacionats amb l'inici de sessió i la gestió d'usuaris. Aquests controls són a la barra d'eines de Visual Studio, a la pestanya *Inici de sessió*, com es veu en la imatge següent:



Controls d'inici de sessió

En la taula següent, es resumeixen les funcionalitats de cadascun d'aquests controls:

Control	Descripció
Login	Mostra una interfície d'usuari per a l'autenticació d'usuari. El control Login conté quadres de text per al nom d'usuari i la contrasenya, i una casella de verificació que permet als usuaris indicar si volen que el servidor emmagatzemi la seva identitat utilitzant els proveïdors d'ASP.NET i que els autentiqui automàticament la pròxima vegada que visitin el lloc. El control Login té propietats per a una presentació personalitzada, per a missatges personalitzats i per a vincles a altres pàgines en què els usuaris poden canviar la contrasenya o recuperar-la si l'han oblidada.
LoginView	El control LoginView permet mostrar informació diferent als usuaris anònims i als que han iniciat una sessió. El control mostra una de les dues plantilles: AnonymousTemplate o LoggedInTemplate. En aquestes plantilles, s'hi poden afegir controls que mostrin informació apropiada per a usuaris anònims i usuaris autenticats, respectivament. El control LoginView també inclou esdeveniments per a ViewChanging i ViewChanged, que permeten escriure controladors per a quan l'usuari inicïi una sessió i canviï l'estat.
LoginStatus	El control LoginStatus mostra un vincle d'inici de sessió per als usuaris que no estan autenticats i un vincle de tancament de sessió per als que ho estan. El vincle d'inici de sessió porta l'usuari a una pàgina d'inici de sessió. El vincle de tancament de sessió restableix la identitat de l'usuari actual perquè sigui un usuari anònim. Es pot personalitzar l'aspecte del control LoginStatus establint les propietats LoginText i LoginImageUrl.
LoginName	El control LoginName mostra el nom d'inici de sessió d'un usuari si l'usuari ha iniciat la sessió mitjançant l'autenticació amb formularis d'ASP.NET. De manera alternativa, si el lloc utilitza autenticació de Windows integrada, el control mostra el nom de compte de Windows de l'usuari.
PasswordRecovery	El control PasswordRecovery permet recuperar les contrasenyes de l'usuari basant-se en l'adreça de correu electrònic que es va utilitzar quan es va crear el compte. El control PasswordRecovery envia un missatge de correu electrònic amb la contrasenya a l'usuari. Aquest control requereix que s'hagi configurat un servidor SMTP, i el seu comportament varia segons les propietats establertes en el proveïdor d'usuaris.

Controls d'inici de sessió

Vegeu també

Per a conèixer-ne més detalladament la configuració i ús, visiteu la pàgina d'MSDN a msdn.microsoft.com.

Control	Descripció
CreateUserWizard	El control CreateUserWizard recull informació dels possibles usuaris. De manera predeterminada, el control CreateUserWizard agrega el nou usuari al sistema de proveïdors d'ASP.NET. Per defecte, la informació sol·licitada per a un usuari consisteix en nom d'usuari, contrasenya, confirmació de contrasenya, adreça de correu electrònic, pregunta de seguretat i resposta de seguretat; aquestes dues últimes depenen de la configuració establerta en el proveïdor d'usuaris.
ChangePassword	El control ChangePassword permet als usuaris canviar la contrasenya. L'usuari ha de proporcionar primer la contrasenya original i, a continuació, crear i confirmar la nova contrasenya. Si la contrasenya original és correcta, la contrasenya de l'usuari es canvia a la nova contrasenya. El control també s'encarrega d'enviar un missatge de correu electrònic sobre la nova contrasenya. El control ChangePassword funciona amb usuaris autenticats i no autenticats. Si l'usuari no s'ha autenticat, el control sol·licita a l'usuari que escrigui un nom d'inici de sessió. Si l'usuari s'ha autenticat, el control emplena el quadre de text amb el nom d'inici de sessió de l'usuari.

