Aplicaciones web

Tras haber visto en qué consiste el desarrollo de interfaces web, las alternativas de las que dispone el programador y el modelo concreto de programación de las páginas ASP.NET web en la plataforma .NET, pasamos ahora a estudiar cómo se construyen aplicaciones web reales, en las cuales suele haber más de una página:

- Primero tendremos que aprender a dominar algunos detalles de funcionamiento del protocolo HTTP, el protocolo a través del cual el navegador del usuario accede al servidor web donde se aloja nuestra aplicación ASP.NET.

- Acto seguido, veremos cómo ASP.NET nos facilita mantener información común a varias solicitudes HTTP realizadas independientemente; esto es, lo que se conoce en el mundo de las aplicaciones web como sesiones de usuario.

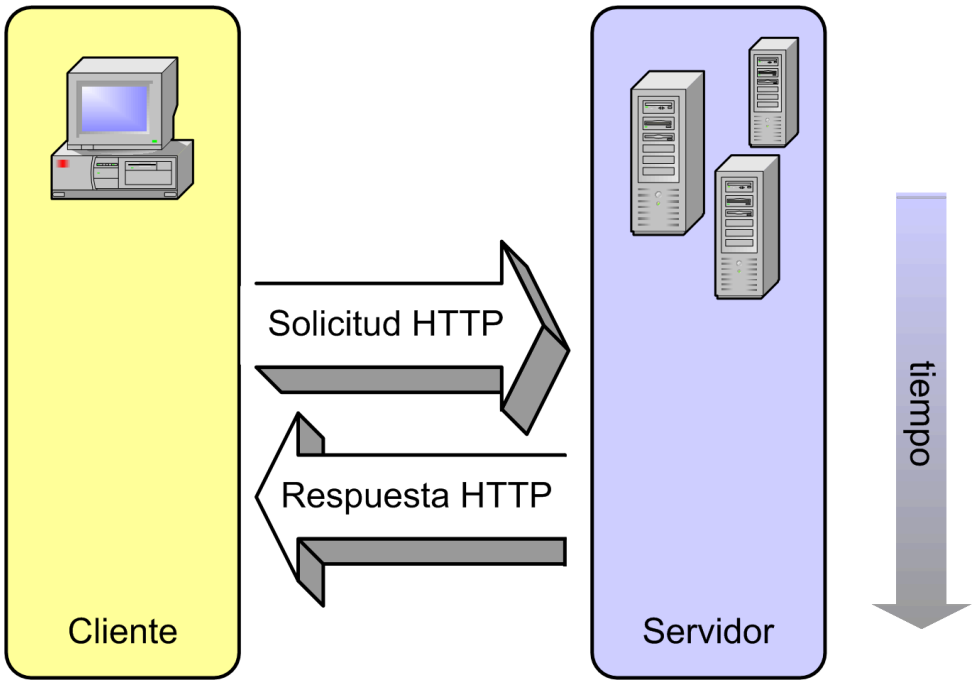
- Finalmente, cerraremos el capítulo describiendo cómo podemos controlar a qué partes de la aplicación puede acceder cada usuario, cómo crear formularios de autentificación y darle permisos al usuario para que realice determinadas tareas en el servidor.

Otros aspectos de interés relacionados con la organización interna de las aplicaciones ASP.NET los dejaremos para el siguiente capítulo. Por ahora, nos centraremos en las cuestiones que afectan más directamente al uso de nuestra aplicación por parte del usuario.

Aplicaciones web

El protocolo HTTP

El protocolo HTTP [HyperText Transfer Protocol] es un protocolo simple de tipo solicitud-respuesta incluido dentro de la familia de protocolos TCP/IP que se utiliza en Internet. Esto quiere decir que, cada vez que accedemos a una página (en general, a un recurso accesible a través de HTTP), se establece una conexión diferente e independiente de las anteriores.



Funcionamiento del protocolo HTTP: Cada solicitud del cliente tiene como resultado una respuesta del servidor y, cada vez que el cliente hace una solicitud, ésta se realiza de forma independiente a las anteriores.

Internamente, cuando tecleamos una dirección de una página en la barra de direcciones del navegador web o pinchamos sobre un enlace, el navegador establece una conexión TCP con el servidor web al que pertenece la dirección especificada. Esta dirección es una URL de la forma `http://...` y, salvo que se indique lo contrario en la propia URL, la conexión con el servidor se establecerá a través del puerto 80 TCP. Una vez establecida la conexión, el cliente envía un mensaje al servidor (la solicitud) y éste le responde con otro mensaje (la respuesta). Tras esto, la conexión se cierra y el ciclo vuelve a empezar. No obstante, hay que

mencionar que, por cuestiones de eficiencia y para reducir la congestión en la red, HTTP/1.1 mantiene conexiones persistentes, lo cual no quiere decir que la interacción entre cliente y servidor varíe un ápice desde el punto de vista lógico: cada par solicitud-respuesta es independiente.

Información técnica acerca de Internet

Como se verá con más detalle en el capítulo dedicado a la creación de aplicaciones distribuidas con sockets, los puertos son un mecanismo que permite mantener varias conexiones abiertas simultáneamente, lo que se conoce como multiplexación de conexiones. Los protocolos TCP y UDP incluyen este mecanismo para que, por ejemplo, uno pueda estar navegando por Internet a la vez que consulta el correo o accede a una base de datos.

Toda la información técnica relacionada con los estándares utilizados en Internet se puede consultar en http://www.ietf.org, la página web del *Internet Engineering Task Force*. IETF es una comunidad abierta que se encarga de garantizar el correcto funcionamiento de Internet por medio de la elaboración de documentos denominados RFCs [*Request For Comments*]. Por ejemplo, la versión 1.0 del protocolo HTTP está definida en el RFC 1945, mientras que la especificación de la versión 1.1 de HTTP se puede encontrar en el RFC 2616.

El protocolo HTTP sólo distingue dos tipos de mensajes, solicitudes y respuestas, que se diferencian únicamente en su primera línea. Tanto las solicitudes como las respuestas pueden incluir distintas cabeceras además del cuerpo del mensaje. En el cuerpo del mensaje es donde se transmiten los datos en sí, mientras que las cabeceras permiten especificar información adicional acerca de los datos transmitidos. En el caso de HTTP, las cabeceras siempre son de la forma `clave: valor`. Un pequeño ejemplo nos ayudará a entender el sencillo funcionamiento del protocolo HTTP.

Cuando accedemos a una página web desde nuestro navegador, la solicitud que se le remite al servidor HTTP es de la siguiente forma:

```
GET http://csharp.ikor.org/index.html HTTP/1.1
If-Modified-Since: Fri, 31 Oct 2003 19:41:00 GMT
Referer: http://www.google.com/search?...
```

La primera línea de la solicitud, aparte de indicar la versión de HTTP utilizada (`1.1` en este

caso), también determina el método utilizado para acceder al recurso solicitado. Este recurso ha de venir identificado mediante un URI [*Universal Resource Identifier*], tal como se define en el estándar RFC 2396. Los métodos de acceso más usados son `GET` y `POST`, que sólo se diferencian en la forma de pasar los parámetros de los formularios. Existe otro método, `HEAD`, que sólo devuelve metadatos acerca del recurso solicitado.

Tras la primera línea de la solicitud, pueden o no aparecer líneas adicionales en las cuales se añade información relativa a la solicitud o al propio cliente. En el ejemplo, `If-Modified-Since` sirve para que el cliente no tenga que descargar una página si ésta no ha cambiado desde la última vez que accedió a ella en el servidor. Por su parte, `Referer` indica la URL del sitio desde el que se accede a la página, algo de vital importancia si queremos analizar el tráfico de nuestro servidor web. Incluso existen cabeceras, como `User-Agent`, mediante las cuales podemos averiguar el sistema operativo y el navegador que usa el cliente al acceder al servidor HTTP.

El final de una solicitud lo marca una línea en blanco. Esta línea le indica al servidor HTTP que el cliente ya ha completado su solicitud, por lo que el servidor puede comenzar a generar la respuesta adecuada a la solicitud recibida. Dicha respuesta puede ser el aspecto siguiente:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Sun, 17 Aug 2003 10:35:30 GMT
Content-Type: text/html
Last-Modified: Tue, 27 Mar 2001 10:34:52 GMT
Content-Length: XXX
<html>
  --- Aquí se envía el texto de la página HTML
</html>
```

Como se puede apreciar, en la primera línea de la respuesta aparece la versión de HTTP empleada, un código de estado de tres dígitos y una breve descripción de ese código de estado. A continuación, aparecen una serie de cabeceras en las que se puede identificar el servidor HTTP que realiza la respuesta (`Server`), la fecha (`Date`), el tipo MIME correspondiente a los datos que se envían con la respuesta (`Content-Type`), la última vez que se modificó el fichero devuelto (`Last-Modified`) y la longitud del fichero de datos que se manda como respuesta a la solicitud (`Content-Length`, donde `XXX` representa, en una respuesta real, el número de bytes de datos que se transmiten). Finalmente, la respuesta concluye enviando los datos asociados al recurso que solicitó el cliente, un fichero HTML tal como indica el tipo de la respuesta (`text/html`).

Como se puede apreciar, en la cabecera de la respuesta HTTP se incluyen bastantes datos que pueden resultar de interés a la hora de controlar la interacción entre el cliente y el servidor. Respecto al primero de esos datos, el código de estado devuelto en la primera línea de la respuesta HTTP, la siguiente tabla resume las distintas categorías a las que pueden pertenecer:

Código	Significado	Ejemplos
1xx	Mensaje informativo	
2xx	Éxito	`200 OK`
3xx	Redirección	`301 Moved Permanently` `302 Resource temporarily moved`

4xx	Error en el cliente	`400 Bad request`
		`401 Unauthorized`
		`403 Forbidden`

5xx	Error en el servidor	`500 Internal Server Error`

Los apartados siguientes nos mostrarán cómo podemos hacer uso de nuestro conocimiento del funcionamiento interno del protocolo HTTP para realizar determinadas tareas que pueden sernos de interés en la creación de aplicaciones web.

En el camino correcto

Como cualquier programador que se precie debe saber, lo ideal a la hora de desarrollar un sistema de cierta complejidad es dividir dicho sistema en subsistemas lo más independientes posibles. En el caso particular que nos ocupa, cuando tenemos una aplicación web con multitud de páginas ASP.NET, lo ideal es que cada página sea independiente de las demás para facilitar su mantenimiento y su posible reutilización.

Esta independencia resulta prácticamente imposible de conseguir si no separamos la lógica de cada página de la lógica encargada de la navegación entre las distintas partes de la aplicación. Además, tampoco nos gustaría que la lógica encargada de la navegación por las distintas partes de nuestra aplicación apareciese duplicada en cada una de las páginas de ésta. En el siguiente capítulo se tratará con mayor detalle cómo organizar la aplicación para mantener su flexibilidad sin que exista código duplicado. Por ahora, nos conformaremos con ver cómo podemos controlar dinámicamente las transiciones de una página a otra sin que estas transiciones tengan que estar prefijadas en el código de la aplicación.

La implementación de las redirecciones resulta trivial en ASP.NET. Cuando queremos redirigir al usuario a una URL concreta, lo único que tenemos que escribir es lo siguiente:

