Exam Manual

LearnSmart

Microsoft .NET 2.0 Framework (70-528)

Web-based Client Development

This LearnSmart exam manual will help prepare ambitious IT professionals for the Microsoft .NET 2.0 Web Applications exam (70-528). By studying this guide, you will become familiar with a plethora of exam-related material, including:

- Creating and Programming a Web Application
- Creating Custom Web Controls
- Tracing, Configuring and Deploying Applications
- And more!

Give yourself the competitive edge necessary to further your career as an IT professional and purchase this exam manual today!

Microsoft .Net Framework 2.0 Web-based Client Development (70-528) LearnSmart Exam Manual

Product ID: 010864
Production Date: July 22, 2011

Warning and Disclaimer

Every effort has been made to make this document as complete and as accurate as possible, but no warranty or fitness is implied. The publisher and authors assume no responsibility for errors or omissions. The information provided is on an "as is" basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this document.

Volume, Corporate, and Educational Sales

PrepLogic offers favorable discounts on all products when ordered in quantity. For more information, please contact PrepLogic directly:

1-800-418-6789
solutions@learnsmartsystems.com

International Contact Information

International: +1 (813) 769-0920

United Kingdom: (0) 20 8816 8036

Table of Contents

Abstract

This Exam Manual is two fold in its purpose. First, it is designed to prepare you for the 70-528 exam. But, in addition to being an exam preparation tool, this manual is designed to be an introductory text to transition individuals familiar with C# and the .NET framework to the new .NET Framework 2.0. This is done by reviewing the new elements of the .NET Framework 2.0 on a very high level. The manual sticks to the basics and the most important information, giving you a quick glance at the material in the hope that you can understand the most important features and extrapolate some of the lesser important details through your own intuition.

What to Know

There's no hiding the fact that both Microsoft exams and the .NET Framework are difficult. If you're reading this Exam Manual purely to prepare for the exam, know that the exam will not just ask you questions regarding specific aspects of the .NET Framework, but it will present detailed questions in coded format and ask you to derive ideas from material that is new to the .NET Framework 2.0 that you may have never used in the past. Without question, you should make certain before you sit for this exam that you spend at LEAST several hours (preferably over the course of several weeks) coding the .NET Framework 2.0 using a free tool, such as Visual Studio Express. Additionally, you should make sure that you memorize as many of the new types, classes, and changes to the framework as you possibly can. Murphy's Law does apply and it's most likely that the one thing that you do not commit to memory will be the one thing they WILL test on. Be careful and be prepared.

Tips

Just like math, there's no substitution for programming with a hands on approach. Before you consider yourself an expert in the .NET, make sure that there's no concept you can't program your way out of. Furthermore, you should guard yourself and make sure that you don't just get stuck in doing things "your way," but that you understand the .NET Framework's methodology for approaching problems, solving difficult memory situations, and handling exceptions. It won't just make you a better programmer; it will help your score!

Create and Configure a Web Application

Create a New Web Application

In .NET Framework 2.0, web applications are created using the Web site project type. This is a new way of handling web applications and replaces the .NET Framework 1.1 Web Application.

To create a new Web site in Visual Studio 2005 you can select **File > New Web Site**. To add a Web site to an existing solution in Visual Studio 2005, select the Solution in Solution Explorer and choose **Add > New Web Site** from the context menu.

This launches the **Add New Web Site** dialog, shown in **Figure 1**, which allows you to select the language for your site (C#, VB.NET or J#) and how you will access your Web site. For the purposes of this manual we're going to assume that you're using C#.

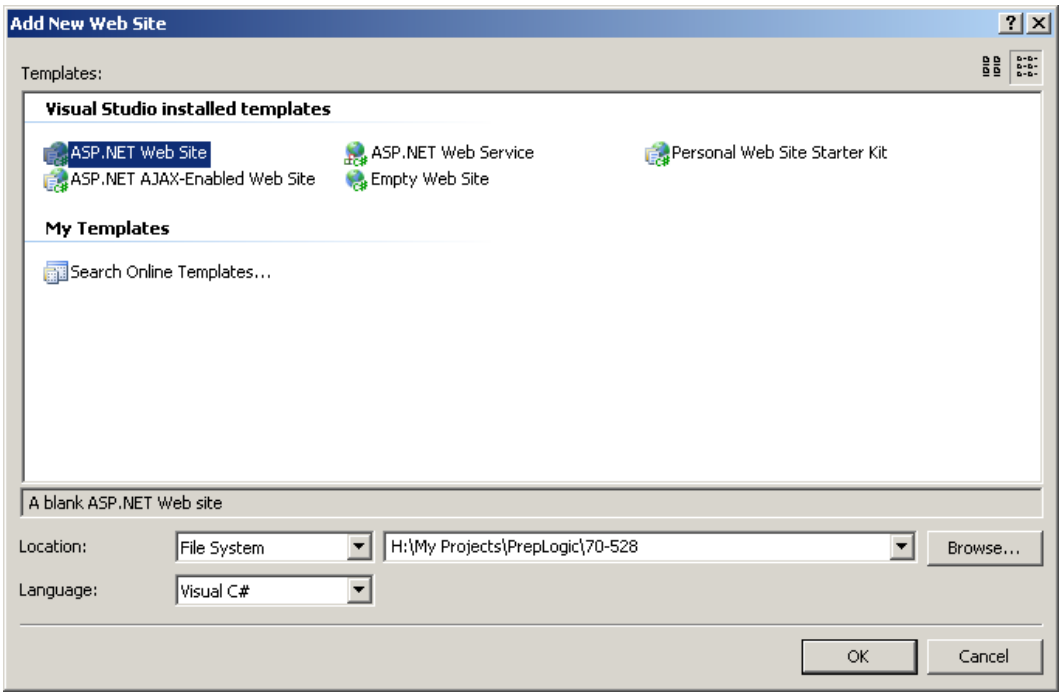

Figure 1 – The *Add New Web Site* dialog.

Depending on your project requirements, you may select how you will access your Web site from the options listed below:

Location	Description
File System	A File System based Web site accesses the file system of your computer directly. When debugging or viewing a File System-based Web site, the lightweight, built-in Web server provided as part of Visual Studio 2005 is used.
HTTP	An HTTP based Web site can access IIS running either locally or remotely. If you enter a remote URL for your Web site, the remote computer must be running Front Page Server Extensions. When debugging or viewing an HTTP based Web site, the IIS server is used.
FTP	An FTP based Web site allows you to connect to a remote server. A second dialog allows you to enter your credentials for connection to the remote server. When using an FTP based Web site, all files are copied locally and any changes made to the files are made to the local copies. Visual Studio automatically propagates any changes to the remove server. When debugging or viewing an FTP based Web site, the locally cached files are used by the lightweight, build-in Web Server provided as part of Visual Studio 2005.

Code-Beside and Inline Programming

When a new Web site is created it will automatically create a Web Form (Default.aspx) and display it in the HTML view within the design window. If you look at the Web Form in Solution Explorer you'll see, as shown in **Figure 2**, that there are two files associated with it.

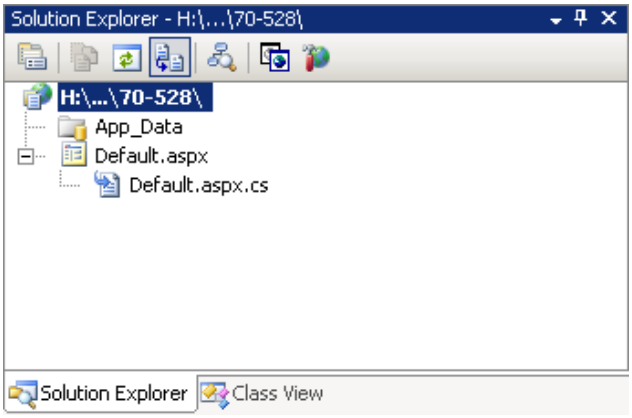

Figure 2 – Solution Explorer showing two files for the Web Form.

This Web Form is an example of "code-beside," and is comparable (albeit a lot simpler) to the "code-behind" model from the .NET Framework 1.1. This isn't the only model available, as shown by the options in the following table:

Name	Description
Code-beside	A separate code file is created for each Web Form with the same name as the Web Form, but with a CS extension (e.g. Default.aspx.cs). Both files must be deployed in order for the Web Form to be viewable.
Inline	All code is contained within a single file (e.g. Default.aspx) in a *<script runat="server">* tag. There is only one file to deploy in order for the Web Form to be viewable.

Code-beside is the preferred model of development as it provides a clean separation between what is code (in the CS file) and what is HTML markup (in the ASPX file).

In addition to Web Forms, User Controls and Master Pages also support both the code-beside and inline models.

Web Site Structure

There are several special folders that make up a Web site. If you look at Figure 2 you'll see that there is an *App_Data* folder created by default. Seven other, special folders are defined in the table below:

Folder	Description
App_Browsers	Contains browser definition files that are used to identify browsers and their capabilities.
App_Code	Contains source code files for classes that are compiled as part of the Web site.
App_Data	Contains any data files for the Web site (including SQL Server Express 2005 databases).
App_GlobalResources	Contains resource files that are global in scope.
App_LocalResources	Contains resource files that are specific to Web Forms, User Controls or Master Pages.
App_Themes	Contains files defining the appearance of Web Forms or Web Controls for the web site.
App_WebReferences	Contains any files that are needed to define references to Web Services.

Dynamic Compilation

The Web site model no longer builds a single assembly containing all of the compiled Web Forms and User Controls for the Web site. Each Web Form or User Control is compiled when it is requested and automatically recompiled as required.

The only exception to this is the *App_Code* special folder, which is compiled into a single assembly available to all Web Forms and User Controls within the Web site.

Add Web Forms to a Web Application

A Web Form can be added to a Web site in two ways:

- By selecting **Website > Add New Item** from the Visual Studio menu.
- By selecting **Add New Item** from the Web site context menu in the Solution Explorer.

Each of these options launches the **Add New Item** dialog, as shown in **Figure 3**.

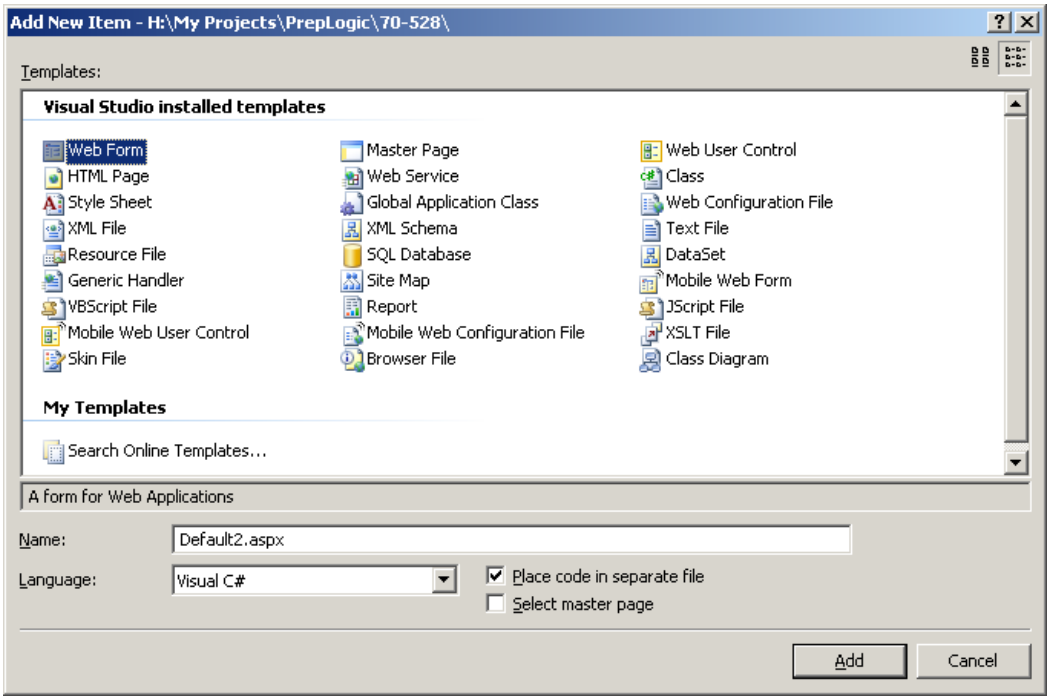

Figure 3 – Adding a Web Form using the *Add New Item* Dialog

From this dialog you can select what you want to create (in this case a Web Form) and specify its name and language. You can also specify whether you want to use the code-beside (by checking the *Place code in separate file* option) or inline code model.

Add and Configure Web Server Controls

Web Server Controls

Web Server Controls are fully programmable and configurable controls that have the ability to detect the browser's capabilities and change their rendering automatically. They are programmable by writing server-side code to respond to events from the client.

A Web Server Control may render itself as a single HTML tag (such as the *Button* or *Label* controls), or as several different HTML tags (such as the *GridView* or *Calendar* controls).

You can tell immediately if a control is a Web Server Control by examining the HTML markup within a Web Form or User Control for tag names prefixed with asp, e.g. <*asp:Button* />.

Creating Web Server Controls

Web Server Controls are added from the Toolbox (**View >Toolbox** or **CTRL+ALT+X**) and are shown in the Standard, Data, Validation, Navigation, Login and WebParts tabs.

They can be added to either the Design or Source view by double-clicking the control or dragging the control from the toolbox onto the Web Form, User Control or Master Page.

You can also add Web Server Controls to a Web Form, User Control or Master Page manually by writing the HTML markup for the control.

Configure the Properties of Web Server Controls

Properties of Web Server Controls can be configured three different ways:

- By selecting the Web Server Control in Design view and changing the properties in the Properties pane (View > Properties Window or F4).
- By selecting the Web Server Control in Source view and adding the properties as attributes to the HTML markup for the control.
- Programmatically, by referencing the control by name and setting the property's value.

Handling Events

Creating Event Handlers

Event handlers can be created in two ways:

- By selecting the Web Server Control in Source view and adding an attribute to the Web Server Control. The attribute name is the event required; its value is the method to call when the event is raised. You must manually write the event handler signature.
- By selecting the control in Design view and viewing the Events in the Properties window (shown in *Figure 4*). Double clicking in the listbox to the right of the event in question will automatically write the event handler signature for you.