Controls d'inici de sessió

Mitjançant l'ús d'aquests controls, aplicant la configuració necessària a cadascun, i estenent-los segons les necessitats del portal, el programador pot implementar en els seus llocs les tasques habituals relacionades amb l'inici de sessió i gestió d'usuaris, utilitzant per sota el proveïdor d'usuaris adequat.

3.1.6. Autenticació Passport

Un altre mètode implementat a ASP.NET per a l'autenticació d'usuaris és utilitzar el sistema d'identitat de Passport de Microsoft. Si tots els usuaris tenen un compte Passport és possible desenvolupar una solució d'una sola signatura, cosa que vol dir que amb aquestes credencials poden accedir al lloc web i a tots els llocs i aplicacions d'Internet que tinguin habilitat aquest tipus d'autenticació.

Excepte les aplicacions i els llocs web de Microsoft mateix (Technet, MSDN, Live, etc.), hi ha molt pocs llocs a Internet que tinguin l'autenticació Passport, de manera que la majoria d'aquests llocs han de recórrer a altres solucions d'autenticació delegada.

3.1.7. Gestió de rols mitjançant Proveïdors

En una aplicació ASP.NET és possible activar l'ús de rols, això és, la possibilitat d'agrupar usuaris per les seves funcions, de manera que es puguin crear diferents rols als quals pertanyeran els usuaris. Uns exemples típics de rols són el rol d'administrador o el rol de gestor, de manera que els recursos a què pot accedir un usuari del portal web depenen del rol a què pertanyen.

Per a activar l'ús de rols en una aplicació web, cal activar aquesta característica en el fitxer *web.config*. La gestió de rols es fa mitjançant l'element *roleManager*, indicant el proveïdor de rols que s'utilitzarà. El sistema de rols es pot usar utilitzant tant l'autenticació basada en Windows com l'autenticació basada en formularis, i en cada cas s'ha d'especificar el proveïdor que cal utilitzar.

Com passa amb el proveïdor d'usuaris, en el fitxer de configuració a escala de màquina *machine.config* hi ha donats d'alta, per defecte, el proveïdor *AspNetSqlRoleProvider* per a emmagatzemar rols en una base de dades SQL Server preparada convenientment, com s'indicava en explicar el proveïdor *AspNetMembershipProvider*, i el proveïdor *AspNetWindowsTokenRoleProvider* utilitzat en l'autenticació de Windows, de manera que els rols representen els grups d'usuaris de Windows.

En una aplicació amb autenticació basada en Windows, la configuració del fitxer *web.config* de l'aplicació és la següent:

```
<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 aplicació configurada així delega l'autenticació d'usuaris al servidor web subjacent, i d'altra banda té activat l'ús de rols en l'aplicació, utilitzant el proveïdor *AspNetWindowsTokenRoleProvider*, de manera que els rols dels usuaris són representats pels grups d'usuaris de Windows.

Una aplicació que utilitzi l'autenticació mitjançant formularis també pot treballar amb rols, usant el proveïdor adequat; si utilitza el repositori SQL Server amb el proveïdor d'autenticació *SqlMembershipProvider*, l'aplicació pot utilitzar el proveïdor *AspNetSqlRoleProvider*, configurant-lo adequadament en el fitxer *web.config*, com s'observa en el quadre següent:

```
<connectionStrings>
   <add name="lamevaCadenaDeConnexio" 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="lamevaCadenaDeConnexio"
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="lamevaCadenaDeConnexio"
             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 aquesta configuració, es mostra una aplicació amb autenticació basada en formularis que utilitza el proveïdor *AspNetSqlMembershipProvider* per a la gestió d'usuaris i el proveïdor *AspNetSqlRoleProvider* per a la gestió de rols. En tots dos casos, la informació s'emmagatzema en la mateixa base de dades, que és l'apuntada per la cadena de connexió *lamevaCadenaDeConnexió*, la qual s'ha preparat prèviament amb l'eina *aspnet_regsql*.

