Desarrollo Profesional de Aplicaciones Web con ASP.NET

Fernando Berzal
Francisco José Cortijo
Juan Carlos Cubero

ISBN 84-609-4245-7

ASP.NET

Hoy en día, resulta bastante común implementar la interfaz de una aplicación utilizando páginas web en vez de las ventanas y los controles específicos de un sistema operativo concreto. En lugar de escribir una aplicación para un sistema operativo concreto, como puede ser Windows, en muchas situaciones es preferible crear aplicaciones web a las que se accede a través de Internet.

Se denominan aplicaciones web a aquellas aplicaciones cuya interfaz se construye a partir de páginas web. Las páginas web no son más que ficheros de texto en un formato estándar denominado HTML [*HyperText Markup Language*]. Estos ficheros se almacenan en un servidor web al cual se accede utilizando el protocolo HTTP [*HyperText Transfer Protocol*], uno de los protocolos de Internet. Para utilizar una aplicación web desde una máquina concreta, basta con tener instalado un navegador web en esa máquina, ya sea éste el Internet Explorer de Microsoft, el Netscape Navigator o cualquier otro navegador. Desde la máquina cliente, donde se ejecuta el navegador, se accede a través de la red al servidor web donde está alojada la aplicación y, de esa forma, se puede utilizar la aplicación sin que el usuario tenga que instalarla previamente en su máquina.

Si las páginas que forman la interfaz de nuestra aplicación las construimos utilizando única y exclusivamente HTML estándar, podemos conseguir que nuestra aplicación funcione sobre prácticamente cualquier plataforma, siempre y cuando dispongamos de un navegador web para el sistema operativo instalado en la máquina desde la que queramos acceder a la aplicación. Una interfaz construida de esta manera nos permite olvidarnos de los detalles específicos de los diferentes entornos gráficos existentes. Dichos entornos gráficos suelen

depender del sistema operativo que utilicemos. Por ejemplo, en Windows utilizaremos las ventanas típicas a las que ya estamos acostumbrados, mientras que en UNIX emplearíamos el sistema X Windows y en Linux podríamos utilizar gestores de ventanas como KDE o Gnome.

Como no podía ser de otra forma, el aumento de la portabilidad de nuestra aplicación lleva consigo ciertas limitaciones en lo que respecta a su usabilidad. Esto se debe a que la interacción del usuario con el ordenador a través de los formularios HTML empleados en las páginas web está mucho más limitada que en el caso de que podamos aprovechar la funcionalidad que ofrecen los controles específicos existentes en la mayoría de los entornos gráficos basados en ventanas.

En el desarrollo de la interfaz de nuestra aplicación podríamos evitar, al menos en parte, la solución de compromiso que supone utilizar formularios HTML para conseguir aplicaciones portables. Existen bibliotecas portables para el desarrollo de interfaces gráficas, como es el caso de *SWING* en la plataforma Java de Sun Microsystems, o de productos como *Qt* (Troll Tech, http://www.troll.no). En el caso de la plataforma Java, ésta se diseñó como una máquina virtual para que pudieran ejecutarse las mismas aplicaciones en distintos sistemas operativos sin tener ni siquiera que recompilar el código. Por su parte, las bibliotecas del estilo de *Qt* ofrecen conjuntos de componentes con implementaciones de los mismos para distintos entornos basados en ventanas. En una de estas bibliotecas de componentes (o tal vez deberíamos decir bibliotecas de interfaces), las distintas implementaciones de un componente particular tienen los mismos interfaces para todas y cada una de las plataformas sobre las que esté implementada la biblioteca. De este modo, si deseamos portar nuestras aplicaciones de un sistema a otro sólo tendremos que recompilar nuestra aplicación utilizando la versión de la biblioteca adecuada para el sistema operativo sobre el que queremos ejecutar la aplicación, sin tener que modificar ni una sola línea en nuestro código fuente. Dicho código fuente, obviamente, lo tendremos que escribir evitando cualquier uso de recursos específicos de una plataforma (algo, de por sí, extremadamente difícil).

Aunque pueda parecer que soluciones como las descritas en el párrafo anterior resuelven nuestros problemas a la hora de desarrollar aplicaciones portables sin limitaciones en cuanto a la funcionalidad de su interfaz, en realidad se limitan a resolver una pequeña parte del problema que supone instalar y mantener un sistema funcionando. De hecho, la etapa del ciclo de vida del software que más recursos consume es la fase de mantenimiento.

El mantenimiento del software consume del cuarenta al ochenta por ciento del coste de un sistema y, probablemente, sea la etapa más importante del ciclo de vida del software (aparte de ser la más olvidada). No debemos pensar que el mantenimiento sólo consiste en reparar aquello que deja de funcionar por un defecto de fabricación o por el desgaste asociado al paso del tiempo. La naturaleza del software hace que nunca se rompa por desgaste. De todos los gastos asociados al mantenimiento del software (que viene a suponer un 60% de su coste total), un 60% corresponde a la realización de mejoras, mientras que sólo un 17% está directamente asociado a la corrección de defectos. Por tanto, el 60% del 60% del coste total del software está asociado a la realización de actualizaciones. Si nuestra aplicación ha de funcionar en muchos puestos diferentes, cada pequeña actualización que realicemos supondrá un esfuerzo considerable a la hora de su distribución e instalación. Y si nuestro sistema tiene

éxito, habrá muchas de esas actualizaciones.

Teniendo los hechos anteriores en cuenta, resulta evidente el atractivo que tiene la implementación de aplicaciones web, puesto que se elimina el problema de la distribución de actualizaciones cada vez que modificamos un sistema. Cuando tengamos que realizar una actualización, nos bastará con modificar la configuración del servidor que da acceso a nuestra aplicación web para que todos los clientes dispongan automáticamente de la versión más reciente de la aplicación. Además, el usuario podrá acceder a nuestra aplicación desde cualquier plataforma y nosotros podremos implementarla utilizando todos los recursos disponibles en la plataforma que elijamos para albergarla, sin restricciones innecesarias (si bien esto no quiere decir que no aspiremos a la construcción de una aplicación flexible que nos facilite el trabajo si algún día tenemos que cambiar de servidor de aplicaciones).

Dada la importancia actual de las aplicaciones web, el siguiente capítulo ofrece una introducción al desarrollo de interfaces web para ayudarnos a comprender en qué consisten y cómo se construyen las aplicaciones de este tipo. En los capítulos posteriores de esta parte del libro veremos con más profundidad algunos de los detalles de la construcción de interfaces web con ASP.NET, la tecnología de la plataforma .NET que nos permite desarrollar aplicaciones web.

Referencias

Los datos que hemos mencionado relacionados con el mantenimiento del software se analizan en el libro de Robert L. Glass titulado *"Facts and fallacies of Software Engineering"* (Addison-Wesley, 2003, ISBN 0-321-11742-5). Este libro es una obra interesante en la que se puede consultar información adicional acerca de cuestiones fundamentales que a menudo se nos olvidan cuando estamos enfrascados en el desarrollo de software.

Interfaces web

En este capítulo aprenderemos en qué consisten las aplicaciones web y mencionaremos algunas de las herramientas que los programadores tenemos a nuestra disposición para construir este tipo de aplicaciones. Para ser más específicos, en las siguientes secciones trataremos los temas que aparecen a continuación:

- En primer lugar, comenzaremos presentando la evolución histórica de las aplicaciones web para comprender cómo se ha llegado a su arquitectura actual desde los modestos comienzos de las páginas web.

- A continuación, pasaremos a describir las principales alternativas de las que dispone el programador para construir sus aplicaciones web.

- Finalmente, comentaremos las soluciones que oferta Microsoft para el desarrollo de interfaces web y empezaremos a ver cómo se construyen aplicaciones web en la plataforma .NET utilizando páginas ASP.NET.

Interfaces web

Evolución de las aplicaciones web

Como dijimos en la introducción a esta parte del libro, las aplicaciones web son aquellas cuya interfaz se construye utilizando páginas web. Dichas páginas son documentos de texto a los que se les añaden etiquetas que nos permiten visualizar el texto de distintas formas y establecer enlaces entre una página y otra.

La capacidad de enlazar un texto con otro para crear un hipertexto es la característica más destacable de las páginas web. Aunque su éxito es relativamente reciente, sus orígenes se remontan al sistema Memex ideado por Vannevar Bush (*"As we may think"*, Atlantic Monthly, julio de 1945). El término hipertexto lo acuñó Ted Nelson en 1965 para hacer referencia a una colección de documentos (nodos) con referencias cruzadas (enlaces), la cual podría explorarse con la ayuda de un programa interactivo (navegador) que nos permitiese movernos fácilmente de un documento a otro.

De hecho, la versión que conocemos actualmente del hipertexto proviene del interés de los científicos en compartir sus documentos y hacer referencias a otros documentos. Este interés propició la creación de la "tela de araña mundial" (*World-Wide Web*, WWW) en el Centro Europeo para la Investigación Nuclear (CERN). Tim Berners-Lee, uno de los científicos que trabajaba allí, ideó el formato HTML para representar documentos con enlaces a otros documentos. Dicho formato fue posteriormente establecido como estándar por el W3C (*World-Wide Web Consortium*, http://www.w3c.org/), el organismo creado por el MIT que fija los estándares utilizados en la web desde 1994.

HTML estático

Inicialmente, las páginas web se limitaban a contener documentos almacenados en formato HTML [*HyperText Markup Language*]. Dichos documentos no son más que ficheros de texto a los que se le añaden una serie de etiquetas. Dichas etiquetas delimitan fragmentos del texto que han de aparecer en un formato determinado y también sirven para crear enlaces de un documento a otro (o, incluso, de una parte de un documento a otra parte del mismo documento). Al final de este capítulo puede encontrar una pequeña introducción al formato HTML para refrescar sus conocimientos o aprender a escribir sus propias páginas web.

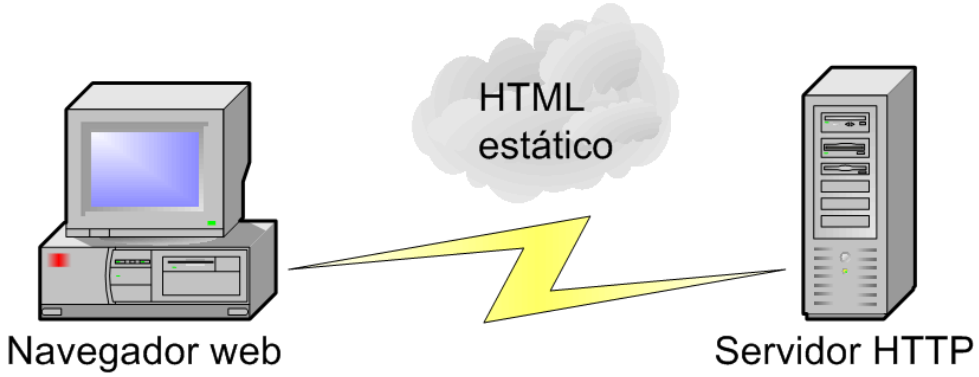
Con unos conocimientos mínimos de HTML, crear un sitio web resulta relativamente sencillo. Sólo hay que preparar los documentos HTML tal y como queramos que los visualicen los visitantes de nuestra página. Cuando podemos predecir con antelación cuál es la información que tenemos que mostrarle al usuario, crear una página web estática resulta la opción más sencilla. Incluso cuando el contenido de nuestra página web ha de cambiar periódicamente, en ocasiones es suficiente con escribir pequeños programas que generen los documentos HTML

a los que accederá el visitante de nuestra página. Este es el caso, por ejemplo, de las páginas web utilizadas por medios de comunicación como periódicos. Cada día, o incluso más a menudo, una aplicación se encarga de generar los documentos HTML con el formato visual más adecuado para nuestro sitio web. Dichos documentos HTML quedan almacenados de forma permanente en ficheros y el usuario accede a ellos directamente.

En realidad, el usuario no accede directamente a los ficheros que contienen los documentos HTML, sino que utiliza un navegador para visualizarlos cómodamente. Dicho navegador es, en realidad, una aplicación cliente que utiliza el protocolo HTTP [*HyperText Transfer Protocol*] para acceder a la máquina en la que hayamos alojado nuestros ficheros en formato HTML. Por tanto, para que los usuarios puedan acceder a nuestra página web, sólo necesitaremos un servidor web que atienda las peticiones HTTP generadas por el navegador web del usuario. En respuesta a esas peticiones, el servidor HTTP que utilicemos le enviará al navegador los documentos que haya solicitado. Cuando nuestro servidor se limita a servir documentos HTML previamente preparados, podemos utilizar cualquier servidor HTTP de los existentes, como el Internet Information Server de Microsoft o el Apache (el cual se puede descargar gratuitamente de http://www.apache.org/).

Aparte del servidor HTTP que hemos de tener funcionando en la máquina que sirva los documentos HTML, nos hará falta disponer de una dirección IP fija para la máquina donde alojemos el servidor HTTP. Dicha dirección resulta imprescindible para que el usuario pueda saber dónde ha de conectarse para obtener los documentos que le interesan. De hecho, la mayoría de los usuario ni siquiera son conscientes de que cada vez que acceden a un sitio web están utilizando una dirección IP a modo de "número de teléfono", sino que escriben en la barra de direcciones de su navegador una URL [*Uniform Resource Locator*] que identifica unívocamente el recurso en Internet al que desea acceder y suele incluir un nombre de dominio. Dicho nombre de dominio es una cadena de texto fácil de recordar que el servicio de nombres DNS [*Domain Name Service*] traduce a la dirección IP necesaria para establecer con éxito una conexión con el servidor. En consecuencia, resulta aconsejable (y no demasiado caro) reservar un nombre de dominio que asociaremos a la dirección IP de la máquina donde instalemos nuestro servidor HTTP. Sólo así podrá el usuario escribir una URL de la forma http://csharp.ikor.org/ para acceder cómodamente a los datos que nosotros previamente habremos dejado a su disposición en nuestro servidor web.

Si bien esta forma de construir un sitio web puede ser suficiente para muchas aplicaciones, en ocasiones necesitaremos que el contenido de nuestra página web se genere dinámicamente en función de las necesidades y deseos de nuestros usuarios. Cuando nos interesa algo más que mostrar siempre los mismos datos de la misma forma, tener que actualizar periódicamente los ficheros HTML no siempre es una buena idea. El inconveniente de utilizar simples ficheros HTML es que estos ficheros son estáticos y, mientras no los actualicemos de forma manual o automática, mostrarán siempre la misma información independientemente de quién acceda a nuestra página. Por ejemplo, si lo que queremos es construir una página web cuyo contenido cambie en función del usuario la visite y de la tarea que desee realizar, la solución pasa ineludiblemente por generar dinámicamente los documentos HTML cada vez que le llega una solicitud a nuestro servidor HTTP. Éste es el principio de funcionamiento de las aplicaciones web que describiremos en el siguiente apartado.



HTML estático: Configuración típica de una "aplicación web" que se limita a ofrecer la información almacenada en páginas HTML a las que el usuario final accede desde su navegador web utilizando el protocolo HTTP.

Aplicaciones web

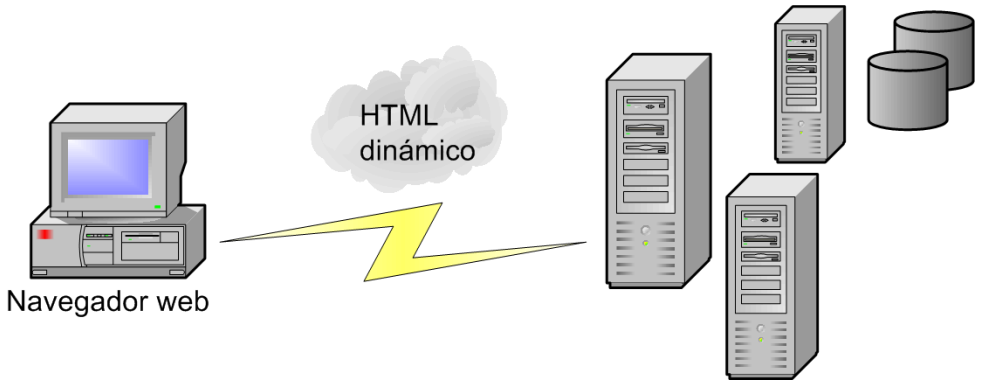
Aunque la utilización de documentos HTML estáticos puede ser la solución más adecuada cuando nuestra página web se limite a ofrecer siempre la misma información o podamos automatizar la realización de actualizaciones de los documentos HTML que la constituyen, la naturaleza dinámica de la web y las expectativas que ha creado en la actualidad hacen necesaria la implementación de aplicaciones web que generen dinámicamente el contenido que finalmente se les ofrece a los usuarios. De esta forma podemos seleccionar, filtrar, ordenar y presentar la información de la forma más adecuada en función de las necesidades de cada momento. Si bien esto se podría conseguir con páginas HTML estáticas si dispusiésemos de espacio suficiente en disco (y, de hecho, esta es una estrategia que se utiliza para disminuir la carga de la CPU de los servidores), las aplicaciones web nos permiten ofrecer la información más actual de la que disponemos al poder acceder directamente a las bases de datos que contienen los datos operativos de una empresa.

La creación de aplicaciones web, en consecuencia, requiere la existencia de software ejecutándose en el servidor que genere automáticamente los ficheros HTML que se visualizan en el navegador del usuario. Exactamente igual que cuando utilizábamos páginas estáticas en formato HTML, la comunicación entre el cliente y el servidor se sigue realizando a través del protocolo HTTP. La única diferencia consiste en que, ahora, el servidor HTTP delega en otros módulos la generación dinámica de las páginas HTML que se envían al cliente. Ya que, desde el punto de vista del cliente, la conexión se realiza de la misma forma y él sigue recibiendo páginas HTML estándar (aunque éstas hayan sido generadas dinámicamente en el servidor), el navegador del cliente es independiente de la tecnología que se utilice en el servidor para generar dichas páginas de forma dinámica.

Desde el punto de vista del programador, existe una amplia gama de herramientas a su disposición. Para generar dinámicamente el contenido que se le ofrece al usuario, puede optar por desarrollar software que se ejecute en el servidor o, incluso, en la propia máquina del usuario. Algunas de las opciones entre las que puede elegir el programador serán comentadas en las siguientes secciones y una de ellas será estudiada con mayor detalle en éste y los siguientes capítulos: las páginas ASP.NET incluidas en la plataforma .NET.

Básicamente, las distintas alternativas disponibles para el desarrollo de aplicaciones web ofrecen la misma funcionalidad. No obstante, en función de las necesidades de cada proyecto y de su envergadura algunas resultarán más adecuadas que otras. Igual que en cualquier otro aspecto relacionado con el desarrollo de software, no existen "balas de plata" y cada tecnología ofrece una serie de facilidades que habremos de estudiar en función de lo que tengamos que hacer. Por ejemplo, el protocolo HTTP es un protocolo simple en el que se establece una conexión TCP independiente para cada solicitud del cliente. Esto es, cada vez que el usuario accede a un fichero de nuestro servidor (o, lo que es lo mismo, a una página generada dinámicamente), lo hace de forma independiente. Por tanto, la herramienta que utilicemos para crear nuestra aplicación web debería facilitarnos el mantenimiento de sesiones de usuario (conjuntos de conexiones independientes relacionadas desde el punto de vista lógico).

En resumen, independientemente de la forma en que implementemos nuestra aplicación web, el navegador del cliente es independiente de la tecnología que se utilice en el servidor, ya que a él sólo le llegará una página HTML estándar que mostrará tal cual. En la siguiente sección repasaremos las formas más comunes de desarrollar aplicaciones web. Usualmente, las páginas web que se le muestran al usuario se generan dinámicamente en el servidor, si bien también se puede introducir cierto comportamiento dinámico en el navegador del cliente a costa de perder parte de la independencia entre el navegador y nuestra aplicación web.



Aplicaciones web: El contenido que se le muestra al usuario se genera dinámicamente para cada solicitud proveniente del navegador web instalado en la máquina cliente.

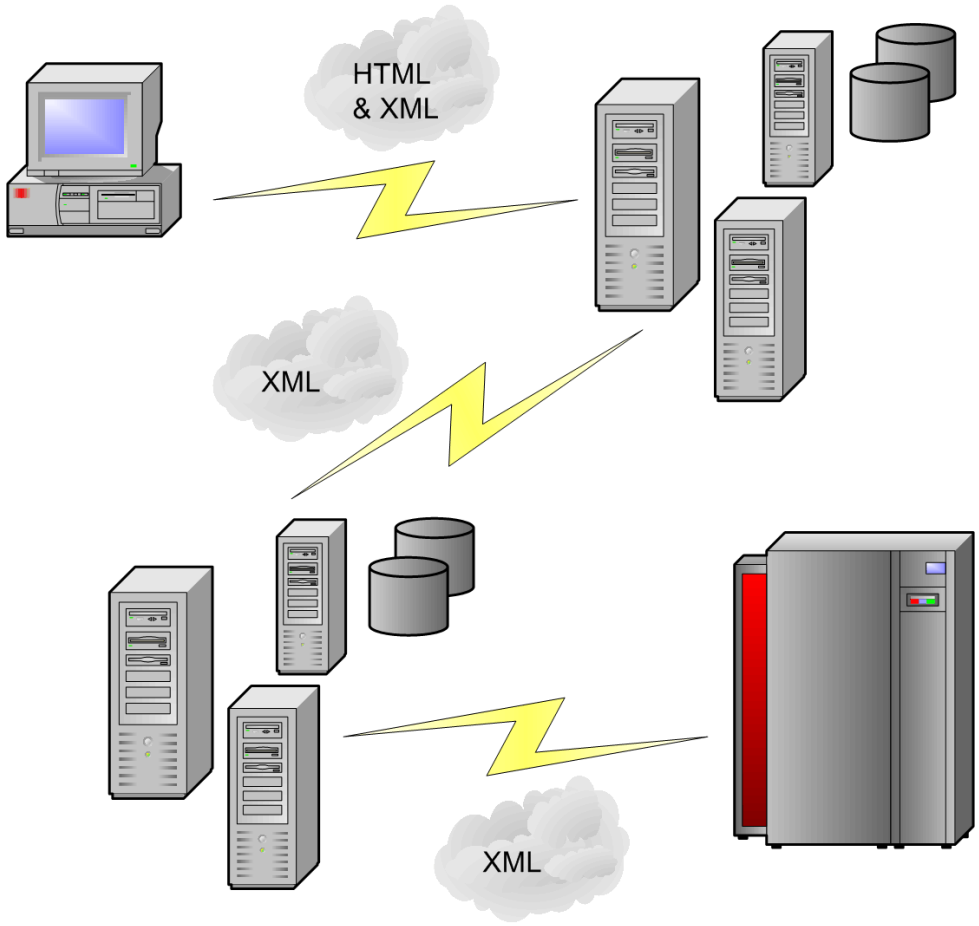
Antes de analizar las distintas herramientas y tecnologías que nos facilitan el desarrollo de aplicaciones web, para terminar de repasar la evolución de las páginas web desde sus comienzos estáticos hasta la actualidad, en el siguiente apartado comentaremos brevemente cuál es la configuración que suelen tener las aplicaciones web más recientes.

Servicios web

Las aplicaciones web han sufrido una evolución análoga a la que ya padecieron las aplicaciones de escritorio que utilizan los recursos propios de cada sistema operativo para construir su interfaz de usuario. Inicialmente, estas aplicaciones se ejecutaban en una única máquina, que era además la máquina donde se almacenaban los datos que manipulaban. Posteriormente, se hicieron populares las arquitecturas cliente/servidor, en las que la interfaz de usuario de las aplicaciones de gestión se ejecuta en la máquina del cliente pero los datos se suelen almacenar en un sistema gestor de bases de datos. La aplicación cliente se conecta al sistema gestor de bases de datos de forma similar a como el navegador web accede al servidor HTTP en una aplicación web como las descritas en el apartado anterior. Finalmente, para determinadas aplicaciones de gestión se han impuesto las arquitecturas multicapa y el uso de *middleware* (por ejemplo, CORBA). En estas aplicaciones, la máquina cliente sólo ejecuta la interfaz de usuario y la lógica de la aplicación se ejecuta en un servidor de aplicaciones independiente tanto de la interfaz de usuario como de la base de datos donde se almacenan los datos.

Las aplicaciones web sólo se distinguen de las aplicaciones de escritorio tradicionales en que, en vez de implementar la interfaz de usuario utilizando un lenguaje particular como C/C++ o Java, se utilizan páginas web como punto de acceso a las aplicaciones. Por consiguiente, no es de extrañar que también se construyan aplicaciones web multicapa. Dichas aplicaciones construyen su interfaz utilizando formularios HTML, implementan su lógica en sistemas distribuidos y suelen almacenar sus datos en sistemas gestores de bases de datos relacionales.

De hecho, en el caso de las aplicaciones web incluso se han propuesto estándares que utilizan los mismos protocolos que las aplicaciones web cliente/servidor como canal de comunicación entre las distintas partes de una aplicación distribuida. Este es el caso de los servicios web, que intercambian mensajes en formato XML utilizando protocolos de transporte como HTTP. Los servicios web, básicamente, establecen un lenguaje común mediante el cual distintos sistemas puedan comunicarse entre sí y, de esta forma, facilitan la construcción de sistemas distribuidos heterogéneos. Dada su importancia actual, les dedicaremos a ellos un capítulo completo en la siguiente parte de este libro. Por ahora, nos centraremos en la construcción de interfaces web. Más adelante ya veremos cómo encajan todas las piezas en la construcción de arquitecturas software.



Servicios web: La lógica de la aplicación se distribuye. El intercambio de mensajes en formato XML y el uso de protocolos estándares de Internet nos permiten mantener conectadas las distintas partes de una aplicación, aunque ésta haya de funcionar en un sistema distribuido heterogéneo.

Desarrollo de aplicaciones para Internet

Como hemos ido viendo en las páginas anteriores, actualmente se observa una tendencia muy definida que fomenta la utilización los estándares de Internet para desarrollar aplicaciones de gestión en medianas y grandes empresas. Si centramos nuestra atención en el desarrollo del interfaz de usuario de estas aplicaciones, lo que encontramos es un uso extensivo de los estándares abiertos utilizados en la web, aquéllos promovidos por el W3C, si bien es cierto que también se utilizan algunas tecnologías propietarias.

La característica común que comparten todas las aplicaciones web es el hecho de centralizar el software para facilitar las tareas de mantenimiento y actualización de grandes sistemas. Es decir, se evita tener copias de nuestras aplicaciones en todos los puestos de trabajo, lo que puede llegar a convertir en una pesadilla a la hora de distribuir actualizaciones y garantizar que todos los puestos de trabajo funcionen correctamente. Cada vez que un usuario desea acceder a la aplicación web, éste se conecta a un servidor donde se aloja la aplicación. De esta forma, la actualización de una aplicación es prácticamente trivial. Simplemente se reemplaza la versión antigua por la versión nueva en el servidor. A partir de ese momento, todo el mundo utiliza la versión más reciente de la aplicación sin tener que realizar más esfuerzo que el de adaptarse a los cambios que se hayan podido producir en su interfaz.

Aunque todas las aplicaciones web se diseñen con la misma filosofía, existen numerosas alternativas a la hora de implementarlas en la práctica. A grandes rasgos, podemos diferenciar dos grandes grupos de aplicaciones web en función de si la lógica de la aplicación se ejecuta en el cliente o en el servidor. Esta distinción nos permitirá, en los dos próximos apartados, organizar un recorrido sobre las tecnologías existentes para el desarrollo de interfaces web.

> En realidad, casi todas las aplicaciones web reales utilizan tecnologías tanto del lado del cliente como del lado del servidor. Utilizar unas u otras en una cuestión de diseño que habrá de resolverse en función de lo que resulte más adecuado para satisfacer las necesidades particulares de cada aplicación.

En el cliente

En principio, todo el software asociado a una aplicación web se puede desarrollar de forma que el trabajo lo realice el servidor y el navegador instalado en la máquina cliente se limite a mostrar páginas HTML generadas en el servidor. De esta forma, al usuario de la aplicación web le basta con tener instalado cualquier navegador web. Esta estrategia es la que resulta

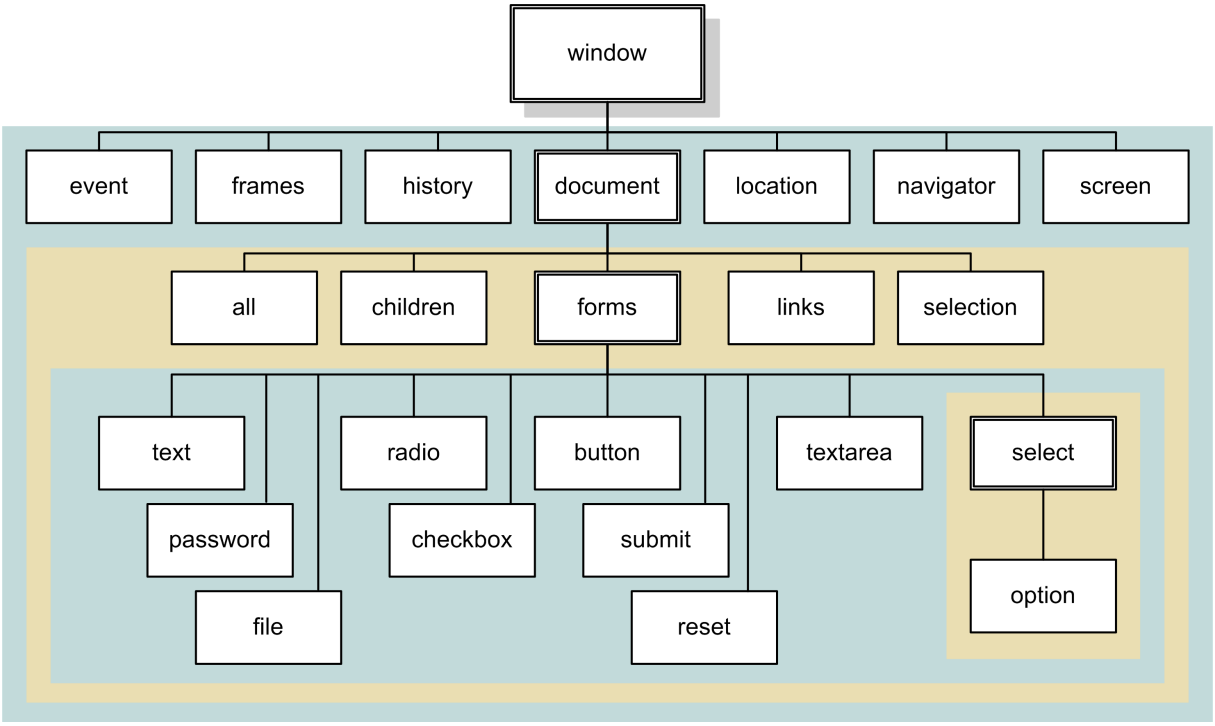
más cómoda para el programador. Sin embargo, desde el punto de vista del usuario final de la aplicación, esta opción no resulta demasiado atractiva. Las limitaciones de los formularios estándar de HTML hacen difícil, si no imposible, construir interfaces de usuario amigables restringiéndonos al estándar.

Las limitaciones del formato HTML para construir interfaces de usuario (algo para lo que nunca fue diseñado) ha propiciado la aparición de numerosas tecnologías que permiten ejecutar código en la máquina del cliente, generalmente dentro del propio navegador web. Con estas tecnologías se consigue mejorar la escalabilidad de las aplicaciones, ya que se realiza menos trabajo en el servidor y éste puede atender a más clientes. Por otro lado, se mejora tanto la productividad como la satisfacción del usuario final, al emplear interfaces de usuario más depuradas y fáciles de manejar. Finalmente, estas tecnologías permiten conseguir aplicaciones muy atractivas desde el punto de vista estético, algo a lo que los programadores no suelen prestar demasiada atención pero que resulta de vital importancia a la hora de vender el producto, ya sea una aplicación a medida para un cliente particular o un sistema que se pone en el mercado a disposición de todo aquél que quiera usarlo desde la comodidad de su casa. Al fin y al cabo, ¿hace cuánto tiempo que no ve una aplicación web nueva en la que no se haya incluido una animación o presentación espectacular en Flash para agradar al cliente? Y eso que la dichosa presentación suele tardar un buen rato en descargarse y sólo supone una pérdida de tiempo para los usuarios habituales de la aplicación web (la cual, en demasiadas ocasiones, ni siquiera funciona correctamente).

A continuación mencionaremos algunas de las herramientas y tecnologías que se suelen utilizar para ejecutar parte de la aplicación web en la máquina del propio cliente:

HTML dinámico y JavaScript

Sin lugar a dudas, la herramienta más utilizada a la hora de dotar a nuestras páginas HTML de cierto comportamiento dinámico. El HTML dinámico (DHTML) se basa en construir un modelo basado en objetos del documento HTML, de forma que podamos acceder fácilmente a los distintos elementos que lo componen (véase la figura). La modificación dinámica de la página HTML se realiza a través de pequeñas macros o *scripts* que suelen incluirse en el mismo fichero que la página, si bien siempre es conveniente intentar separarlas del HTML para no mezclar los detalles del HTML de la interfaz con la lógica que implementan dichas macros.



DHTML DOM [Document Object Model]: Facilita la creación de páginas web dinámicas al ofrecer una forma cómoda de acceder a los distintos elementos que componen una página web.

En HTML dinámico, cada etiqueta HTML se convierte en un objeto con sus propiedades y eventos asociados. Los scripts han de proporcionarle al navegador el código correspondiente a la respuesta prevista por el programador para los distintos eventos que se pueden producir. Esto es, las macros se ejecutan cuando se produce algún evento asociado a alguno de los elementos de la página web de modo análogo a como se programa en cualquier entorno de programación visual para construir interfaces de usuario.

Usualmente, las macros se escriben utilizando JavaScript por cuestiones de portabilidad, si bien navegadores web como el Internet Explorer de Microsoft también permiten otros lenguajes como VBScript [Visual BASIC Script]. En realidad, aunque existe un estándar oficial de JavaScript ratificado por ECMA (por lo que se le suele llamar ECMAScript), cada navegador implementa versiones sutilmente diferentes de JavaScript, con los consiguientes dolores de cabeza que esto conlleva para el programador. Pese a ello, JavaScript resulta una opción atractiva ya que no resulta difícil encontrar en Internet bibliotecas gratuitas de ejemplos que funcionan en los navegadores web más comunes (desde los típicos menús desplegables, banners, relojes y calendarios hasta juegos de ajedrez).

JavaScript es un lenguaje interpretado originalmente llamado LiveScript que Netscape desarrolló para sus productos relacionados con la web. De hecho, JavaScript funciona tanto en navegadores web como en el servidor HTTP de Netscape, al más puro estilo de las páginas ASP de Microsoft.

La sintaxis de JavaScript muy similar a la de Java y resulta fácil de aprender para cualquiera que tenga unos conocimientos básicos de C. Por ejemplo, para declarar una variable no hay que especificar su tipo; basta con asignarle un valor que podemos obtener de alguno de los elementos de nuestra página web (el primer elemento del primer formulario, por ejemplo):

```
var dato = document.forms[0].elements[0].value;
```

Inicialmente, sólo el navegador de Netscape soportaba JavaScript, si bien Microsoft no tardó en incorporar una versión ligeramente modificada de JavaScript denominada JScript (cuando Netscape acaparaba el mercado de los navegadores web y Microsoft era un aspirante). Las resultantes inconsistencias hacen difícil escribir código que funcione correctamente en ambos navegadores, si bien Microsoft ha ganado la batalla y ahora son los demás los que tienen que intentar que sus navegadores interpreten el HTML dinámico de la forma que lo hace el de Microsoft.

Si bien Netscape y Sun Microsystems se han beneficiado mutuamente de su cooperación para facilitar el intercambio de mensajes y datos entre Java y JavaScript, JavaScript es independiente de Java. JavaScript es hoy un estándar abierto, ratificado por ECMA igual que el lenguaje C#, mientras que Java es propiedad de Sun Microsystems.

Controles ActiveX

Otra de las tecnologías que se puede utilizar para implementar parte de las aplicaciones web en el lado del cliente está basada en el uso de controles ActiveX como los que se utilizan en el desarrollo de aplicaciones para Windows. Los controles ActiveX están construidos sobre COM [*Component Object Model*], el modelo de Microsoft para desarrollo de componentes anterior a la plataforma .NET. A diferencia de JavaScript, que es un lenguaje totalmente interpretado, los controles ActiveX se compilan previamente, lo que permite su ejecución más eficiente.

Al ser una tecnología específica de Microsoft, la inclusión de controles ActiveX en páginas web sólo funciona correctamente en el navegador web de Microsoft, el Internet Explorer. Dado su fuerte acoplamiento con los productos de Microsoft, su utilización se suele limitar a las aplicaciones web para intranets. Las intranets constituyen un entorno más controlado que Internet al estar bajo el control de una única organización, por lo que uno puede permitirse el

lujo de que su aplicación web no sea realmente portable.

En cierta medida, se puede decir que los controles ActiveX fueron la primera respuesta de Microsoft a los applets de Java promovidos por Sun Microsystems. La segunda, mucho más ambiciosa, fue la creación de la plataforma .NET.

Applets

Los applets son aplicaciones escritas en Java que se ejecutan en el navegador web. A diferencia de las macros interpretadas de JavaScript, un applet de Java es una aplicación completa compilada para la máquina virtual de Java (similar a la máquina virtual de la plataforma .NET). Los applets se adjuntan a las páginas web y pueden ejecutarse en cualquier navegador que tenga instalada una máquina virtual Java (básicamente, un intérprete del código intermedio que genera el compilador de Java).

Si bien su utilización se puede haber visto limitada por algunos problemas de rendimiento (la ejecución de un applet es, en principio, más lenta que la de un control ActiveX equivalente), los litigios existentes entre Microsoft y Sun Microsystems han ocasionado que su uso se haya realizado con cautela en la construcción de aplicaciones web. A pesar del eslogan de Java, "escribe una vez, ejecuta en cualquier sitio", la existencia de distintos navegadores y de sus sutiles diferencias restan atractivo a la construcción de interfaces web a base de applets. De hecho, el uso de Java está más extendido en el servidor, donde es muy común implementar las aplicaciones con servlets y páginas JSP.

Cuando se utiliza un applet, se descarga del servidor web el código intermedio del applet correspondiente a la máquina virtual Java; esto es, sus *bytecodes*. Al no tener que difundir el código fuente de la aplicación y disponer de una plataforma completa para el desarrollo de aplicaciones, se suelen preferir los applets para las partes más complejas de una aplicación web mientras que se limita el uso de JavaScript a pequeñas mejoras estéticas de los formularios HTML.

Igual que JavaScript, los applets tienen la ventaja de funcionar sobre cualquier navegador que se precie (Netscape Navigator, Internet Explorer de Microsoft o HotJava de Sun). Además de por su portabilidad, garantizada con que exista un intérprete de bytecodes para la máquina virtual Java, los applets destacan por su seguridad: cada aplicación se ejecuta en un espacio independiente [sandbox] que, en principio, no puede acceder al hardware de la máquina del cliente (salvo que éste, explícitamente, lo autorice).

Plug-ins específicos

Los navegadores web pueden extenderse con *plug-ins*. Los *plug-ins* son componentes que permiten alterar, mejorar o modificar la ejecución de una aplicación en la que se instalan. Por ejemplo, los navegadores web suelen incluir *plug-ins* para visualizar documentos en formato PDF (de Adobe), ejecutar presentaciones Flash (de Macromedia), escuchar sonidos

RealAudio, mostrar imágenes vectoriales en formato SVG [Scalable Vector Graphics], ejecutar applets escritos para la máquina virtual Java o recorrer mundos virtuales VRML [Virtual Reality Markup Language].

Para que nuestra página web incluya un fichero para el cual necesitemos un plug-in específico, basta con incluir una etiqueta `EMBED` en el HTML de la página web. Los plug-ins, usualmente, son gratuitos. Basta con descargar de Internet la versión adecuada para nuestro sistema operativo e instalarla una única vez, tras lo cual queda almacenada localmente y podemos utilizarla cuantas veces necesitemos.