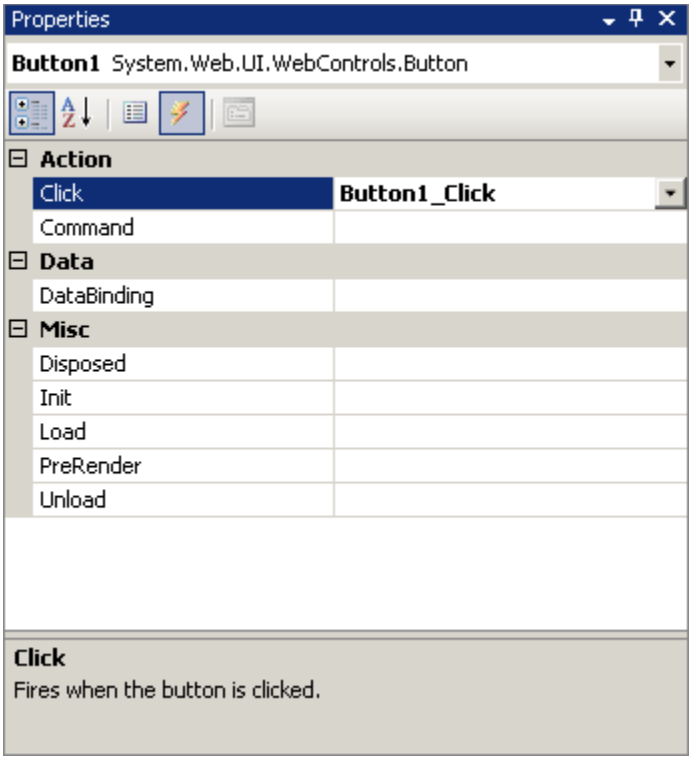

Figure 4 – The Events for a *Button* control.

Postponed Events

Most Web Server Controls cause a PostBack when an event occurs at the client. A *Button*, for instance, raises the *Click* event on the server when the button is clicked.

Some controls, however, don't automatically cause a PostBack, and the event is postponed. A *TextBox* has a *TextChanged* event that doesn't get called when the text changes, instead occurring when another control causes the PostBack.

A PostBack can be forced for postponed events by setting the *AutoPostBack* property to *true*.

Naming Containers and Child Controls

When a Web Server Control is added directly to a Web Form, User Control or Master Page it is available in code using the name of the control directly. Under the covers, the compiler automatically exposes the controls directly contained within the *Controls* collection as properties of the Web Form, User Control or Master Page itself.

However, there are Web Server Controls (in particular the data-bound controls such as *GridView*) that contain controls that are repeated for each item of data that is being displayed. Each of the controls is repeated but they have unique names because the parent control acts as a Naming Container for the child controls.

In addition to the *ID* property of the Web Server Control specified in the HTML markup, each Web Server Control also has a *UniqueID* property. The *UniqueID* property is a fully qualified name for the control based upon the *ID* of its parent control, as well as its own *ID*.

You can find controls within a Naming Container by using the *FindControl* method to search the *Controls* collection for the Naming Container. If the control exists, it is returned as a *System.Web.UI.Control* that can be cast to the correct control type as required.

HTML Server Controls

An HTML Server Control is a normal HTML tag with the addition of the *runat="server"* attribute, and if necessary, the addition of an *ID* attribute. There is a direct mapping between the HTML tag and the controls that are available in code on the server.

Unlike Web Server Controls, HTML Server Controls don't have an event model they can be easily tied into, and the properties of these controls aren't consistent as they are for Web Server Controls.

Not all HTML tags have their own HTML Server Control that they map to. If there isn't a specific HTML Server Control for the HTML tag, the control will be an *HtmlGenericControl*. The *Tag* property then specifies the HTML tag.

Creating HTML Server Controls

HTML Server Controls can be created in two ways:

- By adding a *runat="server"* attribute to an existing HTML tag, ensuring that it has an *ID* attribute.
- By adding the control to the Design or Source view of a Web Form, User Control or Master Page by selecting the required control from the HTML tab of the toolbox.

Configure the Properties of HTML Server Controls

Properties of HTML Server Controls can be configured two different ways:

- By selecting the HTML Server Control in Source view and adding the properties as attributes to the HTML markup for the control.
- Programmatically, by referencing the control by name and setting the property's value.

Examples of Web Server Controls

It would be impossible to list every nuance of every Web Server Control that exists. The following table lists some of the more common controls and gives some examples of their properties and methods.

Control	Description
AdRotator	The *AdRotator* displays randomly selected banners. Configured via an XML file (which contains various properties for the advert such as *ImageUrl* and *NavigateUrl*). The *AdRotator* is displayed at the client using <*a*> and <*img*> tags.
Button	The *Button* is rendered as an <*input type="button"*> tag and raises a *Click* event that is fired when the button is clicked in the client. You can use the *Command* property to distinguish between different buttons that use the same event handler.
Calendar	The *Calendar* is used to display a monthly calendar. The *SelectedDate* property returns the date selected by the user. Several events provide PostBack when the user performs an action – *SelectionChanged* when the user selects a date and *VisibleMonthChanged* if the user selects a different month.
CheckBox	The *CheckBox* is rendered as an <*input type="checkbox"*> control and allows a user to select either true or false. The *CheckChanged* event is raised when the user changes their selection, but is postponed by default. The *Checked* property is used to check the status of the control.
FileUpload	The *FileUpload* control displays as a *TextBox* and a browse *Button* that allows a file to be uploaded to the server. The browse button allows the user to browse their local machine for the file to upload. The PostBack to the server must occur via another control. The upload file is available on the server via the *PostedFile* property.
HyperLink	The *HyperLink* control is rendered at the client as an <*a*> tag. The *NavigateUrl* property sets the required URL.
Image	The *Image* control is rendered as an <*img*> tag. The image required is specified using the *ImageUrl* property.
ImageButton	The *ImageButton* combines the functionality of the *Image* and *Button* controls. It is rendered at the client as an <*input type="image"*> tag and raises a *Click* event when clicked.
ImageMap	The *ImageMap* control is similar to an *ImageButton*, but instead of the entire image being clickable, regions are defined (*CircleHotSpot*, *RectangleHotSpot*, or *PolygonHotSpot*) that allow different actions to be taken based on where the image is clicked. This is rendered as an <*img*> tag with a defined *usemap* attribute that points at a <*map*> tag corresponding to the defined regions.
Label	The *Label* allows you to display text, specified by the *Text* property, and is rendered as a <*span*> tag. This control supports CSS styles (via the *CssClass* property), themes and skins.

LinkButton	The *LinkButton* combines the functionality of the *HyperLink* and *Button* controls. It is rendered as an <*a*> tag but instead of navigating to another page when clicked, a *Click* event is raised at the server.
ListControl	The *ListControl* is an abstract control that provides the basic requirements for all list controls. The *ListControl* has an *Items* property that contains a collection of *ListItem* objects. The *ListItem* has a *Text* property that is displayed to the user and a *Value* property that is posted back to the server. You can populate the *Items* collection either with HTML markup or by using the *DataSource* or *DataSourceID* properties and binding the control to the data. The *SelectedIndex* property returns the index (in *Items*) of the selected item. *SelectedItem* returns the actual *ListItem* selected and *SelectedValue* returns the *Value* of the selected *ListItem*. There are five controls derived from *ListControl*: • *BulletedList* – rendered as unordered (<*ul*>), or ordered (<*ol*>), tags based on the *BulletStyle* property. Unlike the other four derived controls, the BulletedList does not allow the user to select any of the items shown. • *CheckBoxList* – rendered as a series of <*input type="checkbox"*> tags that allow the user to select multiple options. The *SelectedIndex*, *SelectedItem* and *SelectedValue* will only return the details of the first item selected. The *Items* collection must be enumerated and the *Selected* property evaluated to determine all the entries selected. • *DropDownList* – rendered as a <*select*> tag containing <*option*> tags for each *ListItem* to be displayed. • *ListBox* – rendered as a <*select*> tag (with a *size* attribute) and <*option*> tags for each *ListItem* to be displayed. The *SelectionMode* property can be used to allow the user to select multiple entries from the list, in which case *SelectedIndex*, *SelectedItem* and *SelectedValue* will only return the details of the first item selected. The *Items* collection must be enumerated and the *Selected* property evaluated to determine all the entries selected. • *RadioButtonList* – rendered as a grouped series of <*input type="radio"*> tags that allow the user to select a single option.
Literal	The *Literal* control is similar to the *Label* control as it displays text, specified by the *Text* property. It is rendered directly at the client and does not output any tags, neither does it support CSS styles, themes or skins.
Panel	The *Panel* control is output as a <*div*> tag and is a control container – it contains child controls that are displayed on the client. It is useful when you want to show and hide groups of controls together, which is easily accomplished using the *Visible* property.
MultiView	The *MultiView* control is a container control for *View* controls and doesn't generate any tags when it is displayed. Within a *MultiView* you define several View controls and you can select which one is displayed by using the *ActiveViewIndex* property.

View	The *View* control is exclusively used as a child control of the *MultiView* and doesn't generate any tags when it is displayed. It is a container for other controls that are displayed if the *View* is active.
RadioButton	The *RadioButton* control is rendered as an <*input type="radio"*> control and allows the user to select from a mutually exclusive set of options. The *Text* property sets the caption of the *RadioButton* and the *GroupName* property is used to group *RadioButton* controls together. The *Checked* property is used to determine if the user has selected a specific RadioButton.
Table	The *Table* control is rendered as a <*table*> tag and is a container control for *TableRow* controls. The *Rows* property returns a collection of *TableRow* controls.
TableRow	The *TableRow* control is rendered as a <*tr*> tag and is a container for *TableCell* controls. The *Cells* property returns a collection of *TableCell* controls.
TableCell	The *TableCell* control is rendered as a <*td*> tag. It can contain HTML markup, Web Server Controls, HTML Server Controls or plain text.
TextBox	The *TextBox* allows the user to enter text. The *MaxLength* property specifies the maximum number of characters that can be entered and the *Text* property returns the text that is entered. There are three modes for a TextBox, specified using the *TextMode* property: • *SingleLine* – this is the default *TextMode* and it is rendered as an <*input type="text"*> tag. • *MultiLine* – this is rendered as a <*textarea*> tag with the *Columns* and *Rows* properties specifying the size of the area. • *Password* – rendered as an <*input type="password"*> tag. Each character entered by the user is shown as an asterisk.
Wizard	The *Wizard* is a control that is used to display a series of *WizardStep* controls to the user. The Wizard control has a *WizardSteps* property that returns a collection of *WizardStep* controls. Only one *WizardStep* is visible at a time and the current step is identified by the *ActiveStepIndex* property. Each *WizardStep* has a *StepType* that determines which of the built-in navigation options are shown.
Xml	The *Xml* control is used to display an XML file or the results of an XSL transform on an XML file. The *DocumentSource* property is used to point at an external XML file and the *DocumentContent* property is used when the XML is available as a string. An XSL transformation can be performed on the XML document by setting *TransformSource* to a XSL document.

Programmatically Edit Settings in Web.config

You can make changes to the site configuration using the Web Site Administration Tool or by directly editing Web.config. You can also programmatically access Web.config using the *System.Web.Configuration* namespace.

Calling the *GetSection* method on the *WebConfigurationManager* class returns an object that you must cast to the correct type as shown in the following table:

Class	**Configuration Section (within <system.web>)**
AnonymousIdentifcationSection	<anonymousIdentification>
AuthenticationSection	<authentication>
AuthorizationSection	<authorization>
CacheSection	<caching><cache>
ClientTargetSection	<clientTarget>
CompilationSection	<compilation>
CustomErrorsSection	<customErrors>
DeploymentSection	<deployment>
GlobalizationSection	<globalization>
HealthMonitoringSection	<healthMonitoring>
HostingEnvironmentSection	<hostingEnvironment>
HttpCookiesSection	<httpCookies>
HttpHandlersSection	<httpHandlers>
HttpModulesSection	<httpModules>
HttpRuntimeSection	<httpRuntime>
IdentitySection	<identity>
MachineKeySection	<machineKey>
MembershipSection	<membership>
OutputCacheSection	<cache><outputCache>
OutputCacheSettingsSection	<cache><outputCache><outputCacheSettings>
PagesSection	<pages>

ProcessModelSection	<processModel>
ProfileSection	<profile>
RoleMangerSection	<roleManager>
SecurityPolicySection	<securityPolicy>
SessionPageStateSection	<sessionPageState>
SessionStateSection	<sessionState>
SiteMapSection	<siteMap>
SqlCacheDependencySection	<cache><sqlCacheDependency>
TraceSection	<trace>
TrustSection	<trust>
UrlMappingsSection	<urlMappings>
WebControlsSection	<webControls>
WebPartsSection	<webParts>
XhtmlConformanceSection	<xhtmlConformance>

Once you have a class of the correct type you can use the properties and methods to read or write the Web.config settings.

Dynamically Adding Controls to a Web Form

Controls can be added to a Web Form (and indeed any control derived from *System.Web.UI.WebControls.WebControl* class) by calling the *Add* method of the *Controls* collection and passing in the control to be added. This is usually done in an overridden *OnInit* method to ensure that the controls are available before any event handling takes place.

Create Event Handlers

Create Handlers for a Page at Design Time

As with Web Server Controls, the Web Form itself also has several events that occur. However there is no Visual Studio help for creating these events in C# and they must be added manually to the Web Form.

Each Web Form that you create will have the following Load event handler defined automatically:

```
protected void Page_Load(object sender, EventArgs e)
{
}
```

There are 15 events in total for the Page and they all have the same method signature – replace *Load* with the name of the event you wish to handle.

Page events are, by default, automatically called at runtime. Setting the *AutoEventWireup* attribute of the *@Page* directive to false will stop the automatic wiring up of Page events.

Respond to Application and Session Events

There are also events raised that are specific to the entire application and not a specific Web Form or User Control. These events are as follows and are defined in Global.asax:

Event	Description
Application_Start	Raised when the application is started. You can initialize application variables here and perform any necessary logging.
Application_End	Raised when the application is shutdown. You can free any resources used and perform any necessary logging.
Application_Error	Raised when an unhandled exception occurs anywhere within the application. You can send error emails and perform any necessary logging.

There are also two events that are raised on a per-session basis:

Event	Description
Session_Start	Raised when a new session begins.
Session_End	Raised when a session expires, either explicitly by calling *Session.Abandon*, or because the session has timed out.