Els proveïdors ASP.NET proporcionats per tercers per a altres repositoris, com MySQL, SQLite o fitxers XML, també inclouen suport de rols. Si s'ha desenvolupat un proveïdor d'usuaris propi, s'ha de desenvolupar també el proveïdor per al suport a rols en cas de voler utilitzar aquesta característica.

En aquest cas, cal desenvolupar una classe que estengui a *RoleProvider* i proporcioni la implementació als mètodes necessaris, com ara la creació d'un rol, l'eliminació d'aquest rol o la recerca d'usuari per rol, tal com s'observa en el quadre següent:

```
public class elmeuProveidorDeRols : RoleProvider
```

```
{
   public elmeuProveidorDeRols()
   {
   }

   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();
   }
...
```

Com en la gestió d'usuaris, per mitjà de l'eina d'administració de llocs web és possible gestionar els rols o funcions i assignar-los als diferents usuaris segons les necessitats del lloc web.

3.2. Autorització en ASP.NET

Fins ara s'ha parlat dels diferents mecanismes que proporciona ASP.NET per a autenticar usuaris, sia mitjançant el servidor web subjacent o mitjançant l'autenticació amb formularis fent ús d'usuaris del fitxer *web.config* o fent ús del model de proveïdors.

Mitjançant l'autorització, el que es pretén és controlar l'accés dels usuaris als diferents recursos de l'aplicació web, és a dir, controlar a quines pàgines es pot accedir depenent de l'usuari de què es tracti o del rol a què pertanyi.

L'autorització es configura amb l'element *authorization* del fitxer *web.config* de l'aplicació web. Com que els fitxers de configuració segueixen una estructura jeràrquica, el més habitual és que, en cada carpeta de l'aplicació web, hi hagi un fitxer *web.config* en què s'especifiquen les regles d'autorització per a aquesta carpeta. La sintaxi de l'element *authorization* és la següent:

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

Amb els elements *allow* i *deny* s'afegeixen les regles d'autorització; així, quan un usuari mira d'accedir a una pàgina, el mòdul d'autorització recorre en iteració els elements *allow* i *deny*, començant per l'arxiu de configuració més local fins que troba la primera regla d'accés adequada per al compte específic d'usuari. Després, el mòdul d'autorització concedeix o nega l'accés a un recurs d'adreces URL depenent de si la primera regla d'accés trobada és una regla *allow* o *deny*.

Per defecte, la regla especificada en el fitxer de configuració a escala de màquina *machine.config* és *<allow users="*" />*, de manera que es permet l'accés a tots els recursos URL de l'aplicació.

Si es vol denegar l'accés a tots els usuaris que no estiguin autenticats, s'utilitza la configuració següent:

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

Amb aquesta configuració els usuaris anònims tenen denegat l'accés, mentre que a la resta d'usuaris (usuaris autenticats) se'ls aplica la regla per defecte i, per tant, se'ls concedeix accés al recurs sol·licitat.

És possible configurar regles d'autorització tant per a usuaris com per a rols; així, per exemple, si es vol permetre l'accés als usuaris amb rol *Administrador* i denegar l'accés a la resta d'usuaris, la configuració que s'ha d'aplicar és la següent:

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

En els exemples vistos, es configura l'accés els recursos URL de tota la carpeta on hi ha el fitxer *web.config*; no obstant això, també és possible fer configuracions personalitzades per a un recurs URL concret. Per a fer-ho, cal utilitzar l'element *location*, per a indicar el recurs URL concret a què es vol aplicar la regla d'autorització, tal com es veu a continuació:

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

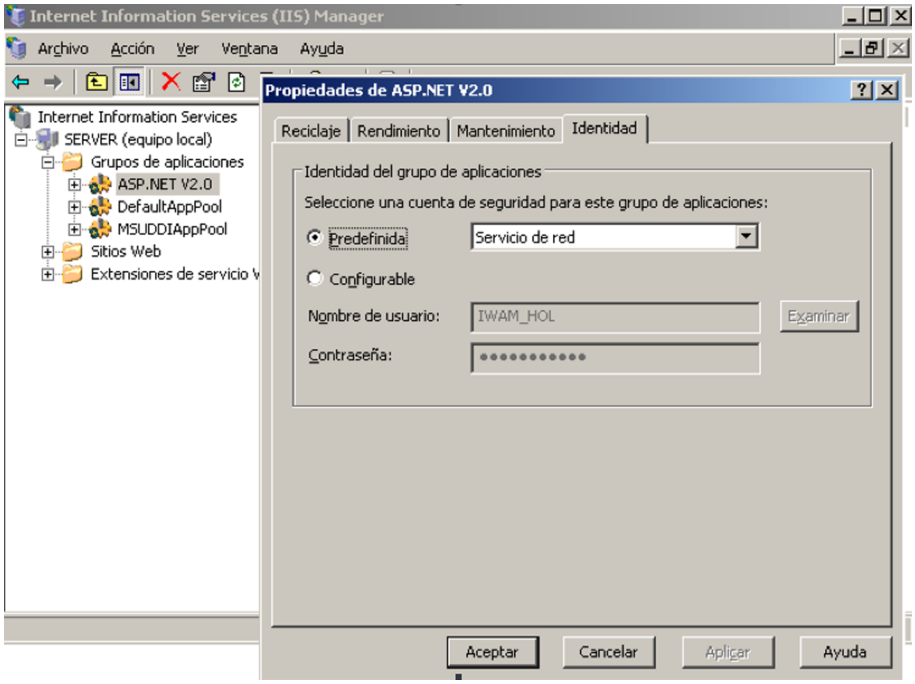
En aquest cas, s'aplica una regla d'autorització al recurs *Administracio.aspx*, de manera que només hi poden accedir els usuaris amb el rol *Administrador*; la resta d'usuaris no tenen autorització per a accedir-hi.

És important destacar que en IIS6 o inferior les regles d'autorització només afecten recursos processats per l'entorn de treball .NET, de manera que no es poden utilitzar per a protegir l'accés a un altre tipus de fitxers, com ara fitxers *zip* o *doc*. En IIS7 és possible utilitzar aquestes regles d'autorització per a protegir tot el contingut, gràcies a la integració total de la plataforma ASP.NET en el servidor d'aplicacions.

3.3. Identitat del codi i personalització

En una aplicació web és molt important conèixer les credencials amb les quals s'executa el procés ASP.NET. En un IIS5 per defecte s'utilitza el compte ASP.NET, mentre que en un IIS6 o IIS7 s'utilitzen les credencials a càrrec de les quals és el grup d'aplicacions a què pertany l'aplicació web.

Per defecte, el grup o *pool* d'aplicacions utilitza l'usuari *Servei de xarxa* o *Network Service*. Aquests comptes han de ser tan poc privilegiats com es pugui i han de concedir accés únicament als recursos que necessiti l'aplicació web. Amb l'eina d'administració d'IIS és possible configurar el grup d'aplicacions que ha d'utilitzar una aplicació web, i també l'usuari a càrrec del qual és l'aplicació, com s'observa en la imatge següent:

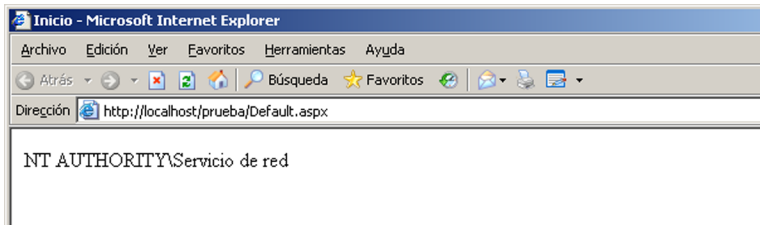


Configuració del grup d'aplicacions

Utilitzant l'objecte *WindowsIdentity* de la classe *Security.Principal* és possible, mitjançant programació, determinar l'usuari rere el qual s'executa l'aplicació. Per a mostrar per pantalla el nom d'usuari que s'utilitza, n'hi ha prou d'utilitzar el fragment de codi següent:

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

En una aplicació web, el grup d'aplicacions de la qual està configurat com es mostra en la imatge anterior, el resultat de l'execució d'aquest codi és el que mostra la imatge següent, independentment que el tipus d'autenticació estigui basat en formularis o en Windows:



Usuari utilitzat pel grup d'aplicacions

No obstant això, per mitjà de l'element *identity* del fitxer de configuració *web.config* és possible activar la personificació. Això permet a ASP.NET executar-se com un procés que utilitza els privilegis d'un altre usuari.

Si en el fitxer de configuració s'activa l'atribut *impersonate* de l'element *identity*, el procés ASP.NET passa a executar-se amb els privilegis de l'usuari que fa la petició; per tant, si es col·loca la configuració següent en el fitxer *web.config*:

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

Independentment del tipus d'autenticació, si el servidor IIS està configurat amb accés anònim, les credencials que utilitza el procés ASP.NET per a executar l'aplicació són les configurades en l'accés anònim.

Si es fa servir l'autenticació bàsica o integrada, les credencials utilitzades són les de l'usuari que fa la petició. També és possible, per mitjà de l'element *identity*, configurar l'aplicació per a utilitzar un compte específic, com es veu en el quadre següent:

```
<configuration>
   <system.web>
      <identity impersonate="true"
            userName="Usuari"
            password="Clau" />
   </system.web>
</configuration>
```

3.3.1. Model SSO en una intranet

Aprofitant les capacitats de personificació d'ASP.NET, en una xarxa corporativa en què hi ha un controlador de domini al qual hi ha subscrits els equips dels diferents usuaris, és possible desplegar un sistema únic d'autenticació que permeti a un usuari, una vegada iniciada la seva sessió amb el compte de domini, accedir als diferents recursos de la intranet sense haver de tornar a autenticar-se.

En la imatge següent, es mostra un petit diagrama de xarxa on es representa una infraestructura en què hi ha un controlador de domini i un servidor web; tots els equips client, tant si són ordinadors de taula, portàtils o PDA com si són altres dispositius, estan subscrits al domini, de manera que els usuaris inicien sessió amb un compte del domini.