```
Response.Redirect("http://csharp.ikor.org");
```

El método `Redirect` de la clase `HttpResponse` redirige al explorador del cliente a la URL especificada utilizando la respuesta HTTP adecuada, mediante el uso del grupo de códigos de estado 3xx (véase la tabla anterior).

Cuando lo único que queremos es enviar al usuario a otra página ASP.NET de nuestra propia aplicación, podemos utilizar el método `Transfer` de la clase `HttpServerUtility`. Basta con teclear:

```
Server.Transfer("bienvenida.aspx");
```

En este caso, la redirección se realiza internamente en el servidor y el navegador del usuario no es consciente del cambio de página. Esto resulta más eficiente que enviarle un código de redirección al navegador del usuario pero puede producir resultados no deseados si el usuario actualiza la página desde la barra de botones de su navegador.

Algo tan sencillo como las redirecciones nos puede servir, no sólo para decidir en tiempo de ejecución la siguiente página que se le ha de mostrar al usuario, sino también para configurar dinámicamente nuestra aplicación. Por ejemplo, podemos almacenar en un fichero XML las URLs correspondientes a los distintos módulos de una aplicación y, en función del contenido de ese fichero, decidir a dónde debemos dirigir al usuario. Esto no solamente disminuye el acoplamiento entre las distintas páginas de la aplicación y su ubicación física , sino que también facilita la implantación gradual de nuevos sistemas conforme se van implementando nuevos módulos. Además, en vez de que cada página sea responsable de decidir a dónde ha de dirigirse el usuario (lo que puede ocasionar problemas de consistencia), se puede garantizar la realización de las acciones de coordinación adecuadas a lo largo y ancho de la aplicación. Simplemente, se elimina de las páginas individuales la responsabilidad de coordinarse con las demás páginas de la aplicación, disminuyendo el acoplamiento e implementado la cohesión de las distintas páginas.

Control del tráfico con manejadores y filtros HTTP

En el apartado anterior hemos visto cómo podemos controlar la navegación por las distintas partes de una aplicación web. Ahora veremos cómo podemos controlar a bajo nivel las peticiones que recibe nuestra aplicación e incluso llegar a interceptar las peticiones HTTP antes de que éstas sean atendidas por una página.

Control de las solicitudes HTTP

Por ejemplo, podríamos centralizar el control de la navegación por nuestra aplicación si creamos un manejador que implemente la interfaz `IHttpHandler`. Este manejador será el

que reciba todas las peticiones HTTP y decida en cada momento cuál es la acción más adecuada en función de la solicitud recibida. De hecho, el interfaz `IHttpHandler` es la base sobre la que se montan todas las páginas ASP.NET y nosotros, si nuestro problema lo justifica, podemos acceder a bajo nivel a las solicitudes y respuestas HTTP correspondientes a la interacción del usuario con nuestra aplicación. Éste es el aspecto que tendría nuestro manejador:

```
using System;
using System.Web;

public class Handler : IHttpHandler
{
  public void ProcessRequest (HttpContext context)
  {
    Command command = ...

    command.Do (context);
  }

  public bool IsReusable
  {
    get { return true;}
  }
}
```

donde `Command` se utiliza para representar una acción concreta de las que puede realizar nuestra aplicación. Dicho objeto se seleccionará en función del contexto de la solicitud HTTP recibida; por ejemplo, a partir de los valores de los parámetros de la solicitud, a los que se puede acceder con `context.Request.Params`. En realidad, el objeto `command` será una instancia de una clase que implemente la interfaz `Command`:

```
using System;
using System.Web;

public interface Command
{
  void Do (HttpContext context);
}
```

De esta forma se aplica un conocido patrón de diseño para separar la responsabilidad de determinar cuál debe ser la acción que la aplicación ha de realizar, de lo que se encarga el manejador, de la ejecución en sí de la acción, de la que se encarga el objeto que implemente la interfaz `Command`. Este es el mismo esquema que se utiliza, por ejemplo, para dotar a un programa de la capacidad de hacer y deshacer acciones. Para esto último sólo tendríamos que añadir a la interfaz `Command` un método `Undo` que se encargase de deshacer cualquier cambio que se hubiese efectuado al ejecutar el método `Do`.

Un lector observador se habrá dado cuenta de que, hasta ahora, hemos creado algunas clases pero aún no las hemos enganchado a nuestra aplicación web para que hagan su trabajo. En realidad, lo único que nos queda por hacer es indicarle al servidor web que todas las peticiones HTTP que reciba nuestra aplicación se atiendan a través de nuestro manejador. Esto se consigue añadiendo lo siguiente en la sección `<system.web>` de un fichero XML denominado `Web.config` que contiene los datos relativos a la configuración de una aplicación ASP.NET:

```
<httpHandlers>
  <add verb="GET" path="*.aspx" type="Handler,ControladorWeb" />
</httpHandlers>
```

donde `Handler` es el nombre de la clase que hace de manejador y `ControladorWeb.dll` es el nombre de la DLL donde está definido nuestro controlador.

Siguiendo el esquema aquí esbozado se centraliza el tráfico que recibe nuestra aplicación, con lo que se garantiza la consistencia de nuestra aplicación web y se mejora su flexibilidad a la hora de incorporar nuevos módulos. A cambio, el controlador incrementa algo la complejidad de la implementación y, si no es lo suficientemente eficiente, puede crear un cuello de botella en el acceso a nuestra aplicación.

Interceptación de los mensajes HTTP

Puede que estemos interesados en tener cierto control sobre las solicitudes que recibe nuestra aplicación pero no deseemos llegar al extremo de tener que implementar nosotros la lógica que determine qué acción concreta ha de realizarse en cada momento. En ese caso, podemos utilizar filtros HTTP por los que vayan pasando las solicitudes HTTP antes de llegar a la página ASP.NET. Esto puede ser útil para monitorizar el uso del sistema, decodificar los datos recibidos o realizar tareas a bajo nivel cualquier otra tarea que requieran el procesamiento de las solicitudes HTTP que recibe nuestra aplicación (como pueden ser, por ejemplo, identificar el tipo de navegador que utilizan los usuarios o la resolución de su pantalla).

En los párrafos anteriores describimos cómo podemos reemplazar el mecanismo habitual de recepción de solicitudes HTTP en las aplicaciones ASP.NET definiendo nosotros mismos una clase que implemente la interfaz `IHttpHandler`. Ahora vamos a ver cómo podemos interceptar esas mismas solicitudes sin tener que modificar el modo de funcionamiento de las páginas ASP.NET de nuestra aplicación. Para ello, deberemos implementar un módulo HTTP.

Los módulos HTTP son clases que implementan la interfaz `IHttpModule`. Por ejemplo, `SessionStateModule` es un módulo HTTP que se encarga de la gestión de sesiones de usuario en ASP.NET, algo que estudiaremos en la siguiente sección de este mismo capítulo. Los módulos HTTP se pueden usar para interceptar los mensajes HTTP correspondientes a

solicitudes y a respuestas en varios momentos de su procesamiento, por lo que, al implementar un módulo HTTP, deberemos decidir cuándo se interceptarán los mensajes.

El siguiente ejemplo muestra cómo podemos crear un módulo HTTP que mida el tiempo que se tarda en atender cada solicitud y lo muestre al final de cada página que ve el usuario:

```
using System;
using System.Web;
using System.Collections;

public class HttpLogModule: IHttpModule
{
  public void Init(HttpApplication application)
  {
    application.BeginRequest += new EventHandler(this.OnBeginRequest);
    application.EndRequest += new EventHandler(this.OnEndRequest);
  }

  // Al comienzo de cada solicitud...

  private void OnBeginRequest (Object source, EventArgs e)
  {
    HttpApplication application = (HttpApplication)source;
    HttpContext context = application.Context;
    context.Items["timestamp"] = System.DateTime.Now;
  }

  // Al terminar de procesar la solicitud...

  private void OnEndRequest (Object source, EventArgs e)
  {
    HttpApplication application = (HttpApplication)source;
    HttpContext context = application.Context;
    DateTime start = (DateTime) context.Items["timestamp"];
    DateTime end   = DateTime.Now;
    TimeSpan time  = end.Subtract(start);
    context.Response.Write("<hr>Tiempo empleado: "+time);
  }

  public void Dispose()
  {
  }
}
```

Igual que sucedía con los manejadores HTTP, el uso de módulos HTTP se indica en el fichero de configuración de la aplicación web, el fichero `Web.config`:

```
<httpModules>
  <add name="HttpLogModule" type="HttpLog.HttpLogModule, HttpLog" />
</httpModules>
```

donde `HttpLog.HttpLogModule` es el nombre de la clase que implementa el interfaz `IHttpModule` y `HttpLog` es el nombre de la DLL en la que está definido el tipo `HttpLog.HttpLogModule`.

Al utilizarse un único fichero de configuración para toda la aplicación web, se evita tener que duplicar código para realizar tareas comunes a las distintas partes de la aplicación. No sólo eso, sino que se pueden añadir y eliminar filtros dinámicamente mientras la aplicación está funcionando, sin tener que modificar una sola línea de código. Los filtros, además, son independientes unos de otros, por lo que se pueden combinar varios si es necesario. Incluso se pueden reutilizar directamente en distintos proyectos, pues los filtros sólo deben dependen del contexto de una solicitud HTTP.

> Ya que los filtros se ejecutan siempre, para todas las solicitudes HTTP que reciba nuestra aplicación, es esencial que su ejecución sea extremadamente eficiente. De hecho, los filtros implementados como módulos HTTP tradicionalmente se han implementado con lenguajes compilados como C o C++ usando ISAPI (un interfaz específico del IIS).

Cuestión de refresco

Las peculiaridades de las interfaces web ocasionan la aparición de problemas de los cuales no tendríamos que preocuparnos en otros contextos. Éste es el caso de un problema bastante común con el que nos encontraremos siempre que nuestra aplicación web deba realizar una tarea relativamente larga. Lo que está claro es que no queda demasiado bien de cara al usuario dejar su ventana en blanco de forma indefinida mientras nuestra aplicación realiza los cálculos que sean necesarios.

Una solución más elegante involucra la utilización de hebras. Como se verá en la siguiente parte de este libro, las hebras nos permiten ejecutar concurrentemente distintas tareas. En nuestro caso, el problema de realizar un cálculo largo lo descompondremos en dos hebras:

- La hebra principal se encargará de mostrarle al usuario el estado actual de la aplicación, estado que se refrescará en su navegador automáticamente gracias al uso de la cabecera `Refresh`, definida en el estándar para las respuestas HTTP.

- Una hebra auxiliar será la encargada de ejecutar el código correspondiente a efectuar todos los cálculos que sean necesarios para satisfacer la solicitud del usuario.

Dicho esto, podemos pasar a ver cómo se implementaría la solución propuesta en la plataforma .NET. Comenzaríamos por implementar nuestra página ASP.NET:

```
...
using System.Threading;

public class Payment : System.Web.UI.Page
{

  protected Guid ID; // Identificador de la solicitud

  private void Page_Load(object sender, System.EventArgs e)
  {
    if (Page.IsPostBack) {
       // 1. Crear un ID para la solicitud
       ID = Guid.NewGuid();
       // 2. Lanzar la hebra
       ThreadStart ts = new ThreadStart(RealizarTarea);
       Thread thread = new Thread(ts);
       thread.Start();
       // 3. Redirigir a la página de resultados
       Response.Redirect("Result.aspx?ID=" + ID.ToString());
    }
  }

  private void RealizarTarea ()
  {
    ...
    Results.Add(ID, resultado);
  }
  ...
}
```

La clase auxiliar `Results` se limita a mantener una colección con los resultados de las distintas hebras que se hayan lanzado, para que la página de resultados pueda acceder a ellos:

```
using System;
using System.Collections;

public sealed class Results
{
  private static Hashtable results = new Hashtable();

  public static object Get(Guid ID)
  {
    return results[ID];
  }

  public static void Add (Guid ID, object result)
  {
    results[ID] = result;
  }

  public static void Remove(Guid ID)
  {
    results.Remove(ID);
  }
```

```
    public static bool Contains(Guid ID)
    {
      return results.Contains(ID);
    }
  }
```

Finalmente, la página encargada de mostrar los resultados se refrescará automáticamente hasta que la ejecución de la hebra auxiliar haya terminado y sus resultados estén disponibles:

```
public class Result : System.Web.UI.Page
{
  protected System.Web.UI.WebControls.Label lblMessage;

  private void Page_Load(object sender, System.EventArgs e)
  {
    Guid ID = new Guid(Page.Request.QueryString["ID"]);

    if (Results.Contains(ID)) {
       // La tarea ha terminado: Mostrar el resultado
       lblMessage.Text = Results.Get(ID).ToString();
       Results.Remove(ID);
    } else {
       // Aún no tenemos el resultado: Esperar otros 2 segundos
       Response.AddHeader("Refresh", "2");
    }
  }
  ...
}
```

Aunque pueda parecer algo complejo, la solución aquí propuesta es extremadamente útil en la práctica. Imagine, por ejemplo, una aplicación de comercio electrónico que ha de contactar con un banco para comprobar la validez de una tarjeta de crédito. No resulta demasiado difícil imaginar la impresión del usuario final cuando la implementación utiliza hebras y cuando no lo hace. En otras palabras, siempre será recomendable tener en cuenta la posibilidad de utilizar hebras cuando el tiempo de respuesta de nuestra aplicación afecte negativamente a su imagen de cara al usuario.

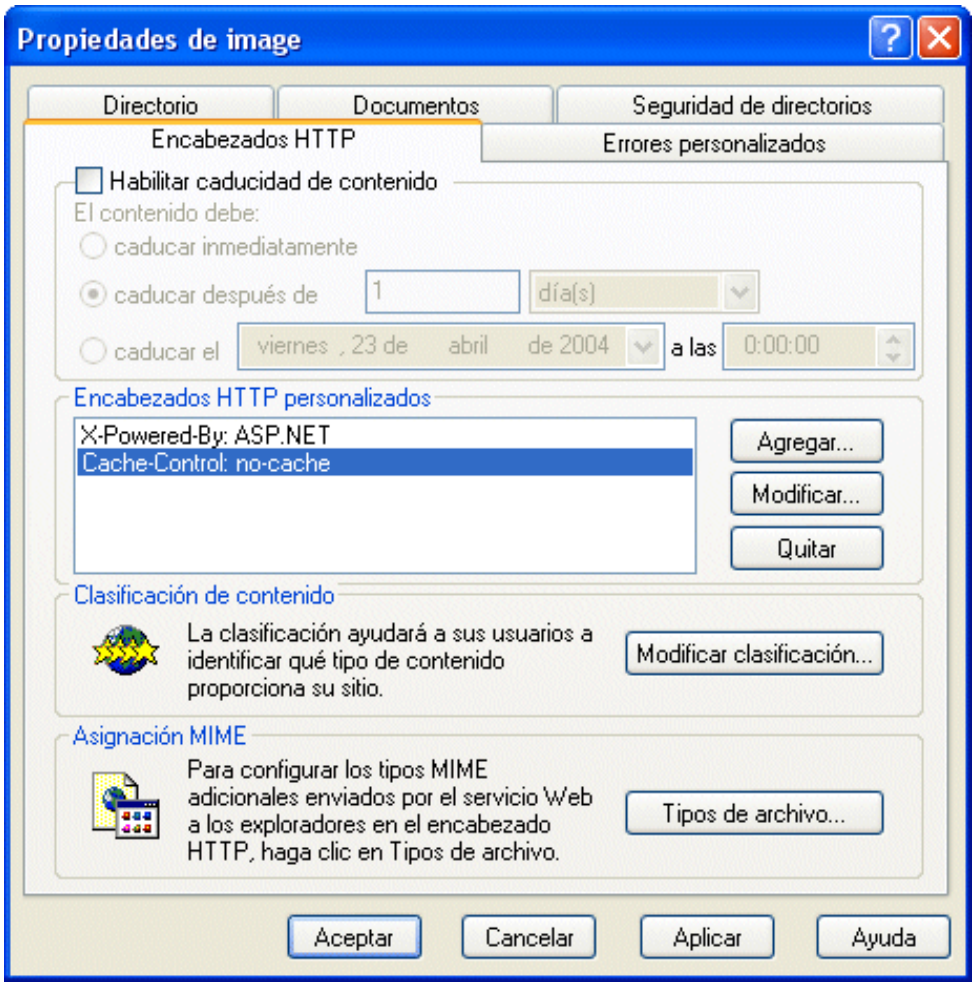
Almacenamiento en caché

En el navegador web del cliente...

Relacionado indirectamente con el refresco de una página web se encuentra otra de las cabeceras estándar que se pueden incluir en las respuestas HTTP. Nos estamos refiriendo a la cabecera `Cache-Control`, que se utiliza en ocasiones para disminuir el tráfico en la red no accediendo al servidor cada vez que, a través del navegador, se accede a una URL.

Si bien el uso de cachés puede ser altamente recomendable para mejorar el tiempo de respuesta de una aplicación, al no tener que acceder reiteradamente al servidor cada vez que se accede a un recurso concreto, también es cierto que, en ocasiones, nos interesará garantizar que no se utiliza ningún tipo de caché al acceder a determinada URL. Esto se consigue con la siguiente cabecera HTTP:

```
Cache-Control: no-cache
```



Definición de cabeceras HTTP específicas para modificar el comportamiento por defecto de una aplicación web. En este caso, para que el usuario nunca acceda por error a versiones antiguas de las imágenes almacenadas en un directorio, se evita que su navegador almacene copias locales de estas imágenes.

Lo anterior es equivalente a incluir lo siguiente en el código de una página ASP.NET:

```
Response.AddHeader("Cache-Control", "no-cache");
```

De esta forma nos aseguraremos de que el usuario siempre accederá a los datos más recientes de los que se disponga en el servidor. En el caso de la agenda de contactos utilizada como ejemplo en el capítulo anterior, cuando el usuario modifica la fotografía de una de las entradas de la agenda, la nueva imagen sobreescribe a la imagen antigua. En tal situación, para mostrar adecuadamente los datos de una persona será necesario deshabilitar la caché para las imágenes, pues si no el navegador del usuario volvería a mostrar la imagen que ya tiene almacenada localmente y no la imagen que reemplaza a ésta.

... y en el servidor web

Independientemente del uso de cachés en HTTP, ASP.NET también permite crear cachés de páginas. Esto puede resultar útil para páginas que se generan dinámicamente pero no cambian demasiado a menudo. El uso de una caché de páginas tiene el único objetivo de reducir la carga en el servidor y aumentar el rendimiento de la aplicación web.

Para que una página ASP.NET quede almacenada en la caché de páginas y no sea necesario volver a generarla basta con usar la directiva `@OutputCache` al comienzo del fichero `.aspx`. Por ejemplo:

```
<%@ OutputCache Duration="60" VaryByParam="none" %>
```

La primera vez que se acceda a la página, la página ASP.NET se ejecutará normalmente pero, al utilizar esta directiva, las siguientes solicitudes que lleguen referidas a la misma página utilizarán el resultado de la primera ejecución. Esto sucederá hasta que la página almacenada en caché caduque, 60 segundos después de su generación inicial en el ejemplo. Después de esos 60 segundos, la página se eliminará de la caché y en el siguiente acceso se volverá a ejecutar la página (además de volver a almacenarse en la caché durante otros 60 segundos).

Cookies

HTTP, por definición, es un protocolo sin estado. Sin embargo, al desarrollar aplicaciones web, mantener su estado resulta imprescindible. Por ejemplo, en un sistema de comercio electrónico debemos ser capaces de almacenar de alguna forma el carrito de la compra de un

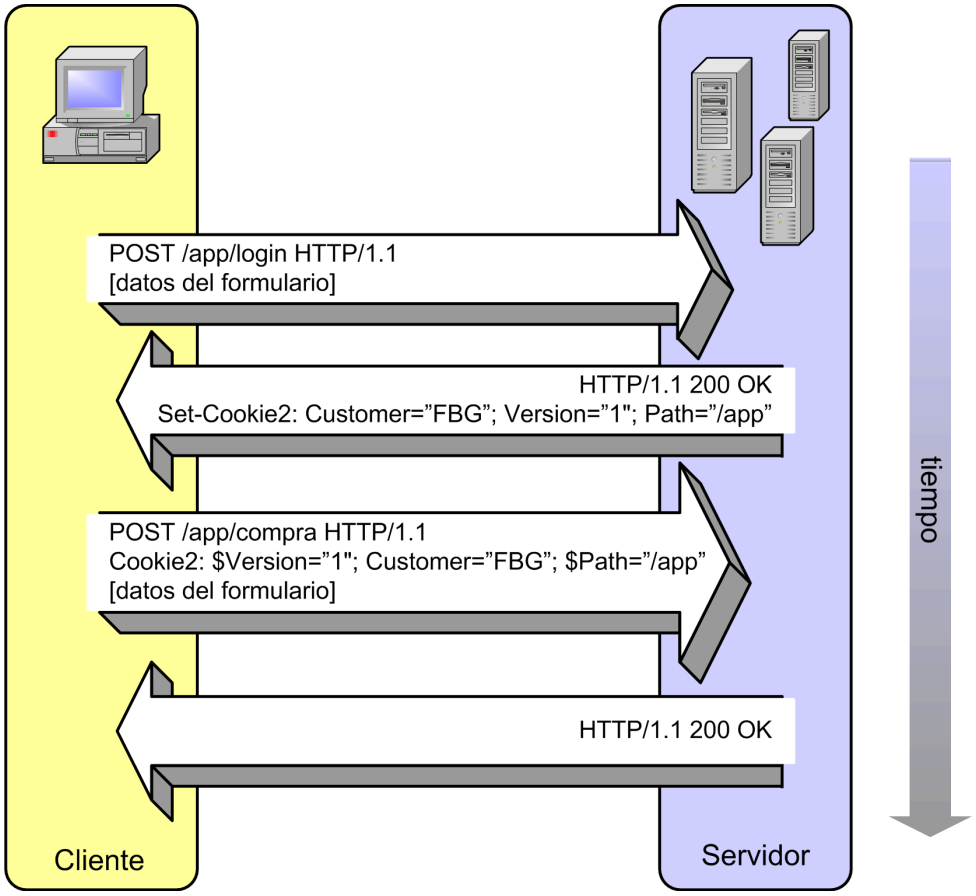
cliente concreto.

Una primera solución a este problema (no demasiado acertada, por cierto) consiste en gestionar sesiones utilizando cookies, tal como se definen en el RFC 2965. Una cookie, "galletita", es una pequeña cantidad de datos almacenada en el cliente. En realidad, se trata simplemente de un par nombre-valor acompañado por una fecha de caducidad. Los datos de la cookie se pasan al servidor como parte de cada solicitud HTTP, de forma que la aplicación web tiene acceso inmediato a esos datos del cliente. Además, la aplicación puede manipular la cookie para almacenar los datos que le interesen en cada momento.

Conforme se va navegando por Internet, muchos sitios van generando cookies en el cliente. En principio, los datos incluidos en la cookie sólo puede utilizarlos el servidor que la creó. La idea es utilizar una cookie por servidor o grupo de servidores y que esa cookie almacene en el cliente toda la información que pueda necesitar el servidor. En el ejemplo del carrito de la compra, la cookie podría incluir la lista de artículos que deseamos comprar.

No obstante, obsérvese que los datos asociados a una cookie se almacenan en la máquina cliente. Nada nos garantiza que los datos en el cliente están almacenados de forma segura. De hecho, usualmente se guardan sin ningún tipo de protección (búsquese, por ejemplo, el subdirectorio `Cookies` en Windows). Por tanto, siempre es recomendable no almacenar en una cookie datos que puedan poner en peligro la privacidad del usuario. Si volvemos al caso anterior, lo más recomendable sería almacenar únicamente en la cookie un identificador que nos permita identificar el carrito de la compra del usuario. El contenido de dicho carrito siempre se mantendría a buen recaudo en el servidor.

Por otro lado, el uso de cookies lo provoca el servidor y no el cliente (`Set-Cookie2`), por lo que en determinadas situaciones puede resultar conveniente configurar el cliente para que ignore los cookies, lo que se consigue no devolviendo la cabecera `Cookie2` mostrada en el siguiente diagrama:



Funcionamiento de las cookies

Las cookies se usan con frecuencia para mantener sesiones de usuario (conjuntos de conexiones HTTP relacionadas desde el punto de vista lógico) y también para analizar los usos y costumbres de los usuarios de portales web. Éste último punto, llevado al extremo, es lo que ha dado mala fama a la utilización de cookies y ha provocado que muchos usuarios, celosos de su privacidad, deshabiliten el uso de cookies en sus navegadores. En realidad, su uso proviene simplemente del hecho de que mantener una pequeña cantidad de datos en el cliente simplifica enormemente el seguimiento de los movimientos del usuario y descarga notablemente al servidor web, que no ha de emplear tiempo en analizar de dónde viene cada solicitud.

Aparte de los posibles problemas de privacidad que puede ocasionar el uso de cookies, también hay que tener en cuenta que la manipulación de las cookies creadas por nuestra aplicación web se puede convertir en un arma de ataque contra nuestra propia aplicación, por lo que debemos ser extremadamente cuidadosos a la hora de decidir qué datos almacenarán

las cookies.

Vistos los pros y los contras del uso de cookies, sólo nos falta ver cómo se pueden utilizar en ASP.NET. Como no podía ser de otra forma, su utilización es muy sencilla. Basta con acceder a las propiedades `Request.Cookies` y `Response.Cookies` de la página ASP.NET. Ambas propiedades devuelven una colección de cookies.

Para establecer el valor de una cookie, basta con escribir

```
Response.Cookies["user"]["name"]  = "BCC";
Response.Cookies["user"]["visit"] = DateTime.Now.ToString();
```

Para acceder a dichos valores bastará con teclear lo siguiente:

```
string   user  = Request.Cookies["user"]["name"];
DateTime visit = DateTime.Parse(Request.Cookies["user"]["visit"]);
```

Por defecto, ASP.NET utiliza cookies para mantener el identificador de la sesión de usuario. A partir de dicho identificador, se pueden recuperar todos los datos relativos al usuario de una aplicación web. De esto se encarga un filtro HTTP denominado `SessionStateModule`. De hecho, una de las tareas típicas de los filtros HTTP en muchas aplicaciones es el control de cookies con diversos fines. Los datos relativos a la sesión de usuario se almacenan en un objeto de tipo `Session` que será objeto de estudio en la siguiente sección de este capítulo.

> Aunque las cookies hayan recibido mucha publicidad, no son el único mecanismo mediante el cual una aplicación web puede almacenar información acerca del cliente. Otras alternativas menos polémicas incluyen el uso de campos ocultos en los formularios HTML, el empleo de parámetros codificados en la propia URL o, incluso, la utilización de bases de datos auxiliares.

Sesiones de usuario en ASP.NET

En las últimas páginas hemos analizado con relativa profundidad el funcionamiento del protocolo HTTP y también hemos visto cómo se pueden aprovechar algunas de sus características para conseguir en nuestras aplicaciones web el comportamiento deseado. No obstante, los mecanismos descritos se pueden considerar de "bajo nivel". Como cabría esperar, la biblioteca de clases de la plataforma .NET nos ofrece otros medios para desarrollar aplicaciones web sin necesidad de tratar directamente con los detalles del protocolo HTTP. En esta sección veremos algunas de las facilidades que nos ofrece la plataforma .NET, es especial aquéllas que nos facilitan el mantenimiento del estado de una aplicación web a pesar de que ésta esté montada sobre un protocolo sin estado como HTTP.

> Obviamente, utilizar las facilidades ofrecidas por la plataforma .NET no implica que el conocimiento del funcionamiento interno de una aplicación web deje de ser necesario. Igual que de cualquier programador se espera un conocimiento básico de la arquitectura de un ordenador y de su funcionamiento interno, un desarrollador de aplicaciones web debe ser consciente en todo momento de lo que sucede por debajo en una aplicación de este tipo. Por este motivo se le han dedicado bastantes páginas al estudio del protocolo HTTP antes de pasar a temas más específicos de ASP.NET.

El contexto de una página ASP.NET

Internamente, todo ASP.NET se construye a partir del interfaz `IHttpHandler`. Este interfaz, que ya hizo su aparición en el apartado anterior de este capítulo cuando vimos cómo se pueden interceptar las solicitudes HTTP, define únicamente dos métodos:

```
IHttpHandler {
  void ProcessRequest (HttpContext context);
  bool IsReusable ();
}
```

El método `ProcessRequest` es el encargado de procesar una solicitud HTTP concreta, a la cual se puede acceder utilizando el objeto de tipo `HttpContext` que recibe como parámetro. El otro método, `IsReusable`, sirve simplemente para indicar si una instancia de

`IHttpHandler` puede utilizarse para atender distintas solicitudes HTTP o deben crearse instancias diferentes para cada solicitud recibida.

A partir del interfaz `IHttpHandler`, si lo deseamos, podemos construir nuestra aplicación web. Nos bastaría con construir una clase que implemente este interfaz y utilice sentencias del tipo `context.Response.Write("<html>...");` para generar la página que verá el usuario en su navegador. No obstante, esto no sería mucho mejor que programar directamente CGIs, por lo que recurriremos a una serie de clases proporcionadas por ASP.NET para facilitarnos el trabajo.

Como ya vimos en el capítulo anterior, las páginas ASP.NET se crean implementando una clase derivada de `System.Web.UI.Page` en la que se separa el código de la presentación en HTML. Internamente, ASP.NET se encargará de compilar nuestra página construyendo una clase que implemente la interfaz `IHttpHandler`.

Aparte de esta clase base a partir de la cual creamos nuestras páginas y de todos los controles ASP.NET que nos facilitan la programación visual de los formularios web, ASP.NET nos proporciona otro conjunto de objetos que nos permite gobernar la interacción entre el cliente y el servidor en una aplicación web. Dentro de esta categoría, los objetos más importantes con los que trabajaremos en ASP.NET se recogen en la siguiente tabla:

Objeto	Representa
`HttpContext`	El entorno en el que se atiende la petición
`Request`	La petición HTTP realizada por el cliente
`Response`	La respuesta HTTP devuelta por el servidor
`Server`	Algunos métodos útiles
`Application`	Variables globales a nivel de la aplicación (comunes a todas las solicitudes recibidas desde cualquier cliente)
`Session`	Variables globales a nivel de una sesión de usuario (comunes a todas las solicitudes de un cliente concreto)

De los objetos recogidos en esta tabla, los dos últimos son los que nos permiten la interacción entre el cliente y el servidor más allá de los límites de un formulario ASP.NET. En el protocolo HTTP, cada par solicitud/respuesta es independiente del anterior. Cuando la interacción se limita a una misma página, ASP.NET se encarga de mantener automáticamente el estado del formulario mediante el uso de `ViewState`, tal como se describió al final del capítulo anterior. Cuando las solicitudes corresponden a páginas diferentes, `Application` y `Session` nos permiten manejar con comodidad las sesiones de usuario en ASP.NET.

En otras palabras, mientras que `HttpContext`, `Request` y `Response` proporcionan el contexto de una solicitud concreta, `ViewState` permite mantener el estado de un formulario concreto y, finalmente, `Session` y `Application` proporcionan un mecanismo sencillo

para almacenar información acerca del estado de una aplicación web (a nivel de cada usuario y a nivel global, respectivamente).

Mantenimiento del estado de una aplicación web

Si bien ASP.NET se encarga de mantener el estado de una página ASP.NET, en cuanto el usuario cambie de página (algo que suele ser habitual en cualquier aplicación web), tendremos que encargarnos nosotros de almacenar la información de su sesión de alguna forma. Para resolver este problema podemos optar por distintas alternativas:

- Almacenar la información de la sesión **manualmente en el cliente**, para lo cual se pueden emplear cookies. Esta solución hace uso del protocolo HTTP a bajo nivel y puede resultar no demasiado buena en función del tipo de aplicación. Baste con recordar la publicidad negativa que ha supuesto para determinadas empresas el uso indiscriminado de cookies.

- Otra opción es crear algún tipo de mecanismo que nos permita almacenar **manualmente en el servidor** los datos de cada sesión de usuario. Esta solución, menos controvertida que la primera, puede requerir un esfuerzo inicial considerable, si bien es cierto que puede ser la solución óptima en determinados entornos distribuidos (granjas de servidores, por ejemplo) y la única viable si deseamos construir un sistema tolerante a fallos.

- Por último, podemos dejar que los datos relativos a las sesiones se almacenen **automáticamente** si empleamos las ya mencionadas colecciones `Session` y `Application`. Dichas colecciones simplifican el trabajo del programador y, como veremos, ofrecen bastante flexibilidad a la hora de desplegar una aplicación web.

Las colecciones `Session` y `Application`, facilitadas por ASP.NET para el mantenimiento de sesiones de usuario en ASP.NET, se pueden ver como arrays asociativos. Un array asociativo es un vector al que se accede por valor en vez de por posición, igual que sucede en el hardware que implementa las memorias caché de cualquier ordenador. En cierto modo, `Session` y `Application` pueden verse como diccionarios en los que se utiliza una palabra para acceder a su definición. Esta estructura de datos es muy común en algunos lenguajes (AWK, por ejemplo) y resulta fácil de implementar en cualquier lenguaje que permita sobrecargar el operador `[]` de acceso a los elementos de un vector, como es el caso de C#.

Para acceder a los datos de la sesión de usuario actual desde una página ASP.NET, no tenemos más que utilizar la propiedad `Session` de la clase que implementa la página ASP.NET. La propiedad `Session` está definida en la clase base `System.Web.UI.Page` y, como todas las páginas ASP.NET derivan de esta clase base, desde cualquier página ASP.NET se puede acceder directamente a la propiedad heredada `Session`.

Por ejemplo, en la página de entrada a nuestra aplicación web podríamos encontrarnos algo como lo siguiente:

```
void Page_Load (Object Src, EventArgs e)
{
  Session["UserName"] = TextBoxUser.Text;
}
```

Una vez establecido un valor para nombre del usuario correspondiente a la sesión actual, podemos utilizar este valor para personalizar la presentación de las páginas interiores de la aplicación:

```
labelUser.Text = (string) Session["UserName"];
```

Lo único que nos falta por ver es cómo se pueden inicializar los valores de las colecciones `Session` y `Application`. Para ello hemos de utilizar algunos de los eventos definidos en `Global.asax.cs`, un fichero opcional en el que se incluye el código destinado a responder a eventos globales relacionados con una aplicación ASP.NET. En dicho fichero se define una clase que hereda de `System.Web.HttpApplication` y en la que se pueden implementar los métodos `Session_Start` y `Application_Start`. Estos métodos se invocan cuando comienza una sesión de usuario y cuando arranca la aplicación web, respectivamente, por lo que es en ellos donde se deben inicializar las colecciones `Session` y `Application`. Aunque no resulte de especial utilidad, podríamos incluir algo como lo siguiente en el fichero `Global.asax.cs`:

```
void Session_Start()
{
  Session["UserName"] = "";
}
```

La colección `Session` se puede considerar como una variable global compartida por todas las solicitudes HTTP provenientes de un mismo usuario. Para saber a qué sesión corresponde una solicitud HTTP concreta se utiliza un identificador de sesión. Dicho identificador se genera automáticamente cuando el cliente accede por primera vez a la aplicación web y se transmite desde el cliente cada vez que éste vuelve a acceder a cualquier página de la aplicación web.

El identificador de la sesión es un número aleatorio de 120 bits que se codifica como una cadena de caracteres ASCII, del estilo de `cqcgvjvjirmizirpld0dyi5`, para que ningún usuario malintencionado pueda obtener información útil a partir de él. Dicho identificador se

puede transmitir mediante una cookie o incrustado en la URL de la solicitud, en función de cómo se configure la aplicación. Por defecto, el identificador de la sesión (que no los datos asociados a la sesión) se transmite utilizando la cookie `ASP.NET_SessionId`. Cuando no se emplean cookies, el identificador de la sesión aparecerá como parte de la URL a la que se accede:

```
http://servidor/aplicación/(uqwkag45e35fp455t2qav155)/página.aspx
```

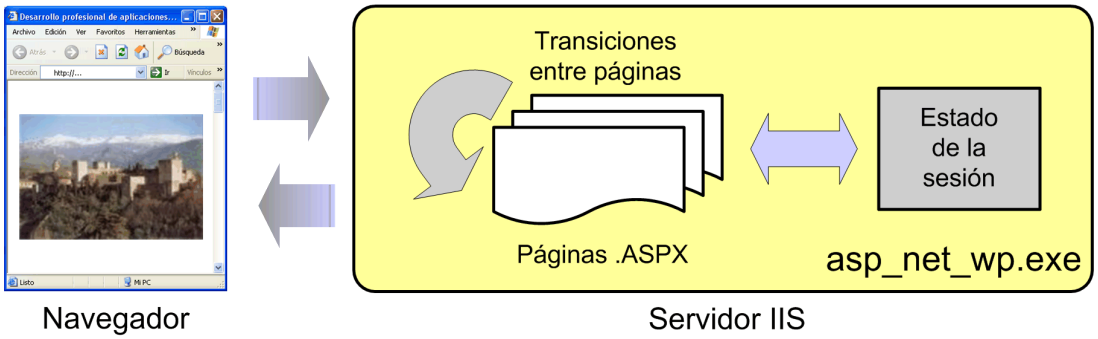
Para seleccionar la forma en la que deseamos transmitir el identificador de la sesión, lo único que tenemos que hacer es modificar el valor del atributo `cookieless` del elemento `sessionState` que aparece en el fichero de configuración de nuestra aplicación web, un fichero XML llamado `Web.config`. Por defecto, este atributo tiene el valor `"false"`, que indica que se utilizará la cookie `ASP.NET_SessionId` para almacenar el identificador de la sesión de usuario:

```
<sessionState
    mode="InProc"
    cookieless="false"
    timeout="30"
/>
```

Simplemente poniendo `cookieless="true"` podemos evitar el uso de cookies en ASP.NET. Lo que es aún mejor, el fichero `Web.config` permite configurar la forma en la que se almacenan los datos correspondientes a las sesiones de los usuarios de nuestra aplicación. Sólo tenemos que jugar un poco con la sección `<sessionState ... />` del fichero `Web.config`. A nuestra disposición tenemos distintos mecanismos para almacenar los datos relativos a las sesiones de usuario. La elección de uno u otro dependerá básicamente del entorno en el que deba funcionar nuestra aplicación.

Por defecto, los datos correspondientes a la sesión del usuario se almacenan en el proceso del servidor que se encarga de ejecutar las páginas ASP.NET, el proceso `aspnet_wp.exe`. Éste es el mecanismo utilizado por defecto, conocido como `InProc:`, y es el que aparece en la sección `<sessionState ... />` del fichero `Web.config` mostrado anteriormente.

Cuando una aplicación web tiene muchos usuarios y un simple ordenador no es capaz de atenderlos a todos a la vez, lo usual es crear un cluster o granja de servidores entre los cuales se reparte la carga de la aplicación. El reparto de la carga se suele realizar dinámicamente, por lo que las peticiones de un usuario concreto no siempre las atiende el mismo servidor. Por tanto, el estado de las sesiones no puede almacenarse `InProc`, sino que ha de centralizarse en algún sitio accesible desde cualquier servidor del cluster.

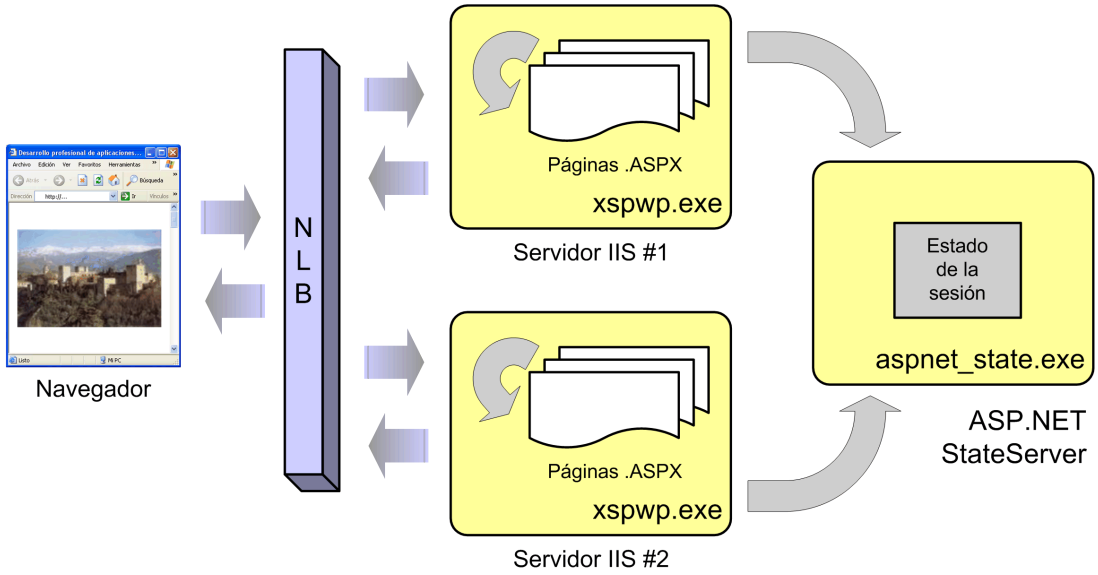


InProc: Los datos correspondientes a las sesiones de usuario se almacenan en la memoria del proceso encargado de ejecutar las páginas ASP.NET.

ASP.NET nos permite disponer de un proceso encargado de mantener el estado de las distintas sesiones de usuario e independiente de los procesos encargados de atender las peticiones ASP.NET. Dicho proceso (`aspnet_state.exe`) es accesible desde cualquiera de los servidores web y permite que las solicitudes HTTP puedan enviarse independientemente a cualquiera de los servidores disponibles, independientemente de dónde provengan. Para utilizar este proceso independiente, conocido como "servidor de estado", hemos de especificar el modo `StateServer` en la sección `<sessionState ... />` del fichero `Web.config` e indicar cuál es la cadena de conexión que permite acceder al servidor de estado. La cadena de conexión consiste, básicamente, en el nombre de la máquina en la que se esté ejecutando dicho servidor de estado y el puerto TCP a través del cual se puede acceder a él:

```
<sessionState mode="StateServer"
    stateConnectionString="tcpip=csharp.ikor.org:42424"
    cookieless="false"
    timeout="30"
/>
```

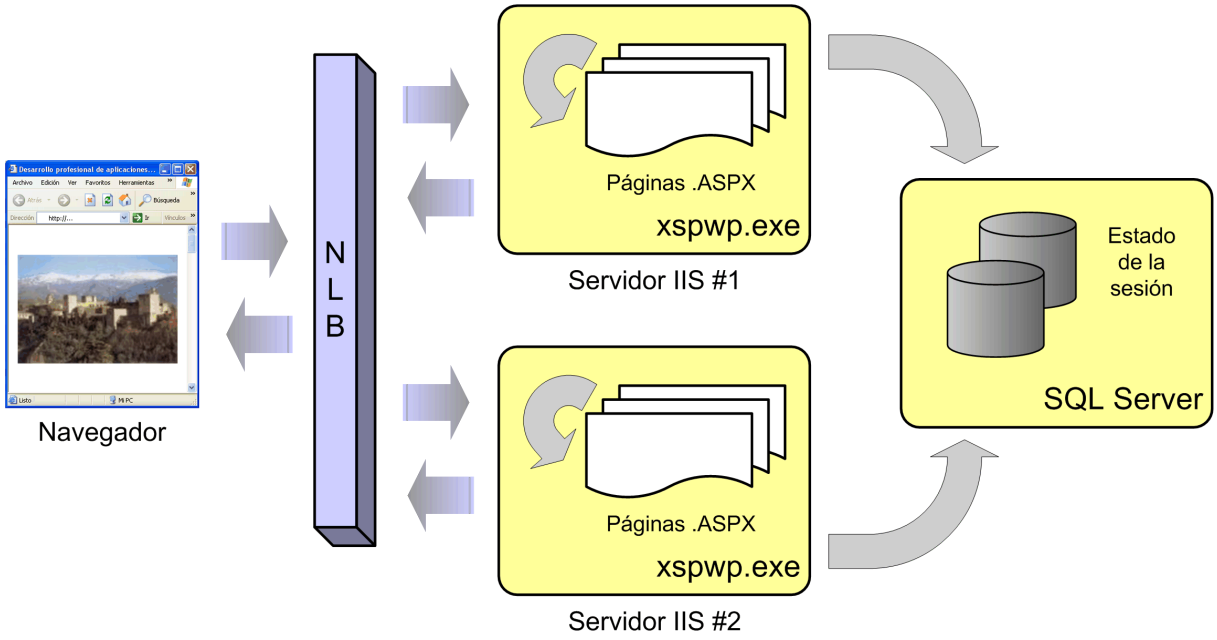
Por último, ASP.NET también nos permite que los datos correspondientes a las sesiones de usuario se almacenen de forma permanente en una base de datos relacional (SQL*Server, como no podía ser de otra forma). Esto permite que el estado de cada sesión se mantenga incluso cuando falle alguna de las máquinas de la granja de servidores (siempre y cuando no falle la base de datos, claro está), lo que permite a una aplicación ASP.NET convencional aspirar al mítico 24x7, funcionar 24 horas al día durante 7 días a la semana sin interrupción alguna.



StateServer: Despliegue de una aplicación web ASP.NET en una granja de servidores. El dispositivo NLB [Network Load Balancer] se encarga de repartir la carga entre varios servidores, mientras que los datos de las sesiones de usuario se almacenan en el servidor de estado.

Para utilizar una base de datos SQL como soporte para el mantenimiento de las sesiones de usuario, lo primero que tenemos que hacer es crear la base de datos en el servidor SQL Server que se vaya a utilizar con este fin. Para ello, no tenemos más que ejecutar una de las macros suministradas, `InstallSqlState.sql` para utilizar una base de datos "en memoria" (`TempDB`) o `InstallPersisSqlState.sql` para crear una base de datos (`ASPState`) que sobreviva ante posibles caídas del SQL Server. A continuación, ya en el fichero `Web.config`, pondremos algo similar a lo siguiente:

```
<sessionState mode="SQLServer"
    sqlConnectionString=
      "data source=csharp.ikor.org;Integrated Security=SSPI;"
    cookieless="false"
    timeout="30"
/>
```



*SqlServer: Uso de una base de datos SQL*Server para almacenar los datos correspondientes a las sesiones de usuario.*

Las facilidades ofrecidas por ASP.NET a través de la modificación del fichero de configuración `Web.config` nos permiten desplegar nuestras aplicaciones ASP.NET en diversos entornos sin tener que modificar una sola línea de código. En el caso de que el éxito de nuestra aplicación lo haga necesario, podemos migrar nuestra aplicación de un servidor a un cluster sin forzar la afinidad de un cliente a un servidor concreto, con lo que se mejora la tolerancia a fallos del sistema. Además, el uso de una base de datos de apoyo nos permite mantener el estado de las sesiones aun cuando se produzca una caída completa del sistema. La selección de una u otra alternativa dependerá del compromiso que deseemos alcanzar entre la eficiencia y la tolerancia a fallos de nuestra aplicación web en lo que se refiere al mantenimiento de las sesiones de usuario.

> Cuando utilizamos un servidor de estado fuera del proceso encargado de atender las peticiones, en los modos `StateServer` o `SQLServer`, los datos almacenados en las sesiones del usuario han de transmitirse de un proceso a otro, por lo que han de ser serializables. Esto es, las clases correspondientes a los objetos almacenados han de estar marcadas con el atributo `[Serializable]`.

Seguridad en ASP.NET

A la hora de construir una aplicación web, y especialmente si la aplicación se construye con fines comerciales, es imprescindible que seamos capaces de seguir los pasos de cada usuario. Además, también suele ser necesario controlar qué tipo de acciones puede realizar un usuario concreto. De lo primero nos podemos encargar utilizando mecanismos de control de sesiones de usuario como los vistos en el apartado anterior. De lo segundo nos ocuparemos a continuación.

Aunque la seguridad es, por lo general, un aspecto obviado por la mayor parte de los programadores a la hora de construir aplicaciones, en un entorno web resulta esencial porque cualquiera con una conexión a la red puede acceder a nuestras aplicaciones. Planificar los mecanismos necesarios para evitar accesos no autorizados a nuestras aplicaciones y servicios web se convierte, por tanto, en algo que todo programador debería saber hacer correctamente.

Cuando hablamos de seguridad en las aplicaciones web realizadas bajo la plataforma .NET, en realidad nos estamos refiriendo a cómo restringir el acceso a determinados recursos de nuestras aplicaciones. Estos recursos los gestiona el Internet Information Server (IIS) de Microsoft, por lo que la seguridad en ASP.NET es un aspecto más relacionado con la correcta configuración del servidor web que con la programación de la aplicación en sí.

Sin entrar en demasiados detalles (que seguro son de interés para los aficionados a las técnicas criptográficas de protección de datos), en las siguientes páginas veremos cómo se pueden implementar mecanismos de identificación de usuarios a través de contraseñas para controlar el acceso a aplicaciones web desarrolladas con páginas ASP.NET. Posteriormente, comentaremos cómo podemos controlar las acciones que el usuario puede realizar en el servidor, para terminar con una breve descripción del uso de transmisiones seguras en la Web.

Autentificación y autorización

Cuando queremos controlar el acceso a una aplicación web, lo normal es que el usuario se identifique de alguna forma. Por regla general, esta identificación se realiza utilizando un nombre de usuario único y una contraseña que el usuario ha de mantener secreta. En la aplicación web, esto se traduce en que el usuario es automáticamente redirigido a un formulario de *login* cuando intenta acceder a un área restringida de la aplicación.

En el fichero de configuración `Web.config`, que ya ha aparecido mencionado en varias ocasiones a lo largo de este capítulo, se incluyen dos secciones relacionadas directamente con la autentificación y la autorización de usuarios. Su aspecto en una aplicación real suele ser similar al siguiente:

```
<configuration>
  <system.web>

    <authentication mode="Forms">
      <forms loginUrl="login.aspx" name=".ASPXFORMSAUTH"></forms>
    </authentication>

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

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

La autentificación consiste en establecer la identidad de la persona que intenta acceder a la aplicación, lo que se suele realizar a través de un formulario de *login*. La autorización consiste en determinar si el usuario, ya identificado, tiene permiso para acceder a un determinado recurso.

Autentificación

La sección de autentificación del fichero `Web.config`, delimitada por la etiqueta `<authentication>`, se utiliza para establecer la política de identificación de usuarios que utilizará nuestra aplicación. ASP.NET permite emplear distintos modos de autentificación, entre los que se encuentran los siguientes:

- `Forms` se emplea para utilizar formularios de autentificación en los que seremos nosotros los que decidamos quién accede a nuestra aplicación.

- `Passport` permite que nuestra aplicación utilice el sistema de autentificación Passport de Microsoft (más información en http://www.passport.com).

- `Windows` se utiliza para delegar en el sistema operativo las tareas de autentificación de usuarios, con lo cual sólo podrán acceder a nuestra aplicación los usuarios que existan previamente en nuestro sistema (por ejemplo, los usuarios de un dominio).

- Finalmente, `None` deshabilita los mecanismos de autentificación, con lo que cualquiera puede acceder a ella desde cualquier lugar del mundo sin restricción alguna.

Cuando seleccionamos el modo de autentificación `Forms`, hemos de indicar también cuál será el formulario encargado de identificar a los usuarios de la aplicación. En el ejemplo anterior, ese formulario es `login.aspx`. Un poco más adelante veremos cómo se puede crear dicho formulario.

Autorización

Después de la sección de autentificación, en el fichero `Web.config` aparece la sección de autorización, delimitada por la etiqueta `<authorization>`. En el ejemplo anterior, esta sección se utiliza, simplemente, para restringir el acceso a los usuarios no identificados, de forma que sólo los usuarios autentificados puedan usar la aplicación web.

Las secciones de autentificación y autorización del fichero `Web.config` restringen el acceso a un directorio y a todos sus subdirectorios en la aplicación web. No obstante, en los subdirectorios se pueden incluir otros ficheros `Web.config` que redefinan las restricciones de acceso a los subdirectorios de nuestra aplicación web. Piense, si no, que sería necesaria una aplicación web independiente para permitir que usuarios nuevos se registrasen en nuestra aplicación web.

Por ejemplo, si en una parte de nuestra aplicación queremos que cualquier persona pueda acceder, incluso sin identificarse, basta con incluir la siguiente autorización en el fichero de configuración adecuado:

```
<authorization>
  <allow users="*" />
</authorization>
```

Esto nos permite tener aplicaciones en las que haya partes públicas y partes privadas, como sucede en cualquier aplicación de comercio electrónico. En ellas, los usuarios pueden navegar libremente por el catálogo de productos ofertados pero han de identificarse al efectuar sus compras.

