



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 aplicación 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.....	125
Componentes de la interfaz	125
El modelo MVC en ASP.NET	127
Controladores en ASP.NET.....	129
Control de la aplicación.....	135
Aspecto visual de la aplicación	140
Filtros con módulos HTTP	142
La directiva #include	143
Plantillas.....	145
Configuración dinámica de la aplicación.....	147
Listas de opciones	149
Vectores simples: Array y ArrayList.....	150
Pares clave-valor: Hashtable y SortedList.....	151
Ficheros XML	153
Conjuntos de datos	157
El control asp:Repeater	158
El control asp:DataList.....	161
El control asp:DataGrid	163
Formularios de manipulación de datos.....	170
Edición de datos	170
Formularios maestro-detalle en ASP.NET.....	174

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

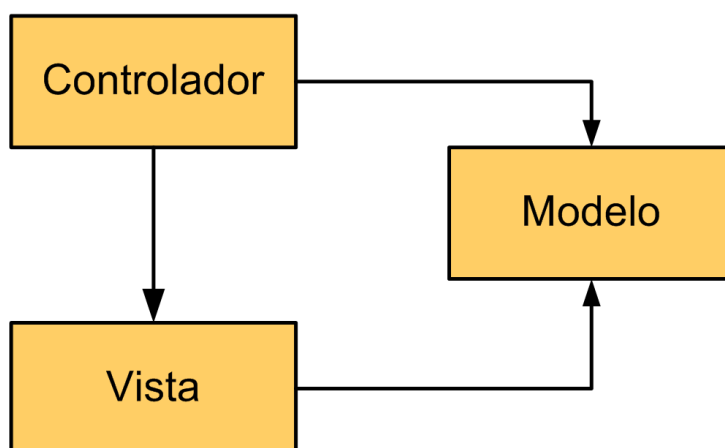
La primera tentación de un programador de aplicaciones web consiste en escribir todo el código de su aplicación en la propia página ASP.NET. Sin embargo, en cuanto la aplicación comienza a crecer, empiezan a aparecer cierta redundancia. La existencia de fragmentos de código duplicados complicarán el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

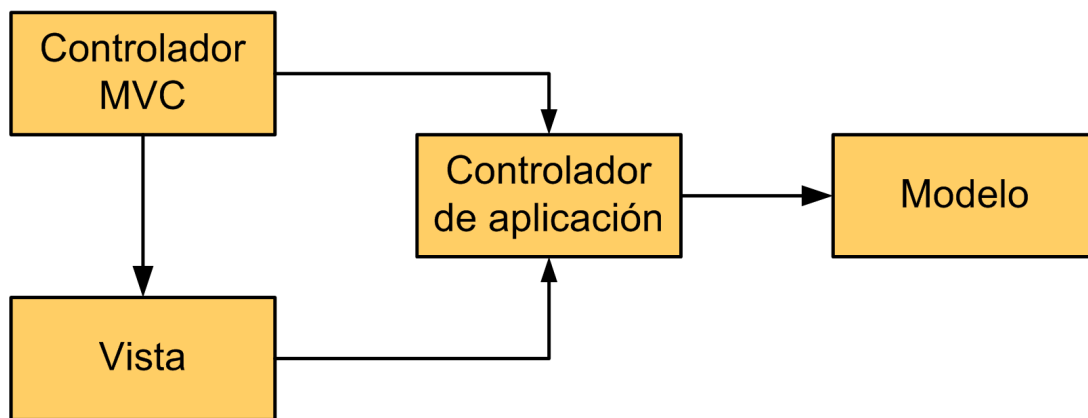
Si bien el recorrido del usuario a través de la aplicación lo determina el controlador de aplicación, las páginas concretas por las que pase el usuario no tienen por qué estar prefijadas en el código del controlador. Lo habitual es almacenar las transiciones entre páginas concretas en un fichero de configuración 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:

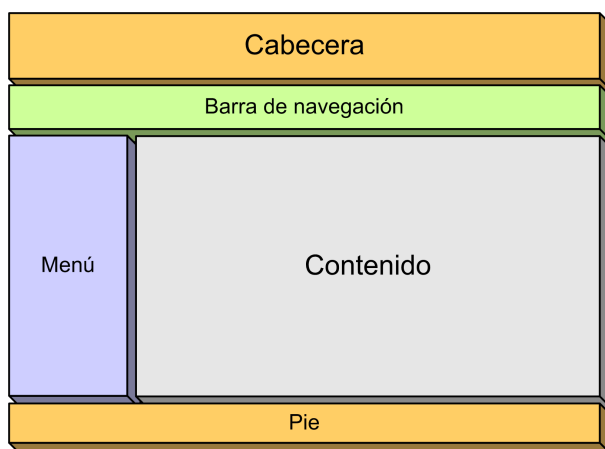
- La **creación de vistas**: Un controlador concreto podría estar asociado a distintas vistas en función de la situación (por ejemplo, cuando se utiliza el mismo controlador para una aplicación web y una aplicación Windows). Lo normal será crear una clase abstracta con varias subclases concretas para 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 página 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 EndRequest (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.

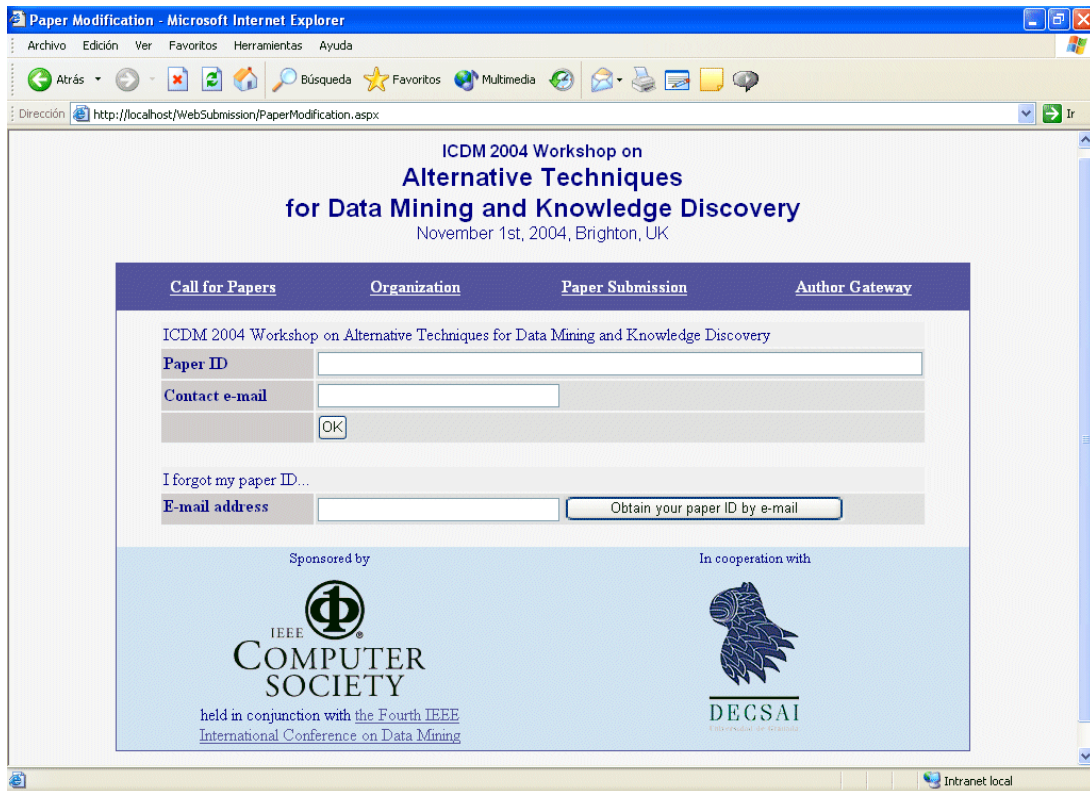
Afortunadamente, ASP.NET nos ofrece un mecanismo más sencillo que nos permite conseguir el mismo resultado sin interferir el en desarrollo de las distintas páginas de una aplicación. Nos referimos a la posibilidad de incluir en la página `.aspx` un comentario con la directiva `#include` al más puro estilo 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. Debemos 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 `DataRowView` 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