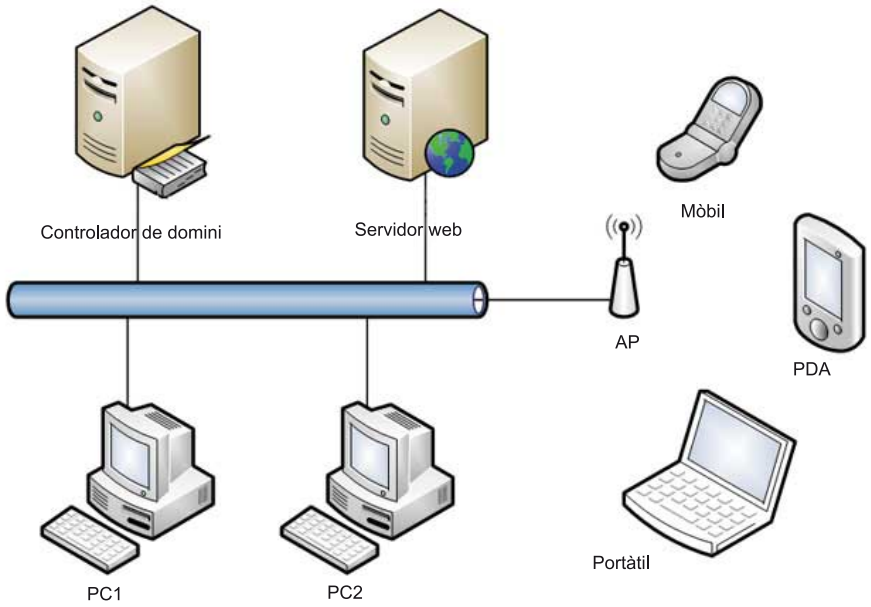
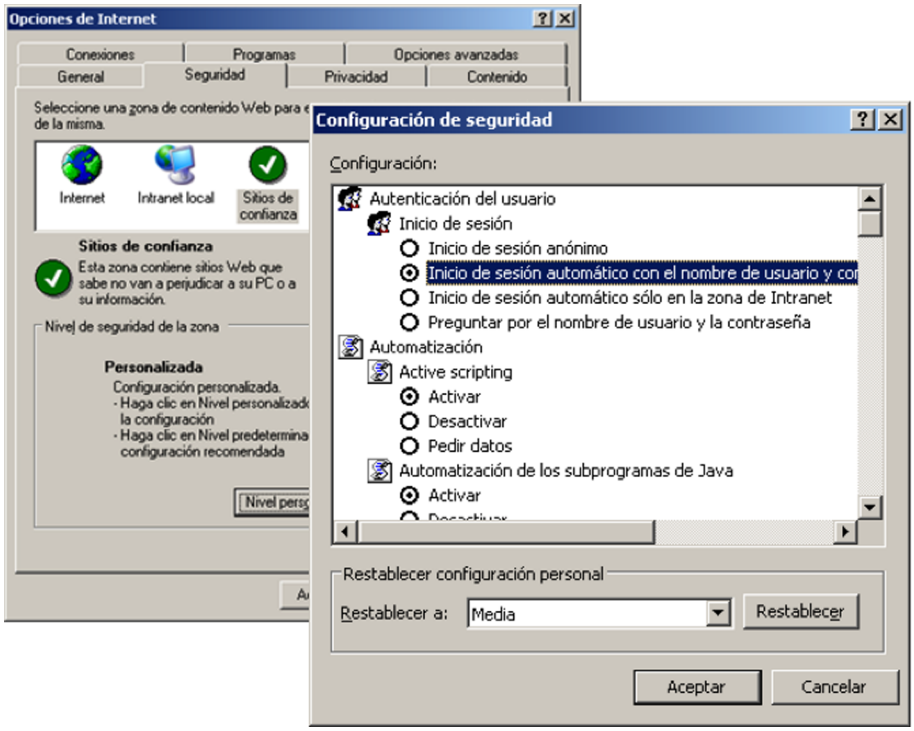


Diagrama de xarxa corporatiu

L'aplicació ASP.NET de la intranet es configura utilitzant el model d'autenticació Windows i s'activa la personalització; en el servidor web, s'especifica l'autenticació de Windows integrada i com a proveïdor de rols en l'aplicació ASP.NET s'indica que s'utilitzarà el proveïdor *AspNetWindowsTokenRoleProvider*. D'aquesta manera, és possible utilitzar els grups d'usuaris del domini en les regles d'autorització de l'aplicació ASP.NET. Aquestes configuracions s'han vist al llarg del manual.

En els navegadors dels equips client, cal configurar l'enviament de credencials de manera automàtica; per a fer-ho, si s'afegeix l'adreça web de l'aplicació de la intranet als llocs de confiança, es poden canviar les opcions de configuració per enviar, de manera automàtica, les credencials amb què s'ha iniciat sessió, tal com es mostra en la imatge següent:



Configuració d'Internet Explorer

D'aquesta manera, quan l'usuari accedeix a la intranet, es transmeten les seves credencials, de manera que el procés ASP.NET s'executa amb els seus privilegis, i accedeix únicament als llocs on té autorització el seu usuari segons els grups a què pertany; si l'aplicació es connecta a la base de dades SQL Server mitjançant l'autenticació integrada, les seves credencials es transmeten al servidor de bases de dades, i es controlen també a aquest nivell els recursos a què té accés l'usuari.