Manage State and Application Data

Manage State by Using Client-Based State Management Options

ASP.NET provides several ways of handling state on the client:

Name	Description
View State	View State is used for storing information between requests for the same page. It is stored as a hidden field in the page and is only useful for temporarily storing values. If you don't need to remember the state of controls between iterations of the page, you can turn View State off by setting the *EnableViewState* property to false. By default the View State is not encrypted and any sensitive information is relatively easy to view. You can add objects to View State by using the *ViewState* property of the Web Form – this is a dictionary of key/value pairs.
Control State	Control State is similar to View State but cannot be turned off. You need to override the *SaveControlState* and *LoadControlState* methods to access Control State. It is not enabled by default and you must call the *RegisterRequiresControlState* method of the *Page* to enable it for a control.
Hidden Fields	Hidden Fields have to be manually added to the Web Form as *HiddenField* controls and the stored data can be accessed using the *Value* property. Hidden Fields are only suitable for storing temporary values between requests for the same page.
Cookies	Cookies can be stored on the client by accessing the *Cookies* collection of the *Request* object. The *Cookies* collection is a dictionary of key/*HttpCookie* pairs. You can make cookies persistent by setting the *Expires* property of an *HttpCookie*, and you can control the scope of the cookie by setting the *Path* property.
Query String	Query String values can be accessed using the *QueryString* property of the *Request* object. The *QueryString* collection is a dictionary of key/*String* pairs. The Query String forms part of the request for the page and needs to be added to other URLs if the value is to be persisted between different pages.

Manage State by Using Sever-Based State Management Options

State on the server can be stored either in Application scope or Session scope. Each user has their own Session state whereas the Application state is shared between all the users; you should never store user-specific data in the Application state.

Both Application state and Session state are stored as key/*Object* pairs and the object being stored must be serializable. When retrieving values from Application or Session state, the Object retrieved must be cast to the correct type before it is used.

Application State

Application state is specific to the server that the application is running on and because it may be accessed by several different requests at the same time, you need to *Lock* the Application state before using it and *UnLock* the Application state when finished.

Session State

Session state is specific to a user and is enabled by default. It can be turned off for an entire Web site by setting the *mode* attribute of *<sessionState>* in Web.config to *off*, or for a specific Web Form by setting the *EnableSessionState* attribute of the *@Page* directive to *false*.

Session state can be stored in several places depending on your requirements:

State	Description
Custom	A custom storage provider is being used.
InProc	This is the default value in which the session state is stored within the context of the Web server. This is fine for a simple Web site, but if you're running the site on multiple Web servers or must have persistent session data between application restarts, you must use SQLServer or StateServer.
SQLServer	All session data is stored in an SQL server database. Session state is available to multiple Web servers so that if one of the Web servers fails, the event will not result in any lost session data.
StateServer	All session data is stored in a separate service (the ASP.NET State Service) on the Web server. Several Web servers can use the service on one Web server. However if the Web server running the ASP.NET State Service fails, then all Web servers will lose the state information.

Implement Globalization and Accessibility

Globalization and Localization

Web Forms, User Controls and Master Pages can be displayed in different languages by using resource files (files with a .resx extension). You can use both local and global resources.

Local Resources

Local resources are specific to a single Web Form, User Control or Master Page and are used to configure for a different language. Local resources are stored in an App_LocalResources folder within the folder containing the Web Form, User Control or Master Page (there may be an App_LocalResources folder in every folder within the Web site).

Local resource files are named using Name[.language].resx where *Name* is the name of the Web Form, User Control or Master Page and *language* is the optional abbreviation for the language to be localized. For example, a Web Form called Default.aspx may have several files within the App_LocalResources folder. Default.aspx.resx is the file to use if no other resource files match. Default.aspx.es.resx is specific to Spanish and Default.aspx.de.resx is specific to German.

There is only limited support for globalization in Visual Studio. Resource files can easily be created using the tools, but linking the resource file contents to the Web Server Control is trickier.

Within the resource file you must specify a name for the resource value – this is a combination of a unique keyword (normally the Web Server Control's name) and the property that you wish to set, with each value separated by a period (e.g. *Button1.Text*.).

Within the Web Form you must then manually add a meta:resourcekey attribute to Button1 that points at the name used in the resource file (e.g. *meta:resourcekey="Button1"*).

At runtime the correct resource file will be selected and the Web Server Control properties (matched to the values in the resource file) will be loaded and set.

Global Resources

Global resources are available to any Web Form, User Control or Master Page in the Web site and should only be used when you need to access the same resource from multiple places. Global Resources are stored in the App_GlobalResources folder at the root of the Web site.

Global resource files are named as required following the Filename[.language].resx, where *Filename* is the name of the resource file and *language* is the optional abbreviation for the language to be localized. The format of a global resource file is the same as a local resource file, except that the name of the resource value doesn't need to include the name of the property.

Within the Web Form, User Control or Master Page you add *<% Resources: Filename, Name %>* where *Filename* is the name of the resource file (minus the language or resx extension) and *Name* is the name of the resource value.

Changing Culture

There may be times when the culture of the browser (used to determine which resources to use by default) is incorrect. In these cases you can override the culture for a Web Form by overriding the *InitializeCulture* method of the Page class and set the *Culture* and *UICulture* properties to the language abbreviation for the required culture, not forgetting to call the *InitializeCulture* method of the base class.

Accessibility

Web Server Controls are designed to be accessible by default. There are however several guidelines that you should follow.

Visual Accessibility

- **Give every image alt text by setting the *AlternateText* property**. Useful when images are disabled or not available. Screen readers will read the alt text descriptions.
- **Use colors correctly**. Easy to read text in a color that contrasts a solid-colored background is better for the user.
- **Flexible page layout**. Modern browsers allow text to be resized, so provide a layout that allows text to be resized without breaking completely.
- **Don't define specific font sizes**. Use tags (e.g. *<h1>*, *<p>*) to control text sizes to make use of the user's preferences.
- **Set table captions**. In order to give users with screen readers the option of skipping tables, provide a *caption* attribute to specify a description of the table.
- **Identify column headers**. Use *<th>* tags to add headings to columns in tables. This simplifies navigation for users with screen readers.
- **Avoid client scripts**. Screen readers will probably not handle client scripts correctly. You should avoid these as the Web Content Accessibility Guidelines bar the use of client scripts.

Forms Accessibility

- **Use *DefaultFocus* to set the cursor position on a form to the location where data entry normally begins**. Typically this is the topmost editable field in a form.
- **Define a tab order**. Use the *TabIndex* property on Web Server Controls so that tabbing between the fields on the page makes sense.
- **Use *DefaultButton* to set a default button**. Default buttons can be accessed by pressing enter. This makes using the form a lot simpler.
- **Have useful link text**. When adding a hyperlink, use descriptive text for the link and avoid links that display as "Click here," or other similar, generalized text.
- **Define access keys for controls**. Use the *AccessKey* property to define keys that can be used in association with the Alt key to access the control.
- **Use *Label* controls to provide access keys for *TextBox* controls**. A *TextBox* doesn't have a description that can be used by screen readers. It is best to associate a descriptive *Label* control with the *TextBox*. Use the *AccessKey* and *AssociatedControlID* properties of the *Label* control to link it to the *TextBox*.
- **Create form sections**. Use the Panel control to create sections on the form and use the *GroupingText* property to describe the controls in that section. This will output *<fieldset>* and *<legend>* tags that make the form easier to navigate.

Implement Site Navigation and Input Validation

The SiteMap Web Server Control

The *SiteMapPath* Web Server Control displays a breadcrumb trail for the Web site based on the contents of a SiteMap file.

When a *SiteMapPath* Web Server Control is rendered, a SiteMap file called *Web.Sitemap* is interrogated to discover the current page's position in the hierarchy. This is then displayed as a breadcrumb trail that allows the hierarchy of pages to the current page to be displayed.

The SiteMap file can be shared with the *TreeView* and *Menu* Web Server Controls to give consistent navigation throughout the site.

Validation Controls

Validation Web Server Controls allow input validation to be performed at either the client or the server. There are five different validation Web Server Controls:

Name	Description
CompareValidator	Used to compare the value of a control to a specific value (using the *ValueToCompare* property) or to another control (using the *ControlToCompare* property). The *Operator* property sets the type of comparison. This can also be used to determine if the data is of a certain type by setting the *Type* property.
CustomValidator	Used to specify custom validation at the client and/or server. A client script is attached using the *ClientFunctionName* property and server validation is handled using the *ServerValidate* event.
RangeValidator	Used to specify that the data entered is between a set of values (specified using *MinimumValue* and *MaximumValue* properties). The *Type* property can be used to specify the type of the required value.
RegularExpressionValidator	Validates based upon a regular expression specified using the *ValidationExpression* property.
RequiredFieldValidator	Used to ensure that a control contains a value. Note that the other controls don't attempt to validate an empty control, so you should always use a *RequiredFieldValidator* as well.

In addition to the properties listed above for the specific controls, each of the Validation Web Server Controls also has a common set of properties:

Property	Description
ControlToValidate	Set to the name of the control to be validated.
Display	Determines how the control is displayed. A value of *None* indicates that there is no output (although the *ErrorMessage* would still be shown in the *ValidationSummary*), *Static* to display the *Text* property and reserve the space for the message, or *Dynamic* to display the *Text* property with no space used if the message doesn't need to be shown.
EnableClientScript	Set to true (the default setting) to perform client-side validation. Client-side validation should not be solely relied on and server-side validation should be performed in tandem.
Enabled	Set to false if the validation control is disabled.
ErrorMessage	The error message that is displayed in the *ValidationSummary* when validation fails.
IsValid	Set to true if the validation is successful.
Text	The message shown when validation fails.

CustomValidator Control
Both client and server validation routines can be added to a CustomValidator.
A client-side JavaScript function needs to be specified as the *ClientFunctionName* property with the following signature:

function ClientFunctionName(source, arguments)

The *source* parameter contains the control that is being validated. The *arguments* parameter is an object that has two properties: *Value*, containing the value being validated; and *IsValid*, that should be set to true if validation succeeds.

Server-side validation is handled via the *ServerValidate* event. This event has two parameters: the *source* parameter is a reference to the validation control raising the event; *args* is a *ServerValidateEventsArgs* object that has *Value* and *IsValid* as properties.

Validating Controls
Controls are validated automatically at the server side and set the *IsValid* property on the Web Form. This should be checked before any event handlers are executed to ensure that execution can go ahead. Validation Web Server Controls can also be grouped using the *ValidationGroup* property to separate different sections of a Web Form and only run the necessary validation. When set on Web Server Controls (the validation controls and the controls that cause a PostBack to occur) only controls within the same *ValidationGroup* are evaluated when the PostBack occurs.

A validation control can be manually validated by calling its *Validate* method. This will set the *IsValid* property of the control and also update the *IsValid* property of the Web Form.

Write an ASP.NET Handler to Generate Images Dynamically

ASP.NET Handlers implement the *IHttpHandler* interface and must provide implementations of the following:

- *IsReusable* property – determines whether the handler can be pooled and reused on subsequent requests.
- *ProcessRequest(HttpContext)* method – responsible for processing the request and writing the output to the response.

Adding an Application Mapping

If you're using a non-standard file extension, then this needs to be mapped to the ASP.NET process. Adding an application mapping is accomplished from the Internet Information Services configuration tool. For a Web site or a configured Application, the Directory tab of the properties dialog has a Configuration button. Clicking this shows the Application Configuration and the Mappings tab shows all the file extension mappings. By clicking the Add button you can add a new mapping, as shown in **Figure 5**.

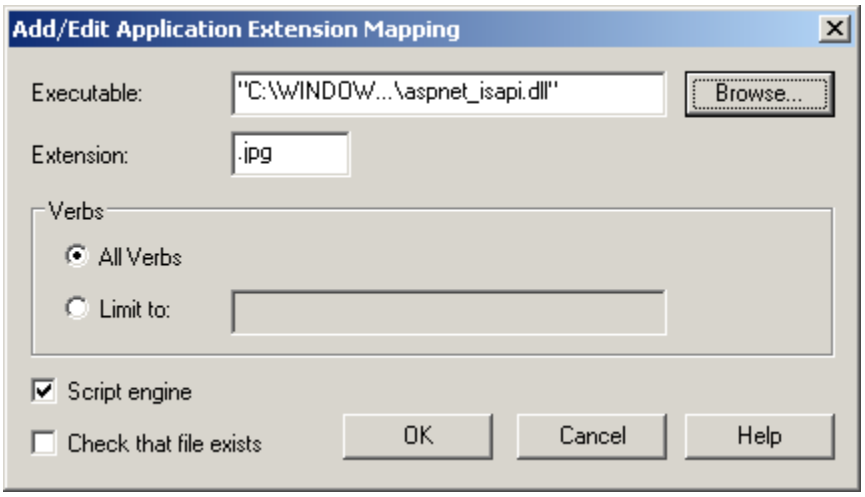

Figure 5 – Adding a File Extension Mapping in IIS

The executable for ASP.NET is the aspnet_isapi.dll file, which, for the .NET Framework v2.0, is at: %WINDOWS%\Microsoft.NET\Framework\v2.0.50727\aspnet_isapi.dll.

Configuring ASP.NET to Use the Correct ASP.NET Handler

Once the mapping is added, you need to add the ASP.NET Handler to Web.Config in the <*httpHandlers*> section of <*system.web*>:

```
<system.web>
 <httpHandlers>
  <add verb="*" path="*.jpg" type="ImageHandler">
 </httpHandlers>
<system.web>
```

Writing the ASP.NET Handler to Process the Image

ASP.NET Handlers need to implement the *IHttpHandler* interface and implement *IsReusable* and *ProcessRequest*. To return an image, we can write the bare bones of the ASP.NET handler as follows:

```
public class ImageHandler : IHttpHandler
{
 public ImageHandler
 {
 }

 public bool IsReusable
 {
  get
  {
   return(true);
  }
 }

 public void ProcessRequest(HttpContext context)
 {
  // set the MIME type correctly
  context.Response.ContentType = "image/jpeg";

  // output the image as necessary
  //  - either create using System.Graphics
  //  - or load from file and return using Response.WriteFile method
 }
```

Configure Application Settings

There are several files that can contain configuration settings:

- Machine.config – stores settings that apply to the entire computer. Stored in the %WINDOWS%\Microsoft.NET\Framework\v2.0.50727\CONFIG folder. It contains settings that apply to all aspects of the .NET Framework and not just settings for ASP.NET.
- Root Web.config – stores settings that apply to all Web sites on the computer. Stored in the %WINDOWS%\Microsoft.NET\Framework\v2.0.50727\CONFIG folder.
- Site Web.config – stores settings that apply to a specific Web site and is stored at the root of the site.
- Folder Web.config – a Web.config file can exist in any folder and contains settings that only apply to that folder. There are only a limited number of settings that can be placed in a folder Web.config file.