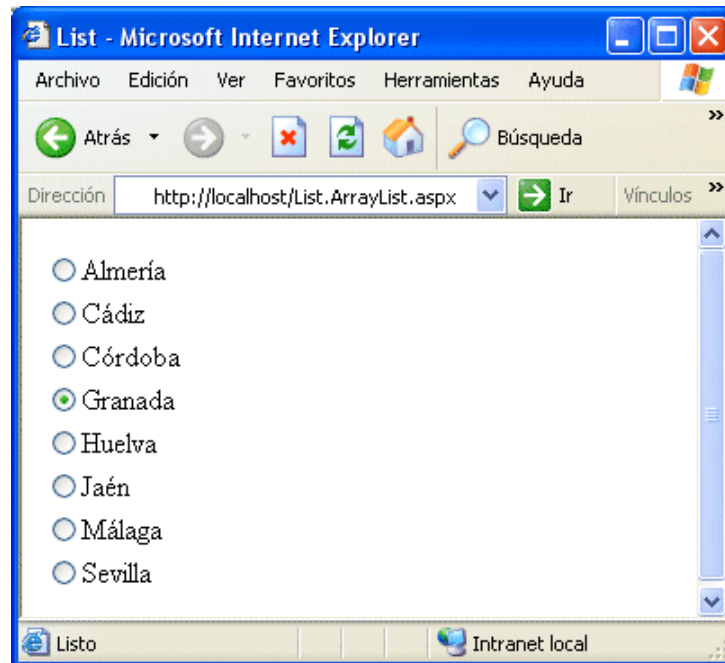
La clase `System.Array` es la clase que representa un vector en la plataforma .NET y sirve como base para todas las matrices en C#. Por su parte, la clase `System.Collections.ArrayList` nos permite crear vectores de tamaño dinámico. Ambas clases implementan el mismo conjunto de interfaces (`ICollection`, `ICollection`, `ICollection`, `ICollection`) 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 hiciera que nuestra aplicación dejara 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 estructura 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 provincia 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.

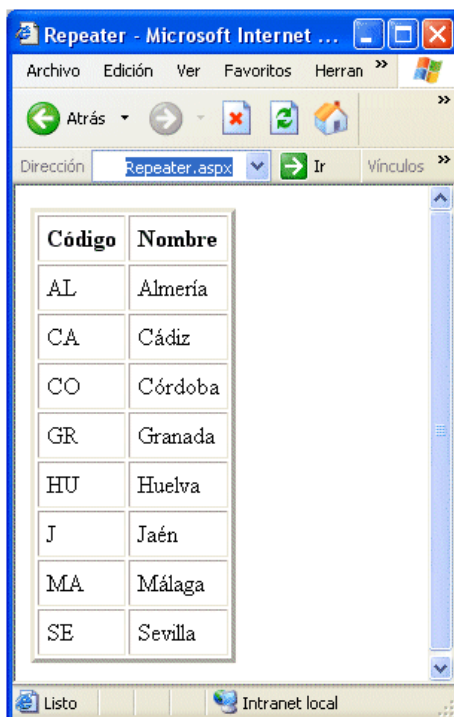
Veamos, en la práctica, cómo funciona el control `asp:Repeater` con el ejemplo que pusimos en la sección anterior. En primer lugar, tenemos que añadir un control `asp:Repeater` al formulario web con el que estemos trabajando. A continuación, podemos obtener el conjunto de datos desde el fichero XML utilizado 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;
...

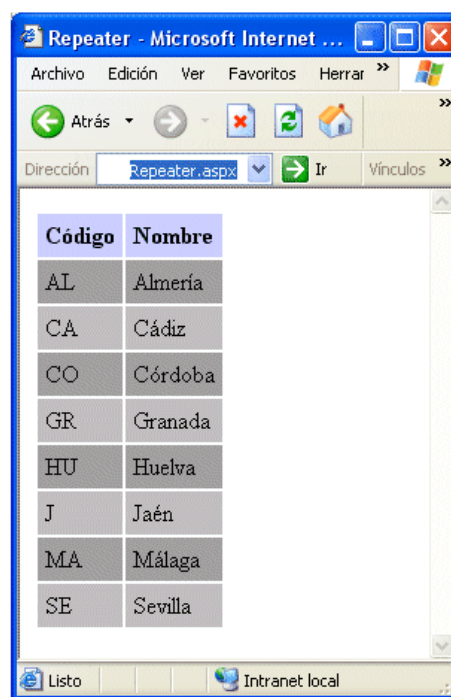
private void Page_Load(object sender, System.EventArgs e)
{
    if (!Page.IsPostBack) {
        DataSet dataset = new DataSet();
        dataset.ReadXml(MapPath("provincias.xml"));
        provincia.DataSource = dataset;
        provincia.DataBind();
    }
}
```

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>
    </table>
  </HeaderTemplate>
  <ItemTemplate>
    <tr>
      <td>
        <%=# DataBinder.Eval(Container.DataItem, "id") %>
      </td>
      <td>
        <%=# DataBinder.Eval(Container.DataItem, "nombre") %>
      </td>
    </tr>
  </ItemTemplate>
  <FooterTemplate>
  </FooterTemplate>
</asp:Repeater>
```