Existen distintos productos que nos permiten construir aplicaciones web utilizando plug-ins, como es el caso de **Curl**, un curioso lenguaje comercializado por un spin-off del MIT. Curl mejora la capacidad de HTML a la hora de construir interfaces de usuario sin salirse del navegador web. Más que por sus características (Curl viene a ser una especie de LaTeX para el desarrollo de interfaces), este producto resulta curioso por el modelo de negocio sobre el que espera prosperar. En vez de pagar los gastos fijos de una licencia para instalar Curl en el servidor web, el que lo utiliza ha de pagar en función de la cantidad de datos que se transmiten comprimidos desde el servidor. Igual que en el caso de los controles ActiveX, esta tecnología puede que tenga éxito en el desarrollo de aplicaciones web para intranets, en situaciones en las que el ancho de banda utilizado por las aplicaciones web tradicionales pueda llegar a ser un obstáculo para su implantación.

En el servidor

Las tecnologías específicas descritas en la sección anterior permiten, fundamentalmente, mejorar la interfaz de usuario de nuestras aplicaciones web en el navegador del cliente, al menos en cierta medida. Independientemente de que utilicemos o no dichas tecnologías, la funcionalidad de las aplicaciones web usualmente la tendremos que implementar en el lado del servidor.

Para construir aplicaciones web que se ejecuten en el servidor disponemos, si cabe, de más alternativas que en el cliente. Además, ya que la aplicación se ejecutará en el servidor, no necesitamos tener ningún plug-in instalado en la máquina del cliente y, en muchas ocasiones, nos bastará con utilizar el HTML dinámico disponible en los navegadores web actuales.

Las aplicaciones que se ejecutan en el servidor pueden recibir información del cliente de distintas formas. Por ejemplo, los datos recogidos por un formulario HTML pueden enviarse al servidor codificados en la propia URL (el identificador unívoco que aparece en la barra de direcciones del navegador) o enviarse en la cabecera del mensaje HTTP que se envía automáticamente cuando el usuario pulsa un botón del formulario. Además, las aplicaciones web pueden almacenar pequeñas cadenas de texto en el navegador web del cliente para realizar tareas como el mantenimiento de sesiones del usuario (las famosas *cookies*).

Generalmente, las herramientas que utilicemos nos simplificar el acceso a los datos facilitados desde el cliente. Una vez que se adquieren estos datos, la aplicación web ha de procesarlos de

acuerdo a sus requisitos funcionales, los cuales pueden involucrar el acceso a bases de datos, el uso de ficheros, el envío de mensajes a otras máquinas utilizando algún tipo de middleware o, incluso, el acceso a otros servidores web, posiblemente utilizando los protocolos asociados a los servicios web.

Finalmente, como resultado de la ejecución de la aplicación web, se ha de generar dinámicamente una respuesta para enviársela al cliente. Dicha respuesta suele ser un documento en formato HTML, si bien también podemos crear aplicaciones web que generen imágenes y documentos en cualquier otro formato en función de las necesidades del usuario.

Entre las ventajas más destacables de las aplicaciones web desarrolladas de esta forma destacan su accesibilidad (desde cualquier punto de Internet), su fácil mantenimiento (no hay que distribuir el código de las aplicaciones ni sus actualizaciones), su relativa seguridad (el código no puede manipularlo el usuario, al que sólo le llega una representación de los datos que le incumban) y su escalabilidad (utilizando arquitecturas multicapa y clusters de PCs resulta relativamente sencillo ampliar en número de clientes a los que puede dar servicio la aplicación).

Dada la gran variedad de herramientas y tecnologías que se pueden utilizar para construir aplicaciones web en el servidor, intentaremos agruparlas en unas pocas categorías mencionando algunas de las más representativas, ya que continuamente aparecen nuevas formas de implementar aplicaciones web e intentar enumerarlas todas sólo serviría para que este libro quedase obsoleto antes de su publicación:

Aplicaciones web compiladas: CGI

CGI es el nombre que se le da a una aplicación web que recibe sus parámetros utilizando el estándar *Common Gateway Interface*, de ahí su nombre. El estándar establece cómo han de comunicarse la aplicaciones con el servidor web. Por extensión, se denomina CGI a un módulo de una aplicación web que se implementa utilizando el estándar CGI en un lenguaje de programación tradicional como C. En realidad, un CGI se encarga únicamente de implementar la respuesta de la aplicación web a un tipo concreto de solicitud proveniente del cliente. Por tanto, una aplicación web estará formada, en general, por muchos CGIs diferentes. Cada uno de ellos será responsable de un contexto de interacción de la aplicación con el usuario (la interacción del usuario con la aplicación que se realiza como una única acción del usuario).

El estándar CGI permite ejecutar programas externos a un servidor HTTP. El estándar especifica cómo se pasan los parámetros al programa como parte de la cabecera de la solicitud HTTP y también define algunas variables de entorno. En realidad un CGI puede ser cualquier programa que pueda aceptar parámetros en su línea de comandos. En ocasiones se utilizan lenguajes tradicionales como C y, otras veces, se emplean lenguajes ideados expresamente para construir aplicaciones web (como es el caso de Perl).

Si escribimos un CGI en C, lo único que tenemos que hacer es escribir un programa estándar

que acepte como parámetros los datos recibidos desde el cliente. Dicho programa ha de generar la respuesta HTTP correspondiente a través de su salida estándar, `stdout` en el caso de C. Usualmente, el programa CGI generará un documento en HTML, si bien también puede redirigir al usuario a otra URL, algo permitido por el protocolo HTTP y que se utiliza mucho para registrar el uso de los enlaces en buscadores y banners publicitarios. Al tener que realizar todo esto a bajo nivel, la tarea se vuelve harto complicada en aplicaciones web de cierta envergadura, lo que nos hará buscar soluciones alternativas como las que se describen en los apartados siguientes.

Habitualmente, un CGI es una aplicación independiente que compilamos para el sistema operativo de nuestro servidor web (esto es, un fichero EXE en Windows). Dicha aplicación se suele instalar en un subdirectorio específico del servidor web, el directorio `/cgi-bin`, si bien esto no es estrictamente necesario. Cada vez que el servidor web recibe una solicitud para nuestra aplicación, el servidor web crea un nuevo proceso para atenderla. Si la ejecución del proceso falla por algún motivo o se reciben más solicitudes que procesos pueden crearse, nuestra aplicación dejará de responder a las solicitudes del usuario.

Dado que lanzar un proceso nuevo cada vez que recibimos una solicitud del usuario puede resultar bastante costoso, los servidores web suelen incluir la posibilidad de implementar nuestros CGIs como bibliotecas que se enlazan dinámicamente con el servidor web y se ejecutan en su espacio de direcciones (DLLs en el caso de Windows). De esta forma se elimina la necesidad de crear un proceso independiente y realizar cambios de contexto cada vez que nuestra aplicación web deba atender una petición HTTP. Éste el mecanismo que utilizan los módulos ISAPI [Internet Server API] del Internet Information Server de Microsoft o los módulos NSAPI [Netscape Server API] del servidor HTTP de Netscape. Facilidades como ISAPI o NSAPI sirven para mejorar el rendimiento de los CGIs y poco más. Un fallo en la implementación de un módulo de nuestra aplicación web puede ocasionar que "se caiga" en servidor web por completo, algo que normalmente no sucederá con los CGIs que se ejecutan de forma independiente al servidor web.

Servlets

El término servlet, por analogía con el término applet, hace referencia a un programa escrito en Java que se ejecuta en el servidor en vez de ejecutarse en el navegador del cliente como los applets.

En realidad, un servlet extiende el comportamiento del servidor web de la misma forma que un CGI tradicional. La principal diferencia es que Java nos ofrecen una completa biblioteca de clases para implementar cómodamente la funcionalidad de nuestra aplicación web.

El uso de servlets, que encapsulan la comunicación con el servidor web y el usuario final de la aplicación, permite que el programador se centre en la lógica de su aplicación sin tener que preocuparse en exceso de los detalles del protocolo HTTP. Además, al ser Java un lenguaje orientado a objetos, las aplicaciones web resultantes son más elegantes, modulares y flexibles que las que se construyen con CGIs.

Aplicaciones web interpretadas: CGI scripts & Scripting languages

Un script CGI es igual que una aplicación CGI, salvo que su implementación se realiza utilizando lenguajes interpretados de propósito específico, de ahí el término con el que a menudo se hace referencia a ellos: *scripting languages*. **Perl**, **PHP** [*Personal Home Page*] y **ColdFusion** (extensión *.cfm*) o **Groovy** (basado en Java) son algunos de los lenguajes más populares pertenecientes a esta categoría, si bien también se pueden utilizar directamente macros del shell de UNIX o incluso lenguajes antiguos como **REXX** [*Restructured EXtended eXecutor*, 1979].

Por ejemplo, a continuación se muestra un pequeño programa escrito en Perl que almacenaríamos en un fichero de nombre `prueba.cgi`:

```
#!/usr/bin/perl
use CGI qw(:standard);

print header,
      start_html,
      h1("CGI de ejemplo"),
          "Su dirección IP es: ", remote_host(),
          end_html;
```

Cuando se ejecuta el programa anterior, se genera automáticamente la cabecera HTTP asociada a la respuesta y un documento HTML que muestra la dirección IP de la máquina que realizó la solicitud HTTP en primera instancia. Si lo ejecutamos localmente obtenemos lo siguiente:

```
...
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>Untitled Document</TITLE></HEAD>
<BODY>
 <H1>CGI de ejemplo</H1>
 Su dirección IP es: localhost
</BODY>
</HTML>
```

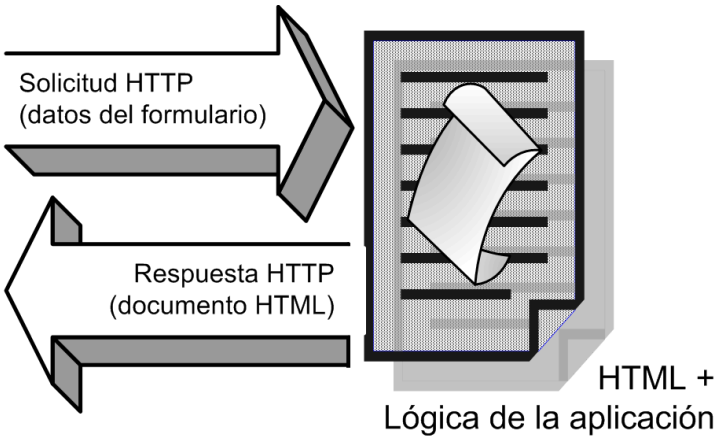
Para ejecutar la aplicación web, el usuario no tendría más que escribir la URL correspondiente en la barra de direcciones de su navegador:

```
http://csharp.ikor.org/cgi-bin/prueba.cgi
```

suponiendo, claro está, que nuestro servidor web fuese `csharp.ikor.org`.

Páginas de servidor: ASP y JSP

Existen otras tecnologías, similares a los lenguajes interpretados como Perl, que nos permiten preparar documentos HTML dentro de los cuales podemos introducir fragmentos de código que será interpretado por el servidor web cuando atienda las solicitudes HTTP que reciba. En otras palabras, en vez de crear programas que incluyan en su interior el código necesario para generar el documento HTML, creamos documentos HTML que incluyen el código de la aplicación en su interior. Las páginas **JSP** [*Java Server Pages*] de Java y las páginas **ASP** [*Active Server Pages*] de Microsoft son los ejemplos más representativos de este tipo de sistemas.



Funcionamiento de las páginas de servidor: Una página ASP/JSP contiene HTML estático intercalado con scripts que se encargan de generar HTML de forma dinámica.

Igual que los lenguajes de propósito específico o los servlets, las páginas de servidor (ASP o JSP) resultan más cómodas para el programador que los CGIs, ya que no tiene que tratar directamente con los mensajes HTTP que se transmiten desde y hasta el navegador web del cliente. Sin embargo, el diseño de las aplicaciones resultantes no suele ser demasiado elegante, pues mezcla la interfaz de usuario con la lógica de la aplicación (y siempre deberíamos aspirar a construir aplicaciones modulares con módulos débilmente acoplados).

Las páginas ASP permiten crear aplicaciones web fácilmente incluyendo en los documentos HTML fragmentos de código escrito en un lenguaje como VBScript, lo más usual, o JScript, la versión de JavaScript de Microsoft. En la plataforma .NET, las páginas ASP.NET reemplazan a las páginas ASP y el resto de este capítulo y los siguientes los dedicaremos a

estudiar la construcción de aplicaciones web utilizando esta tecnología en el servidor HTTP de Microsoft (el Internet Information Server).

JSP (http://java.sun.com/products/jsp/) funciona de forma análoga a las páginas ASP, si bien utiliza el lenguaje de programación Java. En realidad, JSP no es más que una extensión de los servlets Java que permiten programar las aplicaciones web tal como estaban acostumbrados los programadores que utilizaban páginas ASP antes de que JSP existiese. Por este motivo se suele considerar que las páginas JSP son más sencillas que los servlets o los CGIs.

JSP permite generar documentos HTML o XML de forma dinámica combinando plantillas estáticas con el contenido dinámico que se obtiene como resultado de ejecutar fragmentos de código en Java. JSP permite separar mejor la interfaz de usuario de la generación de contenido que las páginas ASP tradicionales, de forma que los diseñadores web pueden modificar el aspecto de una página web sin que eso interfiera en la programación de la aplicación web. Esto se debe a que JSP utiliza etiquetas al estilo de XML para generar el contenido de forma dinámica, generación de la que se harán cargo componentes instalados en el servidor (*JavaBeans*), el cual puede ser un servidor HTTP como Apache, un servidor escrito en Java como Tomcat (conocido familiarmente como "el gato Tom") o cualquiera de los muchos servidores de aplicaciones existentes que se ajustan al estándar J2EE [*Java 2 Enterprise Edition*].

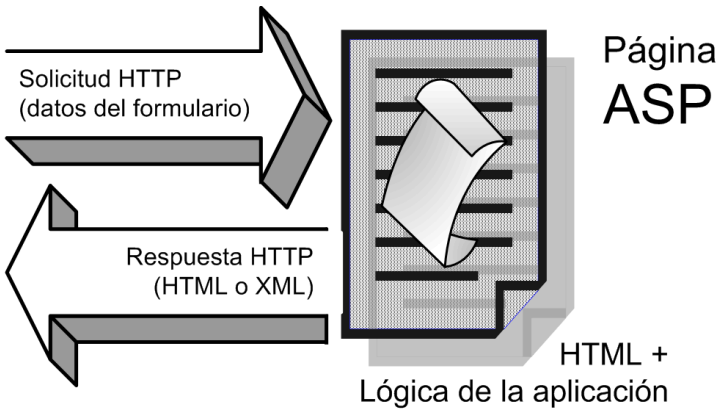
Recordatorio

Algo que nunca debemos olvidar es que tanto ASP como JSP gozan de gran aceptación a pesar de que no fuerzan uno de los principios básicos de diseño de software: la separación entre la interfaz de una aplicación y su lógica interna.

ASP: Active Server Pages

Tal como mencionamos en la sección anterior, ASP es la tecnología de Microsoft que permite desarrollar aplicaciones web que ejecuten en el servidor HTTP de Microsoft, el Internet Information Server (IIS). El desarrollo de aplicaciones utilizando páginas ASP consiste, básicamente, en intercalar macros o fragmentos de código dentro de los documentos HTML que sirven para crear las interfaces de usuario de las aplicaciones web. Los fragmentos de HTML proporcionan la parte estática de lo que ve el usuario mientras que los fragmentos de código generar la parte dinámica. Esto suele conducir a mezclar los detalles de la interfaz con la lógica de la aplicación, algo que, repetimos, no suele ser demasiado recomendable.

Una página ASP no es más que un fichero HTML con extensión .asp (.aspx en el caso de ASP.NET) al que le añadimos algo de código. Este código se pueden implementar utilizando distintos lenguajes interpretados. Por lo general, se emplea una variante de Visual Basic conocida como VBScript [*Visual Basic Script*]. Cuando alguien accede a la página, el Internet Information Server interpreta el código que incluye la página y combina el resultado de su ejecución con la parte estática de la página ASP (la parte escrita en HTML convencional). Una vez interpretada la página ASP, el resultado final es lo que se envía al navegador web instalado en la máquina del usuario que accede a la aplicación.



Funcionamiento de las páginas ASP: La parte estática de la página se envía junto con el resultado de ejecutar el código en el servidor, de ahí el "AS" de ASP.

Para desarrollar páginas ASP con comodidad, el programador dispone de una serie de objetos predefinidos que simplifican su trabajo ocultando los detalles de la comunicación del

navegador web del cliente con el servidor HTTP. Igual que en el caso de las páginas JSP en Java y de otras muchas alternativas para desarrollar aplicaciones web en el servidor, las mencionadas en el apartado anterior de este capítulo, los objetos definidos en ASP proporcionan distintos servicios útiles en el desarrollo de aplicaciones web (véase la tabla adjunta), además de facilidades para acceder a componentes COM (por ejemplo, se puede utilizar ADO [ActiveX Data Objects] para acceder a bases de datos).

Objeto	Encapsula
`Request`	La solicitud HTTP recibida
`Response`	Las respuesta HTTP generada
`Server`	El estado del servidor
`Application`	El estado de la aplicación web
`Session`	La sesión de usuario

Igual que sucede con las demás tecnologías basadas en lenguajes interpretados, para dejar nuestra aplicación web a disposición del usuario basta con escribir las páginas ASP y guardarlas en algún directorio al que se pueda acceder a través del Internet Information Server, sin tener que compilarlas previamente.

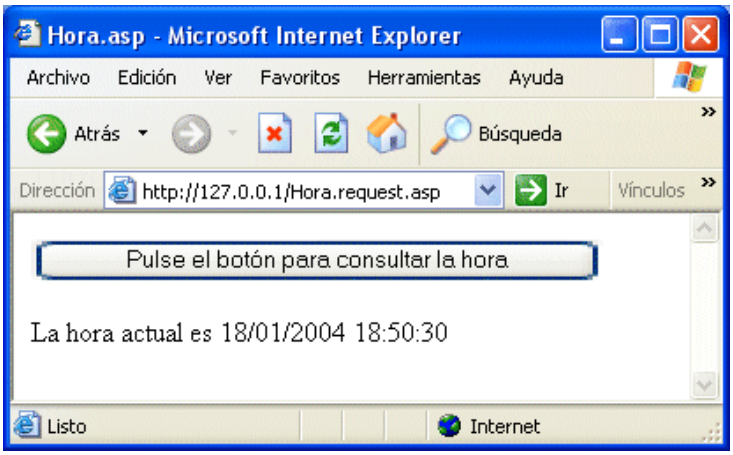
El siguiente ejemplo muestra cómo se pueden crear fácilmente páginas ASP que generen su contenido dinámicamente. Para crear nuestra primera página web, sólo tenemos que crear un fichero de texto llamado `Hora.asp` y colocarlo en algún directorio al que se pueda acceder desde un navegador web utilizando el IIS. Dentro del fichero podemos escribir lo siguiente:

```
<html>
 <head>
  <title>Hora.asp</title>
 </head>
 <body>
  <h2> Hora actual </h2>
  <% Response.Write(Now()) %>
 </body>
</html>
```

El código que aparece entre las etiquetas `<%` y `%>` contiene la parte de la página ASP se interpreta en el servidor antes de enviarle nada al navegador del cliente. El método `Response.Write` sirve para escribir algo en la página HTML que se genera como resultado de la ejecución de la página ASP mientras que la función `Now()` es una función predefinida que nos devuelve la fecha y la hora que marque el reloj del sistema donde esté instalado el servidor web. Siempre que queramos generar dinámicamente parte de la página utilizaremos el objeto `Response` para hacerle llegar al usuario aquello que nos interese.

En el ejemplo anterior, el fragmento de código incluido entre las etiquetas <% y %> se ejecuta cuando el usuario accede por primera vez a la página `Hora.asp`. Una vez que se carga la página, el resultado de la ejecución de la página ASP queda almacenado en la caché del navegador del cliente y la hora no se actualiza hasta que el usuario solicite explícitamente que la página se vuelva a descargar del servidor (pinchando en el botón actualizar de la barra de herramientas del navegador web que esté utilizando).

Podríamos modificar la página ASP que utilizamos en el ejemplo de arriba para mostrar cómo se puede utilizar el objeto `Request`. En esta ocasión, nuestra página ASP incluirá un botón gracias al cual el usuario podrá visualizar la hora actual del servidor en el momento que le interese sin necesidad de conocer el comportamiento interno del navegador web que utilice:



Aspecto visual de nuestra sencilla página ASP.

Para incluir controles en nuestra página ASP a través de los cuales el usuario pueda comunicarse con el servidor web, lo único que tenemos que hacer es incluir un formulario HTML en nuestra página ASP:

```
<html>
 <head>
  <title>Hora.Request.asp</title>
 </head>
 <body>
  <form method="post">
     <input type="submit" id=button name=button
            value="Pulse el botón para consultar la hora" />
     <%
       if (Request.Form("button") <> "") then
          Response.Write "<p>La hora actual es " & Now()
       end if
     %>
```

```
    </form>
  </body>
</html>
```

En esta ocasión, hemos utilizado el objeto `Request` para recoger los datos de entrada que llegan a nuestra página ASP desde el formulario incluido en la página HTML que se visualiza en el navegador web del cliente. `Request` encapsula los datos enviados desde el cliente y nos permite programar nuestra página sin tener que conocer los detalles del protocolo HTTP relativos al envío de datos asociados a un formulario HTML.

Para comprobar los ejemplos anteriores funcionan, sólo hay que guardar los ficheros ASP correspondientes en algún sitio que cuelgue del directorio `wwwroot` del IIS y acceder a él desde el navegador utilizando una URL de la forma `http://localhost/...`.

Como no podía ser de otra forma, en la plataforma .NET se ha incluido una versión mejorada de ASP denominada ASP.NET. Entre los principales inconvenientes asociados a las versiones de ASP anteriores a ASP.NET, a las que se suele hacer referencia por "ASP Clásico", se encuentra el hecho de ASP suele requerir escribir bastante código. Por ejemplo, es necesario escribir código para mantener el estado de la página ASP cuando, por cualquier motivo, el usuario ha de volver a ella. Omitir dicho código provoca la frustrante sensación que todos hemos experimentado alguna vez cuando, tras rellenar un extenso formulario HTML, se nos informa de un error en uno de los campos y nos vemos forzados a volver a rellenar el formulario completo.

Por otro lado, el código incluido en las páginas ASP resulta, además de excesivamente extenso, poco legible y difícil de mantener al estar mezclado con el HTML de la interfaz gráfica. En ASP, el fragmento de código ha de colocarse justo en el sitio donde queremos que su salida aparezca, lo que hace prácticamente imposible separar los detalles de la interfaz de usuario de la lógica de la aplicación. Este hecho, no sólo dificulta la reutilización del código y su mantenimiento, sino que también complica el soporte de nuestra aplicación para múltiples navegadores (recordemos, por ejemplo, que JavaScript y JScript no son exactamente iguales).

Finalmente, el "ASP Clásico" presenta otros inconvenientes que hacen su aparición a la hora de implantar sistemas reales de cierta envergadura. Como muestra, valga mencionar la dificultad que conlleva la correcta configuración de una aplicación ASP de cierta complejidad, los problemas de eficiencia que se nos pueden presentar cuando hemos de atender más peticiones que las que nuestro servidor puede atender, o los innumerables problemas que puede suponer la depuración de una aplicación construida con páginas ASP. En realidad, muchos de estos inconvenientes provienen de las limitaciones de los lenguajes interpretados que se utilizan para escribir los fragmentos de código incluidos en las páginas ASP.

Las limitaciones e inconvenientes mencionados en los párrafos anteriores propiciaron la

realización de algunos cambios notables en ASP.NET. ASP.NET, por tanto, no es completamente compatible con ASP, si bien la mayor parte de las páginas ASP sólo requieren pequeños ajustes para pasarlas a ASP.NET. De hecho, el primero de los ejemplos que hemos mostrado funciona perfectamente como página ASP.NET (sin más que cambiarle la extensión `.asp` por la extensión utilizada por las páginas ASP.NET: `.aspx`). El segundo de los ejemplos funcionará correctamente si tenemos en cuenta que, en Visual Basic .NET ,los paréntesis son obligatorios en las llamadas a los métodos (algo que era opcional en versiones anteriores de Visual Basic).

La siguiente sección de este capítulo y los capítulos siguientes los dedicaremos a tratar los detalles de funcionamiento de ASP.NET, la versión de ASP incluida en la plataforma .NET

ASP.NET: Aplicaciones web en la plataforma .NET

ASP.NET es el nombre con el que se conoce la parte de la plataforma .NET que permite el desarrollo y ejecución tanto de aplicaciones web como de servicios web. Igual que sucedía en ASP, ASP.NET se ejecuta en el servidor. En ASP.NET, no obstante, las aplicaciones web se suelen desarrollar utilizando formularios web, que están diseñados para hacer la creación de aplicaciones web tan sencilla como la programación en Visual Basic (.NET, claro está).

Un ejemplo

Para hacernos una idea de cómo es ASP.NET, retomemos el ejemplo de la sección anterior, que en ASP.NET queda como sigue si empleamos el lenguaje de programación C#:

Ejemplo de página ASP.NET

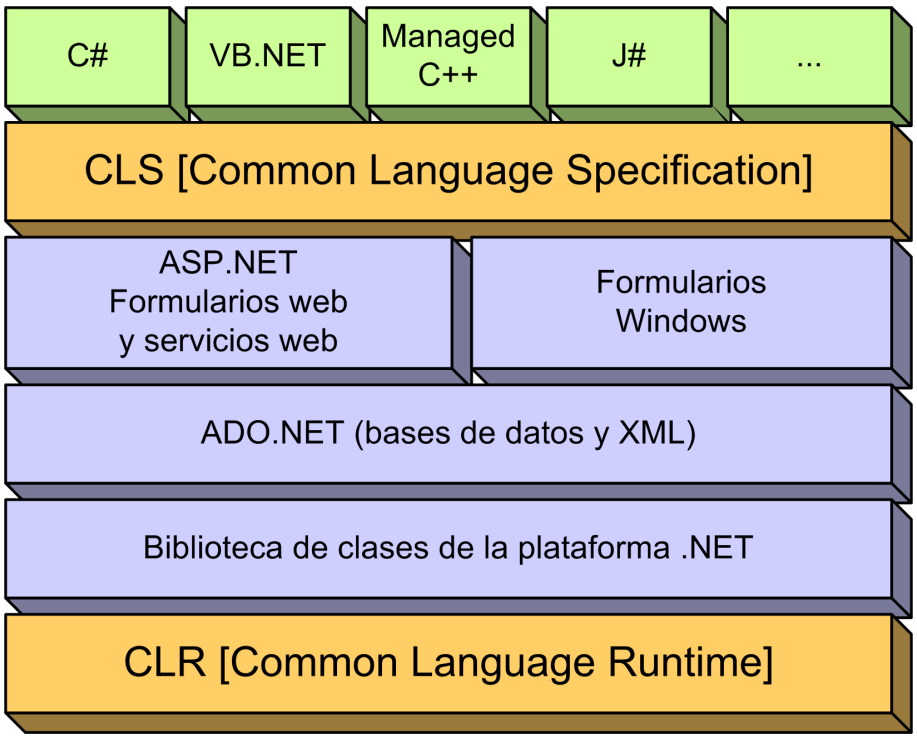
```
<%@ Page language="c#" %>
<html>
 <head>
   <title>Hora.aspx</title>
 </head>
 <script runat="server">
  public void Button_Click (object sender, System.EventArgs e)
  {
    LabelHora.Text = "La hora actual es " + DateTime.Now;
  }
 </script>
 <body>
  <form method="post" runat="server">
    <asp:Button onclick="Button_Click" runat="server"
                Text="Pulse el botón para consultar la hora"/>
    <p>
    <asp:Label id=LabelHora runat="server" />
  </form>
 </body>
</html>
```

Este sencillo ejemplo ilustra algunas de las características más relevantes de ASP.NET. Por ejemplo, podemos apreciar cómo el código de nuestra aplicación ya no está mezclado con las etiquetas HTML utilizadas para crear el aspecto visual de nuestra aplicación en el navegador del usuario. En vez de incluir código dentro de la parte correspondiente al HTML estático, algo que todavía podemos hacer al estilo de las páginas ASP tradicionales, hemos preferido utilizar un par de controles ASP.NET que nos permiten manipular en el servidor los elementos de nuestra página web con más comodidad (de forma similar a como JavaScript

nos da la posibilidad de modificar dinámicamente, en el cliente, el aspecto de los distintos elementos de la página).

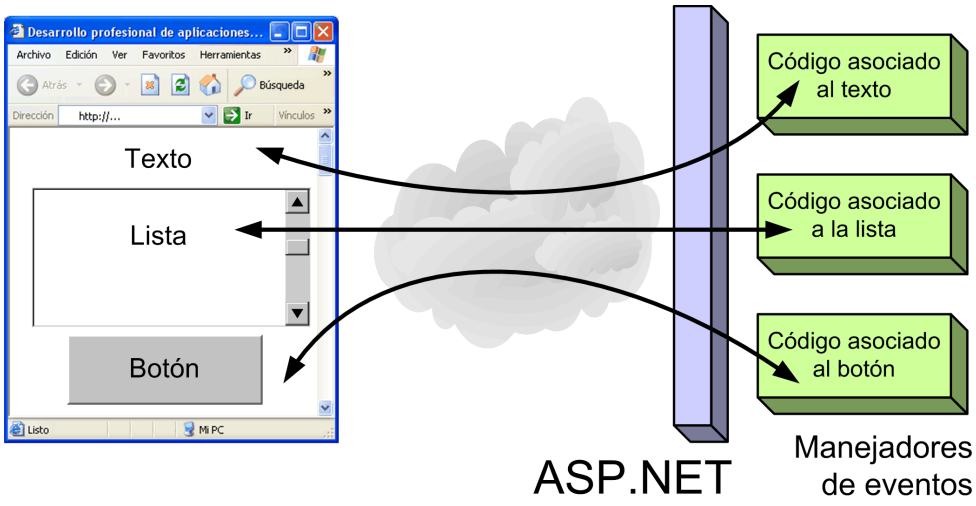
Para probar el funcionamiento del ejemplo anterior, sólo tenemos que guardar la página ASP.NET con la extensión `.aspx` en uno de los directorios de nuestra máquina a los que da acceso el Internet Information Server (por ejemplo, el directorio `/wwwroot`). Una vez hecho esto, podremos acceder a la página desde nuestro navegador web utilizando la URL adecuada. El Internet Information Server se encargará de interpretar la página ASP.NET de la forma adecuada.

ASP.NET forma parte de la plataforma .NET. De hecho, los formularios Windows y los formularios ASP.NET son las dos herramientas principales con las que se pueden construir interfaces de usuario en .NET. Aunque no son intercambiables, ya que aún no existe una forma estándar de crear una interfaz de usuario que funcione tanto para aplicaciones Windows como para aplicaciones web, tanto unos formularios como los otros comparten su posición relativa dentro de la familia de tecnologías que dan forma a la plataforma .NET.



La plataforma .NET: Los formularios ASP.NET y los formularios Windows son las dos alternativas principales de las que dispone el programador para crear las interfaces de usuario de sus aplicaciones.

Igual que sucede en el caso de los formularios Windows (y, de hecho, en cualquier entorno de programación visual para un entorno de ventanas como Windows, desde Visual Basic y Delphi hasta Oracle Developer), la programación en ASP.NET está basada en el uso de controles y eventos. Las páginas ASP.NET, en vez de aceptar datos de entrada y generar su salida en HTML como sucede en ASP, implementan su funcionalidad en fragmentos de código que se ejecutan como respuesta a eventos asociados a los controles de la interfaz con los que puede interactuar el usuario. Esta forma de funcionar le proporciona a ASP.NET un mayor nivel de abstracción, requiere menos código y permite crear aplicaciones más modulares, legibles y mantenibles.

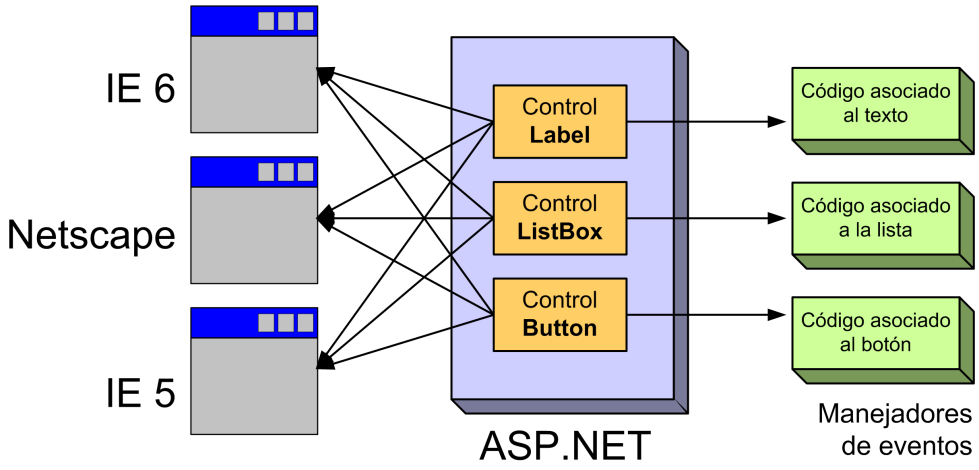


Eventos en ASP.NET: La respuesta de la aplicación web se obtiene como resultado de ejecutar los manejadores de eventos asociados a los controles incluidos en la interfaz de usuario.

Como muestra la figura, el código se ejecuta en el servidor web en función de los manejadores de eventos que definamos para los controles y páginas que conforman la interfaz de nuestra aplicación web. En ASP.NET, todos los controles que aparecen en la interfaz derivan de la clase `System.Web.UI.Control`. La página ASP.NET, en sí misma, también es un objeto. En este caso, la página hereda de `System.Web.UI.Page`, que a su vez deriva de `System.Web.UI.Control`.

Otra de las características destacables de ASP.NET es que las etiquetas que introducimos en la página HTML para incluir controles en la interfaz de usuario son independientes del HTML que después se genera para construir la interfaz de usuario que le llega al navegador del cliente. Es el caso, por ejemplo, de los controles `<asp:Label...>` y `<asp:Button...>` que aparecen en el ejemplo que abre este apartado. ASP.NET se encarga de convertir estas etiquetas en el fragmento de HTML que resulte más adecuado para

mostrar los controles en función del navegador web que utilice el usuario de nuestra aplicación. De esta forma, ASP.NET garantiza la compatibilidad de los controles de nuestra aplicación web con distintos navegadores, sin que el programador tenga que preocuparse demasiado de las diferencias existentes entre los diferentes navegadores web que puede emplear el usuario final para acceder a nuestra aplicación.



ASP.NET se encarga de mostrar los formularios web de la forma que resulte más adecuada para el navegador que utilice el usuario final de la aplicación.

Dos estilos

Si bien el ejemplo incluido en el apartado anterior puede que no nos permita apreciar la importancia de las mejoras de ASP.NET en cuanto a la separación de la interfaz de la lógica de la aplicación, lo cierto es que ASP.NET nos permite aprovechar todas las características de la plataforma .NET para el diseño de aplicaciones modulares utilizando técnicas de orientación a objetos. De hecho, ASP.NET nos permite utilizar dos estilos bien diferenciados para la confección de páginas ASP.NET:

- El primero de ellos consiste en incluir tanto los controles como el código en un único fichero `.aspx`, tal y como hicimos en el ejemplo anterior. A pesar de que los cambios introducidos por ASP.NET suponen una notable mejora respecto al ASP Clásico, esta forma de crear páginas ASP.NET nos impide aprovechar al máximo las ventajas de ASP.NET. Por ejemplo, puede que el responsable de implementar el código asociado a la funcionalidad de la página sea una persona diferente al diseñador gráfico que se encarga del aspecto visual de la aplicación. Tener todo en un único fichero puede provocar más que un simple conflicto de

intereses entre programadores y diseñadores gráficos. Baste con pensar qué podría suceder si el diseñador gráfico crea sus diseños utilizando algún editor visual de HTML, del tipo de los que no se suelen comportar correctamente con los fragmentos del documento HTML que no comprenden.

- Afortunadamente, ASP.NET también nos permite mantener los controles de nuestra interfaz en un fichero `.aspx` y dejar todo el código en un lugar aparte [*code-behind page*]. De esta forma, se separa físicamente, en ficheros diferentes, la interfaz de usuario del código de la aplicación. Esta alternativa es, sin duda, más adecuada que la anterior. En la situación antes descrita, tanto el programador como el diseñador gráfico pueden centrarse en su trabajo y el riesgo de que uno de ellos "estropee" el trabajo del otro es bastante reducido. Incluso aunque no exista la división de trabajo entre programador y diseñador gráfico, mantener el código separado facilita la construcción de aplicaciones web con diferentes aspectos visuales. Esto puede llegar a resultar necesario cuando un mismo producto se personaliza para distintos clientes o cuando nuestra empresa pretende crear una línea de productos sin hipotecar su futuro, ya que tener que mantener código duplicado puede llegar a hundir el mejor de los planes de negocio.

Veamos, a continuación, cómo quedaría nuestro ejemplo de antes si separamos físicamente el diseño de nuestra interfaz de la implementación de los manejadores de eventos que implementan la respuesta de nuestra aplicación cuando el usuario pide la hora.

En la página ASP.NET propiamente dicha, sólo incluiremos el HTML estático que se le envía al cliente junto con las etiquetas correspondientes a los controles ASP.NET que se mostrarán en el navegador del cliente como elementos de un formulario web convencional. Aparte de esto, también hemos de introducir una directiva en la cabecera de nuestra página que le permita saber al IIS dónde está el código que se ejecuta para conseguir el comportamiento dinámico de nuestra página. Todo esto lo pondremos en un fichero llamada `HoraWebForm.aspx`:

```
<%@ Page language="c#" Codebehind="HoraWebForm.aspx.cs"
Inherits="HoraWeb.WebForm" %>
<html>
 <head>
   <title>Hora.aspx</title>
 </head>
 </script>
 <body>
  <form method="post" runat="server">
    <asp:Button id="ButtonHora" runat="server"
                Text="Pulse el botón para consultar la hora" />
    <p>
    <asp:Label id="LabelHora" runat="server" />
  </form>
 </body>
</html>
```

Una vez que tenemos el diseño de nuestro formulario, sólo nos falta implementar su comportamiento dinámico en un fichero aparte. Este fichero de código lo escribiremos como cualquier otro fichero de código en C#, creando para nuestra página una clase que herede de `System.Web.UI.Page`. Tal como podemos deducir de lo que aparece en la directiva que abre nuestro fichero `.aspx`, el código lo guardaremos en el fichero `HoraWebForm.aspx.cs`:

```
namespace HoraWeb
{
  public class WebForm : System.Web.UI.Page
  {
    protected System.Web.UI.WebControls.Button ButtonHora;
    protected System.Web.UI.WebControls.Label LabelHora;

    override protected void OnInit(EventArgs e)
    {
      this.ButtonHora.Click += new
System.EventHandler(this.ButtonHora_Click);
      base.OnInit(e);
    }

    private void ButtonHora_Click(object sender, System.EventArgs e)
    {
      LabelHora.Text = "La hora actual es "+DateTime.Now;
    }
  }
}
```

Como se puede ver, el código asociado a nuestra aplicación web se escribe de forma completamente independiente a su presentación en HTML. En realidad, cuando creamos la página `.aspx`, lo que estamos haciendo es crear una subclase de la clase implementada en el fichero de código. Esta subclase se limita únicamente a organizar los elementos de nuestra interfaz de usuario de la forma que resulte más adecuada.

Como habrá podido observar, la creación de una página ASP.NET con C# resulta relativamente sencilla si se sabe programar en C# y se tienen unos conocimientos básicos de HTML. Sólo tenemos que aprender a manejar algunos de los controles utilizados en la construcción de formularios ASP.NET. Hasta ahora, todo lo hemos implementado a mano con el objetivo de mostrar el funcionamiento de ASP.NET y de desmitificar, en parte, la creación de aplicaciones web. No obstante, como es lógico, lo normal es que creemos las páginas ASP.NET con la ayuda del diseñador de formularios que nos ofrece el entorno de desarrollo Visual Studio .NET. Éste se encargará de rellenar muchos huecos de forma automática para que nos podamos centrar en la parte realmente interesante de nuestras aplicaciones. Pero antes de seguir profundizando en los detalles de ASP.NET tal vez convendría refrescar nuestros conocimientos de HTML.

Apéndice: Aprenda HTML en unos minutos

Sin lugar a dudas, el servicio de Internet más utilizada hoy en día es la World Wide Web (WWW), una aplicación que nos permite enlazar unos documentos con otros fácilmente. Para escribir documentos web, más conocidos como páginas web, se utiliza el formato HTML [*HyperText Markup Language*].