When parsing settings, specific settings override the generic ones. So settings in a Folder Web.config file override those in the Site Web.Config file, which in turn override those in the Root Web.config, which in turn override those in Machine.Config.

Using the Web Site Administration Tool

Manually changing the Web.config file can be tedious and is prone to error. Rather than manually editing the Web.config file, you can use the Edit Configuration option on the ASP.NET tab of the Web site properties dialog to set many of the settings that you require.

The Web Site Administration Tool, shipped with Visual Studio 2005, allows you to configure some of the settings for your Web site:

- **Security** – configures security for the Web site. Users, roles and permissions can all be configured.
- **Application** – allows application settings to be modified. You can also modify SMTP settings, set debugging and tracing options, and also define the default error page to display.
- **Provider** – allows you to configure the database provider to use for the Membership and Roles functionality.

Programming a Web Application

Avoid Unnecessary Processing

When a PostBack occurs the *IsPostBack* property of the Web Form is set to true. You can use check this property to stop any unnecessary processing from occurring.

Cross Page Postbacks

By default, a Web Server Control posts back to the same page. All the button controls (*Button*, *ImageButton* and *LinkButton*) have an extra property (*PostBackUrl*) that allows them to cause the postback to go to a different page.

When a cross-page PostBack occurs, the contents of the original page are available via the *PreviousPage* property of the new Web Form. If the page is not a cross-page PostBack, the *PreviousPage* property will be null.

You can access any of the controls on the previous page by calling the *FindControl* method and specifying the name of the control you wish to access. You can then cast the *Control* object to the correct Web Server Control type.

Redirecting the Client

There are three ways that you can redirect the client to another Web Form:

- Client code or markup – use JavaScript or Web Server Controls (such as a *HyperLink*) to request a new page.
- *Response.Redirect* – sends an HTTP 302 code to the client and the client is responsible for redirecting to the new page. The new URL is shown in the address bar of the browser. You cannot access the state of the previous page (using the *PreviousPage* property), and any state information that you need to pass must be handled using another method (cookies, session state, etc.).

- *Server.Transfer* – the transfer to the new page is handled at the server, and as far as the browser is aware, the original URL is still being viewed. The *Transfer* method has a *preserveForm* property that, if set to true, passes the form and *QueryString* parameters to the new page. When the transfer has taken place, the *PreviousPage* property contains the original page (as it would for a cross-page PostBack).

Page and Application State

There are several objects that are provided as part of the Web Form context. These are available through the static *System.Web.HttpContext.Current* property, or as properties of the Page and UserControl objects. Six of the most useful objects are shown below:

Name	Description
Application	Returns an *HttpApplicationState* object that provides access to the application properties and methods of the Web site.
Request	Returns an *HttpRequest* object that provides access to the information of the current request – headers, cookies, query string, form variables, etc. All information on the *HttpRequest* object is read-only.
Response	Returns an *HttpResponse* object that provides access to the information that is to be sent back to the browser. Most information within an *HttpResponse* object can be written and you can modify the text sent to the browser, add and remove cookies, add headers, etc.
Session	Returns an *HttpSessionState* object that provides access to the current user's session.
Server	Returns an *HttpServerUtility* object that exposes helper methods and properties that allow you to handle requests from the client. You can get the last error that occurred, encode and decode HTML and URLs, and much more.
Trace	Returns a *TraceContext* object that provides methods to write to the trace logs for the Web Form.

Detecting Browser Capabilities

Not all browsers are equal! You should test your Web site in every browser that your users might use. ASP.NET controls automatically adapt to the capabilities of the browser requesting the page but you may need to determine what capabilities the browser has manually.

The *Browser* property of the *HttpRequest* returns an *HttpBrowserCapabilities* object that allows you to check the capabilities of the browser requesting the Web Form.

There are several properties exposed by this object that allow you to check things like:

- Does the browser support JavaScript?
- Does the browser support frames?
- Does the browser support ActiveX?

Handling Exceptions at Page Level

We've seen earlier that we can handle errors at application level by using the *Application_Error* event handler in Global.asax. You can catch individual errors on the page using *try/catch* blocks to deal with any specific problems and present specific error messages to the user.

It is also possible to add a *Page_Error* event handler that will handle any unhandled exceptions for the page.

Rather than using the parameters to the event handler (which are very generic and don't contain a lot of information), you can call *Server.GetLastError()* to return the last exception raised. Once you've handled the error, call *Server.ClearLastError()* so that any application error handling doesn't catch the error as well.

Accessing the Web Form Header

The Web Form Header section (contained with the <*head*> HTML markup) is accessible using the *Header* property of the Web Form. This returns an *HtmlHead* object that has two properties:

Name	Description
StyleSheet	Returns an *IStyleSheet* instance that has two methods – *CreateStyleRule* and *RegisterStyle* which add new styles (in the form of *Style* objects) to the Web Form.
Title	Allows you to programmatically set the title (the <*title*> HTML element) of the Web Form.

Integrating Data in a Web Application by Using ADO.NET, XML and Data-Bound Controls

Data Source Controls

Tabular Data Source Controls

Tabular Data Source controls are used to return data that is table based and gathered from a database or collections of data returned from an object. All Tabular Data Source controls inherit from the *DataSource-Control* abstract class and there are three that are shipped with ASP.NET 2.0:

Name	Description
AccessDataSource	A more specific version of the *SqlDataSource* that can only connect to a Microsoft Access database. You set the *DataFile* property to the correct MDB file and then use as you would for the *SqlDataSource*.
ObjectDataSource	Connects to objects, rather than to a database. The *TypeName* property specifies the name of the type that is being connected to. There are four sets of properties that are used to query the object – for instance, *SelectMethod* and *SelectParameters* specify the method used to select data and there are corresponding methods for Delete, Insert and Update. In addition, the *SelectCountMethod* is used to specify the method used to return the total rows in the data (and is used when paging).
SqlDataSource	Connects to any ADO.NET provider-supported database. The *ConnectionString* property is used to specify the connection string in Web.config to use (this needs to specify both the *connectionString* and *providerName*). There are four sets of properties that are used to query the database - for instance *SelectCommand*, *SelectCommandType* and *SelectParameters* specify the method used to select data. There are corresponding methods for Delete, Insert and Update.

Hierarchical Data Source Controls

Hierarchical Data Source controls are used to return data that isn't tabular in nature, but, essentially, XML in format. All Hierarchical Data Source controls inherit from the *HierarchicalDataSourceControl* abstract base class. There are two shipped with ASP.NET 2.0:

Name	Description
SiteMapDataSource	Exposes the sitemap file in the root of the application as a Data Source.
XmlDataSource	Allows any XML file, specified in the *DataFile* property, to be used as a data source. The *TransformFile* property allows you to specify an XSL transform that is to be applied to the XML before it is returned.

Data-Bound Controls

Data bounds controls are those that connect, or more correctly bind, to data. The data bound controls are all derived from the abstract *BaseDataBoundControl* base class. The controls shipped as part of ASP.NET 2.0 fall into three categories:

- Simple – these controls display tabular data. The *AdRotator* control and all the controls that derive from *ListControl* fall into the simple data bound control category. These controls inherit from the abstract *DataBoundControl* class (which in turn inherits from *BaseDataBoundControl*).
- Composite – these controls display tabular data, but unlike simple data bound controls, the output consists of several controls grouped together. The *GridView*, *DetailsView* and *FormView* are examples of composite data bound controls. These controls inherit from *CompositeDataBoundControl* (which, in turn, inherits from *DataBoundControl* and *BaseDataBoundControl*).
- Hierarchical – these controls are used to display hierarchical data; the *Menu* and *TreeView* controls fall into this category. These controls inherit from *HierarchicalDataBoundControl* (which, in turn, inherits from *BaseDataBoundControl*).

Data binding is the process of taking data from some source and displaying it in a Web Server Control. We can specify the source of the data on a Data Bound control in two different ways:

- Set the *DataSource* property to an object that supports the *IEnumerable* interface (e.g. *SqlDataReader*, *DataTable*, *ArrayList*, etc.). The *DataBind* method must be called, otherwise no data binding will take place.
- Set the *DataSourceID* property to the *ID* of a Data Source control. If both a *DataSource* and a *DataSourceID* are specified, then *DataSourceID* takes precedence. Data binding is automatic.

Display Data using Simple Data Bound Controls

There are six simple data bound controls shipped as part of ASP.NET 2.0:

Name	Description
AdRotator	Used to display randomly selected banners on Web Forms.
BulletedList	Display a bulleted list of read-only items. Derived from the abstract *ListControl* base class.
CheckBoxList	Display a list of check boxes allowing the user to select multiple items. Derived from the abstract *ListControl* base class.
DropDownList	Display a drop down list allowing the user to select a single item. Derived from the abstract *ListControl* base class.
ListBox	Display a list allowing the use to select either a single item or, if enabled, multiple items in the list. Derived from the abstract *ListControl* base class.
RadioButtonList	Display a list of radio buttons allowing the user to select a single item. Derived from the abstract *ListControl* base class.

The ListControl Derived Controls

All list controls derive from the abstract *ListControl* base class; this provides some basic functionality to all of the controls. The five derived controls then provide a different user-interface view of the same data as shown in Figure 6.

Figure 6 – The Five ListControl Derived Controls

The *ListControl* provides an *Items* property that returns a collection of *ListItem* objects, each with a *Text* and *Value* property. The *Text* is displayed to the user and the *Value* is posted back to the server.

You can populate the *Items* collection by manually adding *ListItem* objects in code, by declaring *ListItem* elements in the HTML markup or by data binding and using a data source to return the items.

If you're using data binding to populate the *Items* collection, you'll need to set the *DataTextField* and *DataValueField* properties to indicate which fields from the data are to be used for the *Text* and *Value* properties of the *ListItem* objects, respectively. You can also use the *DataTextFormatString* property to control how the text is formatted.

The *SelectedIndex* property allows you to get or set the index of the item that is currently selected. If the derived control allows multiple items to be selected (which may be the case for certain controls, such as *CheckBoxList* and *ListBox,*), then *SelectedIndex* will return the first item that is selected. You need to iterate through the Items collection and check the *Selected* property of each of the *ListItem* objects to determine the items that were selected.

Similarly, the *SelectedItem* property returns the selected *ListItem* and the *SelectedValue* returns the *Value* of the selected *ListItem*. Again, if the control allows multiple items to be selected, then you should query the *Items* collection to check each *ListItem* to see if it was selected.

In addition to the standard properties, each of the derived controls may have their own properties:

Control	Property	Description
CheckBoxList	RepeatColumns	The number of columns to show before the items are wrapped to the next line.
	RepeatDirection	Set to *Horizontal* or *Vertical*, indicating where the next item will be placed.
ListBox	Rows	The number of rows to display. If the number of items is greater than this value, then scrollbars are added to the *ListBox*.
	SelectionMode	By default, the *ListBox* only allows single selection but setting this to *Multiple* allows multiple items to be selected by the user.
RadioButtonList	RepeatColumns	The number of columns to show before the items are wrapped to the next line.
	RepeatDirection	Set to *Horizontal* or *Vertical*, indicating where the next item will be placed.

The *ListControl* also raises an event, *SelectedIndexChanged*, when the user selection changes between post backs to the server.

The *ListControl* does not automatically post back to the server when the selection is changed and instead relies on another control to cause the post back. Setting the *AutoPostBack* property to true overrides this behavior and causes a post back to occur whenever the user changes their selection.

The AdRotator Control

The AdRotator Control displays randomly selected banners. You can either use data taken from a data source (by specifying the *DataSource* or *DataSourceID* properties) or directly from an XML file containing the data by setting the *AdvertisementFile* property.

The data used as the source for the AdRotator can contain the following information:

Name	Description
AlternativeText	Text to be displayed if the image is unavailable.
Height	An optional value specifying the height of the advert.
ImageUrl	URL of the image to display.
Impressions	An optional number specifying the relative weighting of the advert.
Keyword	An optional keyword that can be used to filter the adverts displayed.
NavigateUrl	URL to navigate to if the ad is clicked.
Width	An optional value specifying the width of the advert.

Display Data using Composite Data Bound Controls

Composite controls are those that display the data using a collection of other controls. There are three composite data bound controls that ship as part of ASP.NET 2.0:

Name	Description
DetailsView	Used to display a single record from the data source in a table with each two-column row corresponding to a field from the data source. Rows can be automatically generated, bound to a single field from the record or created by using a template to customize the appearance of the row.
FormView	Used to display a single record from the data source. Templates can be created to display the data and you full control over what is displayed.
GridView	Displays data in a table with each record in the data source corresponding to a row. Columns can be automatically generated, rendered directly to a single field from the record or created using a template to customize the appearance of a column.

Binding To Records in the Data Source

Both the *DetailsView* and *GridView* automatically generate the rows and columns to be displayed. This can be turned off by setting the *AutoGenerateRows* or *AutoGenerateColumns* properties to false. In this case, you must specify the fields by adding Field definitions to the *Fields* or *Columns* collections.

There are seven types of fields, each derived from the abstract *DataControlField* class, which you can define as:

Name	Description
BoundField	Displays the bound data as text when viewing or as a *TextBox* when in edit mode.
ButtonField	Displays a button that, when clicked, raises the *RowCommand* event. The *ButtonType* property is used to specify the type of button (*Button*, *Image*, *Link*) and the *CommandName* property is used to differentiate between different buttons in the same row.
CheckBoxField	Displays a Boolean field from the data source as a *CheckBox*. Displayed as read-only unless in edit mode.
CommandField	Displays a set of buttons that contains buttons as appropriate. May raise specific events (such as *RowDeleting* and *RowDeleted* for the delete command) but always raises the *RowCommand* event as well.
HyperLinkField	Displays a *HyperLink*. The *Text* property can be set to show the same text for the link in all rows, or you can use the *DataTextField* to specify the field to display to the user and the *DataTextFormatString* to format the output. The *NavigateUrl* property can be set to use the same link for every row, or you can use the *DataNavigateUrlFields* to specify the fields to use to build the URL and the *DataNavigateUrlFormatString* to format the URL correctly.
ImageField	Display an *Image*. You specify the image to display by setting the *DataImageUrlField*; this can be formatted using the *DataImageUrlFormatString*.
TemplateField	If the preset options don't fit your needs, then you can define your own template to output the data in whatever format you require.