3.4. Autenticació en aplicacions Windows Forms

Els conceptes d'*autenticació* i *autorització* són presents no solament en les aplicacions ASP.NET, sinó també en la resta d'aplicacions .NET, incloses les d'escriptori. En una organització amb directori actiu, és possible utilitzar les classes *WindowsIdentity* i *WindowsPrincipal* per a determinar la identitat dels usuaris i protegir les aplicacions d'un ús no autoritzat, incloent aquí assembladors complets, mètodes o fragments de codi.

Implementant classes que heretin de les classes *GenericIdentity* i *GenericPrincipal*, és possible controlar l'autenticació i autorització en entorns no Microsoft, com són aquells en què els usuaris i les regles d'accés resideixen en bases de dades.

3.4.1. Aplicacions autònomes

Amb el terme *autònoma* o *stand-alone* es defineixen les aplicacions que es poden executar i controlar com a entitats independents.

Un exemple d'aquestes aplicacions és un processador de text; aquesta aplicació és completament autònoma i no necessita estar connectada a cap servidor perquè funcioni.

En aquest tipus d'aplicacions, l'autenticació i autorització d'accés als diferents recursos i funcionalitats s'han d'implementar en el codi de l'aplicació mateixa. En aquest apartat, es presenten les classes principals que pot utilitzar un programador per a identificar l'usuari que executa l'aplicació, el grup a què pertany i, segons això, permetre l'accés o l'execució de determinades funcionalitats.

La classe *System.Security.Principal.WindowsIdentity* representa el compte de l'usuari Windows, incloent-hi dades com el nom d'usuari, el tipus d'autenticació i el testimoni o *token* d'autenticació; mitjançant el codi que es presenta a continuació, és possible determinar el nom de l'usuari que executa l'aplicació:

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

L'objecte *WindowsIdentity* té una altra sèrie de propietats i mètodes per a recuperar més informació sobre l'usuari que ha iniciat sessió.

Amb la classe *WindowsPrincipal* és possible accedir al grup d'usuaris a què pertany l'usuari, de manera que és possible fer comprovacions per a autoritzar l'accés a determinats recursos o funcionalitats. El quadre següent mostra el codi necessari per a comprovar si l'usuari pertany al grup administrador o a un grup específic d'usuaris:

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

   if (current.IsInRole(WindowsBuiltInRole.Administrator))
      Console.WriteLine("L'Usuari pertany a administradors");

   if(current.IsInRole(@"DOMINI\GRUP"))
      Console.WriteLine("L'Usuari pertany a GRUP");
}
```

Mitjançant la classe *PrincipalPermission* és possible demanar, de manera declarativa o imperativa (mitjançant programació), que l'usuari que executa l'aplicació pertanyi a un grup d'usuaris per a executar una classe, un mètode o un fragment de codi.

En el quadre següent, es mostra de manera declarativa que per a executar l'aplicació és necessari un usuari autenticat que pertanyi al grup d'administradors locals de la màquina.

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

[PrincipalPermission(SecurityAction.Demand, Authenticated=true, Role="Administradors")]
static void imprimirMissatge()
{
   Console.WriteLine("Hola, món");
}
```

En el mètode *Main* de l'aplicació, s'especifica la política de seguretat que se seguirà en els fils d'aquesta aplicació, mentre que en el mètode *imprimirMissatge* se sol·licita la necessitat de ser un usuari autenticat i pertànyer al grup Administradors de la màquina local.

Si l'aplicació s'executa amb un usuari que no pertany al grup d'administradors de la màquina o, fins i tot pertanyent a aquest grup en un sistema Windows Vista o Windows 7, s'executa sense elevar privilegis, el resultat de l'execució és un error de permisos. En canvi, si l'usuari pertany al grup Administradors, o es fa l'elevació de privilegis en un sistema Windows Vista o Windows 7, el codi es pot executar correctament.