Un documento HTML es un fichero de texto normal y corriente (un fichero ASCII, por lo general). Este fichero incluye ciertas marcas o etiquetas que le indican al navegador, entre otras cosas, cómo debe visualizarse el documento. Las etiquetas, igual que en XML, se escriben encerradas entre ángulos: `<` y `>`. Además, dichas etiquetas usualmente van por parejas: `<etiqueta>` y `</tag>`.

Cualquier documento en formato HTML, delimitado por la pareja de etiquetas `<HTML>` y `</HTML>`, tiene dos partes principales:

- La cabecera (entre `<HEAD>` y `</HEAD>`) contiene información general sobre el documento que no se muestra en pantalla: título, autor, descripción...

- Las etiquetas `<BODY>` y `</BODY>` definen la parte principal o cuerpo del documento.

Documento HTML de ejemplo

```
<HTML>
  <HEAD>
    <TITLE> Título del documento </TITLE>
    <META NAME="Author" CONTENT="Fernando Berzal et al.">
    <META NAME="Keywords" CONTENT="HTML">
    <META NAME="Description" CONTENT="Conceptos básicos de HTML">
  </HEAD>
  <BODY>
    Cuerpo del documento...
  </BODY>
</HTML>
```

El cuerpo del documento HTML puede incluir, entre otros elementos:

- **Párrafos** (delimitados por la etiqueta `<P>`).

- **Encabezados**, empleados para definir títulos y subtítulos (de mayor a menor nivel, con las etiquetas `<H1>` a `<H6>`).

- **Enlaces** para enganchar unas páginas con otras: `<A HREF="url"> texto`

</A>, donde `url` indica la URL del documento al que apunta el enlace y `texto` es el fragmento del documento de texto sobre el cuál ha de pinchar el usuario para acceder al documento enlazado.

- **Imágenes** (en formato GIF, PNG o JPG): `<IMG SRC="fichero" BORDER=0 ALT="texto descriptivo">`, donde `url` indica la URL mediante la cual se puede acceder al fichero que contiene la imagen y `texto` es un texto alternativo que se le presenta al usuario cuando éste no ve la imagen.

- **Listas** numeradas (con `<OL>` y `</OL>`) o no numeradas (con `<UL>` y `</UL>`) cuyos elementos se indican en HTML utilizando la etiqueta `<LI>`.

Con el fin de personalizar la presentación del texto del documento, HTML incluye la posibilidad de poner el texto en negrita (`<B> ... </B>`), en cursiva (`<I> ... </I>`), subrayarlo (`<U> ... </U>`) o centrarlo (con `<CENTER> ... </CENTER>`). Además, se puede modificar el tamaño y color del tipo de letra con `<FONT SIZE="+1" COLOR="#rrggb"> ... </FONT>`, donde `SIZE` indica el tamaño (usualmente, un tamaño relativo al del texto actual, p.ej. +2 +1 -1 -2) y el color se suele representar en hexadecimal como una combinación de rojo, verde y azul. Por ejemplo, el negro es `#000000`, el blanco `#ffffff`, el rojo `#ff0000`, el verde `#00ff00` y el azul `#0000ff`.

Otras etiquetas habituales en los documentos HTML son `<BR>`, que introduce saltos de línea, y `<HR>`, que muestra en el navegador una línea horizontal a modo de separador. En los siguientes apartados presentaremos otros aspectos de HTML que conviene conocer:

Caracteres especiales

En lenguajes derivados de SGML, como es el caso de HTML o XML, las vocales acentuadas, las eñes y otros caracteres "no estándar" en inglés, incluyendo los ángulos que se utilizan para las etiquetas HTML, requieren secuencias especiales de caracteres para representarlos. La siguiente tabla recoge algunas de ellas:

Carácter	Secuencia HTML	Carácter	Secuencia HTML
ñ	ñ	Ñ	Ñ
&	&	€	€
<	<	á	á
>	>	è	è
"	"	ê	ê
©	©	ü	ü
®	®	™	™

Tablas

Las tablas se delimitan con las etiquetas <TABLE> y </TABLE>. Entre estas dos etiquetas se han de incluir una serie de filas delimitadas por <TR> y </TR>. Cada fila, a su vez, incluye una serie de celdas <TD> y </TD>. Por ejemplo:

```
<TABLE border=2>
  <TR bgcolor="#ccccee">
    <TH COLSPAN=2> <IMG SRC="cogs.gif"> Tabla en HTML </TH>
  </TR>
  <TR bgcolor="#e0e0e0">
    <TH> Datos</TH>
    <TH> Valores</TH>
  </TR>
  <TR>
    <TD> Dato 1</TD>
    <TD> Valor 1</TD>
  </TR>
  <TR>
    <TD> Dato 2</TD>
    <TD> Valor 2</TD>
  </TR>
  <TR>
    <TD> Dato 3</TD>
    <TD> Valor 3</TD>
  </TR>
</TABLE>
```

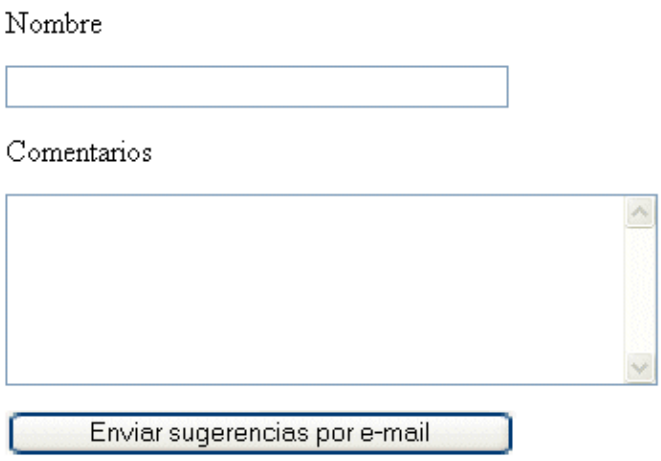
El fragmento de HTML anterior aparecerá en el navegador del usuario como:

Formularios

HTML también permite que el usuario no se limite a leer el contenido de la página, sino que también pueda introducir datos mediante formularios (la base de cualquier aplicación web. Por ejemplo, el siguiente fragmento de HTML contiene un formulario:

```
<FORM METHOD="POST" ACTION="mailto:berzal@acm.org">
  <INPUT TYPE="text" NAME="NOMBRE" SIZE=30 MAXLENGTH=40>
  <TEXTAREA NAME="COMENTARIOS" ROWS=6 COLS=40> </TEXTAREA>
  <INPUT TYPE="submit" VALUE="Enviar sugerencias por e-mail">
</FORM>
```

Este formulario, dentro de una tabla, se mostraría de la siguiente forma en el navegador del usuario:



En los formularios HTML estándar se pueden incluir:

- Cuadros de texto para que el usuario pueda escribir algo (`<INPUT TYPE="TEXT"...>`).
- Cuadros de texto que no muestran lo que el usuario escribe (`<INPUT TYPE="PASSWORD"...>`), indispensables cuando queremos que el usuario introduzca información privada.
- Textos de varias líneas (`<TEXTAREA ...> ... </TEXTAREA>`).

- Opciones que se pueden seleccionar o no de forma independiente (`<INPUT TYPE="CHECKBOX"...>`).
- Opciones mutuamente excluyentes (`<INPUT TYPE="RADIO"...>`).
- Listas para seleccionar valores (`<SELECT> <OPTION> <OPTGROUP>`).
- Ficheros adjuntos (`<INPUT TYPE="FILE"...>`).
- E, incluso, datos ocultos para el usuario (`<INPUT TYPE="HIDDEN"...>`).

Para enviar los datos del formulario a la URL especificada en `FORM ACTION`, que sería la dirección adecuada para nuestra aplicación web, se puede utilizar un botón (`<INPUT TYPE="SUBMIT"...>`) o una imagen (`<INPUT TYPE="IMAGE"...>`). En este último caso, la acción que se realice puede depender de la zona de la imagen que seleccione el usuario, pues, además de los datos rellenados en el formulario, recibiremos las coordenadas de la imagen correspondientes al punto donde el usuario pulsó el botón del ratón.

Hojas de estilo

El principal inconveniente que tiene el formato HTML a la hora de crear páginas web es que debemos indicar la forma junto con el contenido del documento. Cuando hay que mantener un gran número de páginas, sería conveniente disponer de un mecanismo que nos facilitase darles a todas las páginas un formato coherente. Las hojas de estilo en cascada, CSS [*Cascading Style Sheets*] son el estándar del W3C que nos permite facilitar el mantenimiento de un conjunto grande de páginas HTML y asegurarnos de que se mantiene cierta coherencia en su presentación de cara al usuario.

Para emplear una hoja de estilo en la presentación de nuestra página web, sólo tenemos que incluir la siguiente etiqueta antes del cuerpo de nuestro documento HTML:

```
<link REL=STYLESHEET TYPE="text/css" HREF="style.css">
```

donde `style.css` es el fichero que contiene la hoja de estilo que se empleará para visualizar el documento HTML.

El texto de la hoja de estilo ha de escribirse de acuerdo a la siguiente sintaxis:

```
ETIQUETA {
  propiedad1: valor1;
  propiedad2: valor2;
}
```

```
ETIQUETA1, ETIQUETA2 {
  propiedad: valor;
}

.CLASE {
  propiedad: valor;
}
```

donde las etiquetas y las clases son las que se utilizan en los documentos HTML, mientras que las propiedades aplicables a cada elemento y los valores que pueden tomar dichas propiedades están definidas en un estándar emitido por el W3C.

Por ejemplo, podemos hacer que el cuerpo de esta página se visualice con una imagen de fondo y se dejen márgenes alrededor del texto si escribimos la siguiente hoja de estilo:

```
BODY
{
 background-image: url(http://csharp.ikor.org/image/csharp.jpg);
 color: #000000;
 margin-left: 10%;
 margin-right: 10%;
 margin-top: 5%;
 margin-bottom: 5%;
}
```

También podemos definir estilos que nos permitan ver fragmentos de nuestros documentos con el fondo en gris de la misma forma que aparecen los ejemplos de esta sección. Sólo tenemos que definir una clase `ejemplo`:

En el documento HTML:

```
...
<table class="example">
...
```

En la hoja de estilo CSS:

```
.example
{
  background-color: #e0e0e0;
}
```

Incluso podemos definir características comunes para distintas etiquetas de las que aparecen en un documento HTML. La siguiente hoja de estilo hace que el texto incluido en párrafos,

citas, listas y tablas aparezca justificado a ambos márgenes:

```
P, BLOCKQUOTE, LI, TD
{
 text-align: justify;
}
```

Finalmente, algunos navegadores nos permiten modificar la forma en la que se visualizan los enlaces de una página web, para que estos cambien al pasar el cursor del ratón sobre ellos:

```
A
{
  text-decoration: none;
}

A:hover
{
  color: #009999;
}
```

Jugando un poco con las posibilidades que nos ofrecen las hojas de estilo CSS se puede conseguir que nuestras páginas HTML estándar tengan buen aspecto sin tener que plagarlas de etiquetas auxiliares como `FONT`, las cuales lo único que consiguen es que el texto de nuestro documento HTML sea menos legible y más difícil de mantener.

Información adicional

Toda la información relativa a los estándares relacionados con la web se puede encontrar en la página del organismo que los establece, el *World Wide Web Consortium* (W3C). En dicha página encontrará el estándar oficial, si bien, en la práctica, puede que le interese más visitar sitios como *Index DOT*. El ellos encontrará toda la información que necesite acerca de las páginas HTML y las hojas de estilo CSS, incluyendo comentarios acerca de cómo funciona cada etiqueta con las distintas versiones de los navegadores web más populares.

- W3C: http://www.w3c.org

- Index DOT: http://www.blooberry.com/indexdot/index.html

Formularios web

En este capítulo, nos centraremos en la construcción de páginas ASP.NET y adquiriremos los conocimientos necesarios para ser capaces de crear nuestras propias páginas web dinámicas con ASP.NET:

- En primer lugar, veremos en qué consisten los formularios web, una parte fundamental de la plataforma .NET.

- A continuación, estudiaremos los controles que podemos incluir dentro de las interfaces web que construiremos con formularios ASP.NET. Primero, analizaremos los tres tipos de componentes estándar incluidos en las bibliotecas de la plataforma .NET correspondientes a ASP.NET. Después, llegaremos a ver cómo podemos crear nuestros propios controles.

- Cuando ya tengamos una buena noción de lo que podemos incluir en un formulario ASP.NET, aprenderemos algo más acerca de su funcionamiento. En concreto, nos interesará saber qué son los *"post backs"* y cómo se mantiene el estado de una página ASP.NET.

Formularios web

Formularios en ASP.NET

En el capítulo anterior, presentamos una panorámica general del desarrollo de aplicaciones web, en la que mostramos las distintas alternativas de las que dispone el programador y situamos en su contexto la tecnología incluidas en la plataforma .NET para la creación de interfaces web: las páginas ASP.NET.

ASP.NET sustituye a las páginas interpretadas utilizadas en ASP por un sistema basado en componentes integrados en la plataforma .NET. De esta forma, podemos crear aplicaciones web utilizando los componentes que vienen incluidos en la biblioteca de clases de la plataforma .NET o, incluso, creando nuestros propios componentes. Lo usual es que estos últimos los implementemos a partir de los componentes existentes por composición; esto es, encapsulando conjuntos de componentes existentes en un componente nuevo. No obstante, también podemos crear nuevos componentes por derivación, creando una nueva clase derivada de la clase del componente cuyo comportamiento deseamos extender (como en cualquier entorno de programación orientado a objetos).

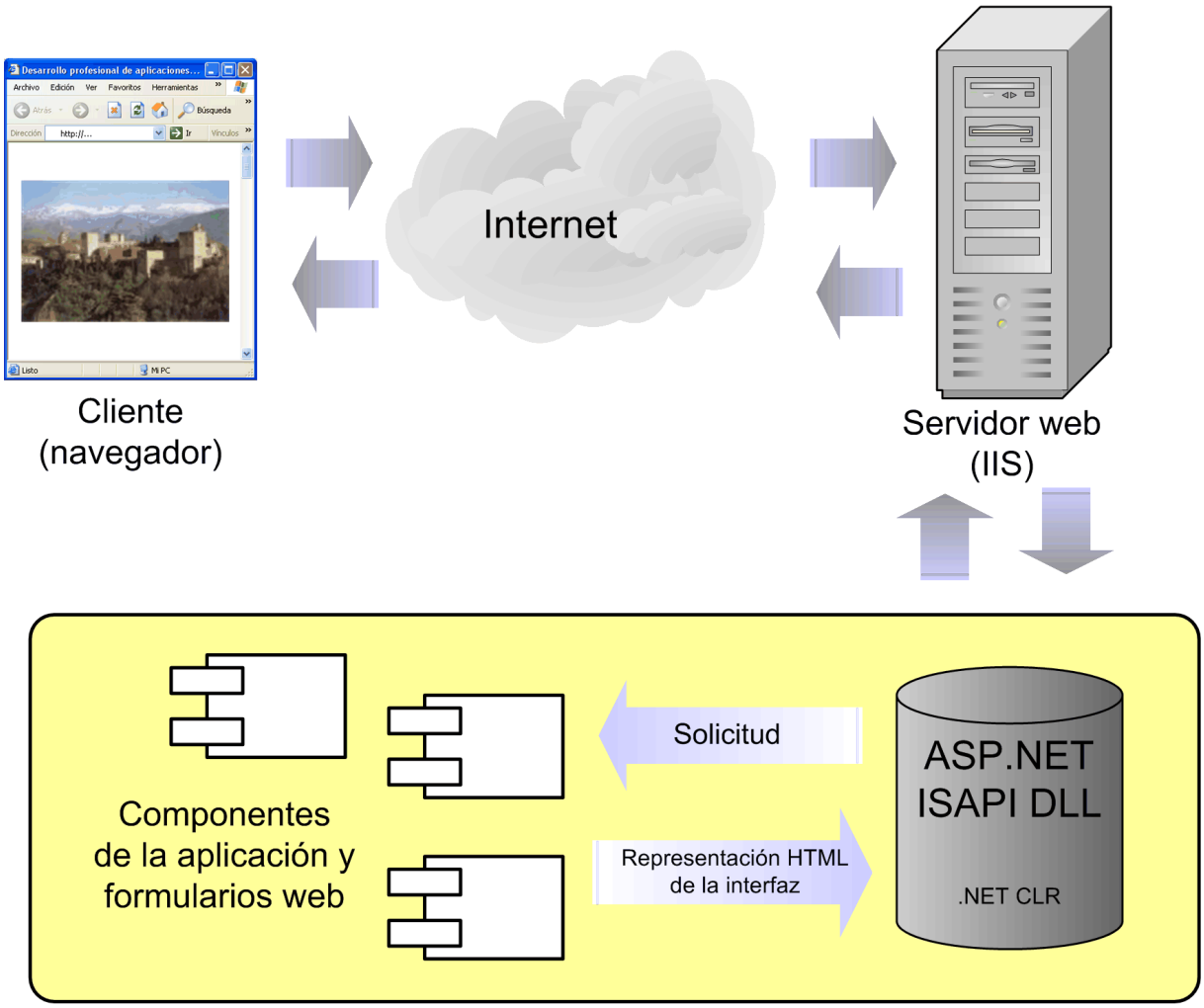
Al construir nuestras aplicaciones utilizando componentes, podemos utilizar un entorno de programación visual (como el Visual Studio .NET). Como consecuencia, al desarrollar aplicaciones web, no hemos de prestar demasiada atención al HTML de nuestras páginas ASP.NET, salvo, claro está, cuando estemos implementando los distintos componentes que utilizaremos para crear los controles de la interfaz de usuario. En otras palabras, los componentes nos permiten crear nuestra aplicación centrándonos en su lógica. Los propios componentes se encargarán de generar los fragmentos de HTML necesarios para construir la interfaz web de la aplicación.

En esta sección veremos en qué consisten y cómo se crean las páginas ASP.NET, aunque antes nos detendremos un poco en analizar cómo se ejecutan las páginas ASP.NET desde el punto de vista físico.

Ejecución de páginas ASP.NET

Los servidores HTTP pueden configurarse de tal forma que las peticiones recibidas se traten de diferentes formas en función del tipo de recurso solicitado. Básicamente, esta decisión la realiza el servidor a partir de la extensión del recurso al que intenta acceder el cliente. En el caso de las páginas ASP convencionales, cuando el usuario intenta acceder a un fichero con extensión `.asp`, el Internet Information Server delega en la biblioteca `asp.dll`, que se encarga de interpretar la página ASP. Cuando se utiliza ASP.NET, el IIS se configura de tal forma que las solicitudes recibidas relativas a ficheros con extensión `.aspx` son enviadas a la biblioteca `aspnet_isapi.dll`.

Como su propio nombre sugiere, la biblioteca `aspnet_isapi.dll` es un módulo ISAPI. Los módulos ISAPI, como vimos en el capítulo anterior, sirven para crear aplicaciones web sin que en el servidor se tengan que crear nuevos procesos cada vez que, como respuesta a una solicitud, se ha de crear dinámicamente una página web. La biblioteca encargada de la ejecución de las páginas ASP.NET (`aspnet_isapi.dll`) encapsula el CLR [*Common Language Runtime*] de la plataforma .NET. De esta forma, podemos utilizar todos los recursos de la plataforma .NET en el desarrollo de aplicaciones web. La DLL mencionada creará las instancias que sean necesarias de las clases .NET para atender las solicitudes recibidas en el servidor web.



Ejecución de páginas ASP.NET: Usando el Internet Information Server como servidor HTTP, una DLL ISAPI se encarga de que podamos aprovechar todos los recursos de la plataforma .NET en el desarrollo de aplicaciones web.

A diferencia de las páginas ASP tradicionales, las páginas ASP.NET se compilan antes de ejecutarse. La primera vez que alguien accede a una página ASP.NET, ésta se compila y se crea un fichero ejecutable que se almacena en una caché del servidor web, un *assembly* si utilizamos la terminología propia de la plataforma .NET. De esta forma, las siguientes ocasiones en las que se solicite la página, se podrá usar directamente el ejecutable. Al no tener que volver a compilar la página ASP.NET, la ejecución de ésta será más eficiente que la de una página ASP convencional.

Configuración del IIS

Para poder ejecutar páginas ASP.NET en el Internet Information Server, primero hemos de indicarle cómo ha de gestionar las peticiones recibidas relativas a ficheros con extensión `.aspx`. Para ello debemos utilizar la herramienta `aspnet_regiis` que se encuentra en el directorio donde se instala la plataforma .NET:

```
<windows>/Microsoft.NET/Framework/v1.X.XXXX
```

donde `<windows>` es el directorio donde esté instalado el sistema operativo Windows (`C:/Windows` o `C:/winNT`, por lo general) y `v1.X.XXXX` corresponde a la versión de la plataforma .NET que tengamos instalada en nuestra máquina. Puede que en nuestro ordenador existan distintas versiones instaladas de la plataforma .NET y, usualmente, escogeremos la más reciente para utilizarla en la creación de nuestras páginas ASP.NET.

Creación de páginas ASP.NET

La biblioteca de clases .NET incluye un conjunto de clases que nos serán de utilidad en la creación de las páginas ASP.NET. Entre dichas clases se encuentra una amplia gama de controles que podremos utilizar en la construcción de interfaces web para nuestras aplicaciones, controles tales como botones, cajas de texto, listas o tablas. Además, la biblioteca de clases estándar también proporciona algunos componentes que nos facilitarán realizar tareas comunes como la gestión del estado de nuestra aplicación web. Conforme vayamos avanzando en la construcción de páginas ASP.NET, iremos viendo cómo funciona cada uno de los componentes suministrados por la plataforma .NET.

Una aplicación web, generalmente, estará formada por varios formularios web. Cada uno de esos formularios lo implementaremos como una páginas ASP.NET. En la plataforma .NET, las páginas ASP.NET se construyen creando clases derivadas de la clase `System.Web.UI.Page`. Dicha clase proporciona la base sobre la que construiremos nuestras páginas ASP.NET, que implementaremos como subclases de `System.Web.UI.Page` en las que incluiremos la funcionalidad requerida por nuestras aplicaciones.

En realidad, para mantener independientes la interfaz de usuario y la lógica asociada a la aplicación, la implementación de las páginas ASP.NET la dividiremos en dos ficheros. En un fichero con extensión `.aspx` especificaremos el aspecto de nuestra interfaz, utilizando tanto etiquetas HTML estándar como etiquetas específicas para hacer referencia a los controles ASP.NET que deseemos incluir en nuestra página. En un segundo fichero, que será un fichero de código con extensión `.cs` si utilizamos en lenguaje C#, implementaremos la lógica de la aplicación.

El siguiente ejemplo muestra el aspecto que tendrá nuestro fichero `.aspx`:

```
<% @Page Language="C#" Inherits="TodayPage" Src="Today.cs" %>

<html>
<body>
   <h1 align="center">
      Hoy es <% OutputDay(); %>
   </h1>
</body>
</html>
```

El fichero anterior incluye todo lo necesario para generar dinámicamente una página web en la que incluiremos la fecha actual, que aparecerá donde está el fragmento de código delimitado por <% y %>, las mismas etiquetas que se utilizan en ASP tradicional para combinar la parte estática de la página (las etiquetas HTML) con la parte que ha de generarse dinámicamente. En esta ocasión, en vez de introducir el código necesario entre las etiquetas <% y %>, hemos optado por implementar una función auxiliar `OutputDay` que definiremos en un fichero de código aparte. El aspecto de este fichero de código, con extensión `.cs`, será el siguiente:

```
using System;
using System.Web.UI;

public class TodayPage:Page
{
  protected void OutputDay()
  {
    Response.Write(DateTime.Now.ToString("D"));
  }
}
```

Este fichero define una clase (`TodayPage`) que hereda de la clase `System.Web.UI.Page`. En nuestra clase hemos incluido un método que se encargará de generar un mensaje en el que se muestre la fecha actual. Para ello empleamos la clase `Response` que representa la respuesta de nuestra página ASP.NET y la clase `DateTime` que forma parte del vasto conjunto de clases incluidas en la biblioteca de clases de la

plataforma .NET.

Técnicamente, el fichero `.aspx` que creamos representa una clase que hereda de la clase definida en el fichero de código con extensión `.cs`, el cual, a su vez, define una subclase de `System.Web.UI.Page`. Esto explica por qué definimos en método `OutputDay()` como `protected`, para poder acceder a él desde la subclase correspondiente al fichero `.aspx`.

Para poner a disposición de los usuario la no demasiado útil aplicación web que hemos creado con los dos ficheros anteriores, sólo tenemos que copiar ambos ficheros a algún directorio al que se pueda acceder a través del IIS (el directorio raíz `wwwroot`, por ejemplo). Cuando un usuario intente acceder a nuestra aplicación web, éste sólo tendrá que introducir la ruta adecuada en la barra de direcciones de su navegador para acceder a la página `.aspx`. Al acceder por primera vez a ella, el código asociado a la página se compilará automáticamente y se generar un *assembly* en la caché del CLR encapsulado por la biblioteca `aspnet_isapi.dll` en el servidor web IIS.

Exactamente igual que en ASP, si el texto de la página ASP.NET cambia, el código se recompilará automáticamente, por lo que podemos editar libremente nuestra página en el servidor y al acceder a ella ejecutaremos siempre su versión actual (sin tener que desinstalar manualmente la versión antigua de la página e instalar la versión nueva en el servidor web). Si, entre solicitud y solicitud, el texto de la página no cambia, las nuevas solicitudes se atenderán utilizando el código previamente compilado que se halla en la caché del servidor web, con lo cual se mejora notablemente la eficiencia de las aplicaciones web respecto a versiones previas de ASP.

Aunque en el ejemplo anterior hayamos incluido un fragmento de código entre las etiquetas `<%` y `%>` dentro de la página `.aspx`, igual que se hacía con ASP, lo usual es que aprovechemos los recursos que nos ofrece la plataforma .NET para construir aplicaciones web de la misma forma que se construyen las aplicaciones para Windows en los entornos de programación visual. Como sucede en cualquiera de estos entornos, la interfaz web de nuestra aplicación la crearemos utilizando controles predefinidos a los cuales asociaremos un comportamiento específico definiendo su respuesta ante distintos eventos. En la siguiente sección de este capítulo veremos cómo se utilizan controles y eventos en las páginas ASP.NET.

Uso de controles en ASP.NET

Utilizando ASP.NET, las interfaces web se construyen utilizando controles predefinidos. Estos controles proporcionan un modelo orientado a objetos de los formularios web, similar en cierto modo al definido por JavaScript. Sin embargo, a diferencia de JavaScript, en ASP.NET no trabajaremos directamente sobre los objetos que representan las etiquetas HTML del documento que visualiza el usuario en su navegador. Lo que haremos será utilizar controles definidos en la biblioteca de clases .NET.

Al emplear controles predefinidos en nuestra interfaz de usuario en vez de especificar directamente las etiquetas HTML de los formularios web, las páginas ASP.NET se convierten en meras colecciones de controles. Desde el punto de vista del programador, cada control será un objeto miembro de la clase que representa la página (aquélla que hereda de `System.Web.UI.Page`). Cuando deseemos asociar a nuestra página un comportamiento dinámico, lo único que tendremos que hacer es asociar a los distintos controles los manejadores de eventos que se encargarán de implementar la funcionalidad de la página, exactamente igual que cuando se crea una interfaz gráfica para Windows utilizando un entorno de programación visual. Además, en nuestras páginas ASP.NET podremos incluir nuestros propios controles, los cuales crearemos a partir de los controles existentes por derivación (creando clases nuevas que hereden de las clases ya definidas) o por composición (encapsulando una página completa para luego utilizarla como un control más dentro de otras páginas).

En definitiva, en vez de utilizar un intérprete que se encargue de ejecutar los fragmentos de código desperdigados por nuestras páginas web, lo que haremos será utilizar un sistema basado en eventos para generar dinámicamente el contenido de las páginas que se le muestran al usuario. Esto nos permitirá acceder a todas las características de C# y de la plataforma .NET para construir aplicaciones web flexibles, modulares y fáciles de mantener.

Además, aparte de proporcionar un modelo orientado a objetos de la aplicación que evita el código "spaghetti" típico de ASP, los controles web constituyen una capa intermedia entre el código de la aplicación y la interfaz de usuario. Entre las ventajas que proporciona este hecho, destaca la compatibilidad automática de las aplicaciones web con distintos tipos de navegadores. La implementación de los distintos controles se encargará de aprovechar la funcionalidad de los navegadores modernos (como JavaScript o HTML dinámico), sin que esto suponga que nuestra aplicación deje de funcionar en navegadores más antiguos (los que se limitan a soportar HTML 3.2).

Dentro de las páginas ASP.NET, los controles se indican en el fichero `.aspx` utilizando etiquetas de la forma `<asp:... />`. Para implementar el ejemplo de la sección anterior utilizando controles, lo único que tenemos que hacer es crear una página ASP.NET con una etiqueta (componente `asp:Label`). Si estamos utilizando Visual Studio .NET como entorno

de desarrollo, creamos un nuevo proyecto de tipo *Aplicación Web ASP.NET*. Al aparecernos un formulario web vacío, modificamos su propiedad `pageLayout` para utilizar el estilo `FlowLayout` típico de las páginas web, en las que los controles no se colocan en coordenadas fijas (a diferencia de la forma habitual de trabajar con formularios Windows). Para que el contenido de la página aparezca centrado, utilizamos uno de los botones de la barra de herramientas igual que si estuviésemos trabajando con un procesador de textos. Finalmente, añadimos una etiqueta [`Label`] que obtenemos del cajón *Web Forms* del cuadro de herramientas. Una vez que tenemos el control en nuestro formulario web, lo normal es que modifiquemos su identificador (propiedad `(ID)`) y el texto que muestra en el formulario (propiedad `Text`). Como resultado de este proceso obtenemos el siguiente fichero `.aspx`, en el que aparece nuestro control como una etiqueta `asp:Label`:

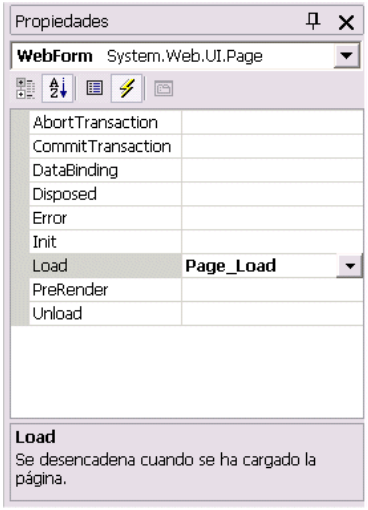
```
<%@ Page language="c#"
         Codebehind="WebControlExample.aspx.cs"
         AutoEventWireup="false"
         Inherits="WebControlExample.WebForm" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
  <HEAD>
    <title>WebForm</title>
    <meta name="GENERATOR" Content="Microsoft Visual Studio .NET 7.1">
    <meta name="CODE_LANGUAGE" Content="C#">
  </HEAD>
  <body>
    <form id="Form1" method="post" runat="server">
      <P align="center">
        <asp:Label id="LabelFecha" runat="server">Hoy es...</asp:Label>
      </P>
    </form>
  </body>
</HTML>
```

El fichero mostrado define los controles existentes en nuestra interfaz de usuario e incluye información adicional acerca de cómo ha de procesarse la página ASP.NET. De todo lo que ha generado el Visual Studio, el detalle más significativo es la inclusión del atributo `runat="server"` en todos aquellos componentes de nuestra interfaz a los que podemos asociarles manejadores de eventos que implementen la lógica de la aplicación como respuesta a las acciones del usuario. Dichos manejadores de eventos se ejecutarán siempre en el servidor y serán los encargados de que nuestras aplicaciones sean algo más que páginas estáticas en las que se muestra información.

IMPORTANTE

Todos los controles en una página ASP.NET deben estar dentro de una etiqueta `<form>` con el atributo `runat="server"`. Además, ASP.NET requiere que todos los elementos HTML estén correctamente anidados y cerrados (como sucede en XML).

Si deseamos lograr el mismo efecto que conseguíamos en la sección anterior al introducir un fragmento de código dentro de nuestro fichero `.aspx`, podemos especificar la respuesta de nuestro formulario al evento `Page_Load`, que se produce cuando el usuario accede a la página ASP.NET desde su navegador. En un entorno de programación visual como Visual Studio .NET, sólo tenemos que buscar el evento asociado al componente `WebForm` y hacer doble click sobre él.



El evento Page_Load nos permite especificar acciones que deseamos realizar antes de mostrarle la página al usuario.

Acto seguido, implementamos la lógica de la aplicación como respuesta al evento dentro del fichero de código C# asociado a la página ASP.NET. En el ejemplo que nos ocupa sólo tenemos que escribir una línea de código dentro del método creado automáticamente por el Visual Studio .NET para gestionar el evento `Page_Load`:

```
private void Page_Load(object sender, System.EventArgs e)
{
  LabelFecha.Text = "Hoy es " + DateTime.Now.ToString("D");
}
```

En realidad, el código de nuestra aplicación es algo más complejo. Aunque nosotros sólo nos preocupemos de escribir el código que muestra la fecha actual, el entorno de desarrollo se encarga de hacer el resto del trabajo por nosotros. El código completo asociado a nuestro formulario, eliminando comentarios y sentencias `using`, será el mostrado a continuación:

```
public class WebForm : System.Web.UI.Page
{
  protected System.Web.UI.WebControls.Label LabelFecha;

  private void Page_Load(object sender, System.EventArgs e)
  {
    LabelFecha.Text = "Hoy es " + DateTime.Now.ToString("D");
  }

  #region Código generado por el Diseñador de Web Forms

  override protected void OnInit(EventArgs e)
  {
    InitializeComponent();
    base.OnInit(e);
  }

  private void InitializeComponent()
  {
    this.Load += new System.EventHandler(this.Page_Load);
  }

  #endregion
}
```

La región de código que no hemos implementado nosotros se encarga, básicamente, de inicializar los componentes de nuestra página con las propiedades que fijamos en el diseñador de formularios (aquéllas que aparecen en el fichero `.aspx`) y, además, les asigna los manejadores de eventos que hemos implementado a los distintos eventos asociados a los componentes de nuestra interfaz; esto es, enlaza nuestro código con los componentes de la interfaz. En el ejemplo mostrado, el único evento para el cual nuestra aplicación tiene una respuesta específica es el evento `Load` del formulario web.

El ejemplo anterior puede conducir a conclusiones erróneas si no se analiza con detalle. Puede parecer que el formulario web construido utilizando controles resulta excesivamente complejo en comparación con la implementación equivalente que vimos en la sección anterior de este capítulo. No obstante, la organización modular de la página ASP.NET simplifica el desarrollo de interfaces complejos y su posterior mantenimiento. Resulta mucho más sencillo modificar el código incluido en un método de una clase que el código disperso entre etiquetas propias de la interfaz de usuario. Además, la comprobación del correcto funcionamiento del código implementado también será más sencilla cuando sólo incluimos en el fichero `.aspx` los controles de la interfaz de usuario e implementamos su funcionalidad aparte. Por otra parte, la complejidad extra que supone utilizar los controles no repercute en el trabajo del programador, ya que el entorno de desarrollo se encarga de generar automáticamente el código necesario para enlazar los controles con el código implementado por el programador.

Como hemos visto, al usar controles en nuestra interfaz web, la lógica de la aplicación se implementa como respuesta de la interfaz de usuario a los distintos eventos que puedan producirse, exactamente igual que en cualquier entorno de programación visual para

Windows. El servidor web será el encargado de interpretar las etiquetas correspondientes a los controles ASP.NET y visualizarlos correctamente en el navegador del usuario, para lo cual tendrá que generar el código HTML que resulte más apropiado.

Una vez que hemos visto el funcionamiento de los controles en las páginas ASP.NET, podemos pasar a analizar los distintos tipos de controles que podemos incluir en nuestras páginas web. En la plataforma .NET, existen tres tipos de controles predefinidos que se pueden utilizar en las páginas ASP.NET:

- Los controles HTML representan etiquetas HTML tradicionales y funcionan de forma similar a los objetos utilizados en JavaScript para manipular dinámicamente el contenido de una página web.

- Los controles web son los controles asociados a las etiquetas específicas de ASP.NET y facilitan el desarrollo de interfaces web utilizando un entorno de programación visual como Visual Studio .NET.

- Por último, existe una serie de controles de validación que se emplean para validar las entradas introducidas por el usuario de una forma relativamente cómoda (aunque no siempre resulta la más adecuada desde el punto de vista del diseño de la interfaz de usuario).

Aparte de estos controles, que ya vienen preparados para su utilización, el programador puede crear sus propios controles para simplificar la creación de interfaces consistentes y evitar la duplicación innecesaria de código en distintas partes de una aplicación web.

En los siguientes apartados de esta sección analizaremos con algo más de detalle los distintos tipos de controles que se pueden utilizar en las páginas ASP.NET y veremos cómo podemos crear nuestros propios controles fácilmente.

Controles HTML

Por defecto, las etiquetas HTML incluidas en una página ASP.NET se tratan como texto en el servidor web y se envían tal cual al cliente. No obstante, en determinadas ocasiones puede que nos interese manipular su contenido desde nuestra aplicación, que se ejecuta en el servidor. Para que las etiquetas HTML sean programables en el servidor, sólo tenemos que añadirles el atributo `runat="server"`.

En el siguiente ejemplo veremos cómo podemos hacer que un enlace HTML apunte dinámicamente a la URL que nos convenga en cada momento. En HTML, los enlaces se marcan con la etiqueta `<A>`. Para poder modificar las propiedades del enlace en HTML, sólo tenemos que añadir el atributo `runat="server"` a la etiqueta estándar `<A>` y asociarle al enlace un identificador adecuado. El fichero `.aspx` de nuestra página ASP.NET quedaría como se muestra a continuación:

```
<html>
  ...
  <body>
    <form id="HTMLControl" method="post" runat="server">
      <a id="enlace" runat="server">¡Visite nuestra página!</a>
    </form>
  </body>
<html>
```

Una vez que hemos marcado el enlace con el atributo `runat="server"`, podemos modificar sus propiedades accediendo a él a través del identificador que le hayamos asociado. En ASP.NET, los enlaces HTML de una página se representan mediante el control `HtmlAnchor`. Una de las propiedades de este control (`HRef`) indica la URL a la que apunta el enlace, por lo que sólo tenemos que establecer un valor adecuado para esta propiedad en el código asociado a alguno de los eventos de la página ASP.NET. El fichero de código resultante tendría el siguiente aspecto:

```
public class HTMLControl : System.Web.UI.Page
{
  protected System.Web.UI.HtmlControls.HtmlAnchor enlace;

  private void Page_Load(object sender, System.EventArgs e)
  {
    enlace.HRef = "http://csharp.ikor.org/";
  }

  override protected void OnInit(EventArgs e)
  {
    this.Load += new System.EventHandler(this.Page_Load);
    base.OnInit(e);
  }
}
```

Si utilizamos Visual Studio .NET, para poder manipular un control HTML en el servidor sólo tenemos que seleccionar la opción "Ejecutar como control del servidor" del menú contextual asociado a la etiqueta HTML en el diseñador de formularios web. Esto hace que se añada al atributo `runat="server"` a la etiqueta en el fichero `.aspx` y que se incluya la declaración correspondiente en la clase que define nuestro formulario, con lo cual ya podemos programar el comportamiento del control HTML en función de nuestras necesidades accediendo a él como un miembro más de la clase que representa la página ASP.NET.

La biblioteca de clases de la plataforma .NET incluye una gama bastante completa de componentes que encapsulan las distintas etiquetas que pueden aparecer en un documento HTML. Dichos componentes se encuentra en el espacio de nombres `System.Web.UI.HtmlControls`.

Las etiquetas más comunes en HTML tienen su control HTML equivalente en ASP.NET. Este es el caso de los enlaces o las imágenes en HTML, los cuales se representan como objetos de tipo `HtmlAnchor` y `HtmlImage` en las páginas ASP.NET, respectivamente. Si, por cualquier motivo, nos encontramos con que no existe un control HTML específico para representar una etiqueta HTML determinada, esto no impide que podamos manipularla desde nuestra aplicación web. Existe un control genérico, denominado `HtmlGenericControl`.