Using Templates to Show Data

When the preset options in a *DetailsView* or *GridView* don't fit your requirements or you're using a *FormView*, then you must use templates to format the data to your requirements.

Within a *TemplateField* (for the *DetailsView* or *GridView*) or directly within the *FormView*, you define templates for each of the different sections of the output. You must, at a minimum, define an *ItemTemplate* but you can also define several other templates.

The *GridView* supports the following templates for the *TemplateField* column:

Name	Description
AlternatingItemTemplate	Defines the template for every other row in the *GridView*, making it easy to distinguish between the rows in the grid.
EditItemTemplate	The template displayed when the row is in Edit mode. Used to display controls (such as a *DropDownList*) that can't be rendered automatically.
FooterTemplate	The content to be displayed as the column footer.
HeaderTemplate	The content to be displayed as the column header.
ItemTemplate	Defines the template for showing the data returned.

You can use the following templates in a *TemplateField* of the *DetailsView* control:

Name	Description
EditItemTemplate	The template displayed when the *DetailsView* is in Edit mode. Used to display controls (such as a *DropDownList*) that can't be rendered automatically.
HeaderTemplate	The content to be displayed as the row header (as the first column in the row).
InsertItemTemplate	The template displayed when the *DetailsView* is in Insert mode.
ItemTemplate	Defines the template for showing the data returned.

You can use the following templates in the *FormView* control:

Name	Description
EditItemTemplate	The template displayed when the *FormView* is in Edit mode.
FooterTemplate	The content to be displayed as the footer of the control after the "data" template.
HeaderTemplate	The content to be displayed as the header of the control before the "data" template.
InsertItemTemplate	The template displayed when the *FormView* is in Insert mode.
ItemTemplate	Defines the template for showing the data returned.

Showing Data in the Template

When using a template you must manually bind to the data source in question. There are two methods of doing this:

- If you only need to show the field from the data source, then you can use the *Eval* method. You would use this method in an *ItemTemplate* or *AlternatingItemTemplate*, or if you wanted to show the column as read-only within the *EditItemTemplate*.
- If you want the data to be editable, then you can use the *Bind* method. You would use this method in an *EditItemTemplate* or *InsertItemTemplate* where the user's entry needs to be populated back to the database.

Display Data using Hierarchical Data Bound Controls

The Menu Control

The *Menu* control is used to display hierarchical data as a menu and is often used in conjunction with a *SiteMapDataSource* for navigating a web site.

The items to be displayed are returned from the *Items* collection as a collection of *MenuItem* objects. Each *MenuItem* has a *ChildItems* collection that also returns a collection of *MenuItem* objects.

The *Menu* control can be populated via code, by specifying the items in markup, or by binding to a data source. Any data source derived from the abstract *HierarchicalDataSourceControl* (such as the *SiteMapDataSource* or *XmlDataSource*) can provide the data to the Menu; you can also bind to an *XmlDocument* or to a *DataSet* (provided it has relationships defined).

An example of the *Menu* control is shown in **Figure 7**.

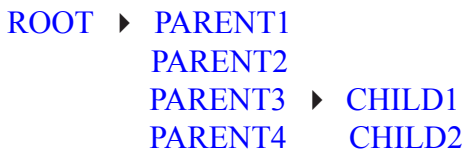

Figure 7 – An example *Menu* control

The TreeView Control

The *TreeView* control is used to display hierarchical data as a tree and can be bound to any data source controls derived from *HierarchicalDataSourceControl*. When used with a *SiteMapDataSource* the *TreeView* provide site navigation.

The items to be displayed are returned from the *Nodes* property as a collection of *TreeNode* objects. Each *TreeNode* has a *ChildNodes* property that also returns a collection of *TreeNode* objects. Each *TreeNode* has a *Text* property that is shown to the user and a Value property that is posted back to the server. *NavigateUrl* is used to specify the URL to be viewed when the node is clicked. If no *NavigateUrl* is specified then clicking the node will cause the node to be selected, raising the *SelectedNodeChanged* event when the page is automatically posted back to the server.

The *TreeView* control can be populated via code, by specifying the items in markup or by binding to a data source.

An example of the *TreeView* control is shown in Figure 8.

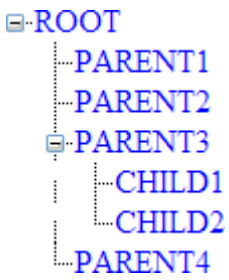

Figure 8 – An example *TreeView* control

Manage Connections and Transactions of Databases

The ADO.NET Data Provider Model

ADO.NET introduces the Data Provider Model for accessing databases. This is an abstract model that provides a common base for all database interactions. There are several different classes, as we'll discover shortly, that all inherit from the same base classes. This allows us, theoretically, to write provider-neutral code.

The ADO.NET Providers

There are four providers installed with the .NET Framework 2.0. These are as follows:

- System.Data.Odbc – for connecting to data sources using ODBC drivers.
- System.Data.OleDb – for connecting to data sources using OLE DB drivers.
- System.Data.OracleClient – for connecting to Oracle 8i Release 3 or later.
- System.Data.SqlClient – for connecting to SQL Server 7 or later

You can return the details of the ADO.NET providers installed on a machine by using the static *GetFactory-Classes* method of the *DbProviderFactories* class:

```
DataTable factories = DbProviderFactories.GetFactoryClasses();
```

This method returns a *DataTable* that contains a *DataRow* for each provider containing the following information:

- Name
- Description
- InvariantName (this is the name used to identify the provider)
- AssemblyQualifiedName (to fully identify the factory class)

Enumerating through Specific Providers

It is also possible to query a provider to return all of the available data sources. For SQL Server, this returns all of the databases running on the current network:

```
DbProviderFactory factory = ~CCC
   DbProviderFactories.GetFactory("System.Data.SqlClient");
DataTable databases = factory.GetCreateDataSourceEnumerator().~CCC
  GetDataSources();
```

For SQL Server, this returns a *DataTable* that contains a *DataRow* for each database containing the following information:

- ServerName
- InstanceName
- IsClustered
- Version

Connection Strings in Web.config

You can write database connection strings and include them on every page. It is far better, from a maintenance point of view, to store the connection string in Web.config:

```
<connectionStrings>
 <add name="connString" providerName="System.Data.SqlClient"
  connectionString="Server=server; Database=database; Uid=user; ~CCC
   Pwd=password;" />
</connectionStrings>
```

You give each connection a *name* and use the *providerName* property to specify the correct ADO.NET provider to use. The *connectionString* contains the provider specific connection string.

The sheer number of different connection strings makes it impossible to list any here. Have a look at http://www.connectionstrings.com/ for a comprehensive list.

It is then possible to access the centralized connection string's details by using a static method of the *ConfigurationManager* object. The *ConnectionStrings* property is indexed on the name of the connection string and returns a *ConnectionStringSettings* object that you can interrogate to return *Name*, *ProviderName* or the actual *ConnectionString*:

```
dbConn.ConnectionString = ConfigurationManager.~CCC
  ConnectionStrings["connString"].ConnectionString;
```

Securing Connection Strings

Connection strings stored in Web.config contain login information that you don't want anyone to see. It is easy to encrypt parts of a Web.config file using the *aspnet_regiis.exe* command line tool.

When encrypted the connection string can be used as normal by your Web site. It is however non-readable to the casual observer.

To encrypt the <*connectionStrings*> element of Web.config you run the following command from the command line:

aspnet_regiis –pef "connectionStrings" "C:\SITE\"

You specify the element within Web.config that you want to encrypt and the full path to the root of the Web site.

You can decrypt the <*connectionStrings*> element in a similar manner:

aspnet_regiis –pdf "connectionStrings" "C:\SITE\"

The DbConnection Object

Connections to databases are managed by classes derived from *DbConnection*. You access a database using a specific ADO.NET data provider:

- OdbcConnection
- OleDbConnection
- OracleConnection
- SqlConnection

Because all the specific classes inherit from *DbConnection*, they all implement a common set of properties and methods.

Creating a database connection is accomplished by instantiating a new instance of the required data provider. You can either pass the connection string to the constructor (which is taken from Web.config) or set the *ConnectionString* property as we have here:

```
SqlConnection dbConn = new SqlConnection();
dbConn.ConnectionString = ConfigurationManager.~CCC
   ConnectionStrings["connString"].ConnectionString;
```

The connection has been created but is not yet open. To open the connection, you call the *Open* method:

```
dbConn.Open();
```

After performing work you then close the connection using the corresponding *Close* method:

```
dbConn.Close();
```

You should only open the connection to the database when you need to and close it as soon as you're finished with it.

Connection Exceptions

All database exceptions inherit from the abstract *DbException* class and there are specific providers for each ADO.NET data provider:

- OdbcException
- OleDbException
- OracleException
- SqlException

You can wrap all your database code in a *try/catch* block and catch either the specific exception, e.g. *SqlException*, or the abstract *DbException* class.

Connection Events

All of the standard ADO.NET data providers implement an *InfoMessage* event that you can use to listen for various events from the connection. You can attach an event handler to this event. The arguments parameter to the event handler depends upon the ADO.NET data provider:

- OdbcInfoMessageEventArgs
- OleDbInfoMessageEventArgs
- OracleInfoMessageEventArgs
- SqlInfoMessageEventArgs

The arguments class is specific to each ADO.NET data provider and there is no common base class.

Connection Pooling

Creating and opening connections to databases is an expensive process. Connection pooling reduces this overhead by making a "pool" of available connections. Pooling is controlled by parameters placed into the database connection string. Pooling, if provided by the underlying data source, is enabled by default:

- OdbcConnection – connection pooling is controlled at the ODBC driver level and is not handled within the data provider.
- OleDbConnection – connection pooling is used if the underlying OLE DB provider supports it.
- OracleClient – all Oracle database connections are pooled by default.
- SqlClient – all SQL Server database connections are pooled by default.

When a request for a connection to a database is made by calling the *Open* method, the connection pool is checked to see if any existing connections are available. If a connection is available, it is returned to the caller.

If no connections are available, and the maximum pool size has not been reached, a new connection is made and returned to the caller.

If no connection is available, and the maximum pool size has been reached, the connection is added to the queue and will wait until a connection becomes available. If no connection becomes available before the connection timeout has been reached, (the *ConnectionTimeout* property of DbConnection) an exception is thrown.

Though a pooled connection is closed by calling the *Close* method, the connection itself is not actually closed. Instead, it is released back to the connection pool and reused.

The DbTransaction Object

Every action performed against the database runs in its own transaction. All actions on the same connection can be enlisted in the same transaction (i.e. all actions complete or all actions fail) by making the connection transactional.

Transactions are started by calling the connection's *BeginTransaction* method:

```
SqlTransaction dbTran = dbConn.BeginTransaction();
```

This returns an ADO.NET data provider specific transaction, derived from the *DbTransaction* base class:

- OdbcTransaction
- OleDbTransaction
- OracleTransaction
- SqlTransaction

Any interactions with the connection are then contained within the same transaction, provided the command object has its *Transaction* property set to the connection's transaction:

```
dbComm.Transaction = dbTran;
```

Once the transaction is complete, you must call the *Commit* method, otherwise all changes will be rolled back:

```
dbTran.Commit();
```

You can also manually roll back a transaction (if for instance an exception occurs) by explicitly calling the *Rollback* method:

```
dbTran.Rollback();
```

Create, Delete and Edit Connected Data

The DbCommand Object

All transactions with the database occur through a command object. All command objects are inherited from *DbCommand* and there is a command object specific to each ADO.NET data provider:

- OdbcCommand
- OleDbCommand
- OracleCommand
- SqlCommand

You can create a command object and connect it to the correct database in two ways:

- By creating a new instance and setting the *Connection* property.
- By calling the *CreateCommand* method of the connection object. This automatically populates the command's *Connection* and *Transaction* properties.

In both cases, you'll have a command object that you can then use to interact with the database. Before you can use the command object, you must ensure the connection to the database is open.

The DbParameter Object

Parameters allow you to pass values to the queries that you execute at runtime. A command object has a *Parameters* property that returns a collection of parameter objects. Each ADO.NET data provider has its own specific parameter object:

- OdbcParameter
- OleDbParameter
- OracleParameter
- SqlParameter

You should always use parameters to pass values to your query, whether it is a SQL query or a stored procedure. Not doing so, especially in the case of SQL queries, leads to the possibility of SQL injection attacks.

Executing Database Queries

Before you can query the database using one of the Execute methods you need to configure at least two other properties of the command:

- *CommandText* – the query to execute. This can be a SQL query or a stored procedure.
- *CommandType* – specifies what type of query the *CommandText* is. It defaults to *Text* indicating that you're passing a SQL query. Setting it to a value of *StoredProcedure* indicates that you're passing the name of a stored procedure.

Once you have configured the command object correctly, you have three common methods for executing the specified query:

Name	Description
ExecuteNonQuery	Use this when the query you're executing has nothing to return. DELETE, INSERT and UPDATE queries are commonly performed using this method.
ExecuteReader	Executes the query against the database and returns the results of the query as a *DbDataReader*.
ExecuteScalar	Use this when the query you're executing only returns a single value. Returns an *Object* that you cast to the correct type.

In addition, the *SqlCommand* object has an *ExecuteXmlReader* method that returns an *XmlReader* object. This is useful for when you are using the FOR XML extension in SQL Server 2000 and above.

Using the DbDataReader Object

The *DbDataReader* object provides a connected way to retrieve data from the database. The *ExecuteReader* method returns the correct *DbDataReader* for the connection. As with all the ADO.NET objects there is a specific object, derived from *DbDataReader*, for each ADO.NET provider:

- OdbcDataReader
- OleDbDataReader
- OracleDataReader
- SqlDataReader

The returned *DbDataReader* may not contain any rows of data and this can be checked using the *HasRows* property.

The *DbDataReader* is a forward-only, read-only view of the data. You call the *Read* method to advance to the first row in the results and also call the *Read* method to advance to subsequent rows. The easiest way to accomplish this is in a *while* loop, as follows:

```
while (dbReader.Read())
{
  // do something with the row of data
}
```

You can access the fields of the current row in several ways:

- By using an indexer on the reader, e.g. dbReader[0]. This returns an *Object* that you must cast to the correct type.
- By using the name of the field, e.g. dbReader["ID"]. This returns an *Object* that you must cast to the correct type.
- By using one of the Get helper methods in conjunction with the *GetOrdinal* method, e.g. dbReader.GetGuid(dbReader.GetOrdinal("ID")). The Get helper methods only accept indexers, so you must use the GetOrdinal method to return the index of the required field. This returns objects of the correct type.