D'aquesta manera és possible, mitjançant el codi de l'aplicació, identificar l'usuari que executa l'aplicació i protegir l'accés als recursos de l'aplicació segons el perfil d'aquest usuari.

3.4.2. Aplicacions client-servidor

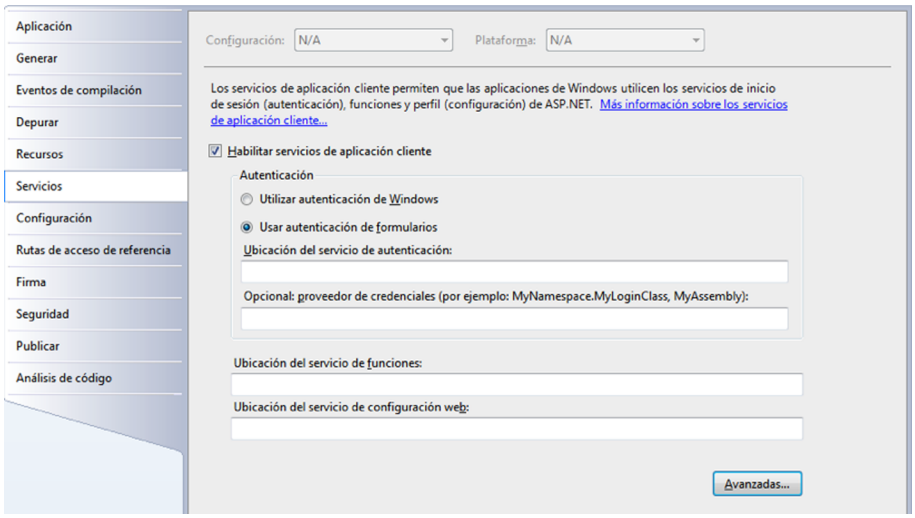
Al contrari que les aplicacions autònomes, hi ha moltes aplicacions que necessiten connexió amb un servidor per a fer diferents tasques, com és el cas d'aplicacions tan conegudes com Windows Live Messenger o Yahoo Messenger, en les quals el procés d'autenticació de l'usuari no es fa a la màquina local, sinó als servidors dedicats a aquest efecte.

Per a aquest tipus d'aplicacions és necessari que, des de l'aplicació client, es reculli el nom d'usuari i la clau i s'enviï al servidor per a verificar-los. L'enviament de credencials es pot fer mitjançant un servei web, que s'encarrega de validar l'usuari.

Amb l'entorn de treball .NET 3.5, Microsoft proporciona una novetat per a les aplicacions Windows Forms i WPF (Windows Presentation Foundation). Es tracta dels serveis d'aplicació client que proporcionen, a aquest tipus d'aplicacions, accés simplificat a l'inici de sessió, rols i serveis de perfil disponible en ASP.

Amb els serveis d'aplicació de client és possible utilitzar un servidor centralitzat per a autenticar els usuaris, determinar les funcions assignades a cada usuari i emmagatzemar informació de configuració d'aplicacions per usuari que pot compartir a la Xarxa.

Si es té una aplicació que s'executa amb l'entorn de treball 3.5, accedint a les propietats del projecte a Visual Studio, amb la pestanya *Serveis* es pot configurar aquesta característica, com es veu en la imatge següent:



Serveis d'aplicació client

Mitjançant la configuració adequada en client i en servidor, és possible aconseguir que les aplicacions es validin en el servidor fent ús dels mecanismes que ja hem vist, que permeten autenticar usuaris contra diferents repositoris d'emmagatzematge com SQL Server o directoris LDAP, i gestionar rols per a les regles d'autorització.

L'ús de clients que utilitzin versions anteriors de l'entorn de treball .NET fa necessari que el programador implementi el seu propi mecanisme per a l'autenticació d'usuari i gestió de rols. Amb les últimes versions de l'entorn de treball .NET s'aconsegueix seguir la mateixa dinàmica de treball, tant si es tracta d'aplicacions ASP.NET com d'aplicacions client.