La siguiente tabla resume los controles HTML disponibles en ASP.NET, las etiquetas a las que corresponden en HTML estándar y una breve descripción de su función en la creación de páginas web:

Control HTML	Etiqueta HTML	Descripción
`HtmlAnchor`	`<a>`	Enlace
`HtmlButton`	`<button>`	Botón
`HtmlForm`	`<form>`	Formulario
`HtmlGenericControl`		Cualquier elemento HTML para el cual no existe un control HTML específico
`HtmlImage`	`<image>`	Imagen
`HtmlInput...`	`<input type="...">`	Distintos tipos de entradas en un formulario HTML: botones (`button`, `submit` y `reset`), texto (`text` y `password`), opciones (`checkbox` y `radio`), imágenes (`image`), ficheros (`file`) y entradas ocultas (`hidden`).
`HtmlSelect`	`<select>`	Lista de opciones en un formulario
`HtmlTable...`	`<table>` `<tr>` `<td>`	Tablas, filas y celdas
`HtmlTextArea`	`<textarea>`	Texto en un formulario

Controles web

La principal aportación de ASP.NET a la creación de interfaces web es la inclusión de controles específicos que aíslan al programador del HTML generado para presentar el formulario al usuario de la aplicación. Como ya mencionamos anteriormente, estos controles

permiten desarrollar aplicaciones web compatibles con distintos navegadores y facilitan la tarea del programador al ofrecerle un modelo de programación basado en eventos.

En el fichero `.aspx` asociado a una página ASP.NET, los controles web se incluyen utilizando etiquetas específicas de la forma `<asp:... />`. La sintaxis general de una etiqueta ASP.NET es de la siguiente forma:

```
<asp:control id="identificador" runat="server" />
```

Dada una etiqueta como la anterior:

- `control` variará en función del tipo de control que el programador decida incluir en su página ASP.NET. La etiqueta concreta de la forma `asp:... />` será diferente para mostrar un texto (`asp:Label`), un botón (`asp:Button`), una lista convencional (`asp:ListBox`) o una lista desplegable (`asp:DropDownList`), por mencionar algunos ejemplos.

- `identificador` especifica el identificador que le asociamos a la variable mediante la cual accederemos al control desde el código de nuestra aplicación.

- Finalmente, la inclusión del atributo `runat="server"` es necesaria para indicar que el programador puede manipular la etiqueta ASP.NET desde el servidor implementando manejadores de eventos para los distintos eventos a los que pueda responder el control representado por la etiqueta ASP.NET.

Igual que sucedía con las etiquetas HTML, en la biblioteca de clases de la plataforma .NET existen componentes que nos permiten manipular dichas las etiquetas ASP.NET desde el código de la aplicación. Dichos componentes encapsulan a las distintas etiquetas ASP.NET y se encuentran en el espacio de nombres `System.Web.UI.WebControls`. De hecho, todos ellos derivan de la clase `WebControl`, que se encuentra en el mismo espacio de nombres.

Uso de controles ASP.NET

Para mostrar el uso de los controles ASP.NET, podemos crear una aplicación web ASP.NET desde el Visual Studio .NET. Dicha aplicación contiene, por defecto, un formulario web vacío al que denominamos `WebControl.aspx`. A continuación, le añadimos un botón al formulario utilizando el control `Button` que aparece en la sección *Web Forms* del *Cuadro de herramientas* de Visual Studio .NET.

Al añadir el botón, en el fichero `.aspx` de la página ASP.NET aparece algo similar a lo siguiente:

Uso de controles ASP.NET

```
<form id="WebControl" method="post" runat="server">
  <asp:Button id="Button" runat="server" Text="Pulse el botón">
  </asp:Button>
</form>
```

Así mismo, en el fichero de código asociado a la página aparece una declaración de la forma:

```
protected System.Web.UI.WebControls.Button Button;
```

Ahora sólo nos falta añadirle algo de funcionalidad a nuestra aplicación. Para lograrlo, buscamos los eventos a los cuales ha de reaccionar nuestro formulario web e implementamos los manejadores de eventos correspondientes. Por ejemplo, desde el mismo diseñador de formularios web del Visual Studio .NET, podemos hacer doble click sobre el botón para especificar la respuesta de este control al evento que se produce cuando el usuario pulsa el botón desde su navegador web:

```
private void Button_Click(object sender, System.EventArgs e)
{
  Button.Text = "Ha pulsado el botón";
}
```

Obviamente, la lógica asociada a los eventos de una aplicación real será bastante más compleja (y útil), si bien la forma de trabajar del programador será siempre la misma: implementar la respuesta de la aplicación frente a aquellos eventos que sean relevantes para lograr la funcionalidad deseada.

Como se puede comprobar, el desarrollo de aplicaciones web con controles ASP.NET es completamente análogo al desarrollo de aplicaciones para Windows. Sólo tenemos que seleccionar los controles adecuados para nuestra interfaz e implementar la respuesta de nuestra aplicación a los eventos que deseemos controlar.

Aunque la forma de desarrollar la interfaz de usuario sea la misma cuando se utilizan formularios web ASP.NET y cuando se emplean formularios para Windows, los controles que se pueden incluir cambian en función del tipo de interfaz que deseemos crear. En el caso de los formularios web, podemos utilizar cualquiera de los controles definidos en el espacio de nombres `System.Web.UI.WebControls` de la biblioteca de clases de la plataforma .NET. La mayoría de ellos corresponden a los controles típicos que uno podría esperar encontrarse en cualquier interfaz gráfica actual, si bien también existen otros muy útiles para el programador en la creación de interfaces más sofisticadas (como es el caso de los componentes `asp:Repeater` y `asp:DataGrid`, que mencionaremos más adelante en este mismo capítulo).

La tabla que ofrecemos a continuación resume, a modo de guía, cuáles son los controles ASP.NET y menciona su función en la creación de interfaces web con ASP.NET:

Control	Descripción
`AdRotator`	Muestra una secuencia de imágenes (a modo de banner)
`Button`	Botón estándar
`Calendar`	Calendario mensual
`CheckBox`	Caja de comprobación (como en los formularios Windows)
`CheckBoxList`	Grupo de cajas de comprobación
`DataGrid`	Rejilla de datos
`DataList`	Muestra una lista utilizando plantillas (*templates*)
`DropDownList`	Lista desplegable
`HyperLink`	Enlace
`Image`	Imagen
`ImageButton`	Botón dibujado con una imagen
`Label`	Etiqueta de texto estático
`LinkButton`	Botón con forma de enlace
`ListBox`	Lista (como en los formularios Windows)
`Literal`	Texto estático (similar a `Label`)
`Panel`	Contenedor en el que se pueden colocar otros controles
`PlaceHolder`	Reserva espacio para controles añadidos dinámicamente
`RadioButton`	Botón de radio (como en los formularios Windows)
`RadioButtonList`	Grupo de botones de radio
`Repeater`	Permite mostrar listas de controles
`Table`	Tabla
`TextBox`	Caja de edición
`Xml`	Muestra un fichero XML o el resultado de una transformación XSL

Como se puede apreciar a partir de la lista de controles de la tabla, la gama de controles estándar ya disponibles es bastante amplia, lo suficiente como para poder construir interfaces gráficos estándar sin tener que preocuparnos demasiado de los detalles de implementación de los controles de la interfaz de usuario.

Controles de validación

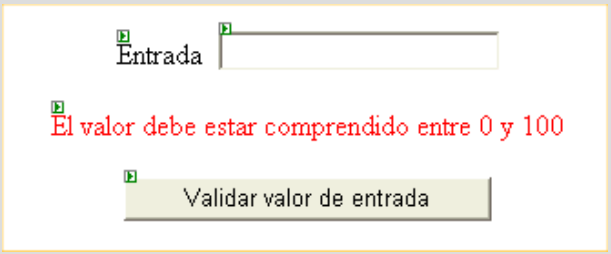
Para finalizar nuestro recorrido por los componentes predefinidos en la biblioteca de clases de la plataforma .NET para la creación de interfaces web, mostraremos un tercer grupo de componentes que puede utilizarse para imponer ciertas restricciones a los datos introducidos por el usuario: los controles de validación.

Los controles de validación son un tipo especial de controles ASP.NET, por lo que también están incluidos en el espacio de nombres `System.Web.UI.WebControls`. Estos controles se enlazan a controles ASP.NET de los descritos en el apartado anterior para validar las entradas de un formulario web. Cuando el usuario rellena los datos de un formulario y alguno de los datos introducidos no verifica la restricción impuesta por el control de validación, este control se encarga de mostrarle un mensaje de error al usuario.

La validación de las entradas de un formulario web se realiza automáticamente cuando se pulsa un botón, ya tenga éste la forma de un botón estándar (`Button`), de una imagen (`ImageButton`) o de un enlace (`LinkButton`). No obstante, se puede desactivar esta validación si establecemos a `false` la propiedad `CausesValidation` del botón correspondiente. Igualmente, se puede forzar la validación manualmente en el momento que nos interese mediante el método `Validate` asociado a la página ASP.NET.

Ejemplo de uso de controles de validación

Podemos forzar a que una entrada determinada esté dentro de un rango válido de valores con un control de validación del tipo `RangeValidator`:



Para ver cómo funcionan los controles de validación, podemos crear una aplicación web ASP.NET con un formulario como el mostrado en la imagen de arriba, en el cual incluimos una etiqueta (`Label`), una caja de texto (`TextBox`), un botón (`Button`) y un control de validación (`RangeValidator`), todos ellos controles incluidos en la pestaña *Web Forms* del cuadro de herramientas de Visual Studio .NET.

Para que nuestro formulario web utilice el control de validación, sólo tenemos que establecer

Ejemplo de uso de controles de validación

la propiedad `ControlToValidate` de este control para que haga referencia al `TextBox` que usamos como entrada. Mediante la propiedad `ErrorMessage` especificamos el mensaje de error que se mostrará (en rojo) cuando el valor introducido por el usuario no cumpla la condición impuesta por el control de validación. Finalmente, esta condición la indicamos utilizando las propiedades específicas del control de validación que hemos escogido: el tipo de dato (`Type=Integer`) y el rango de valores permitidos (entre `MinimumValue` y `MaximumValue`).

Cuando los datos introducidos por el usuario son válidos, la aplicación prosigue su ejecución de la forma habitual. Cuando no se verifica alguna condición de validación, se muestra el mensaje de error asociado al control de validación y se le vuelve a pedir al usuario que introduzca correctamente los datos de entrada. Obviamente, la validación la podríamos haber realizado implementando la respuesta de nuestra aplicación a los eventos asociados utilizando controles web estándar, aunque de esta forma, ASP.NET se encargará de generar automáticamente las rutinas de validación y automatizar, al menos en parte, algunas de las comprobaciones más comunes que se realizan al leer datos de entrada en cualquier aplicación.

Por cuestiones de seguridad, los controles de validación siempre validan los datos de entrada en el servidor, con el objetivo de que un cliente malintencionado no pueda saltarse algunas comprobaciones acerca de la validez de los datos. Aun cuando la comprobación en el servidor siempre se realiza, si el navegador del usuario acepta HTML dinámico, la validación también se efectúa en el cliente. Al realizar la comprobación antes de enviar los datos al servidor, se consigue disminuir la carga del servidoral ahorrarnos un viaje de ida y vuelta cuando los datos de entrada no verifican las restricciones impuestas por los controles de validación.

A pesar de su evidente utilidad, no conviene abusar demasiado del uso de controles de validación porque, desde el punto de vista del diseño de interfaces de usuario, su implementación dista de ser adecuada. Cuando el usuario introduce algún dato erróneo, el error que se ha producido no siempre llama su atención. De hecho, lo único que sucede en la interfaz de usuario es la aparición de una etiqueta, usualmente en rojo, que indica en qué consiste el error. Desafortunadamente, esta etiqueta puede que se encuentre bastante lejos del punto en el cual el usuario tiene fija su atención. Por tanto, para el usuario el error pasa desapercibido y no siempre le resulta obvio darse cuenta de por qué se le vuelve a presentar el mismo formulario que acaba de rellenar. Para evitar situaciones como la descrita resulta aconsejable utilizar el control `asp:ValidationSummary`, que resume en una lista los errores de validación que se hayan podido producir al rellenar un formulario web.

Los controles de validación predefinidos en la biblioteca de clases de la plataforma .NET se encuentran, como mencionamos al comienzo de esta sección, en el espacio de nombres `System.Web.UI.WebControls`. Todos los controles de validación derivan de la clase `System.Web.UI.WebControls.BaseValidator`. En la siguiente tabla se recoge cuáles son y qué utilidad tienen:

Control de validación	Descripción
`CompareValidator`	Compara el valor de una entrada con el de otra o un valor fijo.
`CustomValidator`	Permite implementar un método cualquiera que maneje la validación del valor introducido.
`RangeValidator`	Comprueba que la entrada esté entre dos valores dados.
`RegularExpressionValidator`	Valida el valor de acuerdo a un patrón establecido como una expresión regular.
`RequiredFieldValidator`	Hace que un valor de entrada sea obligatorio.

Controles creados por el usuario

En las secciones anteriores de este capítulo hemos visto cuáles son los controles predefinidos que podemos utilizar en la creación de interfaces web con ASP.NET. En esta, veremos cómo podemos crear nuestros propios controles.

En ASP.NET, el programador puede crear sus propios controles de dos formas diferentes:

- **Por composición:** A partir de una colección de controles ya existentes, el programador decide cómo han de visualizarse conjuntamente. En vez de tener que repetir la disposición del conjunto de controles cada vez que hayan de utilizarse en la aplicación, la colección de controles se encapsula en un único control. Cada vez que se desee, se puede añadir el control a un formulario web para mostrar el conjunto completo de controles a los que encapsula.

- **Por derivación:** Igual que en cualquier otro entorno de programación orientado a objetos, los controles se implementan como clases. Estas clases heredarán, directa o indirectamente, de la clase base de todos los controles web: la clase `System.Web.UI.WebControl` (la cual está definida en la biblioteca `System.Web.dll`).

En esta sección nos centraremos en la primera de las alternativas, la creación de controles de usuario por composición, porque su uso es muy habitual en la creación de aplicaciones web en ASP.NET. Como veremos en los próximos capítulos del libro, los controles ASP.NET creados de esta manera son los que nos permiten construir aplicaciones modulares que sean fácilmente mantenibles y extensibles.

Dado que los controles definidos por composición se utilizan como bloque básico en la construcción de aplicaciones web con ASP.NET, lo usual es que dichos controles se adapten a las necesidades de cada aplicación particular. Esto, obviamente, no impide que podamos crear controles reutilizables en distintas aplicaciones si hemos de realizar el mismo tipo de tareas en diferentes proyectos de desarrollo.

Supongamos ahora que deseamos crear una aplicación web en la cual se almacene información de contacto relativa a diferentes personas, como podría ser el caso de las entradas de una sencilla agenda, un cliente de correo electrónico o una completa aplicación de gestión en la que tuviésemos que mantener información acerca de clientes o proveedores. En cualquiera de las situaciones mencionadas, en nuestra aplicación existirá una clase que encapsule los datos relativos a distintas personas. La implementación de dicha clase podría tener un aspecto similar, aunque nunca igual, al que se muestra a continuación:

```
public class Contact
{
  public string Name;        // Nombre
  public string EMail;       // Correo electrónico
  public string Telephone;   // Teléfono
  public string Mobile;      // Teléfono móvil
  public string Fax;         // Fax
  public string Address;     // Dirección
  public string Comments;    // Anotaciones
  public string ImageURL;    // URL de su fotografía
}
```

Encapsulación

Cualquier purista de la programación orientada a objetos se estremecería al ver un fragmento de código como el anterior, y tendría razones para ello. La idea básica de la utilización de objetos es encapsular la implementación de su interfaz, algo que no se consigue si todas las variables de instancia son públicas. Siempre resulta aconsejable mantener los datos en variables privadas las de una clase y acceder a ellos mediante métodos.

El lenguaje de programación C# nos ofrece un mecanismo muy cómodo para conseguir el mismo resultado: el uso de propiedades. Las propiedades nos permiten seguir utilizando la clase de la misma forma y, al mismo tiempo, mantener la encapsulación de los datos. Si decidimos emplear propiedades, la implementación de la clase anterior resulta algo más extensa. Este es el único motivo por el que, inicialmente, decidimos utilizar variables públicas en el fragmento de código anterior.

Por tanto, cada variable de instancia de las que antes habíamos declarado públicas debería, al menos, convertirse en un fragmento de código como el siguiente:

```
...
private string _variable;

public string Variable {
  get { return _variable; }
  set { _variable = value; }
}
...
```

Una vez que tenemos clases que encapsulan los datos con los que nuestra aplicación ha de trabajar, hemos de crear una interfaz de usuario adecuada para que el usuario de nuestra aplicación pueda trabajar con los datos.

Clases como la anterior se denominan habitualmente clases modelo. Este apelativo proviene del hecho de que estas clases son las que modelan el dominio del problema que nuestra aplicación pretende resolver (la representación de datos de contacto en nuestro ejemplo). Éstas son las clases que suelen aparecer en un diagrama de clases UML o en el modelo de datos de una base de datos.

Para construir la interfaz de usuario asociada a nuestra clase modelo, lo que haremos será crear vistas que nos permitan mostrar de distintas formas los datos que encapsula la clase modelo. Por ejemplo, puede que nos interese mostrar los datos de contacto de una persona en una tabla como la siguiente:

Nombre	Nombre del contacto	
E-mail	mailbox@domain	
Teléfono	999 999 999	
Móvil	999 999 999	
Fax	999 999 999	
Dirección	Calle Localidad CP Provincia	
Comentarios		

Presentación visual de los datos de contacto de una persona

La presentación visual de los datos de contacto requiere la utilización de distintos controles, como las etiquetas que nos permitirán mostrar los datos concretos del contacto o la imagen que utilizaremos para visualizar la fotografía de nuestro contacto. Si en nuestra aplicación tuviésemos que mostrar la información de contacto en diferentes situaciones, resultaría bastante tedioso tener que copiar la tabla anterior y repetir todo el código necesario para rellenarla con los datos particulares de una persona. Aun pudiendo usar copiar y pegar, esta estrategia no resulta demasiado razonable. ¿Qué sucedería entonces si hemos de cambiar algún detalle de la presentación visual de los datos de contacto? Tendríamos que buscar todos los sitios de nuestra aplicación en los que aparezca una tabla como la anterior y realizar la misma modificación en diferentes lugares. Esto no solamente resulta incómodo, sino una situación así es bastante proclive a que, al final, cada vez que mostremos los datos de una persona lo hagamos de una forma ligeramente diferente y el comportamiento de nuestra aplicación no sea todo lo homogéneo que debería ser. En otras palabras, olvidar la realización de las modificaciones en alguna de las apariciones del fragmento duplicado provoca errores. Además, todos sabemos que la existencia de código duplicado influye negativamente en la mantenibilidad de nuestra aplicación y, a la larga, en su calidad.

Por suerte, ASP.NET incluye el mecanismo adecuado para evitar situaciones como las descritas en el párrafo anterior: los controles de usuario creados por composición. Un conjunto de controles como el que utilizamos para mostrar los datos de contacto de una persona se puede encapsular en un único control que después emplearemos en la creación de nuestros formularios web. El mecanismo utilizado es análogo a los `frames` de Delphi o C++Builder.

Para encapsular un conjunto de controles en un único control, lo único que tenemos que hacer es crear un "control de usuario Web", tal como aparece traducido en Visual Studio .NET el término inglés *web user control*, aunque quizá sería más correcto decir "control web de usuario". En realidad, un control de este tipo se parece más a un formulario web que a un control de los que existen en el "cuadro de herramientas" (la discutible traducción de *toolbox* en Visual Studio).

Un control web de usuario nos permite definir el aspecto visual conjunto de una colección de controles a los que encapsula. El control es similar a una página ASP.NET salvo que hereda de la clase `System.Web.UI.UserControl` en vez de hacerlo de `System.Web.UI.Page`.

Por convención, los controles web de usuario se almacenan en ficheros con extensión `.ascx` (en lugar de la extensión `.aspx` de los formularios ASP.NET). Dicho fichero será el que se incluya luego dentro de una página ASP.NET, por lo que en él no pueden figurar las etiquetas `<html>`, `<head>`, `<body>`, `<form>` y `<!DOCTYPE>`, las cuales sí aparecen en un formulario ASP.NET.

Teniendo en cuenta lo anterior, podemos crear un control web de usuario para mostrar los datos de contacto de una persona. Dicho control se denominará `ContactViewer` porque nos permite visualizar los datos de contacto pero no modificarlos y se creará de forma análoga a como se crean los formularios ASP.NET.

En primer lugar, creamos un fichero con extensión `.ascx` que contenga los controles de la interfaz de usuario que nuestro control ha de encapsular. Dicho fichero se denominará `ContactViewer.ascx` y tendrá el siguiente aspecto:

```
<%@ Control Language="c#"
            AutoEventWireup="false"
            Codebehind="ContactViewer.ascx.cs"
            Inherits="WebMail.ContactViewer"%>
<DIV align="center">
  <TABLE id="TableContact" width="90%" border="0">
    <TR>
      <TD width="100" bgColor="#cccccc">
          Nombre
      </TD>
      <TD bgColor="#e0e0e0">
        <asp:Label id="LabelName" runat="server">
          Nombre del contacto
        </asp:Label>
      </TD>
      <TD vAlign="middle" align="center" width="20%" rowSpan="6">
        <asp:Image id="ImagePhoto" runat="server" Height="200px">
        </asp:Image>
      </TD>
    </TR>
    <TR>
      <TD bgColor="#cccccc">
          E-mail
      </TD>
      <TD bgColor="#e0e0e0">
        <asp:Label id="LabelEMail" runat="server">
           mailbox@domain
        </asp:Label>
      </TD>
    </TR>
    <!-- Lo mismo para los demás datos del contacto -->
    ...
    <TR>
      <TD colSpan="3">
        <P>
          <asp:Label id="LabelComments" runat="server">
            Comentarios
          </asp:Label>
        </P>
      </TD>
    </TR>
  </TABLE>
</DIV>
```

Como se puede apreciar, su construcción es similar al de un formulario ASP.NET. Sólo cambia la directiva que aparece al comienzo del fichero. En vez de utilizar la directiva `<%@ Page...` propia de los formularios ASP.NET, se emplea la directiva `<%@ Control..` específica para la creación de controles por composición. Aparte de eso y de la ausencia de algunas etiquetas (`<html>`, `<head>`, `<body>`, `<form>` y `<!DOCTYPE>`), no existe ninguna otra diferencia entre un fichero `.ascx` y un fichero `.aspx`.

Igual que sucedía en las páginas ASP.NET, al crear un control separaremos la lógica de la interfaz implementando el código asociado al control en un fichero de código aparte. A este fichero hace referencia el atributo `Codebehind` de la directiva `<%@ Control..` que aparece al comienzo del fichero `.ascx`. En el ejemplo que estamos utilizando, el contenido del fichero de código `ContactViewer.ascx.cs` será el que se muestra a continuación:

```
public class ContactViewer : System.Web.UI.UserControl
{
  // Controles de la interfaz de usuario

  protected...

  // Modelo asociado a la vista

  public Contact DisplayedContact {
      set { UpdateUI(value); }
  }

  private void UpdateUI (Contact contact) {

    if (contact!=null) {
       LabelName.Text = contact.Name;
       LabelEMail.Text = contact.EMail;
       LabelTelephone.Text = contact.Telephone;
       LabelMobile.Text = contact.Mobile;
       LabelFax.Text = contact.Fax;
       LabelAddress.Text = contact.Address;
       LabelComments.Text = contact.Comments;

       ImagePhoto.ImageUrl = "contacts/"+contact.ImageURL;
    }
  }

  // Código generado por el diseñador de formularios
  ...
}
```

Como se puede apreciar, el código que hemos de implementar se limita a mostrar en las distintas etiquetas de nuestra interfaz los datos correspondientes al contacto que queremos visualizar. Para mostrar la fotografía correspondiente, lo único que hacemos es establecer la URL de la imagen mostrada en la parte derecha de la tabla. Cuando queramos utilizar este control para mostrar los datos de contacto de alguien, lo único que tendremos que hacer es establecer un valor adecuado para su propiedad `DisplayedContact`.

Ya hemos visto en qué consiste un control ASP.NET definido por el usuario, aunque aún no hemos dicho nada acerca de cómo se crean y cómo se utilizan los controles web en la práctica.

Para crear un control web de usuario como el que hemos utilizado de ejemplo en este apartado, podemos utilizar tres estrategias diferentes:

- **Creación manual:** Podemos seguir el mismo proceso que hemos visto en el ejemplo para construir un control ASP.NET escribiendo el contenido del fichero `.ascx` e implementando el código asociado al control en un fichero aparte con extensión `.ascx.cs`. Sólo hemos de tener en cuenta que, al comienzo del fichero `.ascx`, debe aparecer la directiva `<%@ Control...` y que el control debe heredar de la clase `System.Web.UI.UserControl`.

- **Conversión de una página ASP.NET en un control:** Para que resulte más cómoda la creación del control, podemos comenzar creando un formulario ASP.NET convencional. Una vez que el formulario haga lo que nosotros queramos, podemos convertir la página ASP.NET en un control modificando las extensiones de los ficheros. Esta estrategia también es válida cuando ya tenemos la página hecha y queremos reutilizarla. La conversión de una página en un control requiere cambiar la extensión de los ficheros (de `.aspx` y `.aspx.cs` a `.ascx` y `.ascx.cs`), eliminar las etiquetas que no pueden aparecer en el control (`<html>`, `<head>`, `<body>` y `<form>`, así como la directiva `<!DOCTYPE>`), cambiar la directiva que aparece al comienzo de la página (de `<%@ Page...` a `<%@ Control...`) y modificar la clase que implementa la funcionalidad del control para que herede de `System.Web.UI.UserControl` en lugar de hacerlo de `System.Web.UI.Page`.

- **Con Visual Studio .NET**: Cuando estamos trabajando con una aplicación web, podemos añadir un "control de usuario web" desde el menú contextual asociado a nuestro proyecto en el *Explorador de Soluciones* de Visual Studio .NET. Esto nos permite utilizar el diseñador de formularios para implementar nuestro control igual que se implementa un formulario web ASP.NET.

Utilizando cualquiera de las tres estrategias mencionadas, al final obtendremos un control implementado en dos ficheros, uno para definir el aspecto visual de la interfaz de usuario (el fichero con extensión `.ascx`) y otro para el código que implementa la funcionalidad del control (el fichero que tiene extensión `.ascx.cs`).

Una vez que hemos creado el control, sólo nos falta ver cómo utilizarlo. Básicamente, el control que acabamos de mostrar lo emplearemos en la creación de formularios web ASP.NET como cualquier otro de los controles que ya vimos en los apartados anteriores de este capítulo. La única diferencia es que nuestro control no podemos ponerlo en el "cuadro de herramientas" del Visual Studio .NET.

La forma más sencilla de utilizar un control web creado por el usuario es, en el mismo entorno de desarrollo, arrastrar el control desde el *Explorador de Soluciones* hasta el lugar del formulario web en el que deseamos que aparezca nuestro control. Obviamente, primero hemos de incluir el control en nuestro proyecto actual.

Tras arrastrar el control a una página, en el diseñador de formularios veremos algo similar a lo mostrado en la siguiente imagen:

El control que hemos creado aparece en nuestra página ASP.NET como un simple botón. Será cuando ejecutemos la página cuando aparezca el control tal como lo hemos diseñado.

La sencilla operación de arrastrar y soltar que realizamos en el diseñador de formularios web se traduce en una serie de cambios en el fichero de nuestra página ASP.NET. En el ejemplo mostrado, la inclusión del control web en la página ASP.NET provoca que nuestro fichero `.aspx` tenga el siguiente aspecto:

```
<%@ Page language="c#"
         Codebehind="Contacts.aspx.cs"
         AutoEventWireup="false"
         Inherits="WebMail.Contacts" %>
<%@ Register TagPrefix="user"
             TagName="ContactViewer"
             Src="ContactViewer.ascx" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
  <HEAD>
    <title>Información de contacto</title>
  </HEAD>
  <body>
    <form id="FormContacts" method="post" runat="server">
      <P align="center">
        <asp:DropDownList id="ContactList"
                          runat="server" Width="80%"
                          AutoPostBack="True">
        </asp:DropDownList>
        <asp:Button id="ButtonOK" runat="server" Text="Mostrar">
        </asp:Button></P>
      <DIV align="center">
        <P align="center" id="Header" runat="server">
          Información de contacto de
          <asp:Label id="LabelContact" runat="server">
            ...
          </asp:Label></P>
      </DIV>
      <P align="center">
        <user:ContactViewer id="ContactView" runat="server">
        </user:ContactViewer>
      </P>
    </form>
  </body>
</HTML>
```

Si observamos detenidamente la página ASP.NET anterior, podremos apreciar los cambios introducidos por la inclusión de un control web de usuario. Conociendo en qué consisten estos cambios podríamos utilizar un control definido por el usuario sin necesidad de la ayuda del entorno de desarrollo.

En realidad, la utilización de un control web de usuario ocasiona que en nuestro fichero `.aspx` aparecen dos novedades: una directiva al comienzo de la página (la directiva `<%@ Register...`) y una etiqueta ASP.NET que hace referencia a nuestro control (`<user:ContactViewer...`).

La directiva `<%@ Register...` registra el uso del control en nuestra página ASP.NET, especificando el fichero donde se encuentra el control (`Src="ContactViewer.ascx"`) y la etiqueta que utilizaremos para incluir el control en nuestro formulario web (mediante los atributos `TagPrefix` y `TagName`). En cierto modo, esta directiva viene a ser como un `#define` de C.

```
<%@ Register TagPrefix="user"
             TagName="ContactViewer"
             Src="ContactViewer.ascx" %>
```

Una vez que hemos creado la forma de hacer referencia al control con la directiva `<%@ Register...`, para utilizar el control en la página lo incluiremos como incluiríamos cualquier otro control ASP.NET, utilizando la etiqueta que hemos definido (`user:ContactViewer` en el ejemplo). Cuando lo hacemos, hemos de especificar que se trata de un control de servidor; esto es, que el control tiene asociada funcionalidad que ha de ejecutarse al procesar la página ASP.NET en el servidor. Esto lo conseguimos con el atributo `runat="server"`. Finalmente, lo usual es que le asociemos un identificador al control que acabamos de incluir en la página ASP.NET. Esto se logra con el atributo `id="ContactView"`:

```
<user:ContactViewer id="ContactView" runat="server">
</user:ContactViewer>
```

Con el control incluido correctamente en la página ASP.NET, podemos pasar a ver cómo se pueden manipular sus propiedades en tiempo de ejecución. Si estamos trabajando con el Visual Studio .NET, pulsar la tecla `F7` nos llevará del diseñador de formularios web al código asociado a la página.

Si queremos utilizar el control desde el código asociado a la página, hemos de asegurarnos de que el control aparece como una variable de instancia de la clase que implementa la funcionalidad de la página ASP.NET. En otras palabras, en el fichero de código con extensión `.aspx.cs` debe existir una declaración de una variable protegida que haga referencia al

control de usuario incluido en la página. El tipo de esta variable coincidirá con el tipo de nuestro control y su identificador ha de coincidir con el identificador que hayamos indicado en el atributo `id` de la etiqueta `user:ContactViewer`:

```
  protected ContactViewer ContactView;
```

Volviendo al ejemplo anterior, el fichero de código completo asociado a nuestra página ASP.NET será el siguiente:

```
public class Contacts : System.Web.UI.Page
{
  protected System.Web.UI.WebControls.DropDownList ContactList;
  protected System.Web.UI.WebControls.Label LabelContact;
  protected System.Web.UI.WebControls.Button ButtonOK;
  protected System.Web.UI.HtmlControls.HtmlGenericControl Header;

  protected ContactViewer ContactView;

  private void Page_Load(object sender, System.EventArgs e)
  {
    // Inicialización de la lista desplegable
    ...
    UpdateUI(contact);
  }

  // Código generado por el diseñador (OnInit & InitializeComponents)
  ...

  // Manejadores de eventos
  ...

  // Actualización de la interfaz

  private void UpdateUI (Contact contact)
  {
    ContactView.DisplayedContact = contact;
    Header.Visible = true;
    ContactView.Visible = true;
  }
}
```

El código anterior se limita a establecer la propiedad `DisplayedContact` de nuestro control. El valor adecuado para esta propiedad lo obtendremos previamente de algún parámetro que le llegue a nuestra página ASP.NET, de un fichero o de una base de datos. Nuestro control se encargará internamente de visualizar los datos del contacto de la forma adecuada. Al ejecutar nuestra página ASP.NET, el resultado final que se le mostrará al usuario será similar al recogido en la siguiente figura:

Página ASP.NET con un control web definido por el usuario.

Creación de controles a medida

En este apartado hemos visto cómo se pueden crear controles por composición, aunque ya mencionamos al principio que ésta no es la única opción disponible. También se pueden crear controles por derivación a partir de clases que implementen controles ya existentes o directamente a partir de la clase base de todos los controles web: `System.Web.UI.WebControl`.

En la plataforma .NET, la creación de controles web de usuario se puede realizar de tres formas diferentes cuando optamos por crear el control por derivación:

- Podemos crear una clase que herede directamente de alguno de los controles ASP.NET existentes y le añada algo de funcionalidad, sin tener que preocuparnos de la presentación visual del control. Por ejemplo, podemos crear una clase que herede de `TextBox` y sólo permita la introducción de datos en un determinado formato interceptando algunos eventos de un `TextBox` convencional.

Creación de controles a medida

- Podemos crear un control completamente nuevo, para lo cual implementaremos una clase que herede directamente de `System.Web.UI.WebControl`. Además, el control puede que tenga que implementar interfaces como `IPostBackDataHandler` o `IPostBackEventHandler`, los cuales son interfaces relacionados con el funcionamiento de las páginas ASP.NET (que veremos en la siguiente sección de este mismo capítulo). La clase creada de esta forma ha de implementar todo el código que sea necesario para mostrar el control en la interfaz de usuario.

- Como última alternativa, podemos crear un "control compuesto" implementando una clase que simule el funcionamiento de un control creado por composición. En este caso, es necesario que el control incluya el interfaz `INamingContainer` como marcador y redefina el método `CreateChildComponents`, en el cual se crearán los controles que conforman el control compuesto.

Sea cual sea la estrategia utilizada para implementar el control, en la creación de controles por derivación deberemos emplear las técnicas habituales de programación orientada a objetos (encapsulación, herencia y polimorfismo). En otras palabras, hemos de implementar nuestras propias clases "a mano", sin ayuda de un diseñador visual como el diseñador de formularios web de Visual Studio .NET que se puede emplear en la creación de un control por composición.

Los controles creados por composición resultan adecuados para mostrar datos de una forma más o menos estática, mientras que los controles creados por derivación nos permiten ser más flexibles a la hora de generar el aspecto visual del control (que hemos de crear dinámicamente desde el código del control).

Al emplear composición, la reutilización del control se realiza a nivel del código fuente (el fichero `.ascx` y su fichero de código asociado). Por otro lado, si creamos un control por derivación, podemos compilarlo e incluirlo en un *assembly*. Esto nos permite añadir nuestro control al "cuadro de herramientas" de Visual Studio .NET. Además, al tener compilado el control, en tiempo de diseño podremos ver el aspecto visual del control en el diseñador de formularios, así como acceder a sus propiedades y eventos desde la "ventana de propiedades" del Visual Studio .NET.

En definitiva, la elección de una u otra alternativa dependerá, en gran medida, de lo que queramos conseguir y del tiempo del que dispongamos. A la hora de elegir cómo crear nuestros propios controles hemos de optar por la simplicidad de la creación por composición o la flexibilidad de uso que nos ofrecen los controles creados por derivación.

Funcionamiento de las páginas ASP.NET

En lo que llevamos de este capítulo, hemos visto cómo se pueden construir aplicaciones web utilizando formularios con controles de forma completamente análoga al desarrollo de aplicaciones para Windows en un entorno de programación visual. Aunque disponer de un diseñador de formularios para construir aplicaciones web resulta excepcionalmente cómodo para el programador, la semejanza con los entornos de desarrollo para Windows puede conducir a algunos equívocos. El objetivo de esta sección es aclarar aquellos aspectos que diferencian la creación de interfaces web de la creación de interfaces para Windows utilizando entornos de programación visual. La correcta comprensión de estas diferencias resulta esencial para la correcta implementación de aplicaciones web.

En una página ASP.NET, todos los controles cuyo funcionamiento haya de controlarse en el servidor de algún modo deben estar incluidos dentro de un formulario HTML. Los formularios HTML han de ir delimitados por la etiqueta estándar `<form>`. En el caso de ASP.NET, dicha etiqueta ha de incluir el atributo `runat="server"`. Este atributo le indica al servidor web (Internet Information Server) que la página ASP.NET ha de procesarse en el servidor antes de enviársela al cliente. Como consecuencia, el esqueleto de una página ASP.NET será siempre de la forma:

```
<form runat="server">
  ...
  <!-- HTML y controles ASP.NET -->
  ...
</form>
```

El formulario HTML incluido en la página ASP.NET, que ha de ser necesariamente único, es el encargado de facilitar la interacción del servidor con el navegador web que el usuario final de la aplicación utiliza en su máquina. Aunque la forma de programar los formularios web ASP.NET sea prácticamente idéntica a la creación de formularios para Windows, el modo de interacción característico de las interfaces web introduce algunas limitaciones. Estas limitaciones se deben a que cada acción que el usuario realiza en su navegador se traduce en una solicitud independiente al servidor web.

El hecho de que cada solicitud recibida por el servidor sea independiente de las anteriores ocasiona que, al desarrollar aplicaciones web, tengamos que ser conscientes de cuándo se produce cada solicitud y de cómo podemos enlazar solicitudes diferentes realizadas por un mismo usuario. En los dos siguientes apartados analizaremos cómo se resuelven estas dos cuestiones en las páginas ASP.NET. Dejaremos para el capítulo siguiente la forma de identificar las solicitudes provenientes de un único usuario cuando nuestra aplicación consta de varias páginas.

Solicitudes y "postbacks"

Al solicitar una página ASP.NET desde un cliente, en el servidor se dispara el evento `Page_Load` asociado a la página antes de generar ninguna salida. Es en el manejador asociado a este evento donde debemos realizar las tareas de inicialización de la página. Dichas tareas suelen incluir el establecimiento de valores por defecto o el rellenado de las listas de valores que han de mostrarse al usuario.

El evento `Page_Load` se dispara **cada vez** que el usuario accede a la página. Si lo que deseamos es realizar alguna tarea **sólo la primera vez** que un usuario concreto accede a la página, hemos de emplear la propiedad `Page.IsPostBack`. Esta propiedad posee el valor `false` cuando el cliente visualiza por primera vez la página ASP.NET, mientras que toma el valor `true` cuando no es la primera vez que la página ha de ejecutarse para ser mostrada. Esto sucede cuando el usuario realiza alguna acción, como pulsar un botón del formulario web, que tiene como consecuencia volver a generar la página para presentar datos nuevos o actualizados en la interfaz de usuario.

En ASP.NET y otras muchas tecnologías de desarrollo de interfaces web, cuando el usuario realiza una acción que requiere actualizar el contenido de la página que está visualizando, la página "se devuelve al servidor". De ahí proviene el término inglés *post back*. El uso de *postbacks* es una técnica común para manejar los datos de un formulario que consiste en enviar los datos a la misma página que generó el formulario HTML.

Page.IsPostBack

Una vez visto en qué consiste la ejecución repetida de una página ASP.NET como respuesta a las distintas acciones del usuario, estamos en disposición de completar el código correspondiente al ejemplo con el que terminamos la sección anterior de este capítulo (el del uso de controles definidos por el usuario en la creación de formularios web). Para que nuestro formulario muestre en cada momento los datos del contacto que el usuario selecciona de la lista desplegable, hemos de implementar el evento `Page_Load`. Dicho evento se ejecuta cada vez que se accede a la página y ha de seleccionar el contacto adecuado que se mostrará en el navegador web del usuario:

```
private void Page_Load(object sender, System.EventArgs e)
{
  if (!Page.IsPostBack) {

     // Inicialización de la interfaz
     // - Rellenado de la lista desplegable de contactos
     ...
     Header.Visible = false;
     ContactList.SelectedIndex = 0;
```

```
  } else {

     // Visualización del contacto actual
     contact = ...
     UpdateUI(contact);
  }
}

private void InitializeComponent()
{
  this.Load += new System.EventHandler(this.Page_Load);
}
```

Como se puede apreciar, hemos de diferenciar la primera vez en que el usuario accede a la página de las ocasiones en las que el acceso se debe a que el usuario haya pulsado el botón de su interfaz para mostrar los datos de un contacto diferente. La primera vez que el usuario accede a la página, cuando se cumple la condición `!Page.IsPostBack`, se tiene que inicializar el estado de los distintos controles del formulario web. En el ejemplo que nos ocupa, se ha de rellenar la lista desplegable de valores que le permitirá al usuario seleccionar el contacto cuyos datos desea ver.