When using a *DbDataReader* the connection to the database is in use for the entire duration of the access.

Asynchronous Operations

Normally queries are synchronous – the required Execute method is called and the call is blocked until the data is returned. When using SQL Server, you can also perform the same operation asynchronously.

There are three asynchronous operations of the *SqlCommand* object:

- BeginExecuteNonQuery
- BeginExecuteReader
- BeginExecuteXmlReader

Each of these methods is overloaded to provide a callback mechanism or by polling using *WaitHandle* objects.

If you're using polling you call the Begin method, call the *WaitOne* method on the *WaitHandle* and call the corresponding End method to return the results of the asynchronous operation:

```
IAsyncResult async1 = dbComm.BeginExecuteReader();
// do other things
async1.AsyncWaitHandle.WaitOne();
SqlDataReader dbReader = dbComm.EndExecuteReader(async1);
```

The End method will return the same value as the non-synchronous Execute call (e.g. *EndExecuteReader* returns a *SqlDataReader*, the same as *ExecuteReader*).

Bulk Copy with SqlBulkCopy

Transferring data from one source to another can be resource intensive. When copying to SQL Server you have a high-performance option – the *SqlBulkCopy* object.

The *SqlBulkCopy* object is used to wrap a *SqlConnection* object, and provides a *WriteToServer* method that accepts objects of the following type:

- An array of *DataRow* objects.
- A *DataTable*.
- Any *DbDataReader* derived class.

So you can copy from an Oracle database by returning the results to copy as an *OracleDataReader* and then passing the *OracleDataReader* to the *WriteToServer* method.

The *SqlBulkCopy* object has several properties that control the way the copy occurs. *DestinationTable-Name* can be used to set the destination table in the SQL Server database, *BatchSize* allows you to specify how many records are copied at a time, and using the *ColumnMappings* collection you can map columns between the source and destination.

Create, Delete and Edit Disconnected Data

The DataSet Object

The *DataSet* is an in-memory representation of data and is disconnected from the database. It contains a collection of *DataTable* objects (the *Tables* property) and relationships between the *DataTable* objects are stored as a collection of *DataRelation* objects (the *Relations* property).

Unlike most of the objects used for data access, the *DataSet* object is generic – it will happily hold data from any ADO.NET data provider and can hold data from different providers at the same time. You can populate a *DataSet* manually or from an ADO.NET data provider using a *DbDataAdapter* object. You must always create a DataSet before you can populate it:

```
DataSet dsData = new DataSet();
```

The DataTable, DataColumn and DataRow Objects

The *DataTable* object represents a table of data and contains a collection of *DataColumn* objects (the *Columns* property), defining the structure of the table, and a collection of *DataRow* objects (the *Rows* property) holding the actual data.

You can use a *DbDataAdapter* to create and populate a *DataTable*. This creates both the structure (*DataColumn* objects) and content (*DataRow* objects).

You can manually create a *DataTable* and add it to the *Tables* collection of a *DataSet*:

```
DataTable dtPlayers = new DataTable("Players");

    dsData.Tables.Add(dtPlayers);
```

You also need to define the *DataColumn* objects for the *DataTable*:

```
DataColumn dcName = new DataColumn("Name");
dcName.DataType = System.Type.GetType("System.String");
dcName.MaxLength = 64;
dtPlayers.Columns.Add(dcName);
```

You can then add *DataRow* objects to the *DataTable*:

```
DataRow drPlayer = dtPlayers.NewRow();
drPlayer["Name"] = "Alan Shearer";
dtPlayers.Rows.Add(drPlayer);
```

The DataRelation Object

The *DataRelation* object is used to specify the relationship between two *DataTable* objects (which can be from different data sources).

To add a *DataRelation* to a *DataSet* you need to have references to the columns that make up the relationship in both the parent and child tables. You can then use these to create the *DataRelation* and add it to the *Relations* of the *DataSet*:

```
// get the columns in relationship
DataColumn dcParent = dsData.Tables["Clubs"].Columns["ID"];
DataColumn dcChild = dsData.Tables["Players"].Columns["ClubID"];
// create the relationship
DataRelation drPlaysFor = new DataRelation("PlaysFor", dcParent, dcChild);
// add to the data set
dsData.Relations.Add(drPlaysFor);
```

You can navigate between the parent and child relationships by using the *GetParentRow* and *GetChildRows* methods of a *DataRow*.

Binding to a DataSet

Because a *DataSet* can contain more than one *DataTable*, you have two options when using a *DataSet* as a data source for a Web Server Control:

- Set the *DataSource* property to be the specific *DataTable*.
- Set the *DataSource* to the *DataSet* and set the *DataMember* property to the name of the specific *DataTable*.

Copy the Contents of a DataSet

You cannot copy an entire *DataSet*, but you can copy a *DataTable*. To copy both the structure and data of the *DataTable* use the *Copy* method:

```
DataTable dtCopy = dtOriginal.Copy();
```

If you only want to copy the structure of the *DataTable*, use the *Clone* method:

```
DataTable dtClone = dtOriginal.Clone();
```

The Strongly Typed DataSet

Using the *DataSet* object is prone to errors, as there is no compile time checking of table names, column names, etc. You can create a typed *DataSet* using the DataSet Editor within Visual Studio. Select **Add New Item** for the Web site and select the DataSet template.

You can use the graphical designer to add *DataTable* objects to the *DataSet,* define the relationships between the *DataTable* objects, and define any extra methods that you want to return data. An example of a typed DataSet is shown in **Figure 9**.

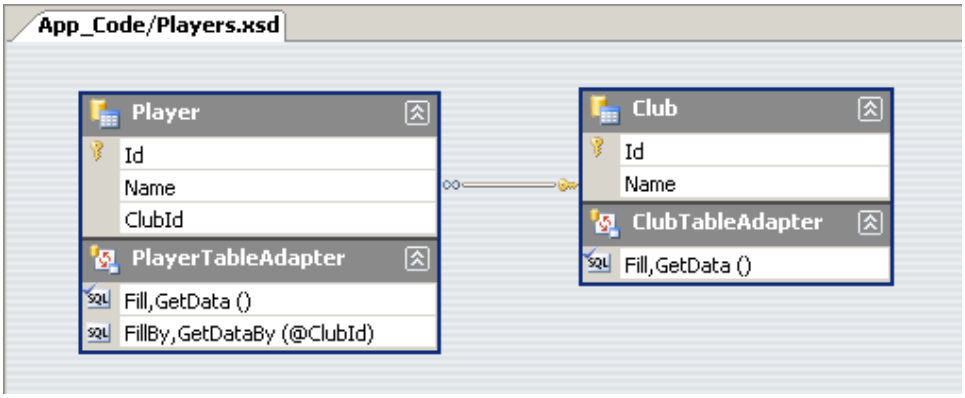

Figure 9 – A typed DataSet in the DataSet Designer

You can then return a typed *DataTable* by using the *TableAdapter* and the correct Get method to return a typed *DataTable* object:

```
PlayerTableAdapter taPlayers = new PlayerTableAdapter();
Clubs.PlayerDataTable dtPlayers = taPlayers.GetDataByClub(1);
```

Each *DataRow* object within the typed *DataTable* is also typed (in this case, as a *PlayerDataRow*) and they have properties for the *DataColumn* objects. So the following is perfectly valid:

```
PlayerDataRow dtPlayer = dtPlayers.Rows[0];
lblPlayerName.Text = dtPlayer.Name;
```

Using the DbDataAdapter Object

Returning a DataTable from a Database

To return data from a database into a *DataSet*, you need to use a *DbDataAdapter* derived object. There are specific ADO.NET data provider versions of the *DbDataAdapter*:

- OdbcDataAdapter
- OleDbDataAdapter
- OracleDataAdapter
- SqlDataAdapter

You create the correct version of the *DbDataAdapter* and then set its *SelectCommand* property to the correctly configured *DbCommand* object. You can then call the *Fill* method to populate a *DataTable* in the *DataSet*:

```
SqlDataAdapter daPlayers = new SqlDataAdapter();
daPlayers.SelectCommand = dbComm;
daPlayers.Fill(dsData,"Players");
```

Modifying a DataTable in Memory

When you have a *DataTable* in memory you are free to modify that data, and add new data to the *Table*, as you see fit. As we'll see shortly, you can also propagate those changes back to the database.

When you're editing a *DataRow*, you can freely edit the contents of each row. However, if you're going to perform several modifications to the same row, you should call the *BeginEdit* method on the *DataRow* to switch into Edit mode. This suspends any events that may be raised (such as validation rules failing). When editing is finished, you call the *EndEdit* method to save the changes to the *DataRow* or the *CancelEdit* method to abort any changes that have been made.

Each *DataRow* has a *RowState* property that indicates what needs to be done to propagate the changes made to it back to the database. It can be one of the following values of the *DataRowState* enumeration:

Name	Description
Added	The *DataRow* has been added to the *DataTable* and needs to be propagated to the database (using an INSERT query).
Deleted	The *DataRow* has been deleted (by calling its *Delete* method) and needs to be propagated to the database (using a DELETE query).
Detached	The *DataRow* has been created but not yet added to the DataTable. It will not be propagated to the database.
Modified	The *DataRow* has been modified and needs to be propagated to the database (using an UPDATE query).
Unchanged	The *DataRow* has not been modified. There is nothing to propagate to the database.

In addition to each *DataRow* having a *RowState*, it may also have several versions of itself in the *DataTable*. By default, when you retrieve the value of a property from a *DataRow*, you return the current value; it is also possible to return different versions of the *DataRow* by specifying the required version from the *DataRowVersion* enumeration:

Name	Description
Current	The current value in the *DataRow*. If the *DataRow* has a *RowState* of *Deleted*, attempting to retrieve this version throws an exception.
Original	The value that populated the *DataRow*. If the *DataRow* has a *RowState* of *Added*, attempting to retrieve this version throws an exception.
Proposed	This is the value as the *DataRow* is being edited. If not in Edit mode, then an exception is thrown.

So, to return the original version of a modified *DataRow*, you specify *Original*:

```
if (drPlayer.RowState == DataRowState.Modified)
{
  string strOriginalName = (string)drPlayer["Name", DataRowVersion.Original];
}
```

Updating a DataTable to the Database

The *DbDataAdapter* can also be used to propagate changes back to the database. Where you use the *Fill* method object to return data from the database, you use the *Update* method to propagate changes back to the database.

In order to update the database, the *DbDataAdapter* needs to know the DELETE, INSERT and UPDATE queries, as well as the SELECT query that it used to populate the *DataTable*.

These are contained in the following properties:

- DeleteCommand
- InsertCommand
- SelectCommand
- UpdateCommand

The *SelectCommand* property of the *DbDataAdapter* is used to retrieve data from the data source when you call the *Fill* method. To propagate changes to the data source when you call the *Update* method, you can manually set the *DeleteCommand*, *InsertCommand* and *UpdateCommand* properties to command objects that will perform the required tasks.

You can also automatically create these extra commands using a *DbCommandBuilder* object (with a specific one for each ADO.NET data provider). Using one is as easy as creating an instance of the *DbCommandBuilder* and specifying the *DbDataAdapter* that you're using:

```
SqlCommandBuilder cbPlayers = new SqlCommandBuilder(daPlayers);
```

The commands to DELETE, INSERT and UPDATE will be created in the *DbDataAdapter* as required. You can also retrieve the created *DbCommand* objects directly from the *DbCommandBuilder* using the *GetDeleteCommand*, *GetInsertCommand* and *GetUpdateCommand* methods.

The DataView Object

The *DataView* object is a window onto a *DataTable* and can be sorted and filtered using the *Sort*, *RowFilter* and *RowStateFilter* properties. A *DataView* implements the *IEnumerable* interface so it can be used as a data source for data binding.

You create a new *DataView* by setting the *Table* property of the *DataView* object to the corresponding *DataTable* or passing the *DataTable* to the constructor:

```
DataView dvPlayers = new DataView(dtPlayers);
```

You can sort the *DataView* by setting the *Sort* property (as you would for an ORDER BY clause in SQL):

```
dvPlayers.Sort = "ClubId DESC, Name";
```

You can also filter the DataView by setting the RowFilter property (as you would for a WHERE clause in SQL):

```
dvPlayers.RowFilter = "Name LIKE 'A%'";
```

You can also filter based on the state of the row using the RowStateFilter property:

```
dvPlayers.RowStateFilter = DataViewRowState.Added;
```

Serializing and Deserializing DataSet Objects

You can serialize and deserialize an entire *DataSet*, or a single *DataTable*, to and from XML using the *WriteXml* and *ReadXml* methods. There are multiple overloads for these methods that allow you to use files or streams to store the XML.

Manage XML Data with the XML Document Object Model

The *XmlNode* is an abstract class that forms the basis of all Document Object Model (DOM) interactions. All of the objects that you will use with the DOM are derived from this base class.

You normally start your interaction with XML using an *XmlDocument* class and this is instantiated as follows:

```
XmlDocument xmlManagers = new XmlDocument();
```

Loading XML into an XmlDocument

You can then load, if you require, an XML document from a stream, file, *TextReader* or *XmlReader* by using the *Load* method. For example, to load an *XmlDocument* from a file:

```
xmlManagers.Load("C:\managers.xml");
```

Searching and Navigating

The *XmlNode* base class provides several properties that can be used for navigation:

Name	Description
ChildNodes	Returns an *XmlNodeList* containing all the XmlNode objects that are children of the current object.
FirstChild	Returns the first child of the current *XmlNode* as an *XmlNode*. If there are no children of the current *XmlNode*, this returns null.
LastChild	Returns the last child of the current *XmlNode* as an *XmlNode*. If there are no children of the current *XmlNode*, this returns null.
NextSibling	Returns the next *XmlNode*. If there is no next *XmlNode*, this returns null.
PreviousSibling	Returns the previous *XmlNode*. If there is no previous *XmlNode*, this returns null.
ParentNode	Returns the parent of the current *XmlNode*. If there is no parent node, this returns null.

It is also possible to search an *XmlDocument* using the following two methods:

Name	Description
GetElementById	Returns an *XmlElement* with the specified ID (the ID must be defined in an associated DTD document).
GetElementsByTagName	Returns an *XmlNodeList* containing all elements in the *XmlDocument* that have the specified name.