Si lo que quisiéramos es restringir el acceso a usuarios o grupos de usuarios particulares, podemos hacerlo incluyendo una sección de autorización similar a la siguiente en nuestro fichero `Web.config`:

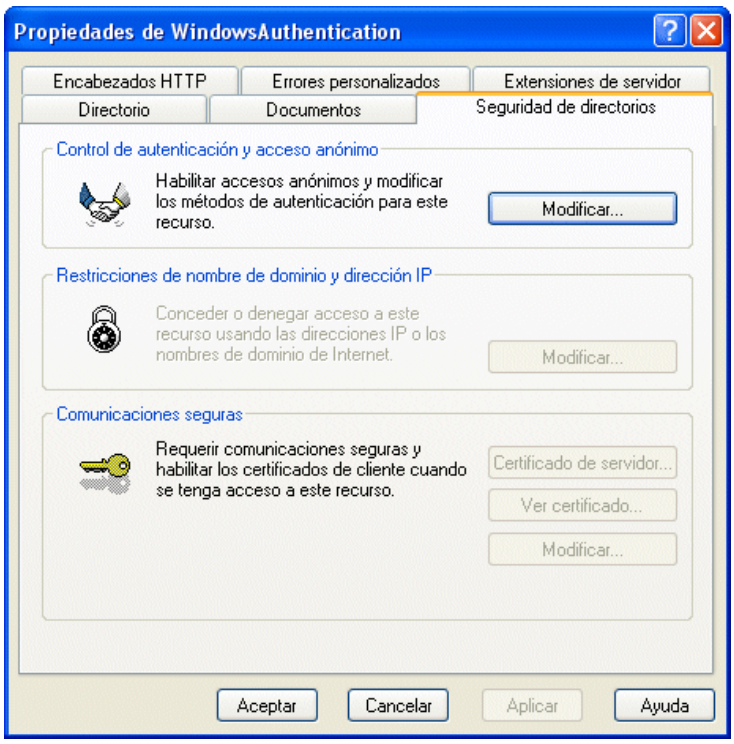
```
<authorization>
  <deny users="*" />
  <allow users="administrador,director" />
</authorization>
```

Autentificación en Windows

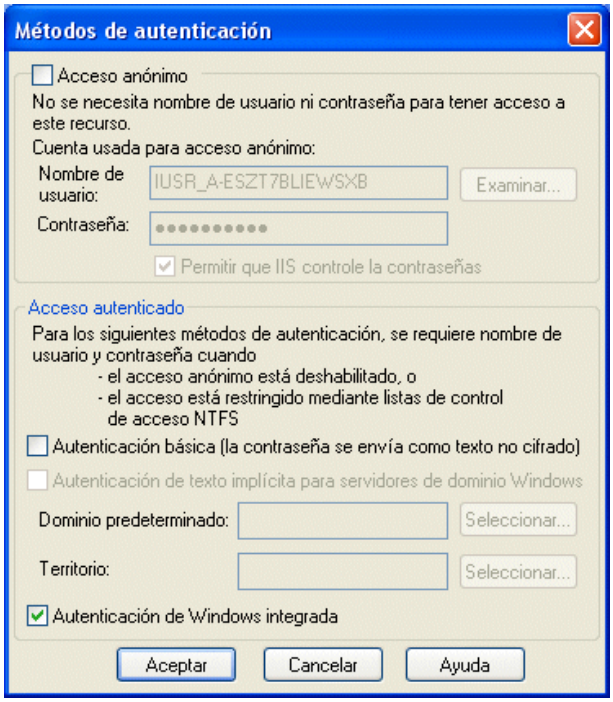
Una vez visto cómo podemos controlar qué usuarios acceden a qué partes de la aplicación, volvamos ahora al problema inicial: ¿cómo identificar a los usuarios en un primer momento?.

Una de las alternativas que nos ofrece ASP.NET es utilizar el sistema operativo Windows como mecanismo de autentificación. Es decir, los nombres de usuario y las claves de acceso para nuestra aplicación web serán los mismos nombres de usuarios y claves que se utilizan para acceder a nuestros ordenadores.

Para utilizar este mecanismo de autentificación, debemos especificar `Windows` como modo de autentificación en el fichero `Web.config`. Además, deberemos eliminar el acceso anónimo a nuestra aplicación desde el exterior. Para lograrlo, hemos de cambiar las propiedades del directorio de nuestra aplicación en el Internet Information Server:



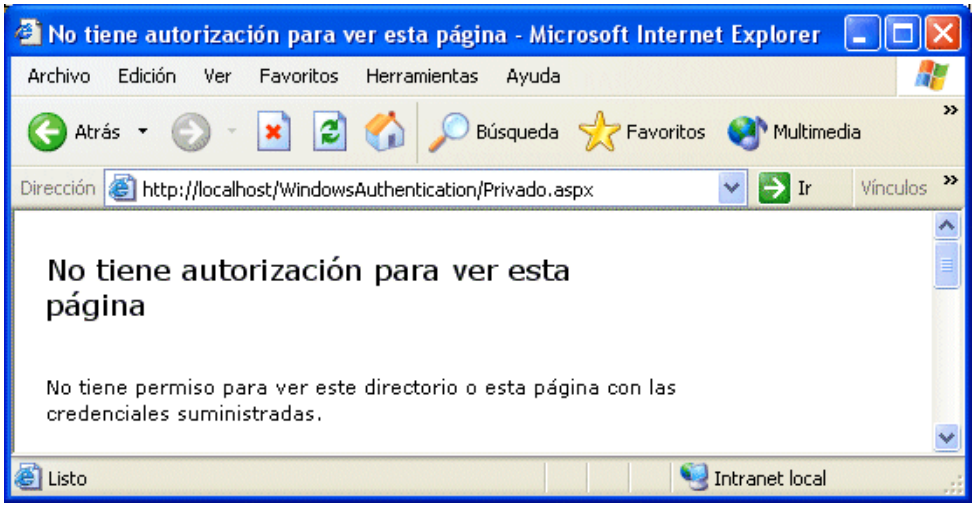
Una vez dentro de las propiedades relacionadas con el control de "autenticación" y acceso anónimo, deshabilitamos el acceso anónimo:



Una vez realizadas las modificaciones pertinentes en las secciones de autentificación y autorización del fichero `Web.config` y deshabilitado el acceso anónimo al directorio de nuestra aplicación, cuando intentamos acceder a la aplicación nos debe aparece una ventana como la siguiente:

Esta ventana nos pide que introduzcamos un nombre de usuario y su contraseña. El nombre de usuario ha de existir en nuestro sistema operativo y la contraseña ha de ser la misma que

utilizamos para acceder al ordenador. Si tras varios intentos no somos capaces de introducir un nombre de usuario válido y su contraseña correspondiente, el servidor web nos devolverá un error de autentificación `"HTTP 401.3 - Access denied by ACL on resource"`:



Este error de autentificación, "acceso denegado por lista de control de acceso", se debe a que el nombre de usuario introducido no está incluido en la lista de usuarios que están autorizados expresamente para acceder a la aplicación web, a los cuales deberemos incluir en la sección `authorization` del fichero de configuración `Web.config`.

Formularios de autentificación en ASP.NET

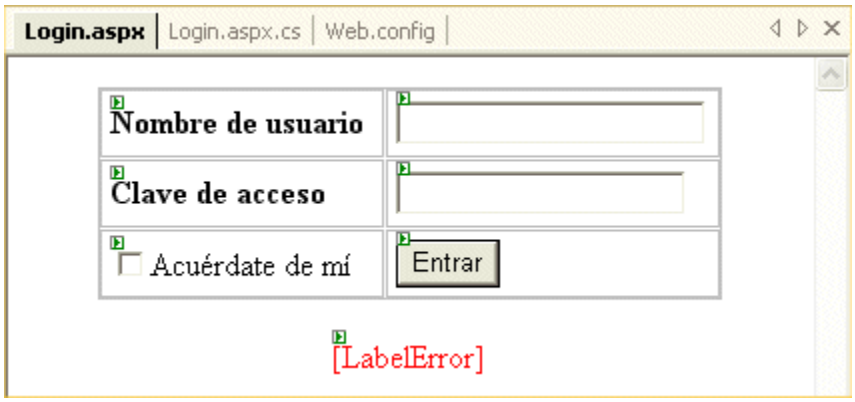
En determinadas ocasiones, no nos podremos permitir el lujo de crear un usuario en el sistema operativo para cada usuario que deba acceder a nuestra aplicación. Es más, posiblemente no nos interese hacerlo. Probablemente deseemos ser nosotros los encargados de gestionar los usuarios de nuestra aplicación y controlar el acceso de éstos a las distintas partes de nuestro sistema.

El modo de autentificación `Forms` es el más indicado en esta situación. Para poder utilizarlo, debemos configurar correctamente el fichero `Web.config`, tal como se muestra a continuación:

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

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

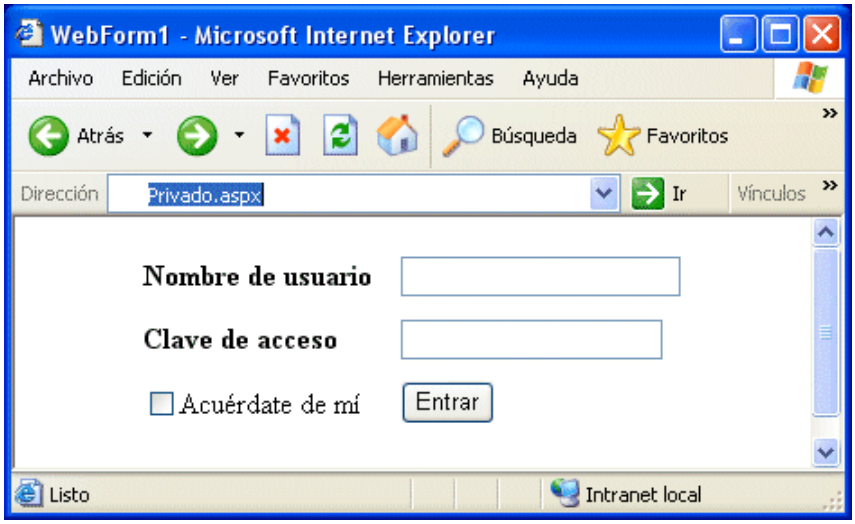
Cuando un usuario no identificado intente acceder a una página cuyo acceso requiera su identificación, lo que haremos será redirigirlo a un formulario específico de *login*, `Login.aspx`. Dicho formulario, al menos, debe incluir dos campos para que el usuario pueda indicar su nombre y su clave:



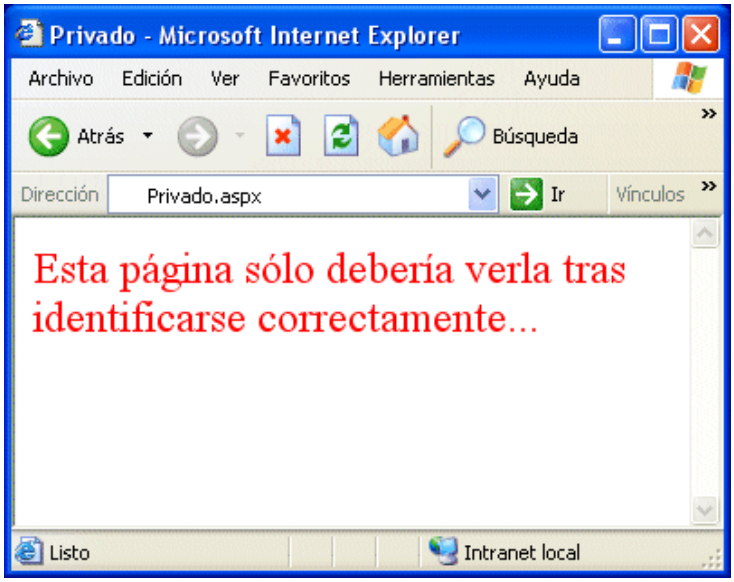
El formulario de identificación incluirá, por tanto, dos controles de tipo `TextBox`. Dado que la contraseña del usuario ha de mantenerse secreta, en el `TextBox` correspondiente a la clave de acceso se ha de especificar la propiedad `TextMode=Password`. El formulario, en sí, lo crearemos igual que cualquier otro formulario. Lo único que tendremos que hacer es comprobar nombre y contraseña. Esta comprobación se ha de realizar de la siguiente forma:

```
if (  textBoxID.Text.Equals("usuario")
   && textBoxPassword.Text.Equals("clave") ) {

   FormsAuthentication.RedirectFromLoginPage(textBoxID.Text,false);

} else {

   // Error de autentificación...
}
```