AutoPostBack

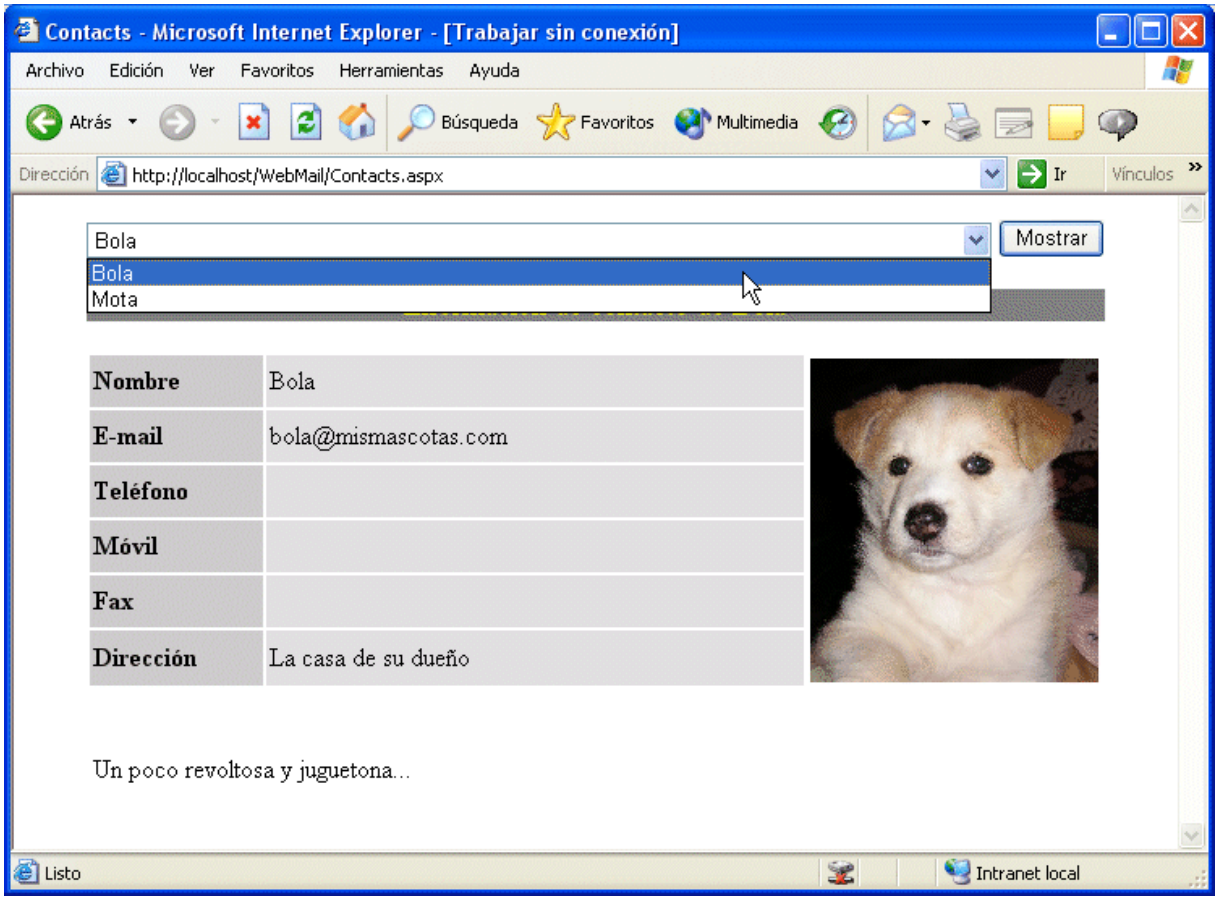
Utilizando únicamente el manejador correspondiente al evento `Page_Load` podemos conseguir una página dinámica cuya actualización se realiza cada vez que el usuario pulsa un botón, pulsación que se traduce en una nueva solicitud al servidor web (*post back* si empleamos la terminología habitual). No obstante, en determinadas ocasiones nos puede interesar que la interfaz de nuestra aplicación web responda a otras acciones del usuario, no sólo a la pulsación final de un botón del formulario.

En el ejemplo que venimos desarrollando en este capítulo, puede que nos interese mostrar los datos de contacto de alguien en el mismo momento en que el usuario selecciona un nombre de la lista desplegable. De esta forma, la respuesta inmediata de la aplicación facilita que el usuario perciba el efecto de las acciones que realiza. Al ver reflejadas sus acciones de forma inmediata en la ventana de su navegador web, el usuario ve reducida la distancia existente entre sus acciones y la respuesta del sistema con el que interactúa, uno de los principios fundamentales del diseño de interfaces de usuario. En el caso de la lista desplegable, debemos implementar la respuesta del control al evento `SelectedIndexChanged`:

```
private void ContactList_SelectedIndexChanged
                 (object sender, System.EventArgs e)
{
    // Contacto seleccionado de la lista
    contact = ...
    UpdateUI(contact);
}
```

```
private void InitializeComponent()
{
  ...
  this.ContactList.SelectedIndexChanged += new
      System.EventHandler(this.ContactList_SelectedIndexChanged);
  ...
}
```

Implementar la respuesta del control a este evento no es suficiente para que la aplicación responda de forma inmediata a un cambio en el elemento seleccionado de la lista desplegable. Si queremos que la página se actualice en el mismo momento en que el usuario selecciona un elemento de la lista, hemos de establecer a `true` la propiedad `AutoPostBack` del control de tipo `DropDownList` que corresponde a la lista desplegable.



Uso de la propiedad AutoPostBack para mejorar la usabilidad de una página ASP.NET: En cuanto el usuario selecciona un contacto de la lista desplegable, en el control de usuario de debajo aparecen todos sus datos.

La propiedad `AutoPostBack` existente en algunos de los controles ASP.NET sirve para que, ante determinados eventos relacionados con acciones del usuario, el estado de los controles de la página se envíe automáticamente al servidor. Esto permite actualizar el contenido de la página conforme el usuario interactúa con la aplicación. Si el retardo existente entre la solicitud del usuario y la respuesta del servidor web no es demasiado elevado, la usabilidad de la aplicación mejora. Sin embargo, el uso indiscriminado de la propiedad `AutoPostBack` genera una mayor carga sobre el servidor web al realizar cada cliente más solicitudes relativas a la misma página. Aparte de consumirse un mayor ancho de banda en la red, conforme aumenta la carga del servidor web, su tiempo de respuesta aumenta y se anulan las ventajas que supone utilizar `AutoPostBack` en primera instancia.

Internamente, el uso de la propiedad `AutoPostBack` para forzar la actualización de la página cuando el usuario realiza alguna acción se traduce en la inclusión de un fragmento de JavaScript en la página que se le envía al cliente. Este fragmento de código se le asocia, en el navegador web del cliente, al evento correspondiente del control cuya respuesta implementamos en la página ASP.NET (que se ejecuta en el servidor). Si volvemos al ejemplo anterior, el control `DropDownList` da lugar al siguiente fragmento de HTML dinámico:

```
<select name="ContactList"  id="ContactList"
        onchange="__doPostBack('ContactList','')"
        language="javascript" style="width:80%;">
  <option value="bola.xml" selected="selected">Bola</option>
  <option value="mota.xml">Mota</option>
</select>
```

Ya que el uso de `AutoPostBack` se traduce en la utilización de HTML dinámico en el cliente, el navegador del usuario ha de ser compatible con JavaScript (ECMAScript para ser precisos). Además, el usuario ha de tener habilitado el uso de secuencias de comandos en su navegador web.

Como consecuencia del uso interno de la versión de JavaScript estandarizada por ECMA, el estándar determina en qué controles y sobre qué eventos se puede emplear la propiedad `AutoPostBack`. Afortunadamente, dicha propiedad está disponible para la mayoría de las situaciones en las que nos pueda interesar refrescar el contenido de la página que ve el usuario. Como se mostró en el ejemplo anterior, podemos hacer que nuestra aplicación responda a un cambio en el elemento seleccionado de una lista, ya sea desplegable (de tipo `DropDownList`) o no (`ListBox`). También podemos conseguir que la aplicación web responda inmediatamente a acciones como el marcado de una caja de comprobación (`CheckBox` y `CheckBoxList`) o la selección de un botón de radio (`RadioButton` y `RadioButtonList`). Incluso podemos lograr que se genere una solicitud al servidor en cuanto el usuario modifique el texto contenido en un control de tipo `TextBox`.

En cualquier caso, cuando decidamos utilizar la propiedad `AutoPostBack` debemos tener en cuenta la carga adicional que esto supone sobre el servidor web y el consumo de ancho de

banda que las solicitudes adicionales suponen. Al sopesar los pros y los contras de la utilización de `AutoPostBack`, nunca debemos olvidar que, aunque mejoremos la realimentación que recibe el usuario como respuesta a sus acciones individuales, esta mejora puede llegar a ser contraproducente si ralentiza la realización de tareas completas.

Estado de una página ASP.NET

A diferencia de las aplicaciones para Windows, en las cuales el usuario interactúa con una instancia concreta de un formulario, en la aplicaciones web cada acción del usuario se trata de forma independiente. En otras palabras, cada vez que se le muestra una página al usuario, la página se construye de nuevo. Esto implica que el objeto concreto que recibe una solicitud del usuario no es el mismo aunque al usuario le dé la sensación de estar trabajando con "su" página. Desde el punto de vista práctico, este hecho tiene ciertas implicaciones que comentamos a continuación retomando el ejemplo de la lista de contactos.

Si recordamos, cuando creamos un control web de usuario para mostrar los datos de contacto de alguien, creamos una clase `ContactViewer` con una propiedad `DisplayedContact` que utilizábamos para mostrar los datos de un contacto concreto. Ingenuamente, puede que nos olvidamos del contexto en el que funcionan las aplicaciones web y escribamos algo como lo siguiente:

```
public class ContactViewer : System.Web.UI.UserControl
{
  ...
  private Contact contact;

  public Contact DisplayedContact {
    get { return contact; }
    set { contact = value; }
  }
  ...
}
```

Aparentemente, podríamos utilizar la variable de instancia `contact` para acceder a los datos que hemos de mostrar en pantalla. Sin embargo, cada vez que el usuario realiza alguna acción que se traduce en una nueva solicitud ál servidor (lo que denominábamos *post back* en la sección anterior), en el servidor web se crea un objeto nuevo encargado de atender la solicitud. Por consiguiente, el valor que habíamos guardado en la variable de instancia del objeto con el que se trabajaba en la solicitud anterior se habrá perdido.

Como consecuencia, la construcción de controles y de páginas en ASP.NET no puede realizarse de la misma forma que se implementaría una clase convencional. En una aplicación Windows, cuando se tiene una referencia a una instancia de una clase, siempre se trabaja con la misma referencia. En ASP.NET, cada acción se realiza sobre un objeto diferente, por lo que

debemos utilizar algún mecanismo que nos permita mantener el estado de una página o de un control entre distintas solicitudes procedentes de un mismo usuario.

Por suerte, los diseñadores de ASP.NET decidieron incluir un sencillo mecanismo por el cual se puede almacenar el estado de una página. Este mecanismo se basa en la utilización de un array asociativo en el cual podemos almacenar cualquier objeto, siempre y cuando éste sea serializable. La forma correcta de implementar una propiedad en el control web de usuario que diseñamos antes sería la siguiente:

```
public class ContactViewer : System.Web.UI.UserControl
{
  ...
  public Contact DisplayedContact {
    get { return (Contact) ViewState["contact"]; }
    set { ViewState["contact"] = value; }
  }
  ...
}
```

En lugar de utilizar una variable de instancia, como haríamos habitualmente al diseñar una clase que ha de encapsular ciertos datos, se emplea el array asociativo `ViewState` para garantizar que el estado de la página se mantiene entre solicitudes diferentes.

En el caso de los controles ASP.NET predefinidos, su implementación se encarga de mantener el estado de los controles cuando se recibe una solicitud correspondientes a un *post back*. Cuando se recibe una nueva solicitud, aunque se trabaje con objetos diferentes, lo primero que se hace es reconstruir el estado de los controles de la interfaz a partir de los datos recibidos del usuario.

Internamente, el estado de una página ASP.NET se define mediante un campo oculto incluido en el formulario HTML correspondiente a la página. Este campo oculto, denominado `__VIEWSTATE`, se le añade en el servidor a cada página que tenga un formulario con el atributo `runat="server"` (de ahí la necesidad de que el formulario HTML de la página ASP.NET incluya este atributo). De esta forma, es la página web que visualiza el usuario la que se encarga, sin que el usuario sea consciente de ello, de mantener el estado de la página ASP.NET en el servidor. Esto lo podemos comprobar fácilmente si visualizamos el código fuente de la página HTML que se muestra en el navegador web de la máquina cliente. En él podemos ver algo así:

```
<input type="hidden" name="__VIEWSTATE"
       value="dDwtMjEwNjQ1OTkwMDs7PiTPnxCh1VBUIX3K2htmyD8Dq6oq" />
```

Este mecanismo, que supone una novedad en ASP.NET con respecto a versiones anteriores de ASP, nos ahorra tener que escribir bastantes líneas de código. Al encargarse `ViewState` de

mantener automáticamente el estado de los controles de los formularios web, el programador puede despreocuparse del hecho de que cada solicitud del usuario se procese de forma independiente.

En ASP clásico, al enviar un formulario, el contenido de la página HTML que ve el usuario se pierde, por lo que el programador debe reconstruir el estado de la página manualmente. Por tanto, si se produce un error, por pequeño que sea, al rellenar alguno de los valores de un formulario, el programador debe implementar el código que se encargue de rellenar los valores que sí son correctos. Esto resulta aconsejable siempre y cuando queramos evitarle al usuario tener que introducir de nuevo todos los datos de un formulario cuando, a lo mejor, se le ha olvidado introducir un dato que era obligatorio. Tener que hacer algo así es demasiado común en muchas aplicaciones web, además de resultar bastante irritante.

Por desgracia, la labor del programador para evitarle inconvenientes al usuario final de la aplicación resulta bastante tediosa y es, en consecuencia, muy propensa a errores. Afortunadamente, en ASP.NET, el formulario web reaparece automáticamente en el navegador del cliente con el estado que sus controles ASP.NET tuviesen anteriormente.

Si seguimos utilizando el estilo tradicional de ASP clásico, una sencilla página podría tener el aspecto siguiente:

```
<%@ Page language="c#" %>
<html>
 <body>

  <form runat="server" method="post">
    Tu nombre:
    <input type="text" name="nombre" size="20">
    <input type="submit" value="Enviar">
  </form>

  <%
    string name = Request.Form["nombre"];

    if (name!=null && name!="") {
       Response.Write("Hola, " + name + "!");
    }
  %>

 </body>
</html>
```

Esta página incluye un formulario en el que el usuario puede introducir su nombre. Cuando el usuario pulsa el botón "Enviar", en su navegador web aparecerá de nuevo el formulario seguido de un mensaje de bienvenida. Sin embargo, el campo del formulario cuyo valor había rellenado aparecerá vacío. Si utilizamos los controles ASP.NET, no obstante, podemos evitar que el valor introducido desaparezca tecleando lo siguiente en nuestro fichero `.aspx`:

```
<%@ Page language="c#" %>
<html>
 <script runat="server">
  void enviar (object sender, EventArgs e)
  {
    label.Text = "Hola, " + textbox.Text + "!";
  }
 </script>
 <body>
  <form runat="server" method="post">
   Tu nombre: <asp:TextBox id="textbox" runat="server" />
   <asp:Button OnClick="enviar" Text="Enviar" runat="server" />
   <p><asp:Label id="label" runat="server" /></p>
  </form>
 </body>
</html>
```

El mantenimiento automático del estado de la página que ofrecen los formularios web cuando utilizamos controles ASP.NET nos sirve para evitar que el usuario "pierda" los datos que acaba de introducir. Como es lógico, aunque el mantenimiento del estado de la página es automático en ASP.NET, puede que nos interese que la página no mantenga su estado. Esto puede suceder cuando el usuario teclea una clave privada o cualquier otro tipo de dato cuya privacidad se haya de preservar. También puede ser recomendable que una página no mantenga su estado cuando se están introduciendo series de datos, ya que mantener el estado de la página podría ocasionar la recepción de datos duplicados cuya existencia habría que detectar en el código de la aplicación.

ASP.NET nos ofrece dos alternativas para indicar explícitamente que el estado de un formulario web no debe mantenerse entre solicitudes independientes aunque éstas provengan de un único usuario:

- A nivel de la página, podemos emplear la directiva `<%@ Page EnableViewState="false" %>` en la cabecera del fichero `.aspx`. Esto deshabilita el mantenimiento del estado para todos controles de la página ASP.NET.

- A nivel de un control particular, podemos establecer su propiedad `EnableViewState` a `false`. De esta forma, podemos controlar individualmente el comportamiento de las distintas partes de una página en lo que se refiere al mantenimiento de su estado.

Resumen del capítulo

En este capítulo hemos aprendido todo lo que necesitamos para ser capaces de construir interfaces web utilizando la tecnología ASP.NET incluida en la plataforma .NET de Microsoft. Para ser más precisos, nos centramos en la construcción de páginas ASP.NET individuales:

- En primer lugar, descubrimos en qué consiste el modelo de programación de los formularios web, que está basado en el uso de controles y eventos.

- A continuación, vimos cómo podemos crear nuestros propios controles para construir interfaces web más modulares y flexibles.

- Finalmente, estudiamos el funcionamiento de las páginas ASP.NET, haciendo especial hincapié en los detalles que diferencian la creación de interfaces web de la construcción de aplicaciones para Windows. En este apartado llegamos a aprender cómo se puede controlar el mantenimiento del estado de una página ASP.NET.

Sin embargo, una aplicación web rara vez consta de una única página, por lo que aún nos quedan por ver algunos aspectos relacionados con la construcción de aplicaciones web. Este será el hilo conductor del siguiente capítulo.

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.

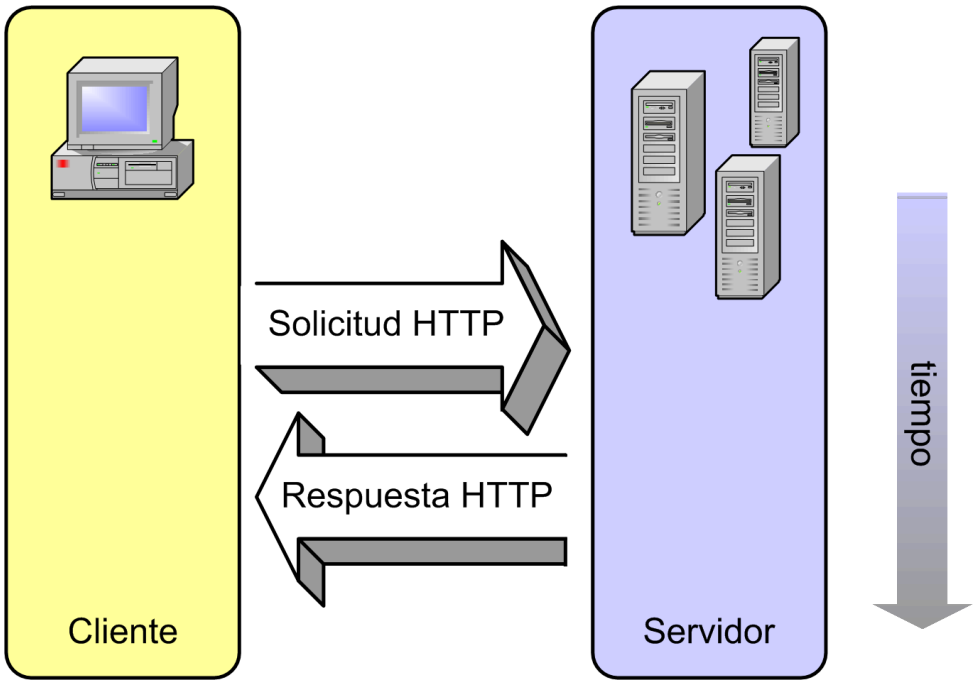
- Finalmente, cerraremos el capítulo describiendo cómo podemos controlar a qué partes de la aplicación puede acceder cada usuario, cómo crear formularios de autentificación y darle permisos al usuario para que realice determinadas tareas en el servidor.

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.

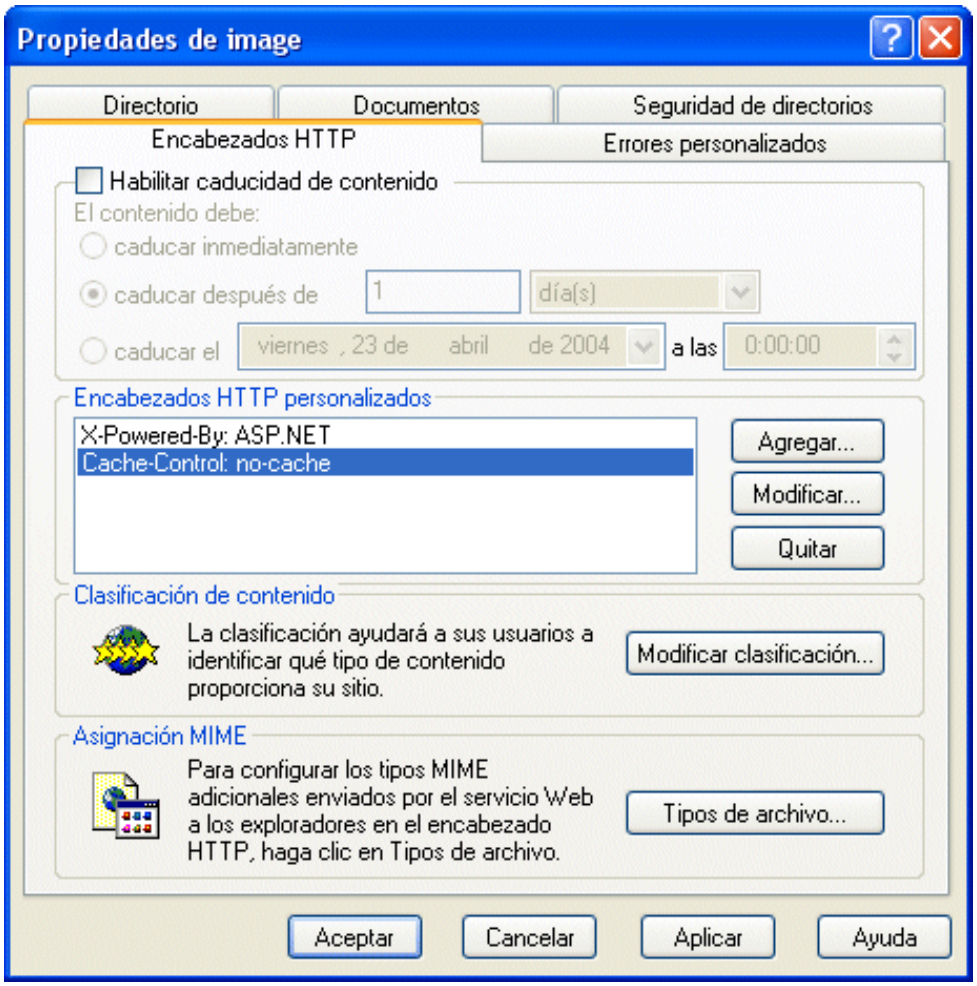
Almacenamiento en caché

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

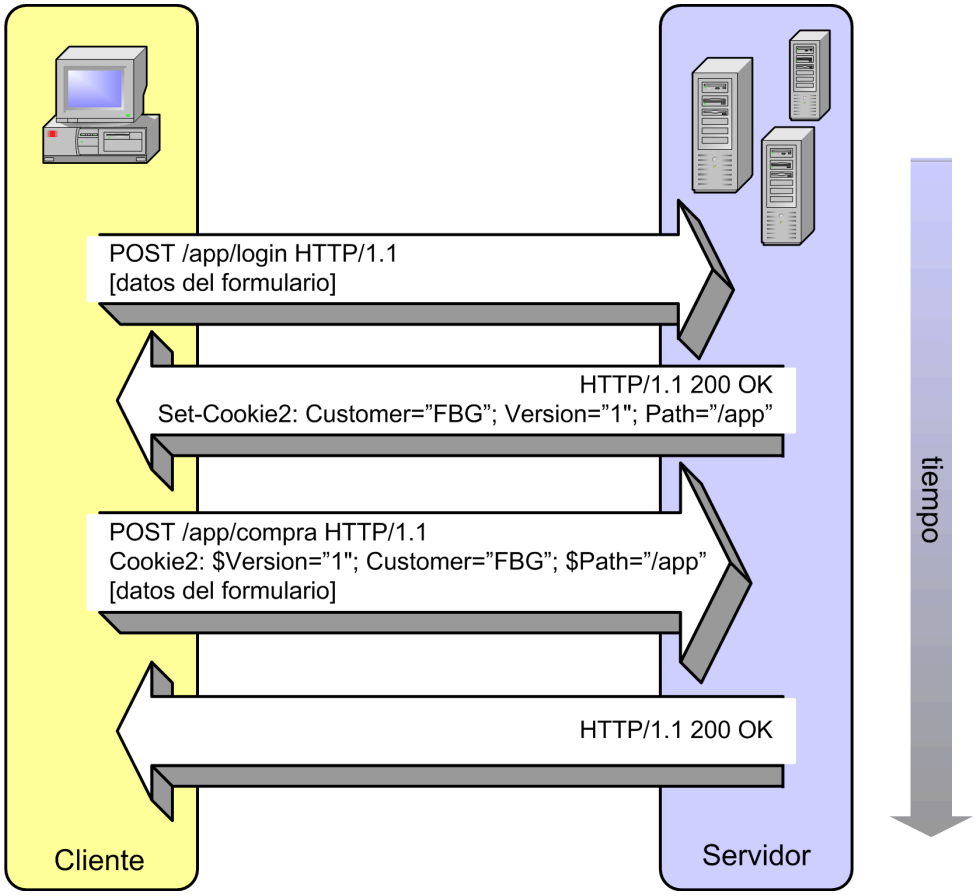
cliente concreto.

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
```

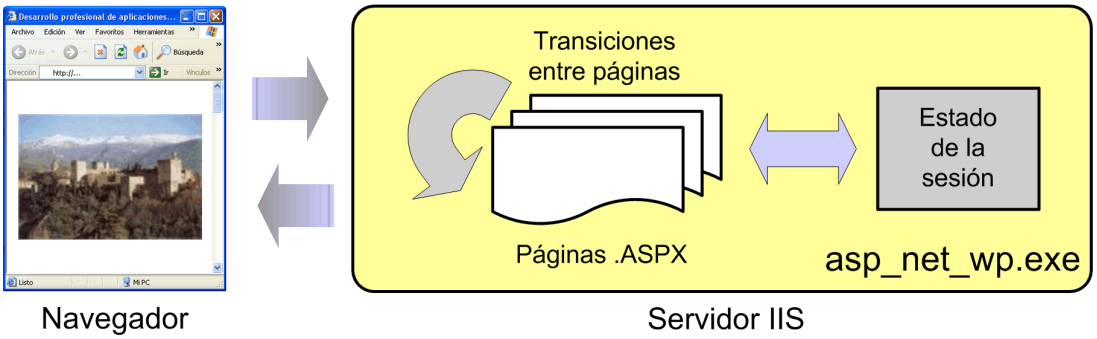
Para seleccionar la forma en la que deseamos transmitir el identificador de la sesión, lo único que tenemos que hacer es modificar el valor del atributo `cookieless` del elemento `sessionState` que aparece en el fichero de configuración de nuestra aplicación web, un fichero XML llamado `Web.config`. 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.

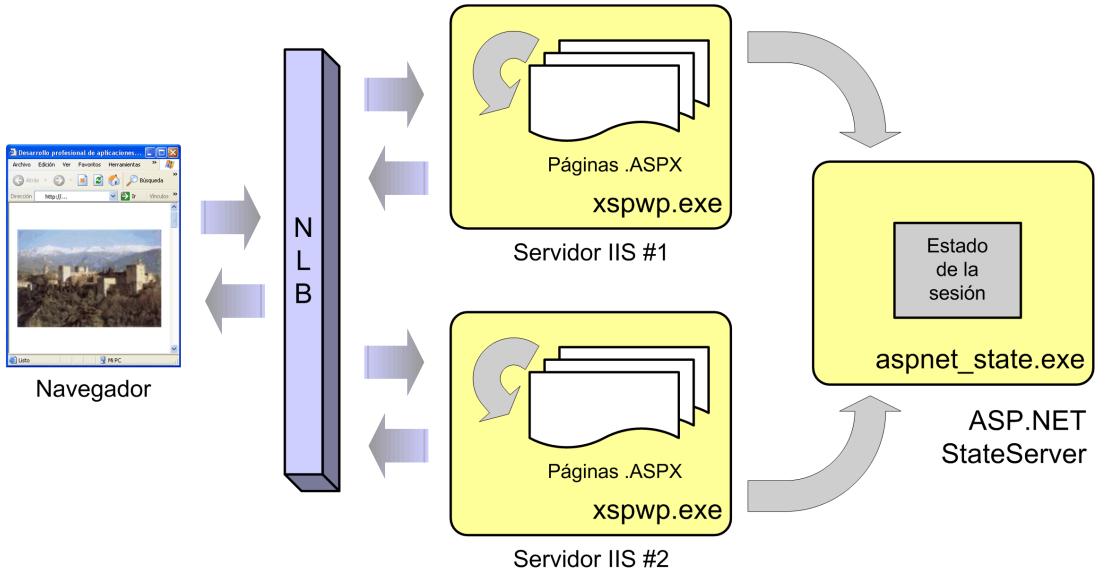


InProc: Los datos correspondientes a las sesiones de usuario se almacenan en la memoria del proceso encargado de ejecutar las páginas ASP.NET.

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"
/>
```

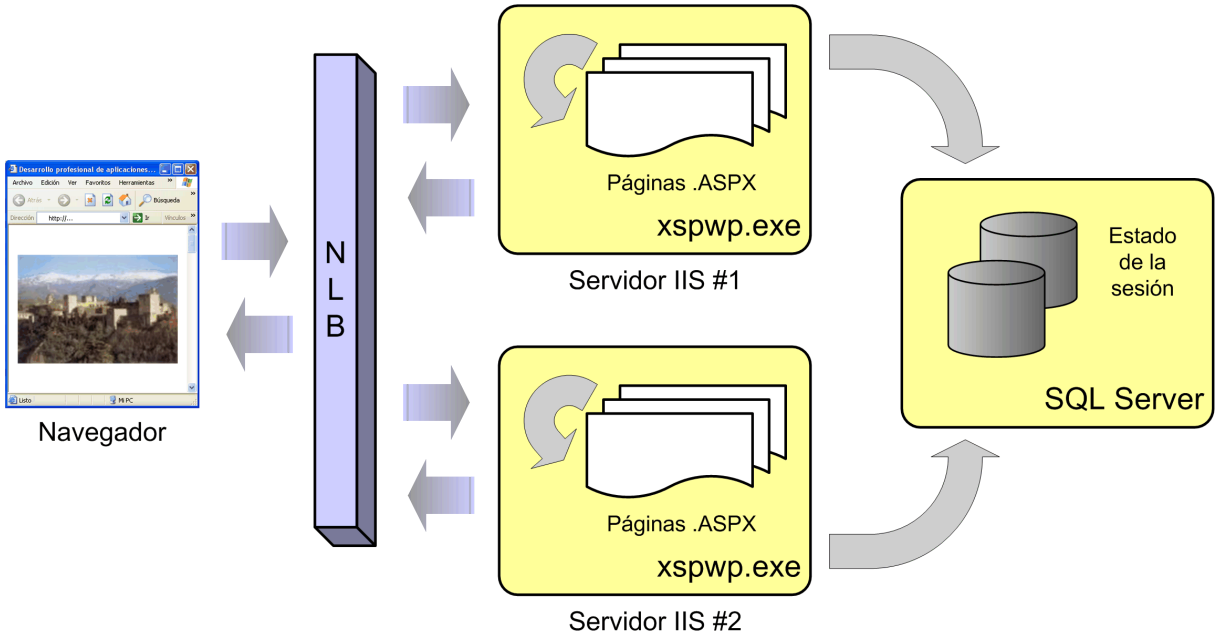
Por último, ASP.NET también nos permite que los datos correspondientes a las sesiones de usuario se almacenen de forma permanente en una base de datos relacional (SQL*Server, como no podía ser de otra forma). Esto permite que el estado de cada sesión se mantenga incluso cuando falle alguna de las máquinas 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`:

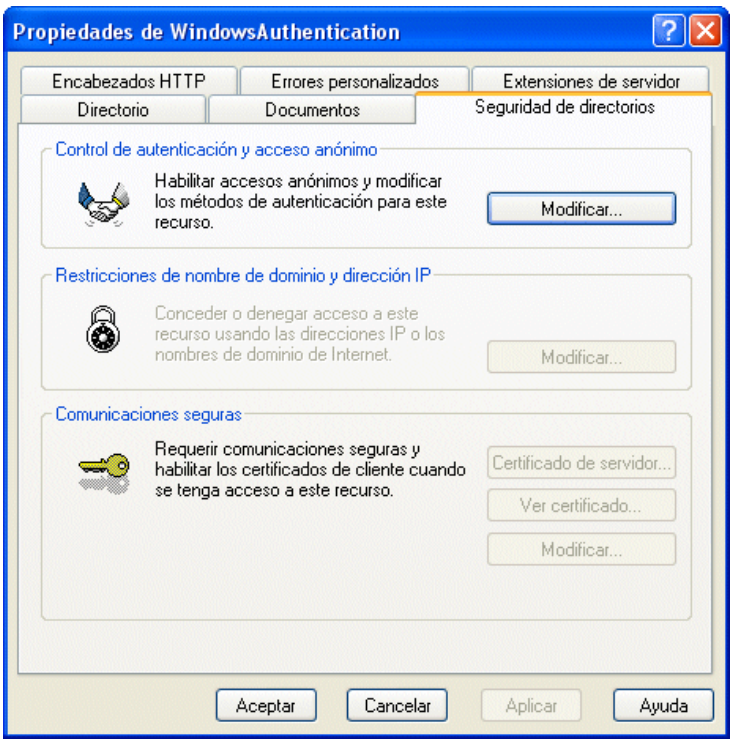
```
<authorization>
  <deny users="*" />
  <allow users="administrador,director" />
</authorization>
```

Autentificación en Windows

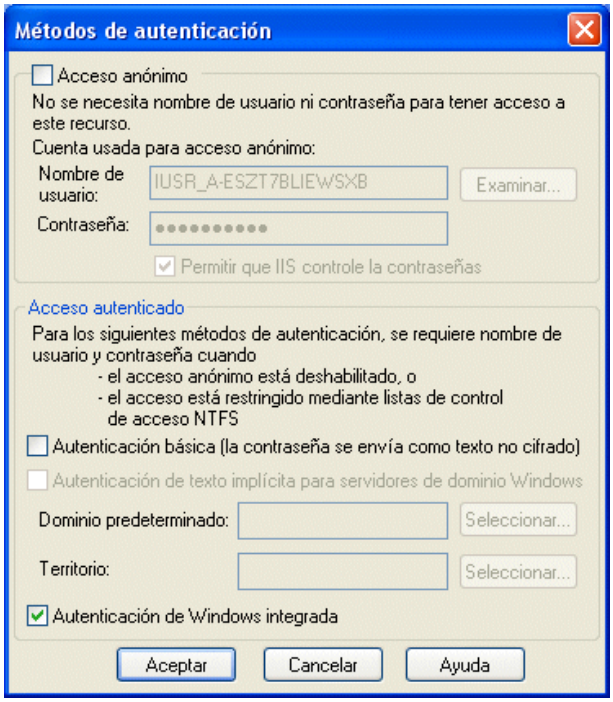
Una vez visto cómo podemos controlar qué usuarios acceden a qué partes de la aplicación, volvamos ahora al problema inicial: ¿cómo identificar a los usuarios en un primer momento?.