Código	Nombre
AL	Almería
CA	Cádiz
CO	Córdoba
GR	Granada
HU	Huelva
J	Jaén
MA	Málaga
SE	Sevilla



Código	Nombre
AL	Almería
CA	Cádiz
CO	Córdoba
GR	Granada
HU	Huelva
J	Jaén
MA	Málaga
SE	Sevilla

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

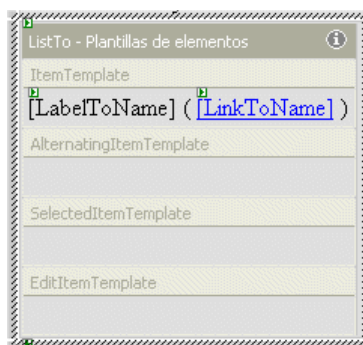
otros controles, que se pueden emplear para visualizar e incluso modificar conjuntos de datos. En concreto, nos referimos a los controles `System.Web.UI.WebControls.DataList` y `System.Web.UI.WebControls.DataGrid` que usaremos en los dos próximos apartados.

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.

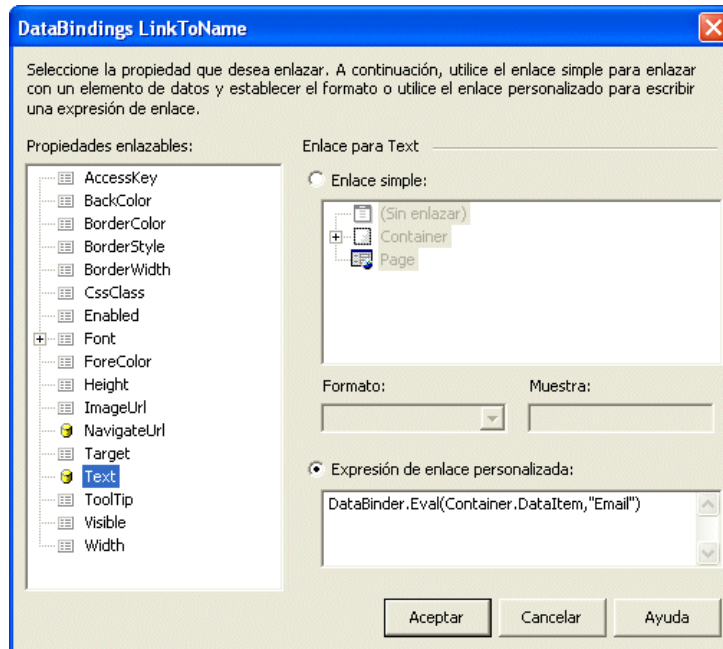
Volviendo al ejemplo de la lista de contactos, supongamos que deseamos mostrar, simultáneamente, datos relativos a varios de nuestros contactos, sin tener que ir eligiéndolos uno a uno. Podemos usar un control de tipo `asp:DataList` y definir visualmente el formato de su plantilla `ItemTemplate`:



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();
```

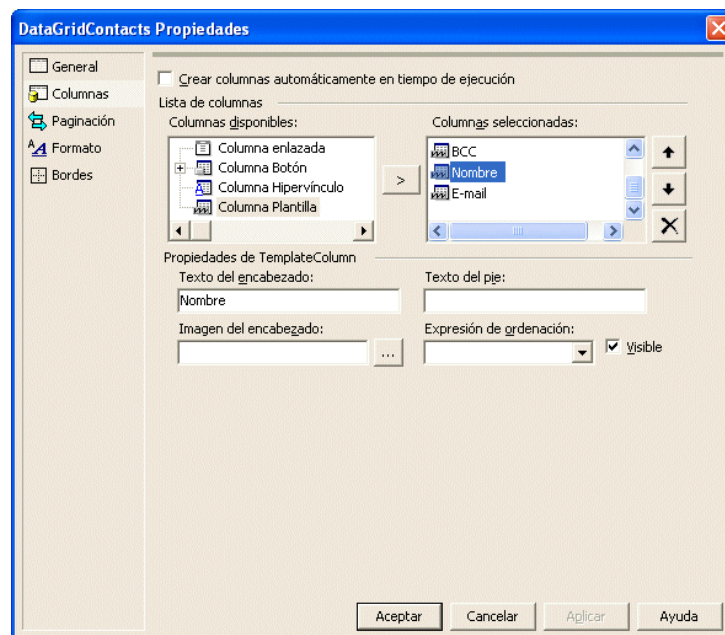
El control `asp:DataGrid`

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`.

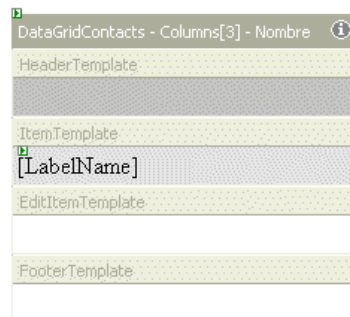
Por ejemplo, supongamos que queremos utilizar nuestra flamante agenda de contactos como parte de un cliente de correo electrónico a través de web, conocido usualmente como *web mail*. En ese caso, lo normal es que tengamos alguna página en nuestra aplicación en la que podamos elegir los destinatarios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

To	CC	BCC	Nombre	E-mail
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	DataBound	DataBound
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	DataBound	DataBound
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	DataBound	DataBound
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	DataBound	DataBound
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	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:

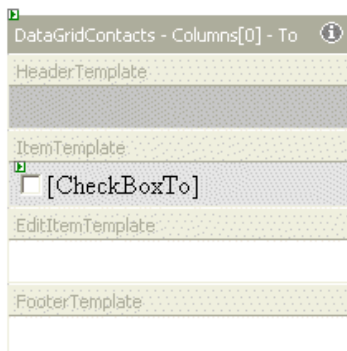
```
DataBinder.Eval(Container.DataItem, "Name")
```

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
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Bola	bola@mismascotas.com
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	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
    }
}

```

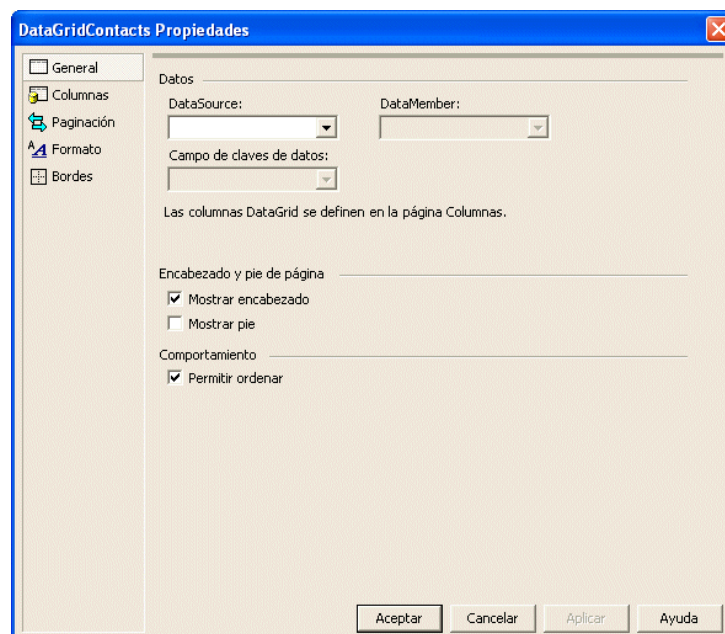
Ordenación por columnas

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.