Cuando un usuario no identificado intenta acceder a cualquiera de los formularios de nuestra aplicación, el usuario es redirigido al formulario de identificación:

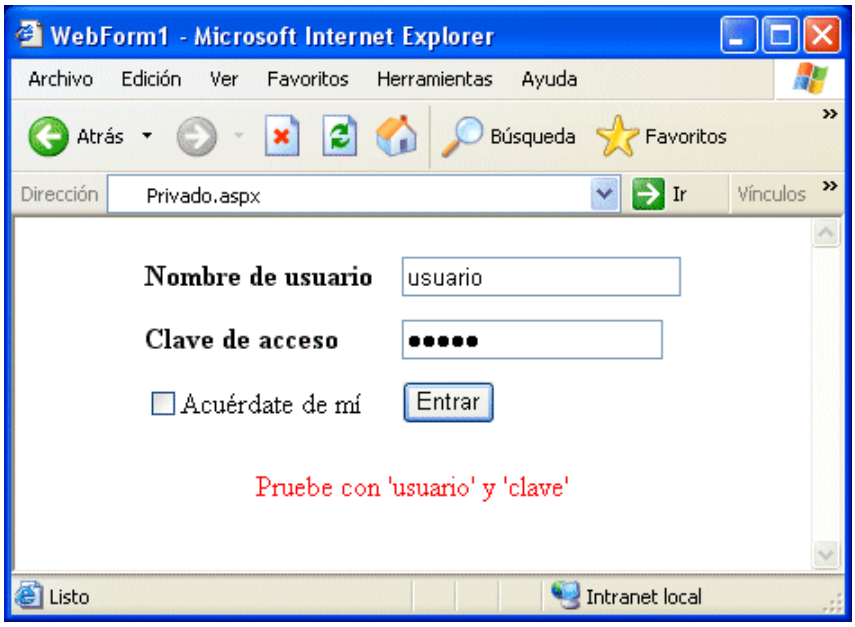


Una vez que el usuario se identifica correctamente, la llamada al método `RedirectFromLoginPage` le indica al IIS que debe darle permiso al usuario para acceder a la página a la que inicialmente intentó llegar. Este método pertenece a la clase `FormsAuthentication`, la cual está incluida en el espacio de nombres `System.Web.Security`. El IIS se encargará de todo lo demás por nosotros y mandará al usuario a la página adecuada:



Sólo cuando nos identifiquemos correctamente deberíamos poder acceder a la aplicación.

Si el usuario no se identifica correctamente, ya sea porque no esté dado de alta o porque no haya introducido correctamente su clave de acceso, se debe mostrar un mensaje informativo de error:



Obviamente, en una aplicación real no deberíamos ser tan explícitos. Cuando el usuario se equivoque en su clave de acceso o su identificador no exista en nuestra base de datos de usuarios registrados, el mensaje de error se le puede mostrar utilizando una etiqueta (`labelMessage.Text=...`), tal y como acabamos de hacer. También podemos redirigir al usuario a otra página mediante `Response.Redirect("http://...");`. Esto último suele ser lo más adecuado si nuestro sistema permite que nuevos usuarios se den de alta ellos mismos.

Permisos en el servidor

Las aplicaciones ASP.NET no suelen residir en un mundo aislado, sino que suelen interactuar con otras aplicaciones y acceder a distintas bases de datos y ficheros.

El proceso que se encarga de ejecutar las páginas ASP.NET (`aspnet_wp.exe`) se ejecuta utilizando una cuenta de usuario llamada ASPNET. Por tanto, si una página ASP.NET ha de acceder a un recurso concreto, se le han de dar los permisos adecuados al usuario ASPNET. Por ejemplo, si la aplicación web ha de escribir datos en un fichero concreto, el usuario ASPNET ha de tener permiso de escritura sobre dicho fichero.

Este modelo de seguridad es bastante sencillo y simplifica bastante el trabajo del administrador del sistema, ya que sólo ha de incluir al usuario ASPNET en las listas de control de acceso de los recursos a los que las aplicaciones web deban tener acceso. Esto es, independientemente de los usuarios que luego utilicen la aplicación, el administrador no tendrá que ir dándole permisos a cada uno de esos usuarios. Además, este modelo tiene otras implicaciones relacionadas con la eficiencia de las aplicaciones web. Si se han de establecer conexiones con una base de datos, todas las conexiones las realiza el mismo usuario, por lo que se puede emplear un *pool* de conexiones, el cual permite compartir recursos de forma eficiente entre varios usuarios de la aplicación (con la consiguiente mejora en su escalabilidad).

Para que las acciones de la aplicación web se realicen con los privilegios del usuario ASPNET, debemos asegurarnos de que el fichero de configuración `Web.config` incluye la siguiente línea:

```
<identity impersonate="false"/>
```

Otra alternativa, disponible cuando la aplicación ASP.NET utiliza el modo de autentificación `Windows`, consiste en utilizar las credenciales de los usuarios de la aplicación. En este caso, el administrador del sistema deberá dar los permisos necesarios a cada uno de los usuarios finales de la aplicación. Aunque resulte más "incómoda" su labor, esta opción es más flexible, ya que permite auditar las acciones que realiza cada usuario y evita los problemas de seguridad que podrían surgir si el servidor web se viese comprometido. Para que nuestra aplicación web funcione según este modelo de seguridad, el fichero `Web.config` deberá incluir las siguientes líneas:

```
<authentication mode="Windows"/>
...
<identity impersonate="true"/>
```

En cualquiera de los dos modelos de seguridad comentados, siempre se debe aplicar el "principio de menor privilegio": darle el menor número posible de permisos a los usuarios del sistema y ocultarles las partes de la aplicación responsables de realizar tareas para las que no tengan autorización. Además, dado el entorno potencialmente hostil en el que ha de funcionar una aplicación web, la interfaz externa de la aplicación ha de ser especialmente cuidadosa a la hora de permitir la realización de cualquier tipo de acción y no realizar ninguna suposición. Toda entidad externa ha de considerarse insegura, por lo que cualquier entrada ha de validarse exhaustivamente antes de procesarse.

Seguridad en la transmisión de datos

Los apartados anteriores han descrito cómo podemos controlar el acceso a las distintas partes de una aplicación y cómo podemos establecer permisos que impidan que un usuario realice acciones para las que no tiene autorización (ya sea malintencionadamente o, simplemente, por error). No obstante, no hemos dicho nada acerca de otro aspecto esencial a la hora de construir aplicaciones seguras: el uso de técnicas criptográficas de protección de datos que protejan los datos que se transmiten entre el cliente y el servidor durante la ejecución de una aplicación web.

Para estos menesteres, siempre se deben utilizadas soluciones basadas en algoritmos cuya seguridad haya sido demostrada, los cuales suelen tener una base matemática que garantiza su inviolabilidad usando la tecnología actual. De hecho, no es una buena idea crear técnicas a medida para nuestras aplicaciones. Aparte de que su diseño nos distraería del objetivo principal de nuestro proyecto (entregar, en un tiempo razonable, la funcionalidad que el cliente requiera), no conviene mezclar los detalles de una aplicación con aspectos ortogonales como puede ser el uso de técnicas seguras de transmisión de datos. Estas técnicas son las que nos permiten garantizar la privacidad y la integridad de los datos transmitidos. Su uso resulta indispensable, y puede que legalmente obligatorio, cuando nuestra aplicación ha de trabajar con datos "sensibles", tales como números de tarjetas de crédito o historiales médicos.

En el caso concreto de las aplicaciones web, que en el cliente se suelen ejecutar desde navegadores web estándar, lo normal es usar la infraestructura que protocolos como HTTPS nos ofrece. El protocolo HTTPS, un HTTP seguro, está basado en el uso de SSL [*Secure Sockets Layer*], un estándar que permite la transmisión segura de datos. El protocolo HTTP manda los datos tal cual, sin encriptar, por lo que cualquier persona que tenga una conexión a cualquiera de las redes por donde se transmiten los datos desde el cliente hasta el servidor podría leerlos. HTTPS nos permite proteger los datos que se transmiten entre el cliente y el servidor. Para utilizar este protocolo, lo único que debemos hacer es instalar un certificado en el IIS.

HTTPS utiliza técnicas criptográficas de clave pública para proteger los datos transmitidos y que sólo el destinatario pueda leerlos. Estas técnicas consisten en utilizar pares de claves emitidas por autoridades de certificación. Estos pares están formados por una clave pública y una clave privada, de tal forma que lo codificado con la clave pública sólo puede leerse si se dispone de la clave privada y viceversa. Si queremos que sólo el destinatario sea capaz de leer los datos que le enviamos, lo único que tenemos que hacer es codificarlos utilizando su clave pública. Como sólo él dispone de su clave privada, sólo él podrá leer los datos que le hayamos enviado. El certificado que hemos de instalar en el IIS no es más que un par clave pública - clave privada emitido por alguna entidad, usualmente externa para evitar que alguien pueda suplantar al servidor.

Aparte de las técnicas criptográficas que impiden que los datos se puedan leer, HTTPS también añade a cada mensaje un código de autentificación HMAC (*Keyed-Hashing for*

Message Authentication, RFC 2104). Este código sirve para que, al recibir los datos, podamos garantizar su integridad. Si alguien manipula el mensaje durante su transmisión, el código HMAC no corresponderá al mensaje recibido.

El uso de estándares como HTTPS permite que nuestras aplicaciones web puedan emplear mecanismos seguros de comunicación de forma transparente. Lo único que tendremos que hacer es configurar adecuadamente nuestro servidor web y hacer que el usuario acceda a nuestras aplicaciones web utilizando URLs de la forma `https://...`. Sólo en casos muy puntuales tendremos que preocuparnos directamente de aspectos relacionados con la transmisión segura de datos. Y no será en nuestras interfaces web, que de eso ya se encarga el IIS.

HTTPS no es la única opción disponible para garantizar la transmisión segura de datos, aunque sí la más usada. existe una variante del protocolo IP usado en Internet, conocido como IPSec, que permite añadir cabeceras de autentificación a los datagramas IP, además de encriptar el contenido del mensaje. Este protocolo, utilizado en "modo túnel", permite la creación de redes privadas virtuales (VPNs). Estas redes son muy útiles cuando una organización tiene varias sedes que han de acceder a sistemas de información internos. Las VPNs permiten que varias redes separadas geográficamente funcionen como si de una única red de área local se tratase. Para ello, pueden alquilar líneas privadas a empresas de telecomunicaciones (como los cajeros automáticos de los bancos) o establecer conexiones seguras a través de Internet entre las distintas redes físicas ("túneles").