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:



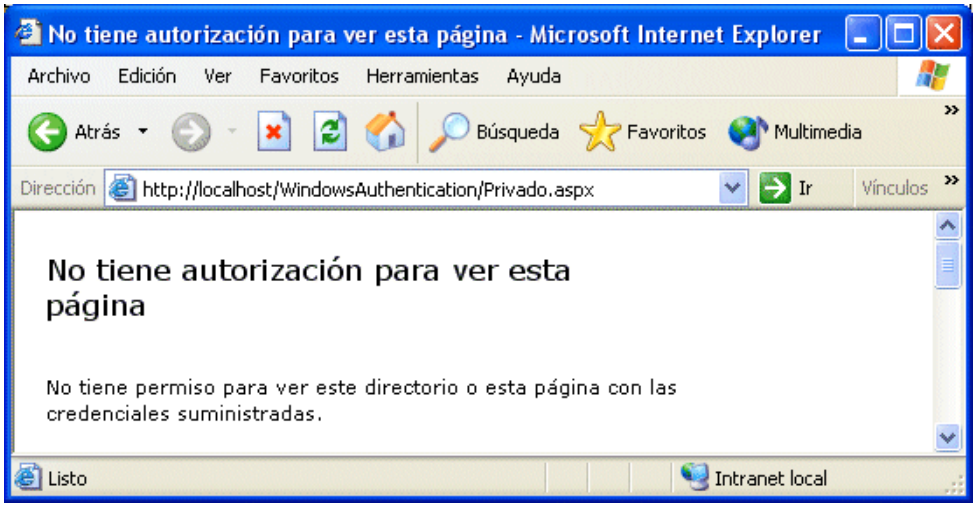
Una vez dentro de las propiedades relacionadas con el control de "autenticación" y acceso anónimo, deshabilitamos el acceso anónimo:



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

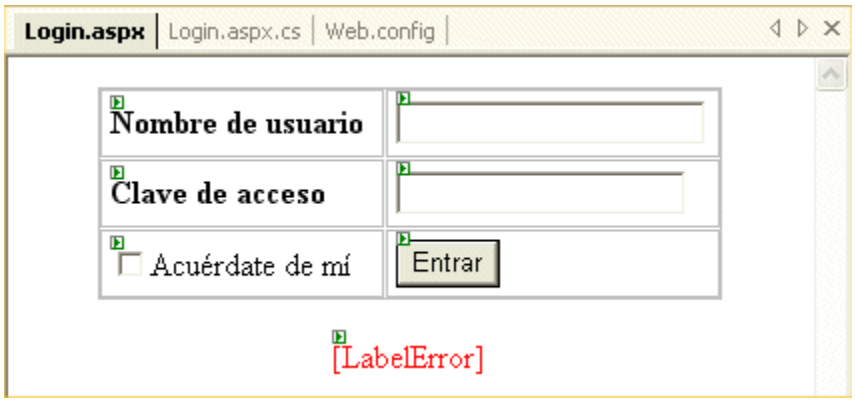
En determinadas ocasiones, no nos podremos permitir el lujo de crear un usuario en el sistema operativo para cada usuario que deba acceder a nuestra aplicación. Es más, posiblemente no nos interese hacerlo. Probablemente deseemos ser nosotros los encargados de gestionar los usuarios de nuestra aplicación 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>
```

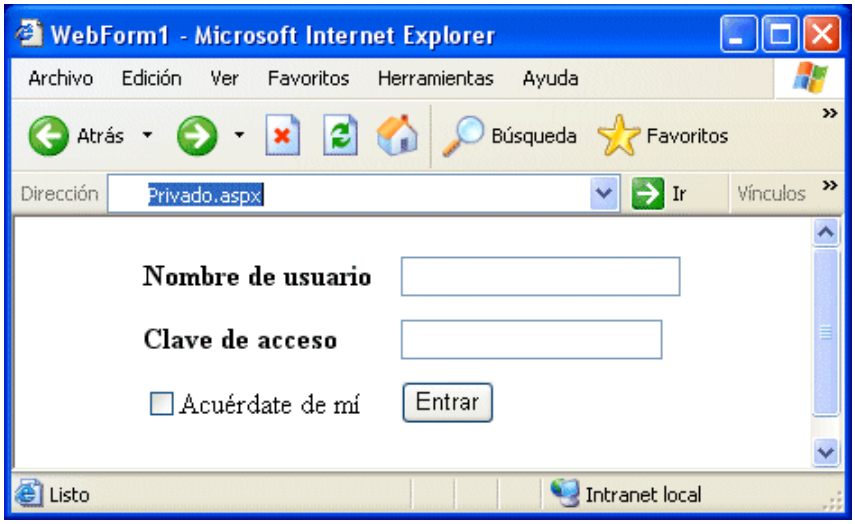
Cuando un usuario no identificado intente acceder a una página cuyo acceso requiera su identificación, lo que haremos será redirigirlo a un formulario específico de *login*, `Login.aspx`. Dicho formulario, al menos, debe incluir dos campos para que el usuario pueda indicar su nombre y su clave:



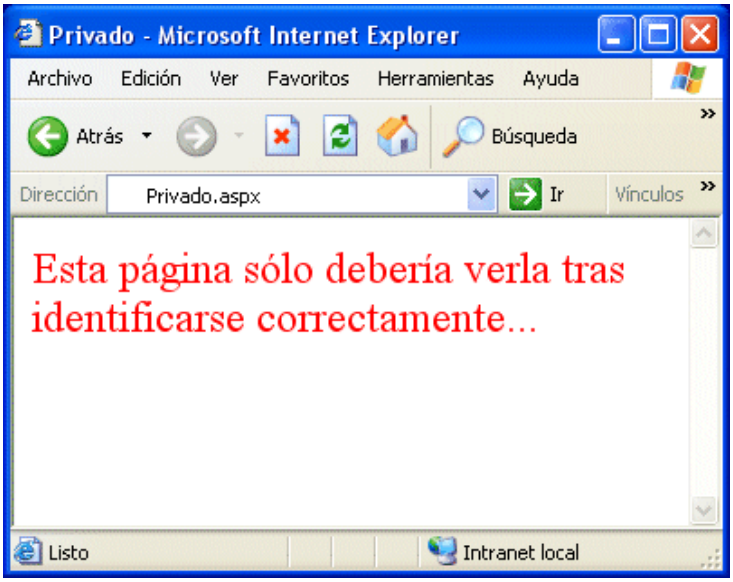
El formulario de identificación incluirá, por tanto, dos controles de tipo `TextBox`. Dado que la contraseña del usuario ha de mantenerse secreta, en el `TextBox` correspondiente a la clave de acceso se ha de especificar la propiedad `TextMode=Password`. El formulario, en sí, lo crearemos igual que cualquier 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:

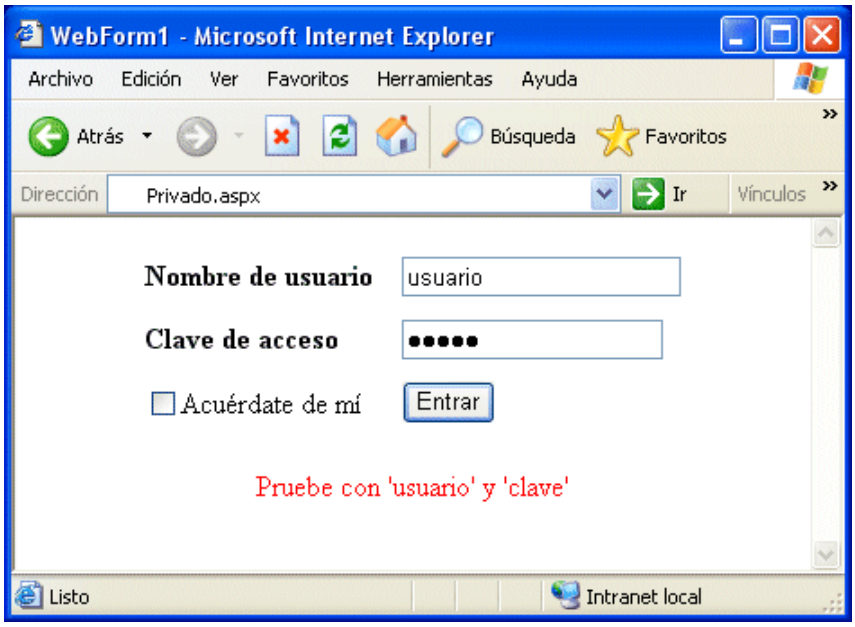


Una vez que el usuario se identifica correctamente, la llamada al método `RedirectFromLoginPage` le indica al IIS que debe darle permiso al usuario para acceder a la página a la que inicialmente intentó llegar. Este método pertenece a la clase `FormsAuthentication`, la cual está incluida en el espacio 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").

ASP.NET en la práctica

En este último capítulo dedicado a ASP.NET, trataremos algunos aspectos de interés a la hora de construir aplicaciones reales. Después de haber visto en qué consiste una aplicaciónes web, cómo se crean formularios web en ASP.NET y de qué forma se pueden realizar las tareas más comunes con las que debe enfrentarse el programador de aplicaciones web, ahora comentaremos algunos de los aspectos relativos a la organización interna de las aplicaciones web:

- En primer lugar, comentaremos brevemente algunos de los patrones de diseño que se suelen emplear a la hora de crear la aplicación web con el objetivo de que ésta mantenga su flexibilidad y su mantenimiento no resulte tedioso.

- A continuación, nos centraremos en los mecanismos existentes que nos facilitan que el aspecto externo de una aplicación web ASP.NET sea homogéneo.

- Una vez descritos los principios en los que se suelen basar las aplicaciones web bien diseñadas, repasaremos cómo se puede hacer que el contenido de nuestra interfaz web cambie dinámicamente utilizando un mecanismo conocido como enlace de datos [*data binding*].

- Finalmente, cerraremos este capítulo centrándonos en la construcción de formularios de manipulación de datos, como ejemplo más común del tipo de módulos que tendremos que implementar en cualquier aplicación.

ASP.NET en la práctica

Organización de la interfaz de usuario

Como Steve McConnell expone en su libro *Software Project Survival Guide*, la mayor parte del trabajo que determina el éxito o el fracaso final de un proyecto de desarrollo de software se realiza antes de que comience su implementación: "si el equipo investiga los requisitos a fondo, desarrolla el diseño con detalle, crea una buena arquitectura, prepara un plan de entrega por etapas y controla los cambios eficazmente, la construcción [del software] destacará por su firme progreso y la falta de problemas graves". Obviamente, la lista de condiciones que se han de cumplir para que un proyecto de desarrollo de software finalice con éxito es extensa, como no podía ser menos dada la complejidad de las tareas que han de realizarse.

En los primeros apartados de este capítulo nos centraremos en una de las bases que sirven de apoyo al éxito final del proyecto: la creación de una buena arquitectura. En concreto, veremos cómo se pueden organizar los distintos elementos que pueden formar parte de la interfaz web de una aplicación realizada con ASP.NET.

Componentes de la interfaz

La primera tentación de un programador de aplicaciones web consiste en escribir todo el código de su aplicación en la propia página ASP.NET. Sin embargo, en cuanto la aplicación comienza a crecer, empiezan a aparecer cierta redundancia. La existencia de fragmentos de código duplicados complicarán el mantenimiento de la aplicación y la realización de nuevas mejoras.

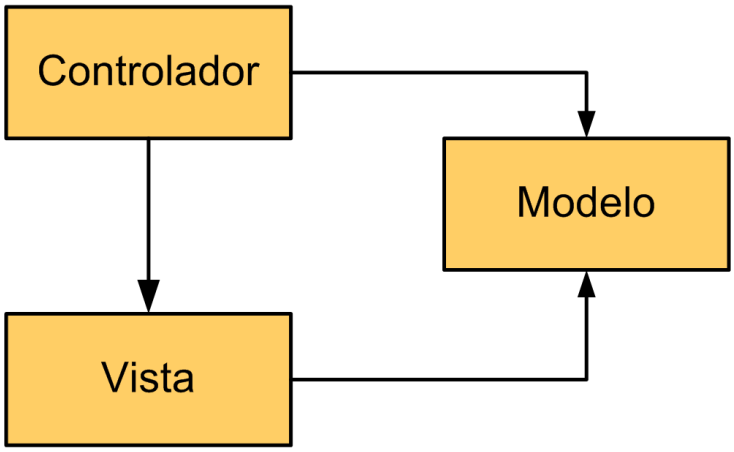
Cuando situaciones como esta se presentan, lo usual es encapsular la lógica de la aplicación en componentes independientes. La redundancia se elimina a cambio de un incremento en la complejidad de la aplicación. Cuanto más compleja sea la aplicación, más difícil será que otros programadores sean capaces de entender correctamente su funcionamiento. Por tanto, la complejidad introducida también dificulta el mantenimiento de la aplicación, aunque sea en un sentido totalmente diferente al causado por la existencia de código duplicado.

Afortunadamente, existen soluciones bien conocidas que consiguen dotar a las aplicaciones de cierta flexibilidad sin incrementar excesivamente su complejidad. En el caso del desarrollo de interfaces de usuario, la solución más conocida es el patrón de diseño MVC [*Model-View-Controller*], en el que se distinguen tres componentes bien diferenciados:

- El **modelo** encapsula el comportamiento y los datos correspondientes al dominio de la aplicación. Habitualmente se construye un modelo de clases del problema con el que se esté trabajando, independientemente de cómo se vayan a presentar los datos de cara al usuario.

- Las **vistas** consultan el estado del modelo para mostrárselo al usuario. Por ejemplo, un mismo conjunto de datos puede verse en forma de tabla o gráficamente, en una pantalla o en un informe impreso. Cada una de las formas de mostrar los constituye una vista independiente y, en vez de tener en cada vista el código necesario para acceder directamente a los datos, cada una de las vistas delega en el modelo, que es el responsable de obtener los datos y realizar los cálculos necesarios.

- Los **controladores**, por último, son los encargados de permitir que el usuario realice acciones. Dichas acciones se traducirán en las respuestas que resulten apropiadas, las cuales pueden involucrar simplemente a las vistas o incluir la realización de operaciones sobre el modelo.



El modelo MVC: Las vistas y los controladores dependen del modelo, pero el modelo no depende ni de la vista ni del controlador. Esto permite que el modelo se pueda construir y probar independientemente de la presentación visual de la aplicación. Además, se pueden mostrar varias vistas de los mismos datos simultáneamente.

El modelo MVC ayuda a modularizar correctamente una aplicación en la cual el usuario manipula datos a través de una interfaz. Si el usuario puede trabajar con los mismos datos de distintas formas, lo habitual es encapsular el código compartido en un módulo aparte con el fin de evitar la existencia de código duplicado. Se puede decir que el modelo contiene el comportamiento común a las distintas formas que tiene el usuario de manipular los datos. De hecho, la existencia de un modelo independiente facilita enormemente la construcción de sistemas que han de ofrecer varios interfaces. Este sería el caso de una empresa que desea disponer de una aplicación Windows para uso interno de sus empleados, una interfaz web para que sus clientes puedan realizar pedidos y un conjunto de servicios web para facilitar el intercambio de datos con sus proveedores.

De hecho, aunque el usuario trabaje con los datos de una única forma, el código correspondiente a la interfaz suele cambiar más frecuentemente que el código correspondiente a la lógica de la aplicación. Por ejemplo, es bastante habitual cambiar el aspecto visual de una aplicación manteniendo su funcionalidad intacta. Eso sucede cada vez que ha de adaptarse la interfaz a un nuevo dispositivo, como puede ser un PDA cuya resolución es mucho más limitada que la de un monitor convencional.

En el caso de las aplicaciones web, los conocimientos necesarios para crear la interfaz (en HTML dinámico por lo general) suelen ser diferentes de los necesarios para implementar la lógica de la aplicación (para la que se utilizan lenguajes de programación de alto nivel). Por consiguiente, la separación de la interfaz y de la lógica de la aplicación facilita la división del trabajo en un equipo de desarrollo.

Por otro lado, cualquier modificación requerirá comprobar que los cambios realizados no han introducido errores en el funcionamiento de la aplicación. En este sentido, mantener los distintas partes de una aplicación lo menos acopladas posible siempre es una buena idea. Además, comprobar el funcionamiento de la interfaz de usuario es mucho más complejo y costoso que realizar pruebas de unidad sobre módulos independientes (algo que describiremos en la última parte de este libro cuando veamos el uso de NUnit para realizar pruebas de unidad).

> Una de las decisiones de diseño fundamentales en la construcción de software es aislar la interfaz de la lógica de la aplicación. Al fin y al cabo, el éxito del modelo MVC radica en esa decisión: separar la lógica de la aplicación de la lógica correspondiente a su presentación.

En los dos próximos apartados veremos cómo se utiliza el modelo MVC en ASP.NET y cómo se pueden crear controladores que realicen tareas comunes de forma uniforme a lo largo y ancho de nuestras aplicaciones.

El modelo MVC en ASP.NET

Las aplicaciones web utilizan una variante del modelo MVC conocida como **MVC pasivo** porque las vistas sólo se actualizan cuando se realiza alguna acción a través del controlador. Esto es, aunque el estado del modelo se vea modificado, la vista no se actualizará automáticamente. El navegador del usuario muestra la vista y responde a las acciones del usuario pero no detecta los cambios que se puedan producir en el servidor. Dichos cambios pueden producirse cuando varios usuarios utilizan concurrentemente una aplicación o existen otros actores que pueden modificar los datos con los que trabaja nuestra aplicación. Las peculiaridades de los interfaces web hacen que, cuando es necesario reflejar en el navegador del usuario los cambios que se producen en el servidor, tengamos que recurrir a estrategias como las descritas al estudiar el protocolo HTTP en el apartado "cuestión de refresco" del

capítulo anterior.

Cuando comenzamos a ver en qué consistían las páginas ASP.NET, vimos que existen dos estilos para la confección de páginas ASP.NET:

- Podemos incluir todo en un único fichero `.aspx` o podemos separar los controles de la interfaz del código de la aplicación. Si empleamos un único fichero `.aspx`, dicho fichero implementa los tres roles diferenciados por el modelo MVC: modelo, vista y controlador aparecen todos revueltos. Entre otros inconvenientes, esto ocasiona que un error introducido en el código al modificar el aspecto visual de una página no se detectará hasta que la página se ejecute de nuevo.

- Si empleamos ficheros diferentes, se separan físicamente la interfaz y la lógica de la aplicación. El fichero `.aspx` contendrá la presentación de la página (la vista según el modelo MVC) mientras que el fichero `.aspx.cs` combinará el modelo con el controlador. El código común a distintas páginas se podrá reutilizar con comodidad y un cambio que afecte a la funcionalidad común no requerirá que haya que ir modificando cada una de las páginas. Además, se podrá modificar el aspecto visual de la página sin temor a introducir errores en el código de la aplicación y podremos utilizar todas las facilidades que nos ofrezca el entorno de desarrollo para desarrollar ese código (al contener el fichero `.aspx.cs` una clase convencional escrita en C#).

Hasta aquí, nada nuevo: el código se escribe en un fichero aparte para facilitar nuestro trabajo (algo que hicimos desde el primer momento). Sin embargo, no estamos utilizando ningún tipo de encapsulación.

Por un lado, se hace referencia a las variables de instancia de la clase desde el fichero `.aspx`. Por ejemplo, podemos tener un botón en nuestra página ASP.NET:

```
<asp:button id="button" runat="server" text="Enviar datos"/>
```

Dicho botón, representado por medio de la etiqueta `asp:button`, hace referencia a una variable de instancia declarada en el fichero `.aspx.cs`:

```
protected System.Web.UI.WebControls.Button button;
```

Por otro lado, el método `InitializeComponent` define el funcionamiento de la página al enlazar los controles de la página con el código correspondiente a los distintos eventos:

```
private void InitializeComponent()
```

```
    {
      this.button.Click += new System.EventHandler(this.Btn_Click);
      this.Load += new System.EventHandler(this.Page_Load);
    }
```

Si bien siempre es recomendable separar la interfaz del código, la vista del modelo/controlador, esta separación es insuficiente cuando tenemos que desarrollar aplicaciones reales. Técnicamente, sería posible reutilizar el código del fichero `.aspx.cs`, aunque eso sólo conduciría a incrementar el acoplamiento de otras partes de la aplicación con la página ASP.NET. Para evitarlo, se crean clases adicionales que separan el modelo del controlador. El controlador contendrá únicamente el código necesario para que el usuario pueda manejar la página ASP.NET y el modelo será independiente por completo de la página ASP.NET.

En la última parte de este libro veremos cómo se crea dicho modelo y cómo, al ser independiente, se pueden desarrollar pruebas de unidad que verifican su correcto funcionamiento. De hecho, tener la lógica de la aplicación separada por completo de la interfaz es algo necesario para poder realizar cómodamente pruebas que verifiquen el comportamiento de la aplicación. En las aplicaciones web, comprobar el funcionamiento la lógica incluida en una página resulta especialmente tedioso, ya que hay que analizar el documento HTML que se obtiene como resultado. Separar la lógica de la aplicación nos facilita enormemente el trabajo e incluso nos permite automatizarlo con herramientas como NUnit, que también se verá en la última parte de este libro.

Por ahora, nos centraremos en cómo se puede implementar el controlador de una aplicación web desarrollada en ASP.NET.:

Controladores en ASP.NET

En el modelo MVC, la separación básica es la que independiza el modelo de la interfaz de la aplicación. La separación entre vistas y controladores es, hasta cierto punto, secundaria. De hecho, en la mayor parte de entornos de programación visual, vistas y controladores se implementan conjuntamente, a pesar de ser una estrategia poco recomendable. En el caso de las aplicaciones web, no obstante, la separación entre modelo y controlador está muy bien definida: la vista corresponde al HTML que se muestra en el navegador del usuario mientras que el controlador se encarga en el servidor de gestionar las solicitudes HTTP. Además, en una aplicación web, distintas acciones del usuario pueden conducir a la realización de distintas tareas (responsabilidad del controlador) aun cuando concluyan en la presentación de la misma página (la vista en el modelo MVC).

En el caso de las aplicaciones web, existen dos estrategias para implementar la parte correspondiente al controlador definido por el modelo MVC, las cuales dan lugar a dos tipos de controladores: los controladores de página (específicos para cada una de las páginas de la aplicación web) y los controladores de aplicación (cuando un controlador se encarga de todas las páginas de la aplicación).

Controladores de página

Usualmente, se utiliza un controlador independiente para cada página de la aplicación web. El controlador se encarga de recibir la solicitud de página del cliente, realizar las acciones correspondientes sobre el modelo y determinar cuál es la página que se le ha de mostrar al usuario a continuación.

En el caso de ASP.NET y de muchas otras plataformas para el desarrollo de aplicaciones web, como JSP o PHP, la plataforma nos proporciona los controladores de página, usualmente mezclados con la vista a la que corresponde la página. En ASP.NET, los controladores toman forma de clase encargada de gestionar los eventos recibidos por la página `.aspx`. Esta estrategia nos facilita el trabajo porque, en cada momento, sólo tenemos que preocuparnos de la acción con la que se ha de responder a un evento concreto. De esta forma, la complejidad del controlador puede ser reducida y, dada la dificultad que conlleva probar el funcionamiento de una interfaz web, la simplicidad siempre es una buena noticia.

Usualmente, las aplicaciones web realizan algunas tareas fijas para todas las solicitudes recibidas, como puede ser la autentificación de usuarios que ya se trató en el capítulo anterior. De hecho, tener un controlador para cada página (y, veces, por acción del usuario) suele conducir a la aparición de bastante código duplicado. Cuando esto sucede, se suele crear una clase base que implemente las funciones comunes a todos los controladores y de la que hereden los controladores de cada una de las páginas de la aplicación. Otra opción consiste en crear clases auxiliares que implementen utilidades de uso común. Este caso deberemos acordarnos siempre de invocar a esas utilidades en todas y cada una de las páginas de la aplicación, mientras que si utilizamos herencia podemos dotar a todos nuestros controladores de cierta funcionalidad de forma automática. Cuando la complejidad de esas operaciones auxiliares es destacable, conviene combinar las dos estrategias para mantener la cohesión de los módulos de la aplicación y facilitar su mantenimiento. Esto es, se puede utilizar una jerarquía de clases para los controladores y que algún miembro de esta jerarquía (presumiblemente cerca de su base) delegue en clases auxiliares para realizar las tareas concretas que resulten necesarias.

Cabecera común para distintas páginas ASP.NET

En ocasiones, hay que mostrar información de forma dinámica en una cabecera común para todas las páginas de una aplicación (por ejemplo, la dirección de correo electrónico del usuario que está actualmente conectado). En vez de duplicar el código correspondiente a la cabecera en las distintas páginas, podemos hacer que la clase base muestre la parte común y las subclases se encarguen del contenido que les es específico. En ASP.NET podemos crear una relación de herencia entre páginas y utilizar un método genérico definido en la clase base que será implementado de forma diferente en cada subclase.

En primer lugar, creamos la página ASP.NET que servirá de base: `BasePage.cs`. En dicha clase incluiremos un método virtual y realizaremos las tareas que sean comunes para las distintas páginas:

Cabecera común para distintas páginas ASP.NET

```
public class BasePage : System.Web.UI.Page
{
  ...

  virtual protected void PageLoadEvent
          (object sender, System.EventArgs e) {}

  protected void Page_Load (object sender, System.EventArgs e)
  {
    if (!IsPostBack) {
       eMailLabel.Text = "csharp@ikor.org";
       PageLoadEvent(sender, e);
    }
  }

  ...
}
```

También se debe crear el fichero que contendrá la parte común del aspecto visual de las páginas: `BasePage.inc`.

```
<table width="100%" cellspacing="0" cellpadding="0">
 <tr>
  <td align="right" bgcolor="#c0c0c0">
   <font size="2" color="#ffffff">
     Hola,
     <asp:Label id="eMail" runat="server">usuario</asp:Label>
   </font>
 </td>
 </tr>
</table>
```

A continuación, se crean las distintas páginas de la aplicación. En los ficheros `.aspx` hay que incluir la directiva `#include` en el lugar donde queremos que aparezca la cabecera:

```
  ...
  <!-- #include virtual="BasePage.inc" -->
  ...
```

Por último, al implementar la lógica asociada a los controladores de las páginas, se implementa el método `LoadPageEvent` particular para cada página sin redefinir el método `Page_Load` que ya se definió en la clase base:

```
public class ApplicationPage : BasePage
{
  ...
  protected override void PageLoadEvent
            (object sender, System.EventArgs e) ...
}
```

Controladores de aplicación

En la mayoría de las aplicaciones web convencionales, las páginas se generan de forma dinámica pero la navegación entre las páginas es relativamente estática. En ocasiones, no obstante, la navegación entre las distintas partes de la aplicación es dinámica. Por ejemplo, los permisos del usuario o de un conjunto de reglas de configuración podrían modificar el "mapa de navegación" de la aplicación. En ese caso, la implementación de los controladores de página comienza a complicarse: aparecen expresiones lógicas complejas para determinar el flujo de la navegación. Por otro lado, puede que nuestra aplicación incluya distintos tipos de páginas, cada uno con sus peculiaridades. Cuantas más variantes haya, más niveles hay que incluir en la jerarquía de clases correspondiente a los controladores. Fuera de control, la jerarquía de controladores puede llegar a ser inmanejable.

En situaciones como las descritas en el párrafo anterior, resulta recomendable centralizar el control de la aplicación en un controlador único que se encargue de tramitar todas las solicitudes que recibe la aplicación web. Un controlador de aplicación, llamado usualmente *front controller*, suele implementarse en dos partes:

- Por un lado, un manejador [*handler*] es el que recibe las peticiones. En función de la petición y de la configuración de la aplicación (que podría almacenarse en un sencillo fichero XML), el manejador ejecuta la acción adecuada y elige la página que se le mostrará al usuario a continuación.

- Por otro lado, el conjunto de acciones que puede realizar la aplicación se modela mediante una jerarquía de **comandos**, utilizando la misma filosofía que permite implementar las operaciones de rehacer/deshacer en muchas aplicaciones. Todos los comandos implementan una interfaz unificada y cada uno de ellos representa una de las acciones que se pueden realizar.

Aparte de facilitar la creación de aplicaciones web realmente dinámicas, usar un controlador centralizado para una aplicación también repercute en otros aspectos del desarrollo de la aplicación. Ya se ha mencionado que comprobar el correcto funcionamiento de una interfaz de usuario requiere mucho esfuerzo. Lo mismo se puede decir de la lógica asociada a los controladores de página de una aplicación web. Al usar un controlador genérico e implementar las acciones de la aplicación como comandos independientes también facilitamos la fase de pruebas de la aplicación.

Respecto a los controladores de página, el uso extensivo de jerarquías de herencia conduce a aplicaciones cuyo diseño es frágil ante la posibilidad de que se tengan que realizar cambios en la aplicación. Además, determinadas funciones es mejor implementarlas de forma centralizada para evitar posibles omisiones. Piense, por ejemplo, en lo que podría suceder si en una página de una aplicación de comercio electrónico se le olvidase realizar alguna tarea de control de acceso. La centralización ayuda a la hora de mejorar la consistencia de una aplicación.

Controlador común para distintas páginas ASP.NET

Supongamos que partimos de una aplicación similar a la descrita en el ejemplo de la sección anterior ("cabecera común para distintas páginas ASP.NET). El método `Page_Load` de la clase base `BasePage` se llama siempre que se accede a una página. Supongamos ahora que el formato de la cabecera mostrada dependa de la solicitud concreta, su URL o sus parámetros. En función de la solicitud, no sólo cambiará la cabecera, sino el proceso necesario para rellenarla. Esto podría suceder si nuestra aplicación web sirve de base para dar soporte a varios clientes, para los cuales hemos personalizado nuestra aplicación. Las páginas individuales seguirán siendo comunes para los usuarios, pero la parte implementada en la clase base variará en función del caso particular, por lo que se complica la implementación de `BasePage`:

```
protected void Page_Load (object sender, System.EventArgs e)
{
  string estilo;

  if (!IsPostBack) {
     estilo = Request["style"];

     if ( estilo!=null && estilo.Equals("vip"))
        ... // Sólo VIPs
     else
        ... // Convencional

     PageLoadEvent(sender, e);
  }
}
```

Conforme se añaden casos particulares, el desastre se va haciendo más evidente. Cualquier cambio mal realizado puede afectar a aplicaciones que estaban funcionando correctamente, sin causa aparente desde el punto de vista de nuestros enojados clientes.

Para crear un controlador de aplicación que elimine los problemas de mantenimiento a los que podría conducir una situación como la descrita, lo primero que hemos de hacer es crear un manejador que reciba las peticiones. En ASP.NET podemos utilizar la interfaz a bajo nivel `IHttpHandler` tal como pusimos de manifiesto en el capítulo anterior al estudiar los manejadores y filtros HTTP:

```
using System;
using System.Web;

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

    command.Do (context);
  }
```

Controlador común para distintas páginas ASP.NET

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

El comando se limita a implementar una sencilla interfaz:

```
using System;
using System.Web;

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

En el caso de una aplicación en la que el usuario debiese tener oportunidad de deshacer sus acciones, lo único que tendríamos que hacer es añadirle un método `Undo` a la interfaz `Command` (e implementarlo correctamente en todas las clases que implementen dicha interfaz, claro está).

En vez de crear directamente los objetos de tipo `Command`, se utiliza una clase auxiliar en la que se incluye la lógica necesaria para elegir el comando adecuado. Esta fábrica de comandos (`CommandFactory`) se encargará de determinar el comando adecuado en función del contexto:

```
using System;
using System.Web;

public class CommandFactory
{
  public static Command Create (HttpContext context)
  {
    string  estilo  = context.Request["style"];
    Command command = new NullCommand();

    if ( estilo!=null && estilo.Equals("vip"))
       command = new VIPCommand(context);
    else
       command = new ConventionalCommand(context);

    return command;
  }
}
```

Para que nuestra aplicación utilice el controlador creado, tal como se describió en el capítulo anterior al hablar de los filtros HTTP, hemos de incluir lo siguiente en el fichero de configuración `Web.config`:

Controlador común para distintas páginas ASP.NET

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

Finalmente, lo último que nos falta es crear nuestra jerarquía de comandos, que serán los que se encarguen de personalizar el funcionamiento de nuestra aplicación para cada cliente:

```
using System;
using System.Web;

public class AspNetCommand : Command
{
  public void Do (HttpContext context)
  {
    ...
    context.Server.Transfer(url);
  }
}
```

En realidad, lo único que hemos hecho ha sido reorganizar las clases que implementan la interfaz de nuestra aplicación y reasignar las responsabilidades de cada una de las clases. Si lo hacemos correctamente, podemos conseguir un sistema modular con componentes débilmente acoplados que se pueden implementar y probar de forma independiente. En la siguiente sección veremos una forma más general de conseguir una aplicación web fácilmente reconfigurable.

Control de la aplicación

En una aplicación web, cada acción del usuario se traduce en una solicitud que da lugar a dos respuestas de ámbito diferente. Por un lado, se ha de realizar la tarea indicada por el usuario. Aparte, se le ha de mostrar al usuario la página adecuada, que puede no estar directamente relacionada con la tarea realizada. De hecho, una misma página puede ser la respuesta del sistema ante distintas acciones y una misma acción puede dar lugar a distintas páginas de respuesta en función del contexto de la aplicación.

En general, a partir del estado actual de la aplicación (almacenado fundamentalmente en los diccionarios `Application` y `Session`) y de la acción que realice el usuario, el sistema ha de decidir dos cosas: las acciones que se han de realizar sobre los objetos con los que trabaja la aplicación y la siguiente página que ha de mostrarse en el navegador del usuario.

En el apartado anterior, vimos cómo los controladores se encargan de tomar ambas decisiones. Obviamente, el resultado no destaca por su cohesión. En el mejor de los casos, la cohesión del controlador es puramente temporal. Es decir, las operaciones se incluyen en un

módulo porque han de realizarse al mismo tiempo.

Para organizar mejor una aplicación, podemos dividir en dos la parte responsable de la interfaz de usuario. Esta parte de la aplicación se denomina habitualmente capa de presentación. En la capa de presentación distinguir dos tipos bien diferenciados de componentes:

- Los **componentes de la interfaz de usuario** propiamente dichos: Los controles con los que los usuarios interactúan.

- Los **componentes de procesamiento de la interfaz de usuario** (componentes de procesamiento para abreviar): Estos componentes "orquestan los elementos de la interfaz de usuario y controlan la interacción con el usuario".

Esta distinción entre controles de la interfaz y componentes de procesamiento nos permite diferenciar claramente las dos responsabilidades distintas que antes les habíamos asignado a los controladores MVC:

- Los controles de la interfaz engloban las vistas del modelo MVC con las tareas realizadas por los controladores MVC para adquirir, validar, visualizar y organizar los datos en la interfaz de usuario.

- Los componentes de procesamiento se encargan de mantener el estado de la aplicación y controlar la navegación del usuario a través de sus distintas páginas.

Cuando un usuario utiliza una aplicación, lo hace con un objetivo: satisfacer una necesidad concreta. Una técnica popular para realizar el análisis de los requisitos de una aplicación consiste en emplear casos de uso. Los casos de uso describen conjuntos de acciones que ha de realizar un usuario para lograr sus objetivos. Cada acción se realiza en un contexto de interacción determinado (una página en el caso de las aplicaciones web) y el usuario va pasando de un contexto de interacción a otro conforme va completando etapas del caso de uso que esté ejecutando. Mientras que los controles de la interfaz se encargan de garantizar el funcionamiento de un contexto de interacción, los componentes de procesamiento controlan el flujo de control de un caso de uso.

> Estos "componentes de procesamiento" se suelen denominar *controladores de aplicación*, si bien hay que tener en cuenta que estamos refiriéndonos a algo totalmente distinto a lo que vimos en el apartado anterior cuando estudiamos la posibilidad de usar controladores MVC centralizados para una aplicación web. En ocasiones, para evitar malentendidos, se denominan *controladores frontales* a los controladores MVC centralizados, haciendo referencia a que sirven de punto de acceso frontal para las aplicaciones web.

Los componentes de procesamiento de la interfaz de usuario, controladores de aplicación de aquí en adelante, son responsables de gestionar el flujo de control entre las distintas páginas involucradas en un caso de uso, decidir cómo afectan las excepciones que se puedan producir y mantener el estado de la aplicación entre distintas interacciones (para lo cual acumulan datos recogidos de los distintas páginas por las que va pasando el usuario) y ofrecerle a los controles de la interfaz los datos que puedan necesitar para mostrárselos al usuario. De esta forma, se separa por completo la visualización de las páginas del control de la navegación por la aplicación.

Esta estrategia nos permite, por ejemplo, crear distintas interfaces para una aplicación sin tener que modificar más que las vistas correspondientes a los distintos contextos de interacción en que se puede encontrar el usuario. El controlador de la aplicación puede ser, por tanto, común para una aplicación Windows, una aplicación web y una aplicación para PDAs o teléfonos móviles. Además, se mejora la modularidad del sistema, se fomenta la reutilización de componentes y se simplifica el mantenimiento de la aplicación.

Un controlador de aplicación se suele implementar como un conjunto de clases a las que se accede desde la interfaz de usuario, a modo de capa intermedia entre los controles de la interfaz de usuario y el resto de la aplicación. La siguiente clase ilustra el aspecto que podría tener el controlador de una aplicación web de comercio electrónico para el caso de uso correspondiente a la realización de un pedido:

```
public class ControladorPedido
{
  private Cliente   cliente;      // Estado actual del pedido
  private ArrayList productos;
  ...

  public void MostrarPedido()
  {
    if (application.WebInterface) {
       ... // Interfaz web
       System.Web.HttpContext.Current.Response.Redirect
              ("http://csharp.ikor.org/pedidos/Pedido.aspx");
    } else {
       ... // Interfaz Windows
       FormPedido.Show();
    }
  }

  public void ElegirFormaDePago()
  {
    ... // Seleccionar la forma de pago más adecuada para el cliente
  }

  public void RealizarPedido()
  {
    if (ConfirmarPedido()) {
       ... // Crear el pedido con los datos facilitados
    }
  }
```

```
    public bool ConfirmarPedido()
    {
      ... // Solicitar una confirmación por parte del usuario
    }

    public void Cancelar()
    {
      ... // Vuelta a la página/ventana principal
    }

    public void Validar()
    {
      ... // Comprobar que todo está en orden
    }
  }
```

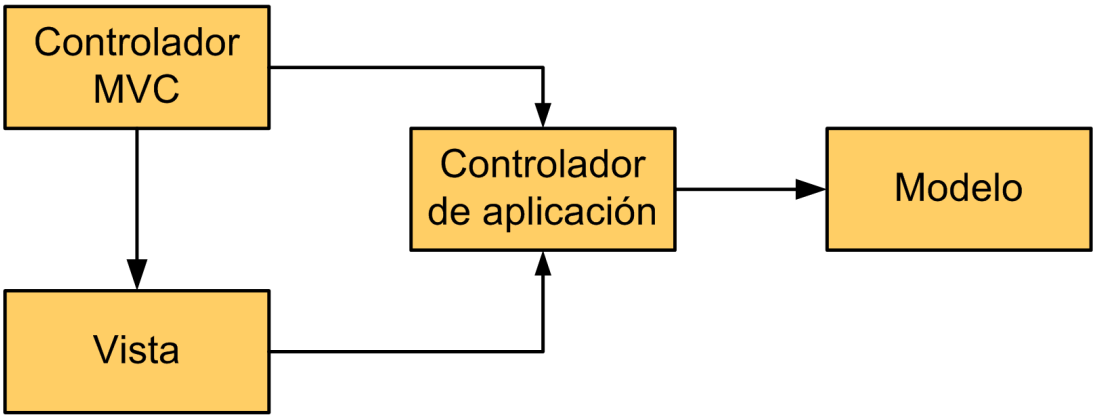
Si bien el recorrido del usuario a través de la aplicación lo determina el controlador de aplicación, las páginas concretas por las que pase el usuario no tienen por qué estar prefijadas en el código del controlador. Lo habitual es almacenar las transiciones entre páginas concretas en un fichero de configuración aparte. La lógica condicional para seleccionar la forma adecuada de mostrar la interfaz de usuario también se puede eliminar del código del controlador de aplicación si se crea un interfaz genérico que permita mostrar un contexto de interacción a partir de una URL independientemente de si estamos construyendo una aplicación Windows o una aplicación web.

Separar físicamente el controlador de aplicación de los controles de la interfaz de usuario tiene otras ventajas. Por ejemplo, un usuario podría intentar realizar un pedido y encontrarse con algún tipo de problema. Entonces, podría llamar por teléfono para que un comercial completase los pasos pendientes de su pedido sin tener que repetir el proceso desde el principio. De hecho, quizá utilice para ello una aplicación Windows que comparte con la aplicación web el controlador de aplicación.

Por otro lado, los controladores de aplicación permiten que un usuario pueda realizar varias tareas en paralelo. Por ejemplo, el usuario podría tener abiertas varias ventanas de su navegador y dejar la realización de un pedido a medias mientras consultaba el catálogo de precios de productos alternativos, por ejemplo. Al encargarse cada controlador de mantener el estado de una única tarea, se consigue una simplificación notable en el manejo de actividades concurrentes.

En el caso de las aplicaciones web ASP.NET, los controladores de aplicación son los objetos almacenados como estado de la aplicación en la variable `Session`. Los manejadores de eventos de las páginas `.aspx` accederán a dichos controladores en respuesta a las acciones del usuario.

> En el caso de las aplicaciones Windows, los formularios incluirán una variable de instancia que haga referencia al controlador de aplicación concreto. Usar variables globales nunca es una buena idea. Y mucho menos en este ámbito.



El controlador de aplicación se coloca entre el modelo y la interfaz de usuario para orquestar la ejecución de los casos de uso.

Centremos ahora nuestra atención en el diseño del controlador de aplicación. Básicamente, tenemos que considerar tres aspectos fundamentales a la hora de construir el controlador de aplicación:

- La **creación de vistas**: Un controlador concreto podría estar asociado a distintas vistas en función de la situación (por ejemplo, cuando se utiliza el mismo controlador para una aplicación web y una aplicación Windows). Lo normal será crear una clase abstracta con varias subclases concretas para cada tipo de interfaz. En función de la configuración de la aplicación, se instanciará una ventana o una página `.aspx`. Lo importante es no tener que incluir la selección de la vista concreta en el código del controlador. Esto es, todo lo contrario de lo que hicimos para ilustrar el aspecto general de un controlador de aplicación.

- La **transición entre vistas**: Esto resulta más sencillo si le asignamos un identificador único a cada vista. Este identificador se traducirá a la URL correspondiente en el caso de las aplicaciones web, mientras que en una aplicación Windows conducirá a la creación de un formulario de un tipo concreto. De hecho, el mapa de navegación completo de una aplicación puede almacenarse como un documento XML fácilmente modificable que describa el diagrama de estados de cada caso de uso del sistema.

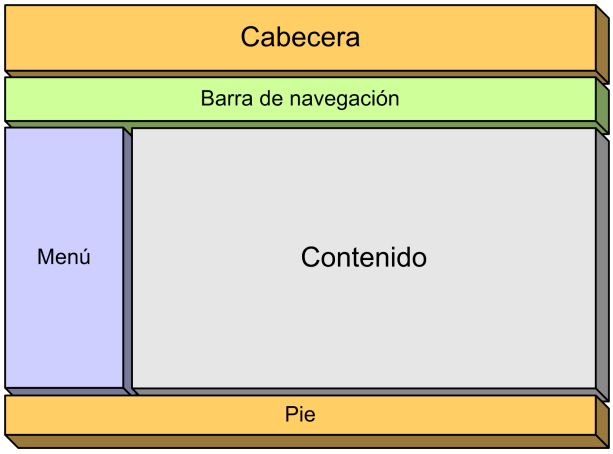
- El **mantenimiento del estado** de las tareas realizadas por el usuario: Lo más sencillo es que el estado de la aplicación se mantenga en memoria o como parte del estado de la aplicación web. En ocasiones, no obstante, puede interesar almacenar el estado del controlador de forma permanente (en una base de datos, por ejemplo) para permitir que el usuario pueda reanudar tareas que dejase a medias en sesiones anteriores.

Aspecto visual de la aplicación

En el apartado anterior hemos visto distintas formas de organizar las partes de nuestra aplicación relacionadas con la interfaz de usuario. Las decisiones de diseño tomadas nos ayudan a mejorar la calidad del diseño interno de nuestra aplicación, si bien esas decisiones no son visibles desde el exterior. De hecho, a la hora de desarrollar una aplicación, resulta relativamente fácil olvidar algunos de los factores que influyen decisivamente en la percepción que tiene el usuario de nuestra aplicación. Al fin y al cabo, para el usuario, el diseño de la aplicación es el diseño de su interfaz. Y en este caso nos referimos exclusivamente al aspecto visual de la aplicación, no al diseño modular de su capa más externa. De ahí la importancia que tiene conseguir un aspecto visual coherente en nuestras aplicaciones (aparte de un comportamiento homogéneo que facilite su uso por parte del usuario).