Additionally, it is possible to search any *XmlNode* derived class using XPath using either of the following methods:

Name	Description
SelectNodes	Returns an *XmlNodeList* containing all nodes that match the XPath expression.
SelectSingleNode	Returns the first node that matches the XPath expression.

Modifying an XmlNode

Modifying the children of an *XmlNode* is accomplished using the following methods:

Name	Description
AppendChild	Adds the new node to the end of the child nodes.
InsertAfter	Adds the new node to the child nodes after the specified child node.
InsertBefore	Adds the new node to the child nodes before the specified child node.
PrependChild	Adds the new node at the start of the child nodes.
RemoveAll	Removes all the child nodes.
RemoveChild	Removes the specified node from the child nodes.
ReplaceChild	Replaces the specified child node with the new node.

You cannot directly create a new *XmlNode*. You must use one of the *Create* methods of the *XmlDocument*:

Name	Description
CreateCDataSection	Creates an *XmlCDataSection* object that can be added to an *XmlElement*.
CreateComment	Creates an *XmlComment* object that can be added to an *XmlDocument* or *XmlElement*.
CreateElement	Creates an *XmlElement* with the specified name that can be added to *XmlDocument* and *XmlElement* nodes.
CreateTextNode	Creates an *XmlTextNode* that can be added to an *XmlElement*.

Modifying the Attributes of an XmlElement

The *XmlElement* is slightly different than the normal *XmlNode*. In addition to being able to have children, it can also have attributes. These are handled slightly differently as compared to other *XmlNode* objects. The *XmlElement* has an *Attributes* property that returns an *XmlAttributeCollection*. This is a collection of the *XmlAttribute* objects for the element. You can retrieve a specific *XmlAttribute* by specifying the name of the attribute or the index to the *Attributes* property:

```
XmlAttribute xmlName = xmlPlayer.Attributes["Name"];
```

You must use the methods of the *XmlAttributeCollection* class to modify the attributes for the *XmlElement*:

Name	Description
Append	Adds the new attribute at the end of the attributes.
InsertAfter	Adds the new attribute after the specified attribute.
InsertBefore	Adds the new attribute before the specified attribute.
Prepend	Adds the new attribute at the start of the attributes.
Remove	Removes the specified attribute.
RemoveAll	Removes all the attributes.

Saving an XmlDocument to XML

You can save an *XmlDocument* in the same way you can load it – to a stream, file, *TextWriter* or *XmlWriter*. For example to save an *XmlDocument* to a file:

```
xmlManagers.Save("C:\managers.xml");
```

Read and Write Xml Data Using XmlReader and XmlWriter

The XmlReader Object

The *XmlReader* object is the abstract base class of the three derived classes that allow us to work with XML documents:

Name	Description
XmlNodeReader	Provides non-cached, forward-only access to an XmlNode.
XmlTextReader	Provides a non-cached, forward-only way to read an XML document.
XmlValidatingReader	Validates an existing XmlReader against the specified schema (a DTD, XDR or XSD document).

Read Xml Data using the XmlReader

The *XmlReader*, although abstract, has a static *Create* method that allows you to specify a stream, file or *TextReader* that it will use to create a valid *XmlReader* of the correct type.

You can configure the *XmlReader* using the *XmlReaderSettings* class that is optionally passed as a parameter to the *Create* method.

In its simplest form, you can create an *XmlReader* and iterate it through all the nodes in the document as follows:

```
XmlReader xmlRead = XmlReader.Create("C:\managers.xml");
while(xmlRead.Read())
{
        // do what we need to
}
```

The *XmlReader* has properties to determine the node type (*NodeType*), the name of the node (*Name*) the value of the node (*Value*), and the number of attributes (*AttributeCount*). You can use the indexer (by index or name) to return the value of the attributes of the node.

Read XML Data using the XmlTextReader

The *XmlTextReader* is a derived version of the *XmlReader*. The recommended practice is to use the *Create* method of *XmlReader* to access XML documents. It is, however, still possible to create an *XmlTextReader* manually to read from a stream, file or *TextReader*.

You can create an *XmlTextReader* (assuming we want to read from a file) as follows:

```
XmlTextReader xmlRead = new XmlTextReader("C:\managers.xml");
```

Because *XmlTextReader* derives from *XmlReader* you access the child nodes in exactly the same way:

```
while(xmlRead.Read())
{
        // do what we need to
}
```

Read Nodes using the XmlNodeReader

The *XmlNodeReader* is used to iterate over the children of an *XmlNode*. You create an *XmlNodeReader* object by specifying the *XmlNode* you want to iterate over:

```
XmlNodeReader xmlRead = new XmlNodeReader(xmlPlayers);
```

Because *XmlNodeReader* derives from *XmlReader*, you read the child nodes in exactly the same way:

```
while(xmlRead.Read())
{
        // do what we need to
}
```

Validating XML Documents

The *XmlReader* can be configured to provide validation (by setting the *ValidationType* property of the *XmlReaderSettings*). It is also possible to use the obsolete *XmlValidatingReader* to validate an XML document.

You create a validating *XmlReader* as follows:

```
XmlReaderSettings xmlSettings = new XmlReaderSettings();
xmlSettings.ValidationType = ValidationType.DTD;
XmlReader xmlValidRead = XmlReader.Create(xmlRead, xmlSettings);
```

You normally create an *XmlValidatingReader* by passing in an existing *XmlReader* derived class and specifying the type of validation in the *ValidationType* property:

```
XmlValidatingReader xmlValidRead = new XmlValidatingReader(xmlRead);
xmlValidRead.ValidationType = ValidationType.DTD;
```

For both the *XmlReader* and *XmlValidatingReader*, validation of the XML occurs during a call to the *Read* method. If the *XmlReader* encounters an exception, it throws an *XmlSchemaValidationException*. If the *XmlValidatingReader* encounters a validation error, it throws an *XmlException*. You can override this behavior by catching the validation events in a *ValidationEventHandler*.

You catch the validation events for the *XmlReader* by attaching an event handler to the *XmlReader Settings* class:

```
xmlSettings.ValidationEventHandler += new ValidationEventHandler ~CCC
  (this.ValidationEvent)
```

For the *XmlValidatingReader*, you attach the event handler as follows:

```
xmlValidRead.ValidationEventHandler += new ValidationEventHandler ~CCC
  (this.ValidationEvent)
```

By attaching to the *ValidationEventHandler* event, you can catch any validation errors that occur and stop *any exceptions* being thrown (you then have to handle a validation failure manually).

We then have an event handler that will be called when there is validation error:

```
private void ValidationEvent (object sender, ValidationEventArgs args)
{
 // do something with the validation errors
}
```

The *ValidationEventArgs* exception has three properties that we can use to determine what has occurred:

Name	Description
Exception	A reference to the *XmlSchemaException* object which contains detailed information about the exception.
Message	A description of the validation event.
Severity	A value from the *XmlSeverityType* enumeration, either *Error* or *Warning*, which indicates the severity of the validation event.

The XmlWriter Object

The *XmlWriter* object is the abstract base class for writing XML. There is only one derived class:

Name	Description
XmlTextWriter	Provides non-cached, forward-only way to write an XML document.

Write Data using the XmlWriter

The *XmlWriter* is used to write an XML document from scratch. You cannot directly create an *XmlWriter*. Instead, you use the *Create* method to create an instance.

You can configure the *XmlWriter* using the *XmlWriterSettings* class that is optionally passed as a parameter to the *Create* method.

The *XmlWriter* can write to a stream, file, *StringBuilder*, *TextWriter* or an existing *XmlWriter*. In its simplest form, you can create an *XmlWriter* to write to a file as follows:

```
XmlWriter xmlWrite = XmlWriter.Create("C:\managers.xml");
```

You then use various methods of the *XmlWriter* to write nodes to the XML document. The commonly used methods are as follows:

Name	Description
WriteAttributeString	Writes an attribute to the current element with the specified name, value and optional prefix and namespace.
WriteCData	Writes the specified text (as a CDATA block) as a child of the current element.
WriteComment	Writes the specified text (as a comment) as a child of the current element.
WriteElementString	Writes a complete element as a child of the current element with the specified name, value, and optional prefix and namespace. This element can have no attributes or children.
WriteEndElement	Writes the closing element tag to the currently open element.
WriteStartElement	Writes the opening element tag with the specified name and optional prefix and namespace. Once the element is started, any subsequent Write methods will be on the opened element. You must call the *WriteEndElement* method to close the element and move back up the hierarchy.

When writing using the *XmlWriter*, you must remember where you are in the document. If you start an element using the *WriteStartElement* method, you must call the *WriteEndElement* method to step back up a level. Otherwise, any other nodes you write will be children of the element.

Once you have constructed the XML document correctly, you call the *Close* method to close the underlying object. Any open, started elements will be automatically ended in the correct order before the document is closed.

Write Data Using the XmlTextWriter

The *XmlTextWriter* is a derived version of the *XmlWriter*. The recommended practice is to use the *Create* method of *XmlWriter* to access XML documents. It is, however, still possible to create an *XmlTextWriter* that will manually write to a stream, file or *TextWriter*.

You can create an *XmlTextWriter* (assuming we want to write to a file) as follows:

```
XmlTextWriter xmlWriter = new XmlTextWriter("C:\managers.xml",~CCC
  Encoding.Unicode);
```

Because *XmlTextWriter* derives from *XmlWriter*, you can then write to the XML file as you would for *XmlWriter*.

Creating Custom Web Controls

Create a Composite Web Application Control

Create a User Control

User Controls are files in your Web site with an .ascx extension (this ensures that IIS will not render the control directly). You can manually create a file with an .ascx extension in your Web site, or you can use Visual Studio and select **Web User Control** from the **Add New Item** dialog.

User Controls are identified within the .ascx file with the *@Control* directive and can use either the code-beside or code-behind model. The *@Control* directive indicates this.

For code-beside, you have a *ClassName* specified in the directive:

```
<%@ Control Language="C#" ClassName="WebUserControl" %>
```

And for code-behind you have *CodeFile* and *Inherits* attributes in the directive:

```
<%@ Control Language="C#" AutoEventWireup="true"
 CodeFile="WebUserControl.ascx.cs" Inherits="WebUserControl" %>
```

Convert a Web Form to a User Control

There may be occasions when you need to turn a Web Form into a User Control. There is a relatively simple process for this:

1. Remove the <*html*>, <*body*> and <*form*> tags.
2. Change the *@Page* directive to a *@Control* directive.
3. Rename the ASPX file to ASCX.
4. If using code-behind, then rename the ASPX.CS file to ASCX.CS and change the *CodeFile* attribute of the *@Control* directive appropriately.
5. Change the class defined in ASCX.CS to inherit from *UserControl* instead of from the *Page*.

Adding a User Control to a Web Form or a User Control

You can add a User Control to a Web Form, or another User Control, in Visual Studio by dragging it from the Solution Explorer onto the design view of the Web Form. This adds the required markup to the page. You can also manually add a User Control to a Web Form by adding the markup manually. You first need to add a reference to the User Control by adding a *@Register* directive:

```
<%@ Register Src="WebUserControl.ascx" TagName="WebUserControl"
 TagPrefix="uc1" %>
```

The *Src* property identifies the relative path (from the current Web Form) to the User Control. The *TagPrefix* and *TagName* are used to define the markup required to add the User Control to the Web Form. The above tag would be added to the Web Form as follows:

```
<uc1:WebUserControl ID="WebUserControl1" runat="server" />
```

Manipulate User Control Properties

All User Controls inherit from the *System.Web.UI.UserControl* class and as such have several inherited properties that can be manipulated by adding the properties directly to the HTML markup in Source view or by using the Properties window when in Design View.

To add your own properties, you need to add public properties to the User Control. You cannot simply add public variables to a User Control and use these as properties. You must add a property with *get* and *set* accessors:

```
private string m_Name = String.Empty;
public string Name
{
 get
 {
  return(m_Name);
 }
 set
 {
  m_Name = value;
 }
}
```

Handling Events in User Controls

Events can be added to User Controls in the same way as they are added for a Web Form. A User Control understands the page lifecycle and has many of the same events as a Web Form (e.g. *Init*, *Load*, *PreRender*). Web Controls contained within the User Control can also raise their own events. You respond to them in the same way as you would on a Web Form.

Dynamically Loading User Controls

Rather than defining User Controls directly on a Web Form, it is also possible to load User Controls dynamically. The *Page* and *UserControl* classes both have a *LoadControl* method that accepts the relative path to the User Control.

This returns a *Control* instance that you can add to the Controls collection. To add a dynamically loaded control to a Web Form, you can execute the following:

```
Control ctlControl = this.LoadControl("WebUserControl.ascx");
this.Controls.Add(ctlControl);
```

You may also need to cast the returned control to the correct type if you want to set any properties of the User Control:

```
WebUserControl ctlWebUserControl = (WebUserControl) ~CCC
 this.LoadControl("WebUserControl.ascx");
this.Controls.Add(ctlWebUserControl);
```

You also need to add a *@Reference* directive so that the types can be used:

```
<%@ Reference Control="WebUserControl.ascx" %>
```

Create a Templated User Control

Templated User Controls allow a page designer to specify the UI for the User Control. There are a series of steps that you can follow to implement a Templated User Control:

1. Add a User Control to your Web site. This will be the Templated User Control.
2. Add a class to the *App_Code* folder that derives from the *Control* class and implements the *INamingContainer* interface. This is the container class that provides the data for the template and you should have a public property for each item of data you want to make available to the template.
3. Implement a property of the *ITemplate* type in the User Control (created in Step 1). Apply the *TemplateContainer* attribute specifying the container class (created in Step 2) and apply the *PersistenceMode* attribute and set it to *PersistenceMode.InnerProperty*.
4. Add properties to the User Control that allow you to specify the data that is to be available to the template via the container class.
5. In the *Page_Init* method of the User Control, first check that there is a template specified. If there is, then create a new instance of the data container, set the properties on it as required, and pass the data container to the *InstantiateIn* method of the *ITemplate*. Add the data container to the *Controls* collection.

Use the Templated User Control

You've already seen how to use templated controls when looking at data controls, earlier. Templated User Controls are no different.

You need to define the template as part of the User Control declaration. So, if we have an *ITemplate* property (see Step 3 above) of UsersTemplate, then you define the User Control as follows:

```
<uc1:WebUserControl ID="WebUserControl1" runat="server">
  <UsersTemplate>
    Name: <%# Container.Name %>
  </UsersTemplate>
</uc1:WebUserControl>
```

Within the UsersTemplate you can define whatever HTML markup you require. To access the data container (see Step 2 above), you use inline data-binding to access the *Container* object and any properties, in this case Name, that are defined on it.