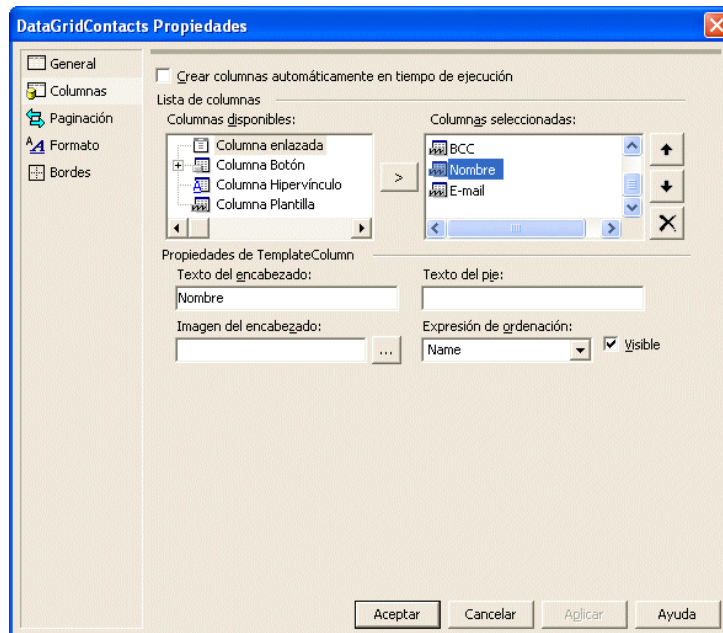
En primer lugar, debemos asegurarnos de que nuestro control tiene habilitada la opción "Permitir ordenar" [AllowSorting] que aparece en la pestaña "General" del generador de propiedades del DataGrid:



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
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	DataBound	DataBound
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	DataBound	DataBound
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	DataBound	DataBound
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	DataBound	DataBound
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	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.

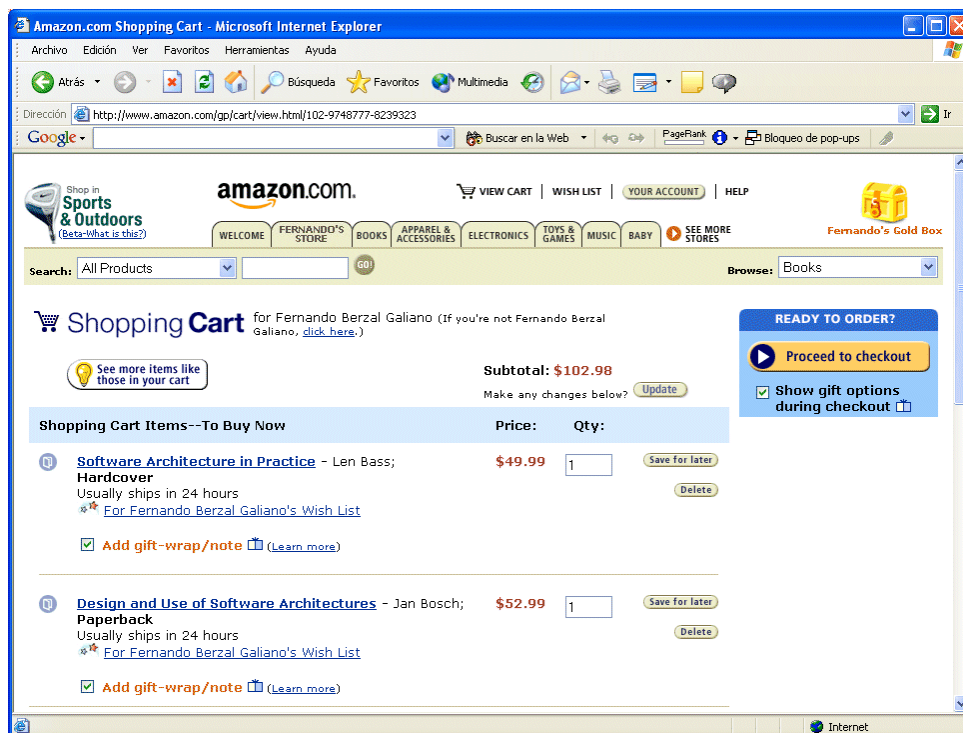
Formularios maestro-detalle en ASP.NET

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 encargarse).

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.