En este apartado, veremos cómo se pueden utilizar distintas estrategias para conseguir en nuestras aplicaciones un aspecto visual consistente. Además, lo haremos intentando evitar cualquier tipo de duplicación, ya que la existencia de fragmentos duplicados en distintas partes de una aplicación no hace más que dificultar su mantenimiento. Por otro lado, también nos interesa separar la implementación de nuestra implementación de los detalles relacionados con el diseño gráfico de la interfaz de usuario, de forma que ésta se pueda modificar libremente sin afectar al funcionamiento de la aplicación.

Cualquier aplicación web, sea del tipo que sea, suele constar de distintas páginas. Esas páginas, habitualmente, compartirán algunos elementos, como pueden ser cabeceras, pies de página, menús o barras de navegación. Por lo general, en el diseño de la aplicación se creará un boceto de la estructura que ha de tener la interfaz de la aplicación, tal como se muestra en la figura.



El espacio disponible en la interfaz de usuario se suele distribuir de la misma forma para todas las páginas de la aplicación web.

En la creación de sitios web, la distribución del espacio disponible para los distintos componentes de las páginas se puede efectuar, en principio, de dos formas diferentes: utilizando marcos (*frames*) para cada una de las zonas en las que dividamos el espacio disponible, o empleando tablas con una estructura fija que proporcione coherencia a las distintas páginas de nuestra aplicación. Si utilizamos marcos, cada marco se crea como una página HTML independiente. En cambio, si optamos por usar tablas, cada páginas deberá incluir repetidos los elementos comunes a las distintas páginas. Si nuestro análisis se quedase ahí, obviamente nos decantaríamos siempre por emplear marcos.

Desgraciadamente, el uso de marcos también presenta inconvenientes. En primer lugar, una página creada con marcos no se verá igual siempre: en función del navegador y del tamaño del tipo de letra que haya escogido el usuario, una página con marcos puede resultar completamente inutilizable. En segundo lugar, si los usuarios no siempre acceden a nuestras páginas a través de la página principal, como puede suceder si utilizan los enlaces proporcionados por un buscador, sólo verán en su navegador el contenido de uno de los marcos. En este caso, no podrán navegar por las distintas páginas de nuestro sistema si elementos de la interfaz como la barra de navegación se muestran en marcos independientes.

Las desventajas mencionadas de los marcos han propiciado que, casi siempre, las interfaces web se construyan utilizando tablas. Desde el punto de vista del programador, esto implica que determinados elementos de la interfaz aparecerán duplicados en todas las páginas de una aplicación, por lo que se han ideado distintos mecanismos mediante los cuales se pueden generar dinámicamente las páginas sin necesidad de que físicamente existan elementos duplicados.

Filtros con módulos HTTP

La primera idea que se nos puede ocurrir es interceptar todas las solicitudes que llegan al servidor HTTP, de tal forma que podamos realizar tareas de preprocesamiento o postprocesamiento sobre cada solicitud HTTP. En ASP.NET, esto se puede realizar utilizando módulos HTTP.

Los módulos HTTP funcionan como filtros que reciben las solicitudes HTTP antes de que éstas lleguen a las páginas ASP.NET. Dichos módulos han de ser completamente independientes y este hecho permite que se puedan encadenar varios módulos HTTP y construir filtros complejos por composición. Usualmente, estos módulos se utilizan para realizar tareas a bajo nivel, como puede ser el manejo de distintos juegos de caracteres, el uso de técnicas criptográficas, la compresión y decompresión de datos, la codificación y decodificación de URLs, o la monitorización del funcionamiento del sistema. No obstante, también se pueden utilizar como "decoradores", para añadirle funcionalidad de forma transparente a las páginas ASP.NET a las que accede el usuario. Sin tener que modificar las páginas en sí, que seguirán siendo independientes, se puede hacer que los módulos HTTP se encarguen de las tareas comunes a todas las páginas de una aplicación web.

Entre las ventajas de los módulos HTTP destaca su independencia, al ser totalmente independientes del resto de la aplicación. Esta independencia les proporciona una gran flexibilidad, ya que pueden combinarse libremente distintos módulos HTTP. Además, permite su instalación dinámica en una aplicación, algo conocido como *hot deployment* en inglés. Finalmente, al ser componentes completamente independientes, son ideales para su reutilización en distintas aplicaciones. Un claro ejemplo de esto es el módulo `SessionStateModule`, que se encarga del mantenimiento de las sesiones de usuario en todas las aplicaciones web ASP.NET.

ASP.NET proporciona una serie de eventos que se producen durante el procesamiento de una solicitud HTTP a los que se pueden asociar fragmentos de código. Simplemente, tendremos que crear un módulo HTTP, que no es más que una clase que implemente la interfaz `IHttpModule`. Dicha clase incluirá un método `Init` en el que indicaremos los eventos que estemos interesados en capturar. El siguiente ejemplo muestra cómo podemos añadir lo que queramos al comienzo y al final de la respuesta generada por nuestra aplicación:

```
using System;
using System.Web;

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

```
  private void BeginRequest (Object source, EventArgs e)
  {
    HttpApplication application = (HttpApplication)source;
    HttpContext     context = application.Context;
    context.Response.Write("<h1>Cabecera</h1>");
  }

  private void EndReq   uest (Object source, EventArgs e)
  {
    HttpApplication application = (HttpApplication)source;
    HttpContext context = application.Context;
    context.Response.Write("<hr> &copy; Berzal, Cortijo, Cubero");
  }

  public void Dispose()
  {
  }
}
```

Una vez que tengamos implementado nuestro módulo HTTP, lo único que nos falta es incluirlo en la sección `httpModules` del fichero de configuración `Web.config` para que se ejecute en nuestra aplicación web.

```
<httpModules>
  <add name="DecoratorModule" type="DecoratorModule, biblioteca" />
</httpModules>
```

donde `DecoratorModule` es el nombre de la clase que implementa el módulo y `biblioteca` es el nombre del assembly que contiene dicha clase (esto es, `biblioteca.dll`).

> Al ejecutarse siempre que se recibe una solicitud HTTP, y con el objetivo de evitar que se conviertan en un cuello de botella para nuestras aplicaciones, es esencial que los módulos HTTP sean eficientes.

La directiva #include

Utilizando módulos HTTP se puede conseguir que nuestras páginas ASP.NET le lleguen al usuario como una parte de la página web que se visualiza en su navegador. Lo mismo se podría lograr si se crean controles por composición que representen la cabecera, la barra de navegación, el menú o el pie de las páginas. En este caso, no obstante habrá que colocar a mano dichos controles en cada una de las páginas de la aplicación, lo que resulta más tedioso

y propenso a errores.

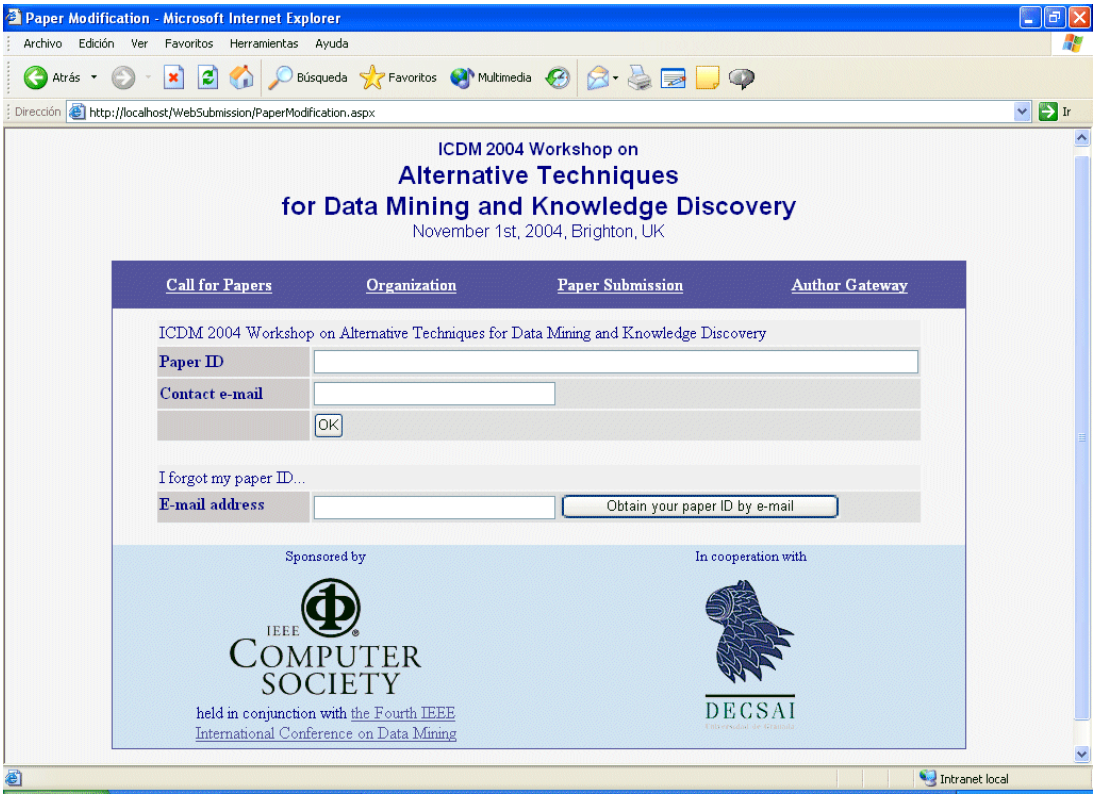
Afortunadamente, ASP.NET nos ofrece un mecanismo más sencillo que nos permite conseguir el mismo resultado sin interferir el en desarrollo de las distintas páginas de una aplicación. Nos referimos a la posibilidad de incluir en la página `.aspx` un comentario con la directiva `#include` al más puro estilo del lenguaje de programación C. Este mecanismo, en cierto modo, ya lo estudiamos cuando analizamos el uso de controladores de página en ASP.NET y, en su versión más básica, su utilización se puede resumir en los siguientes pasos:

- Se crean las partes comunes de las páginas en ficheros HTML independientes.
- Se implementan las distintas páginas de la aplicación como si se tratase de páginas independientes.
- Se incluyen las directivas `#include` en el lugar adecuado de las páginas `.aspx`.

Por ejemplo, puede que deseemos añadir a una página web la posibilidad de que el visitante nos envíe ficheros a través de la web, lo que puede ser útil en una aplicación web para la organización de un congreso o para automatizar la entrega de prácticas de una asignatura. El formulario en sí será muy simple, pero nos gustaría que quedase perfectamente integrado con el resto de las páginas web de nuestro sistema. En ese caso, podemos crear un fichero `header.inc` con la cabecera completa de nuestras páginas y otro fichero `footer.inc` con todo lo que aparece al final de nuestras páginas. En vez de incluir todo eso en nuestra página `.aspx`, lo único que haremos será incluir las directivas `#include` en los lugares correspondientes:

```
<%@ Page language="c#" Codebehind="Include.aspx.cs"
         AutoEventWireup="false" Inherits="WebFormInclude" %>
<HTML>
  <body>
    <!-- #include virtual="header.inc" -->
    <form id="WebForm" method="post" runat="server">
      ...
    </form>
    <!-- #include virtual="footer.inc" -->
  </body>
</HTML>
```

Esta estrategia de implementación es la ideal si las partes comunes de nuestras páginas son relativamente estáticas. Si no lo son, lo único que tenemos que hacer es crear una jerarquía de controladores de página, tal como se describió cuando estudiamos el uso del modelo MVC en ASP.NET.



Aspecto visual de un formulario ASP.NET creado para la organización de un congreso. El fichero .aspx sólo incluye las directivas #include y los controles propios del formulario. El aspecto visual final es el resultado de combinar el sencillo formulario con los detalles de presentación de la página, que se especifican en los ficheros header.inc y footer.inc.

Plantillas

En vez de que cada página incluya los detalles correspondientes a una plantilla general, podemos seguir la estrategia opuesta: crear una plantilla general en la cual se va sustituyendo en contenido manteniendo el aspecto general de la página. Para implementar esta estrategia en ASP.NET, se nos ofrecen dos opciones: usar una página maestra o utilizar un control maestro.

Página maestra

El uso de una página maestra se puede considerar como una forma de crear un controlador

MVC de aplicación. El usuario siempre accederá a la misma página `.aspx` pero su contenido variará en función del estado de la sesión. En la página maestra se reserva un espacio para mostrar un control u otro en función de la solicitud recibida. El evento `Page_Load` de la página maestra contendrá, entonces, algo similar a lo siguiente:

```
void Page_Load (object sender, System.EventArgs e)
{
  Control miControl = LoadControl ("fichero.cs.ascx");

  this.Controls.Add (miControl);
}
```

En el código de la página maestra se tendrá que decidir cuál es el control concreto que se mostrará en el espacio reservado al efecto. Mantener el aspecto visual de las páginas de la aplicación es una consecuencia directa de la forma de implementar la aplicación, igual que el hecho de que no se pueden utilizar los mecanismos proporcionados por ASP.NET para restringir el acceso a las distintas partes de la aplicación (por el simple hecho de que siempre se accede a la misma página). Además, como las solicitudes han de incluir la información necesaria para decidir qué página mostrar, este hecho hace especialmente vulnerable a nuestra aplicación frente a posibles ataques. Deberemos ser especialmente cuidadosos para que un usuario nunca pueda acceder a las partes de la aplicación que se supone no puede ver.

Control maestro

Para evitar los inconvenientes asociados a la creación de una página maestra, se puede utilizar un control maestro que se incluya en distintas páginas ASP.NET. El control maestro realiza las mismas tareas que la página maestra, salvo que, ahora, el usuario accede a distintas páginas `.aspx`.

El control maestro se crea como cualquier otro control definido por el usuario en ASP.NET, si bien ha de definir el aspecto visual general de las páginas de nuestra aplicación e incluir una zona modificable. En esta zona modificable, que puede ser una celda de una tabla, se visualizará el contenido que cambia dinámicamente de una página a otra de la aplicación.

La aplicación, por tanto, constará de un conjunto de páginas `.aspx`, cada una de las cuales incluye una instancia del control maestro. Al acceder a una página, en el evento `Form_Load`, se establecerá el control concreto que ha de mostrar el control maestro.

Configuración dinámica de la aplicación

Los capítulos anteriores sirvieron para que nos familiarizásemos con la construcción de aplicaciones web en la plataforma .NET. En las secciones anteriores vimos cómo se pueden organizar las distintas partes de una aplicación web directamente relacionadas con la interfaz de usuario. Por otro lado, analizamos distintas formas de crear aplicaciones web consistentes y homogéneas, además de fácilmente mantenibles. Ahora, ha llegado el momento de dotar de contenido a las páginas `.aspx` que conforman nuestras aplicaciones ASP.NET.

Los controles incluidos como parte de la biblioteca de clases de la plataforma .NET ofrecen un mecanismo conocido con el nombre de "enlace de datos" [*data binding*] mediante el cual podemos asociarles valores de forma dinámica a sus propiedades. Este mecanismo nos permite, por ejemplo, mostrarle al usuario conjuntos de datos de una forma flexible, cambiar las opciones existentes a su disposición o modificar el aspecto de la aplicación en función de las preferencias del usuario, todo ello sin tener que modificar una sola línea de código e independientemente de dónde provengan los datos que permiten modificar las propiedades de los controles de nuestra aplicación. Esos datos pueden provenir de una simple colección, de un fichero XML o de un `DataSet` que, posiblemente, hayamos obtenido a partir de los datos almacenados en una base de datos convencional.

A grandes rasgos, se puede decir que existen tres formas diferentes de asociarles valores a las propiedades de los controles de la interfaz gráfica:

- Como primera opción, podemos especificar los valores que deseemos como atributos de las etiquetas asociadas a los controles (esto es, dentro de `<asp:... />`). Si estamos utilizando un entorno de desarrollo como el Visual Studio .NET, estos valores los podemos fijar directamente desde la ventana de propiedades. Obviamente, así se le asignan valores a las propiedades de los controles de forma estática, no dinámica.

- Una alternativa consiste en implementar fragmentos de código que hagan uso del modelo orientado a objetos asociado a los controles de nuestros formularios. Esto es, como respuesta a algún evento, como puede ser el evento `Form_Load` de un formulario web, establecemos a mano los valores adecuados para las propiedades que deseemos modificar. Ésta es una opción perfectamente válida para rellenar valores o listas de valores cuando éstos se obtienen a partir del estado de la sesión de usuario.

- Por último, la alternativa más flexible consiste en obtener de alguna fuente de datos externa los valores que vamos a asociar a una propiedad de un control. Esta estrategia es extremadamente útil para simplificar la implementación en situaciones más complejas. La plataforma .NET nos ofrece un mecanismo, conocido con el nombre de "enlace de datos", que nos facilita el trabajo en este

contexto. En primer lugar, debemos disponer de un objeto que contenga los valores en cuestión, ya sea una simple colección (como un vector, instancia de la clase `Array` en .NET) o algo más sofisticado (como un `DataSet` de los usados en ADO.NET). A continuación, simplemente tenemos que asociarle ese objeto a la propiedad adecuada del control. El enlace de datos se realizará automáticamente cuando invoquemos el método `DataBind` asociado al control.

El mecanismo de enlace de datos ofrece varias ventajas con respecto a la forma tradicional de rellenar dinámicamente los valores de las propiedades de un control (que, recordemos, consiste en implementar las asignaciones correspondientes). La primera de ellas es que nos permite separar claramente el código de nuestra aplicación de la interfaz de usuario. Al crear un nivel extra de indirección, se desacoplan los conjuntos de valores de las propiedades de los controles a las que luego se asignan. Además, el mecanismo está implementado de tal forma que podemos enlazar las propiedades de un control a una amplia variedad de fuentes de datos. Es decir, los valores pueden provenir de colecciones (de cualquier tipo de los definidos en la biblioteca de clases .NET, desde `Array` hasta `Hashtable`), pero también pueden provenir de conjuntos de datos ADO.NET (como el versátil `DataSet`, la tabla `DataTable`, la vista `DataView` o el eficiente `DataReader`), los cuales, a su vez, pueden construirse a partir de una consulta sobre una base de datos relacional o a partir de un fichero XML. De hecho, cualquier objeto que implemente la interfaz `IEnumerable` puede usarse como fuente de datos. La existencia de esta interfaz permite que, vengan de donde vengan los valores, la implementación del enlace sea siempre la misma, lo que simplifica enormemente el mantenimiento de la aplicación cuando cambian nuestras fuentes de datos.

Otro aspecto que siempre debemos tener en mente es que el enlace no es del todo dinámico en ASP.NET. A diferencia de los formularios Windows, los cambios que se produzcan en los controles ASP.NET que muestran los datos no se propagan al conjunto de datos. En realidad, el mecanismo de enlace de datos en ASP.NET se limita a rellenar las propiedades que hayamos enlazado una única vez. Si, por el motivo que sea, nuestro conjunto de datos cambia y queremos que ese cambio se refleje en la interfaz de usuario, seremos nosotros los únicos responsables de hacer que los cambios se reflejen en el momento adecuado.

De forma análoga a las etiquetas JSP que se emplean para construir aplicaciones web con Java, en las páginas ASP.NET se pueden incluir expresiones de enlace de datos de la forma `<%# expresión %>`. Un poco más adelante veremos algunos ejemplos que ilustrarán su utilización en la práctica. En las siguientes secciones, comenzaremos a ver cómo funciona el enlace de datos en una amplia gama de situaciones.

El mecanismo de enlace de datos se puede utilizar tanto en la construcción de aplicaciones Windows como en la creación de aplicaciones web con ASP.NET. No obstante, en este último caso, el enlace es unidireccional. Los formularios Windows permiten automatizar las actualizaciones en el conjunto de datos. En ASP.NET, las actualizaciones tendremos que implementarlas manualmente.

Listas de opciones

Comencemos por el caso más sencillo (y común) que se nos puede presentar a la hora de crear la interfaz de usuario de nuestra aplicación. Es habitual que, en un formulario, determinados campos puedan sólo tomar un valor dentro de un conjunto de valores preestablecido, como pueden ser el estado civil o la nacionalidad de una persona, el método de pago o la forma de envío de un pedido. En el caso de ASP.NET, para este tipo de campos utilizaremos alguno de los siguientes controles:

- un conjunto de botones de radio `asp:RadioButtonList`, para mostrar un conjunto de opciones mutuamente excluyentes,
- un conjunto de botones de comprobación `asp:CheckBoxList`, cuando las distintas opciones no son mutuamente excluyentes,
- una lista convencional de tipo `asp:Listbox`, o, incluso,
- una lista desplegable de valores `asp:DropDownList` (cuando no dispongamos de demasiado espacio en pantalla).

Sea cual sea el control utilizado, los valores permitidos se definen utilizando componentes de tipo `asp:ListItem`. Si, por ejemplo, deseásemos mostrar una lista de opciones que permitiese al usuario seleccionar una zona geográfica determinada, podríamos escribir a mano lo siguiente (o emplear el editor de propiedades para la colección `Items` del control correspondiente a la lista):

```
<html>
 <body>
  <form runat="server">
    <asp:RadioButtonList id="provincia" runat="server">
     <asp:ListItem value="GR" text="Granada" />
     <asp:ListItem value="AL" text="Almería" />
     <asp:ListItem value="MA" text="Málaga" />
     <asp:ListItem value="J"  text="Jaén" />
     <asp:ListItem value="CO" text="Córdoba" />
     <asp:ListItem value="SE" text="Sevilla" />
     <asp:ListItem value="CA" text="Cádiz" />
     <asp:ListItem value="HU" text="Huelva" />
    </asp:RadioButtonList>
  </form>
 </body>
</html>
```

Sin embargo, resulta mucho más adecuado utilizar una fuente de datos independiente para rellenar la lista. Si no, ¿qué sucedería cuando nuestra aplicación abarcase un área geográfica

más amplia? Tendríamos que ir modificando, una por una, todas las listas de este tipo que tuviésemos repartidas por la aplicación.

El mecanismo de enlace de datos nos es extremadamente útil en estas situaciones, ya que nos permite rellenar las listas de valores permitidos a partir de los datos obtenidos de alguna fuente externa que tengamos a nuestra disposición. Si la lista de opciones se tuviese que modificar, sólo habría que modificar la fuente de datos, que debería ser única, y no los elementos de la lista en todos los lugares donde apareciesen. En definitiva, se separa por completo la interfaz de la aplicación de los datos con los que trabaja la aplicación, con lo que su mantenimiento resultará más sencillo.

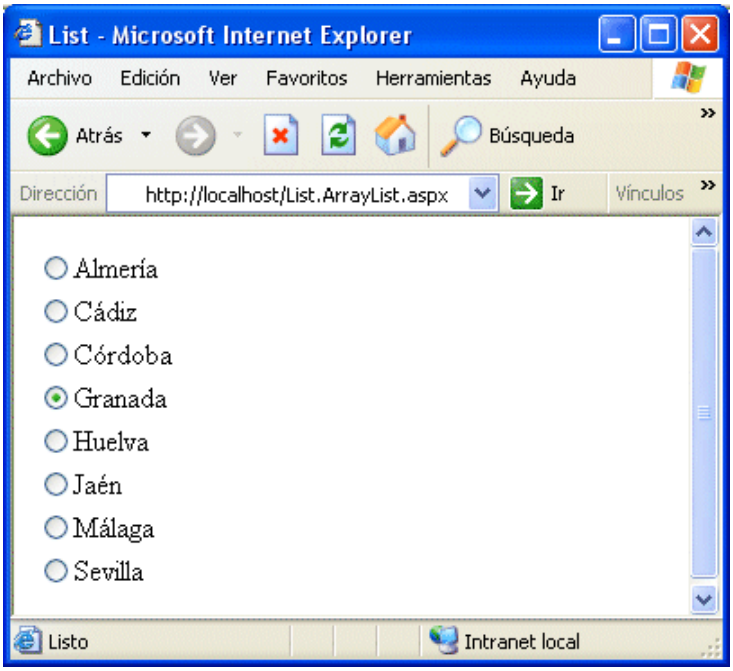
Ahora veremos cómo podemos rellenar dinámicamente una lista de opciones mostrada a partir de distintas fuentes de datos. En concreto usaremos distintos tipos de colecciones y ficheros XML que nos permitirán configurar dinámicamente muchas partes de nuestras aplicaciones.

Vectores simples: Array y ArrayList

La clase `System.Array` es la clase que representa un vector en la plataforma .NET y sirve como base para todas las matrices en C#. Por su parte, la clase `System.Collections.ArrayList` nos permite crear vectores de tamaño dinámico. Ambas clases implementan el mismo conjunto de interfaces (`IList, ICollection, IEnumerable, ICloneable`) y pueden utilizarse para enlazar colecciones de datos a las propiedades de un control.

Al cargar la página ASP.NET, en el evento `Form_Load`, obtenemos la lista de valores y se la asociamos a la propiedad `DataSource` de la lista de opciones. Para rellenar la lista de opciones con los valores incluidos en la colección de valores, no tenemos más que llamar al método `DataBind()` del control:

```
private void Page_Load(object sender, System.EventArgs e)
{
  if (!Page.IsPostBack) {
     ArrayList list = new ArrayList();

     list.Add("Granada");
     list.Add("Almería");
     list.Add("Málaga");
     list.Add("Jaén");
     list.Add("Córdoba");
     list.Add("Sevilla");
     list.Add("Cádiz");
     list.Add("Huelva");
     list.Sort();

     provincia.DataSource = list;
     provincia.DataBind();
  }
}
```



Lista de opciones creada dinámicamente a partir de una colección.

Para simplificar el ejemplo, en el mismo evento `Form_Load` hemos creado la lista de tipo `ArrayList` y hemos añadido los valores correspondientes a las distintas opciones con el método `Add()`. Una vez que tenemos los elementos en la lista, los hemos ordenado llamando al método `Sort()`. Si hubiésemos querido mostrarlos en orden inverso, podríamos haber usado el método `Reverse()` una vez que los tuviésemos ordenados.

Finalmente, el enlace de la colección con el control se realiza mediante la propiedad `DataSource` del control y la llamada al método `DataBind()` cuando queremos rellenar la lista. En este caso, los elementos del vector se utilizan como texto (`Text`) y como valor (`Value`) asociado a los distintos elementos `ListItem` de la lista, si bien esto no tiene por qué ser así siempre, como veremos a continuación.

Pares clave-valor: Hashtable y SortedList

Generalmente, cuando utilizamos listas de opciones en la interfaz de usuario, una cosa es el mensaje que vea el usuario asociado a cada opción y otra muy distinta es el valor que internamente le asignemos a la opción seleccionada. El mensaje varía, por ejemplo, si nuestra

aplicación está localizada para varios idiomas, mientras que el valor asociado a la opción será fijo. Incluso aunque nuestra aplicación sólo funcione en castellano, es buena idea que, en las opciones de una lista, texto y valor se mantengan independientes. El texto suele ser más descriptivo y puede variar con mayor facilidad. Además, no nos gustaría que un simple cambio estético en el texto de una opción hiciese que nuestra aplicación dejase de funcionar correctamente.

Cuando queremos asignarle dos valores a las propiedades de cada una de las opciones de una lista, un simple vector no nos es suficiente. Hemos de utilizar colecciones que almacenen pares clave-valor, como es el caso de las colecciones que implementan la interfaz `IDictionary`, como pueden ser `Hashtable` o `SortedList`.

Una tabla hash, implementada en la biblioteca de clases .NET por la clase `System.Collections.Hashtable`, es una estructuras de datos que contiene pares clave-valor y está diseñada para que el acceso a un valor concreto sea muy eficiente, *O(1)* utilizando la terminología habitual en el análisis de algoritmos. En nuestro ejemplo de la lista de provincias, podríamos utilizar una tabla hash para almacenar como clave el código correspondiente a una provincias y como valor su nombre completo. La clave la asociamos al valor de la opción (`DataValueField`) y, el nombre, al texto de la etiqueta asociada a la opción (`DataTextField`):

```
private void Page_Load(object sender, System.EventArgs e)
{
  if (!Page.IsPostBack) {
     Hashtable table = new Hashtable();

     table.Add("GR", "Granada");
     table.Add("AL", "Almería");
     table.Add("MA", "Málaga");
     table.Add("J",  "Jaén");
     table.Add("CO", "Córdoba");
     table.Add("SE", "Sevilla");
     table.Add("CA", "Cádiz");
     table.Add("HU", "Huelva");

     provincia.DataSource = table;
     provincia.DataValueField="Key";
     provincia.DataTextField="Value";
     provincia.DataBind();
  }
}
```

El único inconveniente de usar tablas hash es que no se puede elegir el orden de presentación de los elementos de la lista. Para que las opciones de la lista aparezcan ordenadas tenemos que usar una colección como la proporcionada por la clase `System.Collections.SortedList`, que se emplea exactamente igual que la tabla hash y mantiene sus elementos ordenados de forma automática.

Ficheros XML

Hoy en día, la opción más comúnmente utilizada para configurar dinámicamente una aplicación consiste en la utilización de ficheros XML auxiliares que contengan los datos necesarios para configurar adecuadamente una aplicación. A partir de datos guardados en ficheros XML, se puede personalizar el menú de opciones visible para un usuario en función de sus permisos, cambiar el aspecto visual de las páginas de la aplicación en función de las preferencias de un usuario concreto o, incluso, instalar distintas "versiones" de una misma aplicación para distintos clientes.

En el caso de una aplicación web ASP.NET, para leer un fichero XML sólo tenemos que teclear lo siguiente:

```
DataSet dataset = new DataSet();
dataset.ReadXml ( Server.MapPath("fichero.xml") );
```

La función `ReadXml()` construye un conjunto de datos a partir del contenido de un fichero XML. La función `Server.MapPath()` es la que nos permite buscar el fichero dinámicamente en el lugar adecuado; esto es, en el directorio donde tengamos instalada la aplicación web.

Si volvemos a nuestro ejemplo simplón, el de la lista de provincias, podemos crear un fichero XML auxiliar para rellenar la lista de opciones a partir de dicho fichero. Este fichero puede tener el siguiente formato:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<provincias>
 <provincia>
  <id>AL</id>
  <nombre>Almería</nombre>
 </provincia>
 ...
 <provincia>
  <id>GR</id>
  <nombre>Granada</nombre>
 </provincia>
 ...
</provincias>
```

Una vez que tenemos el fichero con los datos, lo único que tenemos que hacer en el código de nuestra aplicación es cargar dicho fichero en un conjunto de datos creado al efecto y enlazar el conjunto de datos al control que representa la lista de opciones. El resultado es bastante sencillo y cómodo de cara al mantenimiento futuro de la aplicación:

```
private void Page_Load(object sender, System.EventArgs e)
{
  DataSet dataset;

  if (!Page.IsPostBack) {
     dataset = new DataSet();
     dataset.ReadXml(MapPath("provincias.xml"));

     provincia.DataSource = dataset;
     provincia.DataValueField="id";
     provincia.DataTextField="nombre";
     provincia.DataBind();
  }
}
```

Obsérvese cómo se realiza en enlace de datos con los elementos del fichero XML: el elemento `id` se hace corresponder con el valor asociado a una opción de la lista, mientras que el elemento `nombre` será lo que se muestre como etiqueta asociada a cada opción.

Caso práctico

En capítulos anteriores pusimos como ejemplo una sencilla lista de contactos pero nos dejamos en el tintero cómo se crea dinámicamente la lista desplegable a partir de la cual podemos ver los datos concretos de cada uno de nuestros contactos. Ahora, ya estamos en disposición de ver cómo se construye dinámicamente esa lista para completar nuestra sencilla aplicación de manejo de una agenda de contactos.

El formulario que nos permite ver los datos de una lista de contactos.

Antes de ver cómo se rellena esa lista desplegable, no obstante, nos hacen falta algunas rutinas auxiliares que nos permitan leer y escribir los datos asociados a cada contacto. Para lograrlo, podemos utilizar las facilidades que nos ofrece la plataforma .NET para serializar objetos en XML, algo que podemos implementar de forma genérica en una clase auxiliar:

```
using System
using System.IO;
using System.Xml;
using System.Xml.Serialization;

public class Utils
{
  // Serialización

  public static void Serialize (object obj, string filename)
  {
    XmlSerializer serializer = new XmlSerializer (obj.GetType());
    TextWriter writer = new StreamWriter (filename, false,
                System.Text.Encoding.GetEncoding("ISO-8859-1"));
    serializer.Serialize (writer, obj);
    writer.Close();
  }

  // Deserialización

  public static object Deserialize (Type type, string filename)
  {
    object obj = null;

    XmlSerializer serializer = new XmlSerializer (type);
    TextReader stream = new StreamReader (filename,
                System.Text.Encoding.GetEncoding("ISO-8859-1"));
    XmlReader  reader = new XmlTextReader(stream);

    if (serializer.CanDeserialize(reader))
       obj = serializer.Deserialize (reader);

    reader.Close();
    stream.Close();

    return obj;
  }
}
```

Las utilidades definidas en la clase anterior, cuando las aplicamos sobre objetos de tipo `Contact` (la clase que utilizamos para encapsular los datos que teníamos de un contacto concreto), dan lugar a ficheros XML de la siguiente forma:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<Contact xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Name>Mota</Name>
  <EMail>mota@mismascotas.com</EMail>
  <Address>La casa de su dueño</Address>
  <Comments>Yorkshire Terrier</Comments>
  <ImageURL>image/mota.jpg</ImageURL>
</Contact>
```

Ahora bien, volvamos a nuestra aplicación web. Supongamos que los datos relativos a nuestros contactos los almacenamos en un subdirectorio denominado `contacts` del directorio que aloja nuestra aplicación web. Para leer desde un fichero XML los datos de un contacto, lo único que tendremos que hacer es escribir algo similar a lo siguiente:

```
private Contact GetContact (string filename)
{
  FileInfo fi = new FileInfo
                    ( Server.MapPath("contacts")+"/"+filename );

  return (Contact) Utils.Deserialize(typeof(Contact), fi.FullName );
}
```

Por tanto, ya sabemos cómo leer los datos de un contacto. Lo único que nos falta es rellenar la lista desplegable de opciones que nos servirá de índice en nuestra agenda de contactos. Eso es algo que podemos hacer fácilmente a partir de lo que ya sabemos:

```
private void Page_Load(object sender, System.EventArgs e)
{
  DirectoryInfo di;
  SortedList    list;
  FileInfo[]    fi;
  Contact       contact;

  if (!Page.IsPostBack) {

      list = new SortedList();

      di = new DirectoryInfo( Server.MapPath("contacts") );

      if (di!=null) {

          fi = di.GetFiles();

          foreach (FileInfo file in fi) {
              contact = GetContact( file.Name );
              list.Add(contact.Name, file.Name);
          }

          ContactList.DataSource = list;
          ContactList.DataTextField = "Key";
          ContactList.DataValueField = "Value";
          ContactList.DataBind();

          Header.Visible = false;
          ContactList.SelectedIndex = 0;
      }

  } else {
      UpdateUI();
  }
}
```

A partir del contenido del directorio donde almacenamos los datos, vemos los ficheros que hay y creamos una entrada en la lista desplegable. Para saber en cada momento cuál es el contacto cuyos datos hemos de mostrar, lo único que tenemos que hacer es ver cuál es la opción seleccionada de la lista desplegable. Dicha opción viene dada por la propiedad `SelectedItem` de la lista:

```
private void UpdateUI ()
{
  string  filename;
  Contact contact;

  if (ContactList.SelectedItem!=null) {
     filename = ContactList.SelectedItem.Value;
     contact = GetContact(filename);
     LabelContact.Text = ContactList.SelectedItem.Text;
     ShowContact(contact);
  }
}
```

Finalmente, sólo nos queda hacer que, siempre que el usuario seleccione una de las opciones de la lista, los datos que se muestren correspondan a la opción seleccionada. Para lograrlo, sólo tenemos que redefinir la respuesta de nuestro formulario al evento `SelectedIndexChanged` de la lista desplegable:

```
private void ContactList_SelectedIndexChanged
              (object sender, System.EventArgs e)
{
  UpdateUI();
}
```

Con esto se completa la sencilla agenda de contactos que empezamos a construir hace un par de capítulos al comenzar nuestro aprendizaje sobre ASP.NET. No obstante, aún nos quedan algunos temas por tratar, entre los que se encuentra la visualización de conjuntos de datos en los formularios ASP.NET por medio de controles como los que se estudiarán en la siguiente sección.

Conjuntos de datos

La mayor parte de las aplicaciones trabajan con conjuntos de datos. Dado que es bastante normal tener que mostrar un conjunto de datos en una tabla o en un informe, también lo es que la biblioteca de clases de la plataforma .NET incluya controles que permitan visualizar conjuntos de datos. En esta sección veremos los tres controles que se utilizan con este fin en la plataforma .NET y mostraremos cómo se les pueden enlazar conjuntos de datos.

El control asp:Repeater

La etiqueta `asp:Repeater` corresponde a un control ASP.NET que permite mostrar listas de una forma más versátil que los controles vistos en la sección anterior. El control `asp:Repeater`, implementado por la clase `System.Web.UI.WebControls.Repeater`, permite aplicar una plantilla a la visualización de cada uno de los elementos del conjunto de datos.

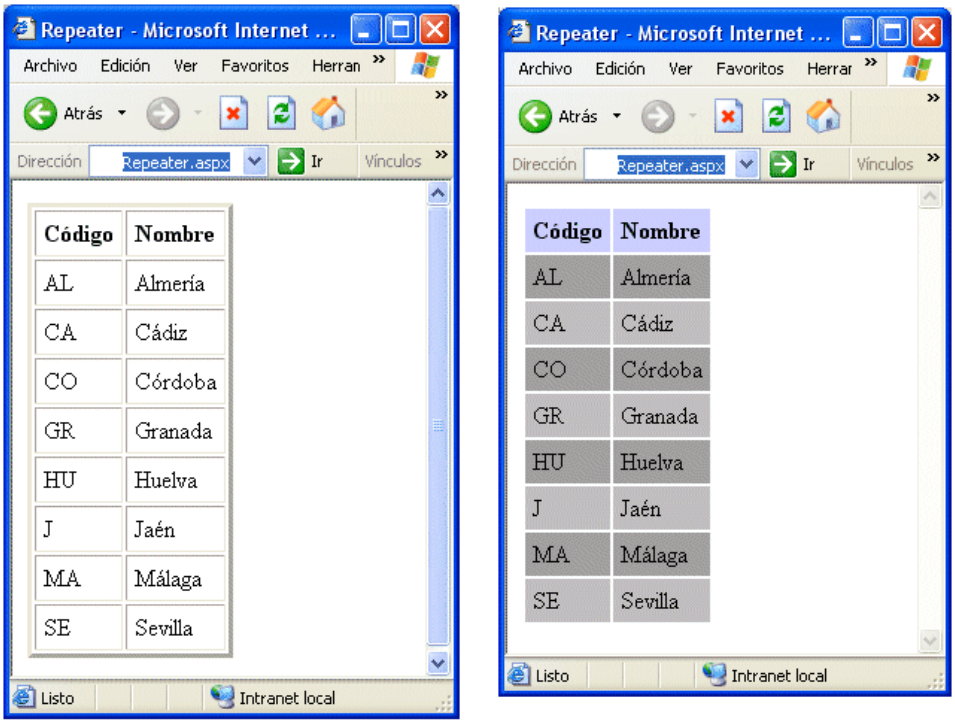
Dichas plantillas, que deberemos crear a mano, especifican el formato en el que se mostrará cada elemento, usualmente mediante el uso de las etiquetas asociadas a la creación de tablas en HTML. Para crear una tabla, se incluye la etiqueta `<table>` de comienzo de la tabla y la cabecera de la misma en la plantilla `HeaderTemplate`, que corresponde a la cabecera generada por el control `asp:Repeater` antes de mostrar los datos. De forma análoga, la etiqueta `</table>` de cierre de la tabla se incluye en la plantilla `FooterTemplate` del control `asp:Repeater`. Por su parte, cada uno de los elementos del conjunto de datos se mostrará tal como especifiquemos en la plantilla `ItemTemplate`, que estará delimitada por las etiquetas HTML `<tr>` y `</tr>`, correspondientes a una fila de la tabla. Como mínimo, un control `asp:Repeater` debe incluir la definición de la plantilla `ItemTemplate`. Todas las demás plantillas son opcionales.

Veamos, en la práctica, cómo funciona el control `asp:Repeater` con el ejemplo que pusimos en la sección anterior. En primer lugar, tenemos que añadir un control `asp:Repeater` al formulario web con el que estemos trabajando. A continuación, podemos obtener el conjunto de datos desde el fichero XML utilizado en el último ejemplo de la sección anterior. Dicho conjunto de datos hemos de enlazarlo al control `asp:Repeater`, lo que conseguimos usando su propiedad `DataSource`:

```
protected System.Web.UI.WebControls.Repeater provincia;
...

private void Page_Load(object sender, System.EventArgs e)
{
   if (!Page.IsPostBack) {
      DataSet dataset = new DataSet();
      dataset.ReadXml(MapPath("provincias.xml"));
      provincia.DataSource = dataset;
      provincia.DataBind();
   }
}
```

Por último, hemos de especificar cómo se visualizará cada dato del conjunto de datos en el control `asp:Repeater`, para lo cual no nos queda más remedio que editar el código HTML de la página ASP.NET. Mediante el uso de plantillas definimos el aspecto visual de nuestro conjunto de datos:

```
<asp:Repeater id="provincia" runat="server">
  <HeaderTemplate>
    <table border="3" cellpadding="5">
      <tr>
        <th>Código</th>
        <th>Nombre</th>
      </tr>
  </HeaderTemplate>
  <ItemTemplate>
    <tr>
      <td>
        <%# DataBinder.Eval(Container.DataItem, "id") %>
      </td>
      <td>
        <%# DataBinder.Eval(Container.DataItem, "nombre") %>
      </td>
    </tr>
  </ItemTemplate>
  <FooterTemplate>
    </table>
  </FooterTemplate>
</asp:Repeater>
```



Visualización de conjuntos de datos con asp:Repeater

El resultado de la ejecución de nuestro formulario con el control `asp:Repeater` se muestra en la parte izquierda de la figura. Para entender su funcionamiento, debemos tener en cuenta lo siguiente:

- La plantilla `<HeaderTemplate>` se muestra al comienzo de la salida asociada al control `asp:Repeater`. En este caso, lo único que hace es crear la cabecera de la tabla.

- Para cada elemento del conjunto de datos enlazado al `asp:Repeater` se utiliza la plantilla `<ItemTemplate>`. Esta plantilla la rellenamos con las etiquetas HTML correspondientes a la definición de una fila de la tabla e incluimos expresiones del tipo `<%# DataBinder.Eval(Container.DataItem, "campo") %>` para visualizar los valores que toma `campo` en los elementos de nuestro conjunto de datos.

- Por último, la plantilla `<FooterTemplate>` se utiliza para finalizar la salida asociada al control `asp:Repeater`, que muchas veces no es más que cerrar la etiqueta con la que se abrió la tabla.

Obviamente, la tabla mostrada en la parte derecha de la figura anterior es mucho más vistosa que la sobria tabla que se visualiza con las plantillas que hemos definido para el control `asp:Repeater`. Para construir una tabla de este tipo, lo único que tenemos que hacer es definir la plantilla opcional `<AlternatingItemTemplate>`. Si incluimos el código HTML necesario para utilizar distintos colores en `<ItemTemplate>` y `<AlternatingItemTemplate>` podemos conseguir que las filas de la tabla se muestren con colores alternos, como en la figura, de forma que se mejore la legibilidad de los datos y el atractivo visual de la tabla. Como su propio nombre sugiere, la plantilla `<AlternatingItemTemplate>` determina la visualización de elementos alternos del conjunto de datos. Cuando no se define, `<ItemTemplate>` se utiliza para todos los elementos del conjunto de datos. Cuando se define, las filas pares e impares se mostrarán de forma distinta en función de lo que hayamos especificado en las plantillas `<ItemTemplate>` y `<AlternatingItemTemplate>`.

Finalmente, el control `asp:Repeater` incluye la posibilidad de utilizar una quinta plantilla opcional: la plantilla `<SeparatorTemplate>`. Esta plantilla se puede utilizar para describir lo que queremos que aparezca entre dos elementos del conjunto de datos.

En resumen, el control `asp:Repeater` muestra los elementos de un conjunto de datos generando un fragmento de código HTML a partir de las plantillas que hayamos definido. En el caso más sencillo, se define una única plantilla, la plantilla `<ItemTemplate>`. En el caso más complejo, se pueden utilizar hasta cinco plantillas que se utilizarán para generar la salida en el siguiente orden: `Header - Item - Separator - AlternatingItem - Separator - Item - ... - Footer`.

Una de las limitaciones del control `asp:Repeater` es que no admite cómodamente la implementación funciones de selección ni de edición. Afortunadamente, en ASP.NET existen

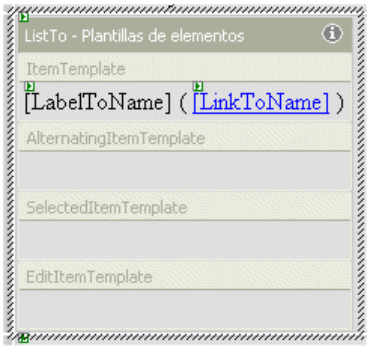
otros controles, que se pueden emplear para visualizar e incluso modificar conjuntos de datos. En concreto, nos referimos a los controles `System.Web.UI.WebControls.DataList` y `System.Web.UI.WebControls.DataGrid` que usaremos en los dos próximos apartados.

El control asp:DataList

El control `asp:DataList` es similar a `asp:Repeater`, si bien incluye por defecto una tabla alrededor de los datos. Además, se puede diseñar su aspecto de forma visual desde el diseñador de formularios del Visual Studio .NET. Desde el diseñador de formularios, podemos seleccionar el control `asp:DataList` y pinchar con el botón derecho del ratón para acceder a las opciones de *"Editar plantilla"* incluidas en su menú contextual. Desde ahí podemos definir una amplia gama de propiedades del control de forma visual.

El control `asp:DataList`, además de las cinco plantillas que ya mencionamos al hablar del control `asp:Repeater`, incluye dos plantillas adicionales: `SelectedItemTemplate` y `EditItemTemplate`. Estas plantillas se usan, respectivamente, para resaltar el elemento seleccionado de la lista y permitir la edición de los valores correspondientes a uno de los elementos del conjunto de datos.

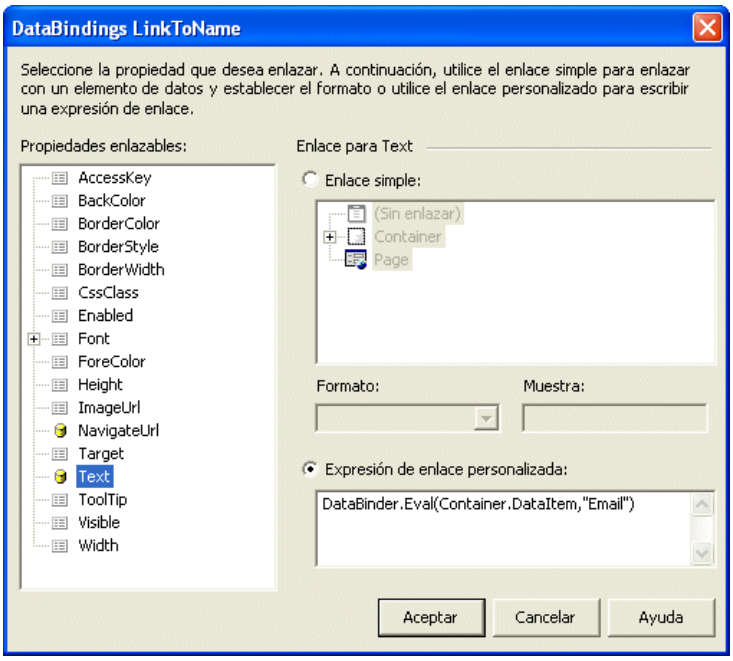
Volviendo al ejemplo de la lista de contactos, supongamos que deseamos mostrar, simultáneamente, datos relativos a varios de nuestros contactos, sin tener que ir eligiéndolos uno a uno. Podemos usar un control de tipo `asp:DataList` y definir visualmente el formato de su plantilla `ItemTemplate`:



Edición visual del formato correspondiente a las plantillas de un control asp:DataList

En este caso, cada vez que se muestre un contacto, aparecerá una etiqueta `asp:Label` con su nombre acompañada de un enlace `asp:HyperLink` en el que figurará su dirección de correo electrónico. Mediante la opción `(DataBindings)` que aparece en el editor de

propiedades de los controles, podemos enlazar el nombre y la dirección de correo con los controles correspondientes, tal como se muestra en la figura:



Manualmente, tenemos que ir seleccionando las propiedades adecuadas de los controles, como puede ser la propiedad `NavigateUrl` del control `asp:HyperLink`, e indicar cuáles van a ser los valores que se les van a asociar a dichas propiedades. En el caso de la propiedad `NavigateUrl`, escribiremos algo como lo siguiente:

```
"mailto:"+DataBinder.Eval(Container.DataItem,"Email")
```

Lo que hacemos visualmente desde el entorno de desarrollo se traduce internamente en una plantilla similar a las que ya vimos al ver el funcionamiento del control `asp:Repeater`. Al final, nuestro fichero `.aspx` contendrá algo como lo siguiente:

```
<asp:DataList id="ListTo" runat="server">
 <ItemTemplate>

  <asp:Label id=LabelToName runat="server"
   Text=
    '<%# DataBinder.Eval(Container.DataItem,"Name") %>'>
  </asp:Label>

  (
```

```
    <asp:HyperLink id=LinkToName runat="server"
     Text=
      '<%# DataBinder.Eval(Container.DataItem,"Email") %>'
     NavigateUrl=
      '<%# "mailto:"+DataBinder.Eval(Container.DataItem,"Email") %>'>
    </asp:HyperLink>
    )
    </ItemTemplate>
  </asp:DataList>
```

Igual que siempre, lo único que nos queda por hacer es obtener un conjunto de datos y asociárselo a la lista. Por ejemplo, podríamos crear una tabla de datos a medida que contenga únicamente aquellos datos que queremos mostrar en el control `asp:DataList`:

```
    DataTable dt = new DataTable();
    DataRow   dr;
    int       i;

    dt.Columns.Add(new DataColumn("Name", typeof(string)));
    dt.Columns.Add(new DataColumn("Email", typeof(string)));

    for (i=0; i<contacts.Length; i++) {
        dr = dt.NewRow();
        dr[0] = contacts[i].Name;
        dr[1] = contacts[i].Email;
        dt.Rows.Add(dr);
    }

    list.DataSource = dt;
    list.DataBind();
```

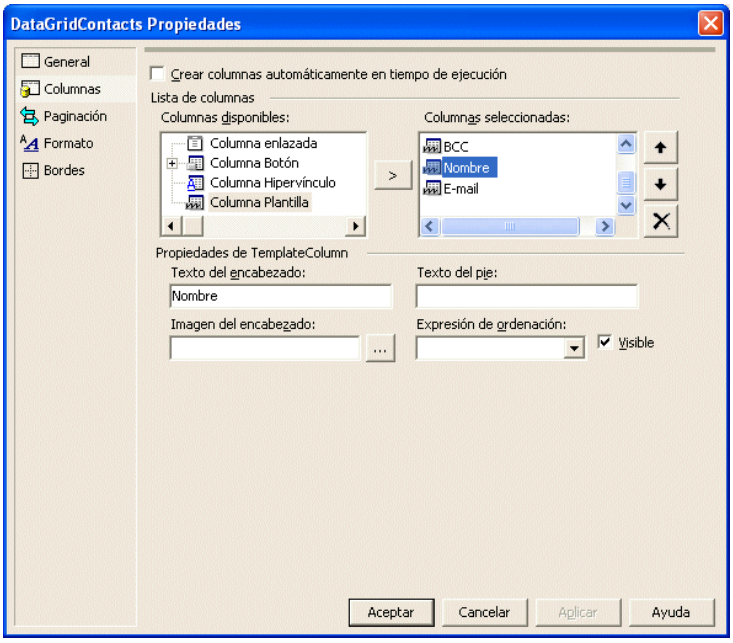
El control asp:DataGrid

Aunque el control `asp:DataList` es bastante versátil, la verdad es que la mayor parte de las veces que queremos mostrar un conjunto de datos, lo que nos interesa es mostrar esos datos en forma de tabla con varias columnas perfectamente alineadas. En ese caso, lo que tenemos que hacer es recurrir al control `asp:DataGrid`.

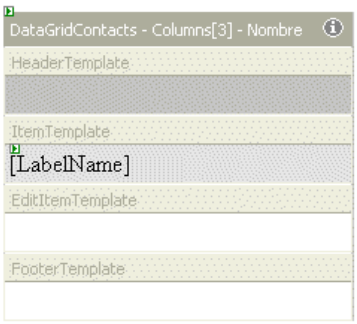
Por ejemplo, supongamos que queremos utilizar nuestra flamante agenda de contactos como parte de un cliente de correo electrónico a través de web, conocido usualmente como *web mail*. En ese caso, lo normal es que tengamos alguna página en nuestra aplicación en la que podamos elegir los destinatarios de un nuevo mensaje. Algo similar a lo mostrado en la siguiente figura:

To	CC	BCC	Nombre	E-mail
☐	☐	☐	DataBound	DataBound
☐	☐	☐	DataBound	DataBound
☐	☐	☐	DataBound	DataBound
☐	☐	☐	DataBound	DataBound
☐	☐	☐	DataBound	DataBound

El control `asp:DataGrid` es bastante más complejo que el control `asp:DataList`, si bien es cierto el entorno de desarrollo nos ayuda mucho para poder diseñar nuestra tabla de forma visual. Para crear las columnas de la tabla, hemos de seleccionar la opción *"Generador de propiedades"* del menú contextual asociado al control. En la ventana que nos aparece podemos definir las columnas que incluirá nuestra tabla como columnas de tipo plantilla:



Una vez que tenemos definidas las columnas de nuestra tabla, podemos editar las plantillas asociadas a ellas mediante la opción *"Editar plantilla"* del menú contextual asociado al `DataGrid`. En el caso de las columnas que contienen una simple etiqueta, el proceso es completamente análogo al que se utilizó en la sección anterior al usar el control `DataList`. Primero, se crea la plantilla en sí:



La propiedad `Text` de etiqueta que aparece en la plantilla la enlazaremos con uno de los valores contenidos en el conjunto de datos de la misma forma que siempre:

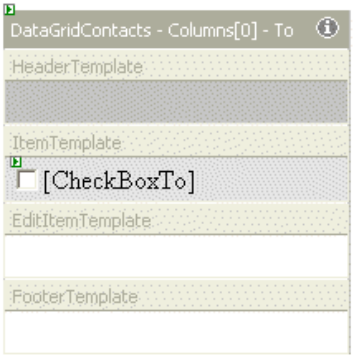
```
DataBinder.Eval(Container.DataItem, "Name")
```

Seleccionando la opción *"Terminar edición de plantilla"* del menú contextual volvemos a la vista general de la tabla para poder editar la plantillas asociadas a las demás columnas. Una vez preparadas las plantillas correspondientes a todas las columnas, lo único que nos falta por hacer es enlazar el control a un conjunto de datos (mediante su propiedad `DataSource`) e invocar al método `DataBind()` para rellenar la tabla con los datos, igual que se hizo con el control `DataList`.

Antes de dar por cerrado este apartado, no obstante, veamos cómo se implementan en ASP.NET dos de las funciones que con más asiduidad se repiten en aplicaciones web que han de trabajar con conjuntos de datos: la posibilidad de seleccionar un subconjunto de los datos y la opción de ordenar los datos de una tabla en función de los valores que aparecen en una columna.

Selección de subconjuntos de datos

Cuando queremos darle la posibilidad al usuario de que seleccione algunos de los elementos de un conjunto de datos mostrado en una tabla, lo normal, cuando usamos aplicaciones web, es que se le añada a la tabla una columna en la que aparezca un control de tipo `CheckBox`:



La plantilla de la columna se crea exactamente igual que cualquier otra plantilla. En tiempo de ejecución, tal como se muestra en la siguiente figura, el control se repetirá para cada una de los elementos del conjunto de datos mostrado en la tabla:

To	CC	BCC	Nombre	E-mail
☐	☐	☐	Bola	bola@mismascotas.com
☐	☐	☐	Mota	mota@mismascotas.com

Cuando queramos saber cuáles son los elementos del conjunto de datos que ha seleccionado el usuario, lo único que tendremos que hacer es acceder secuencialmente a los elementos de tipo `DataGridItem` que componen el `DataGrid` en tiempo de ejecución, tal como se muestra en el siguiente fragmento de código:

```
CheckBox cb;
foreach (DataGridItem dgi in this.DataGridContacts.Items) {
  cb = (CheckBox) dgi.Cells[0].Controls[1];
  if (cb.Checked) {
     // Fila seleccionada
  }
}
```

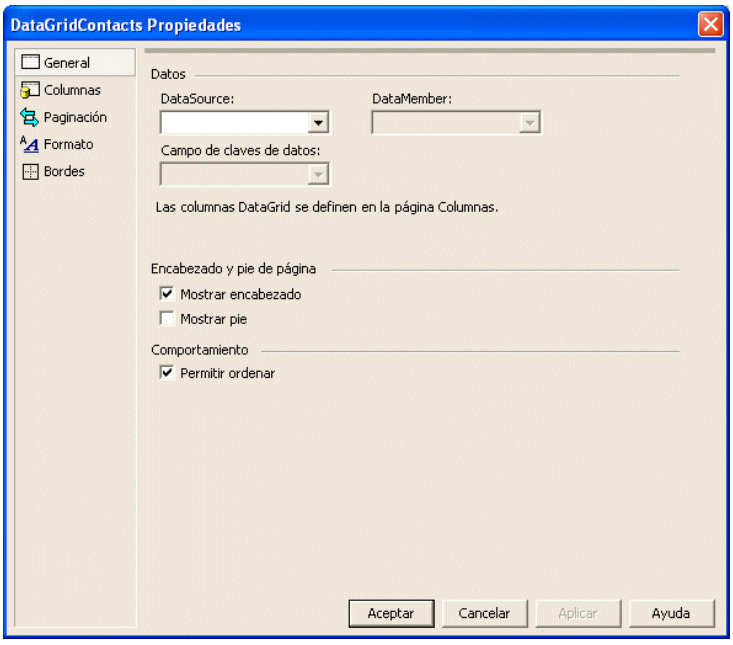
Ordenación por columnas

Otra de las funciones que resultan bastante útiles al trabajar con tablas es la posibilidad de ordenar su contenido en función de los valores que aparezcan en una columna determinada.

Esta función resulta bastante vistosa y su implementación es prácticamente trivial en el caso del control `asp:DataGrid`.

Aunque se puede conseguir el mismo resultado modificando directamente el fichero `.aspx` correspondiente a la página ASP.NET donde esté incluida la tabla y, de hecho, el proceso se puede realizar a mano sin demasiada dificultad, aquí optaremos por aprovechar las facilidades que nos ofrece el Visual Studio .NET a la hora de manipular directamente el control `asp:DataGrid` desde el propio editor de formularios web.

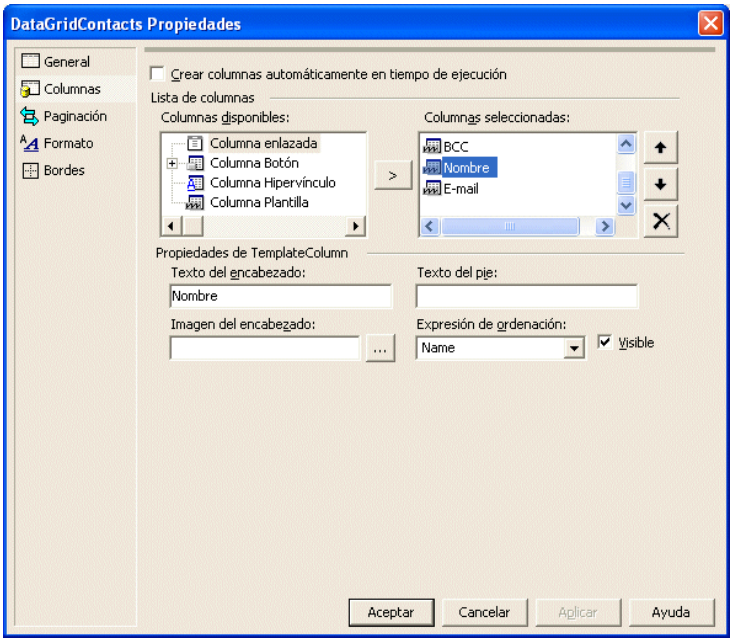
En primer lugar, debemos asegurarnos de que nuestro control tiene habilitada la opción *"Permitir ordenar"* [`AllowSorting`] que aparece en la pestaña *"General"* del generador de propiedades del `DataGrid`:



Esto se traduce, simplemente, en la inclusión del atributo `AllowSorting` en el control `asp:DataGrid` que aparece en el fichero `.aspx`:

```
<asp:DataGrid id="DataGridContacts" runat="server" AllowSorting="True">
```

A continuación, hemos de seleccionar las expresiones de ordenación para las distintas columnas de nuestra tabla, algo que también podemos hacer desde el generador de propiedades del `DataGrid`:



Igual que antes, la selección se traduce en la inclusión de un atributo: `SortExpression`. En este caso, el atributo aparece asociado a las plantillas correspondientes a las distintas columnas de la tabla:

```
<asp:TemplateColumn SortExpression="Name" HeaderText="Nombre">
...
<asp:TemplateColumn SortExpression="EMail" HeaderText="E-mail">
```

En cuanto lo hacemos, automáticamente cambia el aspecto de la cabecera de la tabla. Donde antes aparecían simples etiquetas en las cabeceras de las columnas, ahora se muestran enlaces sobre los que se puede pinchar:

To	CC	BCC	Nombre	E-mail
☐	☐	☐	DataBound	DataBound
☐	☐	☐	DataBound	DataBound
☐	☐	☐	DataBound	DataBound
☐	☐	☐	DataBound	DataBound
☐	☐	☐	DataBound	DataBound

Lo único que nos falta por hacer es implementar el código necesario para realizar la ordenación de la tabla cuando el usuario pincha sobre los enlaces que encabezan las columnas "ordenables" de la tabla. Eso lo haremos como respuesta al evento `SortCommand`:

```
private void DataGridMessages_SortCommand (object source,
    System.Web.UI.WebControls.DataGridSortCommandEventArgs e)
{
  DataTable dt = (DataTable)Session["contactTable"];

  DataView dv = new DataView(dt);
  dv.Sort = e.SortExpression;

  DataGridContacts.DataSource = dv;
  DataGridContacts.DataBind();
}
```

Lo único que tenemos que hacer es crear una vista sobre la tabla de datos y ordenarla. Una vez que tenemos los datos ordenados, enlazamos la vista al control e invocamos al método `DataBind` para que se refresquen los datos de la tabla con los datos ordenados de la vista recién construida.

Paginación en ASP.NET

Cuando los conjuntos de datos son grandes, lo normal es que sólo le mostremos al usuario una parte de los datos y le demos la posibilidad de moverse por el conjunto de datos completo. Es lo que se conoce con el nombre de paginación.

La paginación consiste, básicamente, en no trabajar con más datos de los que resulten estrictamente necesarios. En el caso de las aplicaciones web, esto se traduce en una mejora en la escalabilidad de las aplicaciones al no tener que transmitir más que aquellos datos que los que se muestran en una página al usuario.

Como no podía ser de otra forma, el control `DataGrid` facilita la posibilidad de usar paginación (sin más que poner la propiedad `AllowPaging` a `true`). De hecho, podemos especificar el número de datos que queremos mostrar por página (`PageSize`), así como dónde y cómo tienen que aparecer los botones que se usarán para navegar por el conjunto de datos (`PagerStyle`). Cuando se cambia de página, se produce el evento `OnPageIndexChanged`, que es el que debemos interceptar para mostrar los datos adecuados.

Formularios de manipulación de datos

Aparte de mostrar datos, las aplicaciones también tienen que manipularlos: realizar inserciones, actualizaciones y borrados. Por tanto, a la hora de implementar la interfaz de usuario, una de las tareas más habituales con las que ha de enfrentarse un programador es la creación de formularios de manipulación de datos, también conocidos como *formularios CRUD* porque esas son las iniciales en inglés de las cuatro operaciones básicas que realizan (creación, lectura, actualización y borrado de datos). Dada su importancia práctica, este es el tema con el que terminaremos nuestro recorrido por ASP.NET.

Usualmente, los formularios de manipulación de datos se construyen en torno a tres elementos:

- Un objeto que representa a una instancia de la entidad que el formulario manipula.
- Un mecanismo que permita mostrar el estado del objeto.
- Un mecanismo que permita modificar el estado del objeto.

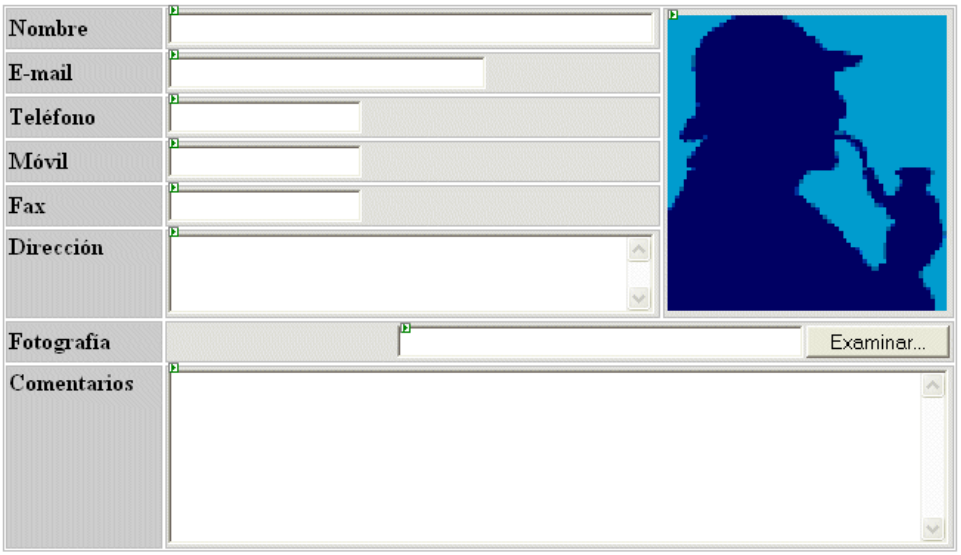
Si nos fijamos una vez más en nuestro ejemplo de la agenda de contactos, podemos diferenciar fácilmente los componentes que desempeñan los tres roles que acabamos de enumerar. Los objetos de tipo `Contact` encapsulan las entidades con las que trabaja nuestra aplicación. Por su parte, el control `ContactViewer`, que creamos por composición, proporciona el mecanismo adecuado para mostrar los datos encapsulados por un objeto de tipo `Contact`. Las operaciones de manipulación de los datos de un contacto (edición e inserción) requerirán la inclusión en nuestro formulario de nuevos mecanismos, tal como se describe en el apartado siguiente.

> La operación que nos queda por cubrir, el borrado, resulta trivial en la práctica, pues nos basta con añadir al formulario un botón con el que borrar el contacto que se esté visualizando en ese momento.

Edición de datos

Si, en nuestro ejemplo de la agenda, queremos dotar al formulario de la posibilidad de añadir o modificar contactos, lo único que tenemos que hacer es crear un nuevo control que permita modificar el estado de un objeto de tipo `Contact`. Lo primero haremos será crear un control análogo a `ContactViewer`. A este control lo denominaremos `ContactEditor`.

No obstante, como lo que tenemos que hacer es permitir que el usuario pueda modificar los datos de un contacto, en el control `ContactEditor` incluiremos componentes adecuados para esta tarea. En este caso particular, podemos usar controles de tipo `TextBox` para los distintos datos de contacto de una persona y un control de tipo `HtmlInputFile` para poder cargar las imágenes correspondientes a las fotos.



El control ContactEditor permite modificar los datos de un contacto.

Igual que hicimos con el control `ContactViewer`, a nuestro control le indicaremos el contacto con el que trabajar mediante una propiedad:

```
public Contact DisplayedContact
{
  get { return UpdatedContact(); }
  set { ViewState["contact"] = value; UpdateUI(); }
}
```

La actualización de la interfaz de usuario para mostrar los datos de un contacto determinado es trivial:

```
private void UpdateUI ()
{
  Contact contact = (Contact) ViewState["contact"];
```

```
    if (contact!=null) {
       TextBoxName.Text = contact.Name;
       TextBoxEMail.Text = contact.EMail;
       TextBoxTelephone.Text = contact.Telephone;
       TextBoxMobile.Text = contact.Mobile;
       TextBoxFax.Text = contact.Fax;
       TextBoxAddress.Text = contact.Address;
       TextBoxComments.Text = contact.Comments;
       ImagePhoto.ImageUrl = "contacts/"+contact.ImageURL;
    }
  }
```

Del mismo modo, la obtención del estado actual del contacto resulta bastante fácil. Sólo tenemos que añadir algo de código para cargar la imagen correspondiente a la foto si el usuario ha seleccionado un fichero:

```
  private Contact UpdatedContact ()
  {
    Contact contact = (Contact) ViewState["contact"];

    if (contact!=null) {
       contact.Name = TextBoxName.Text;
       contact.EMail = TextBoxEMail.Text;
       contact.Telephone = TextBoxTelephone.Text;
       contact.Mobile = TextBoxMobile.Text;
       contact.Fax = TextBoxFax.Text;
       contact.Address = TextBoxAddress.Text;
       contact.Comments = TextBoxComments.Text;

       // Fichero
       if (  (FilePhoto.PostedFile != null)
          && (FilePhoto.PostedFile.ContentLength>0) ) {

          string filename  = contact.Name;
          string extension = "";

          if (FilePhoto.PostedFile.ContentType.IndexOf("jpeg")>-1)
              extension = "jpg";
          else if (FilePhoto.PostedFile.ContentType.IndexOf("gif")>-1)
              extension = "gif";
          else if (FilePhoto.PostedFile.ContentType.IndexOf("png")>-1)
              extension = "png";

          string path = Server.MapPath("contacts/image")
                      + "/"+filename+"."+extension;
          try {
              FilePhoto.PostedFile.SaveAs(path);
              contact.ImageURL = "image/"+filename+"."+extension;
          } catch {
          }
       }
    }

    return contact;
  }
```

Recordemos que, para almacenar el estado de un control, hemos de utilizar la colección `ViewState` asociada a la página ASP.NET como consecuencia de la naturaleza de la interfaz web, donde cada acceso a la página se realiza con una conexión TCP independiente. Además, para poder almacenar los datos del contacto en `ViewState`, la clase `Contact` debe estar marcada como serializable con el atributo `[Serializable]`.

El ejemplo anterior muestra cómo se pueden construir interfaces modulares en ASP.NET para la manipulación de datos. No obstante, también podríamos haber optado por organizar nuestra interfaz de una forma diferente. En lo que respecta a la forma de acceder a los datos, la capa de presentación de una aplicación (la parte de la aplicación que trata directamente con la interfaz de usuario) se puede organizar atendiendo a tres estilos bien diferenciados:

- Como hemos hecho en el ejemplo, podemos acceder directamente al **modelo de clases** que representa los datos con los que trabaja la aplicación. Éste es el método tradicional de implementación de una aplicación basada en el uso de técnicas de diseño orientado a objetos.

- Por otro lado, también podríamos utilizar algún tipo de **paso de mensajes** que independice la interfaz del resto de la aplicación, como sucedía en el caso de los controladores de aplicación. Con esta estrategia, se puede aislar por completo la capa de presentación del resto de la aplicación (incluida la lógica que gobierna la ejecución de casos de uso por parte del usuario).

- Finalmente, se puede optar por emplear directamente **componentes de acceso a los datos**, ejemplificados en la plataforma .NET por las clases `DataSet` y `DataReader`. Esta última opción, la más cómoda cuando se utiliza un entorno de programación visual, es también la peor desde el punto de vista del diseño, pues las distintas partes de la aplicación quedan fuertemente acopladas entre sí a través de los conjuntos de datos que se transmiten de una parte a otra de la aplicación. De hecho, una consecuencia directa de este enfoque es que la interfaz de usuario queda ligada inexorablemente al diseño físico de nuestros conjuntos de datos. Es más, si se permite libremente la modificación de los conjuntos de datos, al encontrarnos un error difícilmente seremos capaces de localizar su origen. No nos quedará más remedio que buscar por todas las partes de la aplicación por donde haya podido pasar el conjunto de datos.

Independientemente de la estrategia que utilicemos internamente para acceder a los datos, de cara al usuario, la presentación de la aplicación será similar. Por ejemplo, nuestra aplicación siempre incluirá formularios en los que aparezcan conjuntos de datos (para los que podemos usar controles como `DataList` o `DataGrid`). Cuando el usuario quiera modificar alguno de los datos que esté viendo, podemos usar un formulario independiente para editar los datos

(con un control del estilo de `ContactEditor`) o, incluso, podemos permitirle al usuario que modifique los datos sobre el mismo formulario (usando, por ejemplo, las plantillas `EditItemTemplate` disponibles en los controles `DataList` y `DataGrid`). La funcionalidad de la aplicación será la misma en ambos casos y la elección de una u otra alternativa dependerá de la situación. Para tomar la decisión deberemos considerar factores como la facilidad de uso de la aplicación, su mantenibilidad, el esfuerzo de desarrollo requerido o, simplemente, la consistencia que siempre hemos de procurar entre las distintas partes de una aplicación.

Formularios maestro-detalle en ASP.NET

Otro de los temas recurrentes en la construcción de interfaces de usuario es la creación de formularios maestro-detalle. Este tipo de formularios consiste, básicamente, en mostrar los datos relativos a un objeto y, debajo, representar un conjunto de datos relacionados directamente con ese objeto. Un ejemplo típico es el formulario que se nos presenta en una aplicación de comercio electrónico cuando vamos a comprar algo. Junto con los datos generales de nuestro pedido (dirección de envío, forma de pago, etcétera) aparece el conjunto de artículos que deseamos comprar. Obviamente, en un formulario de este tipo, sólo deben aparecer los artículos incluidos en nuestro pedido (y no los artículos que otros clientes hayan podido encargar).

En la práctica, lo único que tenemos que hacer es combinar en una misma página los elementos que ya hemos aprendido a utilizar por separado. A continuación se mencionan brevemente algunas de las decisiones de diseño que conscientemente deberíamos tomar para crear un formulario maestro-detalle:

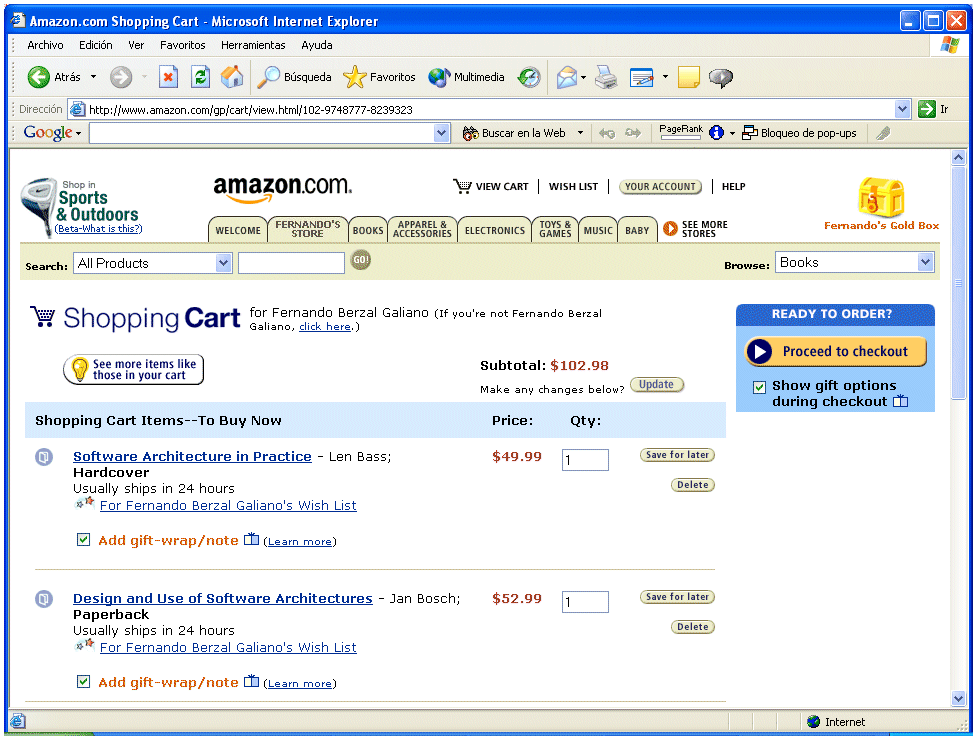
- En primer lugar, podemos crear un par de controles para, respectivamente, visualizar y editar los datos correspondientes al maestro, siguiendo exactamente los mismos pasos con los que construimos los controles `ContactViewer` y `ContactEditor`. Este par de controles, que se puede servirnos para crear formularios simples, podemos reutilizarlo para construir la parte del formulario correspondiente al maestro. En el caso de un formulario para la realización de un pedido, en el maestro aparecerían datos como el nombre del cliente, la dirección de envío, la forma de pago o el importe total del pedido.

- Respecto a los detalles, lo normal es que utilicemos un único control `DataList` o `DataGrid` para visualizar simultáneamente todos los datos relacionados con la entidad principal del formulario. Durante la realización de un pedido, esto nos permitiría ver de golpe todos los artículos incluidos en el pedido.

- Si nuestro maestro-detalle requiere mostrar muchos datos, deberíamos utilizar el mecanismo de paginación facilitado por el control `DataGrid` para mejorar la eficiencia de nuestra aplicación. Este mecanismo podría llegar a ser necesario, por ejemplo, si quisiéramos mostrar en un formulario todos los clientes que han comprado un artículo concreto o el historial de pedidos de un cliente habitual.

- Dado que los detalles mostrados en un formulario de este tipo "pertenecen" a la entidad que aparece como maestro, la modificación de los datos correspondientes a los detalles debería realizarse directamente sobre el control `DataGrid` en el que se visualizan. De ese modo, los artículos del pedido se modifican sobre la misma tabla de detalles del pedido y el usuario no pierde de vista el contexto en el que se realiza la operación.

- Por último, debemos asegurarnos de que el enlace de datos se realiza correctamente: siempre que el usuario cambia de maestro, los datos que se muestran como detalle deben corresponder al maestro actual. Esto es, si pasamos de un pedido a otro, los artículos mostrados han de corresponder al pedido que se muestre en ese momento. En otras palabras, siempre que actualicemos los datos visualizados en el maestro con `DataBind()`, hemos de asegurarnos de actualizar correctamente los datos correspondientes al detalle, algo imprescindible para que las distintas vistas del formulario se mantengan sincronizadas.



El formulario maestro-detalle correspondiente a un pedido en Amazon.com, un conocido portal de comercio electrónico.

Comentarios finales

Pese a que, como programadores, siempre nos tienta dedicarnos exclusivamente al diseño interno de nuestra aplicación, nunca podemos olvidar el siguiente hecho: **"Para el usuario, la interfaz es la aplicación"** (Constantine & Lockwood, *"Software for use"*, 1999). El aspecto visual del sistema y su facilidad de uso son los factores que más decisivamente influyen en la percepción que los usuarios tienen de los sistemas que construimos para ellos.

Si bien nunca debemos perder de vista este hecho, tampoco podemos restarle un ápice de importancia al correcto diseño de nuestras aplicaciones. Al fin y al cabo, un diseño correcto se suele traducir en una aplicación fácil de usar para el usuario y fácil de mantener para el programador. Y ambas son características muy deseables para los sistemas que nosotros construyamos.

En las siguientes referencias, se puede encontrar información adicional relativa a la construcción de la capa de presentación de una aplicación, desde patrones de diseño generales como el modelo MVC hasta consejos concretos acerca del diseño modular de interfaces web con ASP.NET:

- Martin Fowler: *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2002, ISBN 0321127420

- David Trowbridge, Dave Mancini, Dave Quick, Gregor Hohpe, James Newkirk, David Lavigne: *Enterprise solution patterns using Microsoft .NET*, Microsoft Press, 2003, ISBN 0735618399

- *Application architecture for .NET: Designing applications and services*, Microsoft Press, 2003, ISBN 0735618372

- *Design and Implementation Guidelines for Web Clients*, Microsoft Corporation, 2003

Aparte de las referencias bibliográficas citadas, se puede encontrar casi de todo en distintas web y grupos de noticias de Internet. Por ejemplo, La página oficial de ASP.NET (http://asp.net/) puede ser un buen punto de partida para profundizar en el dominio de esta tecnología de desarrollo de interfaces web.