In order for the data binding to occur, you must call the *DataBind* method of the Templated User Control.

Create a WebControl Derived Custom Control

Create a Custom Web Control

User Controls are specific to the Web site in which they are defined. Alternatively, you can create a control that derives from *WebControl* in a class library and reuse that control in multiple Web sites.

You create a Custom Web Control by creating a class that inherits from the *WebControl* class. You then add properties to the derived class that you can use when rendering the control.

Unlike User Controls, there is no design surface for Custom Web Controls and you must override the *Render* method and manually write the HTML markup for the control directly to the *HtmlTextWriter* object passed to the *Render* method.

It is possible, however, to inherit from any Web Control (since they all derive from *WebControl*) to use as the basis of a Custom Web Control. If you inherit from the *TextBox* class, you will get all the properties and rendering available to it. You change the rendering of the control in the *Render* method override:

```
protected override void Render(HtmlTextWriter output)
{
  // add whatever markup is required before the TextBox
  base.Render(output);
  // add whatever markup is required after the TextBox
}
```

Adding a Custom Web Control to the Toolbox

You can manually add Custom Web Controls to a Web Form or User Control by creating an instance of the Custom Web Control, setting the properties and then adding it to the *Controls* collection. This quickly becomes tedious.

You can, instead, add Custom Web Controls to the Visual Studio toolbox, provided the control is contained within a class library.

To add the control to the Toolbox, select the Choose Items options from the Tollbox's context menu. Browsing to the class library allows you to add the controls within it to the Toolbox.

Custom Web Controls added to the toolbox use a default image, but it is possible to use your own image by applying the *ToolboxBitmap* attribute to the class definition. This allows you to specify a 16x16 bitmap that is used and can be a file on disk or a resource within the class library.

Individualize a Custom Web Control

When your custom control is placed onto a Web Form or User Control, it is rendered using a standard declaration:

```
<cc1:WebCustomControl ID="WebCustomControl1" runat="server" />
```

It is possible to change the markup that is used using the *ToolboxData* attribute. To change the HTML markup to always add a Name attribute, you would add the following *ToolboxData* attribute:

```
[ToolboxData(@"<{0}:WebCustomControl runat=""server"" Name="""" />")]
public class WebCustomControl : System.Web.UI.WebControl
```

Within the HTML markup {0} is a placeholder for the TagPrefix. By default, this is cc suffixed with a number. It is possible to change this prefix by adding the TagPrefix attribute:

```
[assembly: TagPrefix("WebCustomControls","wcc")]
```

You must specify the namespace containing the controls and the prefix that you want to use. As it is an assembly attribute, the prefix will apply to all controls within the assembly with that namespace. If you want to use different prefixes for different controls within the same assembly, you must put the controls in different namespaces.

Create a Custom Designer for a Custom Web Control

When a Custom Web Control is added to a page, the *Render* method is called and the output from that is displayed in the Design view in Visual Studio. This may not be correct, especially when your control uses property values that have not been set and the default values make no sense when rendering.

In these cases, you need to create a Custom Designer that will generate the design time view of your control. You need to inherit from the *ControlDesigner* class and then override the *GetDesignTimeHtml* method to return the HTML that you want to display in the designer.

```
namespace WebCustomControls
{
  private class CustomDesigner : System.Web.UI.Design.ControlDesigner
  {
    override string GetDesignTimeHtml()
    {
      // return the HTML to be shown in designer
    }
  }
}
```

You attach your Custom Designer to your Custom Web Control using the Designer attribute:

```
[Designer("WebCustomControls.CustomDesigner")]
public class WebCustomControl : System.Web.UI.WebControl
```

Create a Composite Server Control

A Composite Server Control is a control that contains other controls. This is similar to a User Control but there is no designer available. Like Custom Web Controls, you can also add Composite Server Controls to a class library for reuse in other projects.

A Composite Server Control inherits from the *CompositeControl* class which, in turn, inherits from *WebControl*, so all of what you've learned for Custom Web Controls also applies to Composite Server Controls.

To add controls to your Composite Server Control you need to override the *CreateChildControls* method and add the controls to the *Controls* collection. You will, effectively, end up with a tree of controls that are rendered correctly to the Web Form or User Control that contains the Composite Server Control.

Handling Events in Composite Server Controls

A Composite Server Control can contain several Web Controls and these controls can raise their own events. If you have a *Button* in the Composite Server Control, you will be able to catch the Click event as long as you add a handler for the event in the *CreateChildControls* method:

```
protected override CreateChildControls()
{
 button btnSubmit = new Button():
 btnSubmit.Text = "Click Me";
 btnSubmit.Click += new EventHandler(btnSubmit_Click);
 Controls.Add(btnSubmit);

 // add whatever other controls are required
}
```

This adds an event handler for the click event (of type *EventHandler*). The event handler then needs to be implemented:

```
private void btnSumbit_Click(object sender, EventArgs e)
{
 // handle the click event
}
```

Bubbling Events from Composite Server Controls

There are also times when we want the event to be available outside of the Composite Server Control. Rather than just handling the event internally, we want the hosting Web Form or User Control to be able to provide event handling – we need to bubble the event.

To bubble the event, we need to create an event that the Web Form or User Control can attach to. This needs to be public and, unless we're creating our own delegate and *EventArgs* derived class, it needs to provide the same event handler as we've trapped in the Composite Server Control.

To bubble a *Button* click event, we would create a public *EventHandler event* property:

```
public event EventHandler Submitted;
```

We then need to raise this event within the *Click* event handler for the button:

```
private void btnSubmit_Click(object sender, EventArgs e)
{
 Submitted(this, e);
}
```

We can then use the Submitted event within the Web Form or User Control to handle the *Button* being clicked by using the Properties window in Design View or by adding the event handler in code.

Develop a Templated Custom Control

We've already seen how to build a Templated User Control and building a Templated Custom Control is only slightly more complex. There are a series of steps you can follow to build a Templated Custom Control within a class library:

1. Add a class to the class library that derives from the *Control* class and implements the *INamingContainer* interface. This is the container class that provides the data for the template and you should have a public property for each item of data that you want to make available to the template.
2. Add a class to the class library that derives from the *Control* class and implements the *INamingContainer* interface. This is the templated control.
3. Add the *ParseChildren(true)* attribute to the template control (from Step 2).
4. Create properties on the template control (from Step 2) that are of type *ITemplate*. These are populated by the page designer on the Web Form or User Control containing the Templated Custom Control.
5. The *ITemplate* properties need to have their *PersistenceMode* attribute set to *PersistenceMode.InnerProperty*.
6. The *ITemplate* properties need to have their *TemplateContainer* attribute set to the type of the data container (from Step 1).
7. Override the *DataBind* method of the template control (from Step 2) to call the *EnsureChildControls* method on the base class.
8. In the *CreateChildControls* method of the template control, instantiate the correct template (via the properties defined in Step 4) by passing in a populated data container (from Step 1) using the *InstantiateIn* method. Add the data container to the *Controls* collection.

Tracing, Configuring and Deploying Applications

Use a Web Setup Project

Deploying an ASP.NET Web site can be very simple. A Web site is typically file based and can be deployed simply by copying the files. If you need to make an update, you can overwrite the existing files. This makes updating a Web site very easy, as you typically don't need administrator rights to update the files.

There are times, however, when you need more control over how your application is installed. It may have pre-requisites, it may need registry entries, etc. If you want your Web site to be downloaded over the internet, then you'll want an installer, as this allows the Web site to be installed even if the user doesn't know how to configure a Web server.

Creating a Web Setup Project

A Web Setup Project is similar to a standard Setup Project, except that it supports the extra configuration options required by Web sites. You create a Web Setup Project with these steps:

1. Open your existing Web site and select **Add > New Project** from the File menu.
2. In the Add New Project dialog expand **Other Project Types** in the Project Types list and select the **Setup and Deployment** option.
3. Select the **Web Setup Project** in the templates list and give your setup project a name (this is the name that the installer will be given).

4. Click OK. This will add the Setup Project to your solution.
5. In the File System editor (shown in the main pane) right-click on Web Application Folder and select **Add > Project Output**.
6. In the Add Project Output Group dialog ensure that the Web site is selected in the Project drop-down list and select the Content Files option. Click OK.

You've now created a Web Setup Project for your Web site. By default, a Web Setup Project is empty and the final two steps above add the required files from the Web site.

You need to manually build a Web Setup Project, as it doesn't build by default with the rest of your Web site. Select the Web Setup Project in Solution Explorer and select Build from the context menu.

Configuring Deployment Options

Your Web site may not require any additional configuration but there are times when you may need to add additional configuration options.

Launch Conditions

Launch Conditions can be used to restrict the machines that your Web site can be installed on. You can view the Launch Conditions editor by selecting **View > Launch Conditions** from the Web Setup Project context menu and this launches the editor, shown in *Figure 10*.

Figure 10 – Launch Conditions Editor

By default, there is one launch condition in a Web Setup Project and this checks that at least version 4 of IIS is installed. It does this by making use of the Search Target Machine and Launch Conditions nodes:

- Search Target Machine – contains search criteria for the installation. These can be file, registry or Windows Installer based criteria. In *Figure 10*, the "Search for IIS" is a registry search that looks for the version number of IIS.
- Launch Conditions – contains the conditions that must be met before the installation can continue. These can be based on defined search criteria (as "IIS Condition" is in Figure 10) or other criteria (such as .NET Framework version).

Custom Wizard Pages

The default installation process allows you to specify the Web Server and virtual directory that is to be created. You may also want to display a license agreement or change Web.config settings during the install. In cases such as these, you'll need to add Custom Wizard Pages.

You can view the User Interface editor by selecting **View > User Interface** from the Web Setup Project context menu.

From the User Interface Editor you can add Custom Wizard Pages. You can then reference the data contained on the Custom Wizard Pages in any Custom Actions that you add.

Custom Actions

Custom Actions allow you to perform specific installation options by calling an executable or script during installation. Information entered in Custom Wizard Pages can be passed to the executable or script.

You can view the Custom Actions editor by selecting **View > User Interface** from the Web Setup Project context menu.

Registry Entries

You should store configuration entries in Web.config but there may be times when you need to store information in the Registry. You can view the Registry editor by selecting **View > Registry** from the Web Setup Project context menu and this launches the editor shown in *Figure 11*.

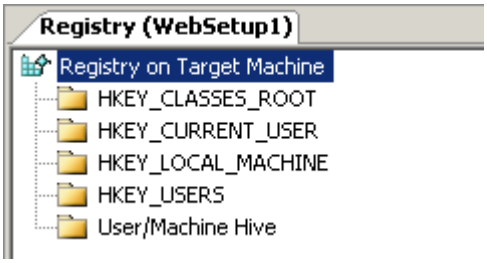

Figure 11 – Registry Editor

You can add keys to the subset of keys shown in *Figure 11*. To add a nested key, you need to add all keys down to the key you require (only keys that don't already exist are created). You then need to either make the key creation conditional (which you can base on criteria specified in the Launch Conditions editor), or you must set the *AlwaysCreate* property to true. Keys can also be automatically deleted on uninstall by setting the *DeleteOnUninstall* property.

Deploying Web Applications

Two files are generated when you build a Web Setup Project:

- setup.exe – This is a wrapper for the Windows Installer file and is the file that most users will be familiar with.
- <project>.msi – This is a Windows Installer file that contains the setup process. It is called by setup.exe to perform the install.

The Windows Installer file contains the setup process and running it installs the Web site. During the setup process you can specify the Site and Virtual Directory for the Web site as shown in Figure 12.

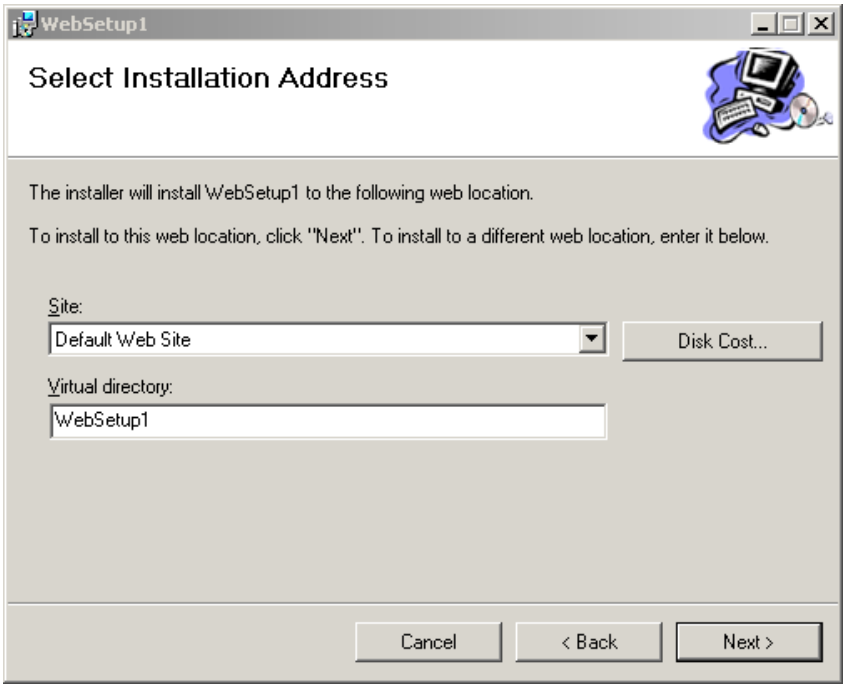

Figure 12 – Specifying the installation details for the Web site

Copy a Web Site using the Copy Web Site Tool

If you simply need to move a Web site from server to server, then logging onto the server to install a Web Setup Project can be impractical. In these situations you can use the Copy Web Site tool.

The Copy Web Site tool can connect to a remote server using several different methods:

- File System – connect to a Web site on a local drive or shared folder.
- Local IIS – connect to an IIS Web site on the local machine.
- FTP Site – connect to a remote Web site using FTP.
- Remote Site – connect to a Web site using FrontPage Server Extensions. You can also choose to use an SSL connection for security.

You can use the Copy Web Site tool to copy an entire Web site to/from a remote site or you can synchronize the local and remote sites. Synchronization only changes those files that are changed between the two Web sites and will also detect version conflicts between the two Web sites.

Precompile a Web Site using the Publish Web Site Tool

When a page is first requested from a Web site, the Web site is first compiled from MSIL to native code. This is a quick process but the first few hits on a Web site may be delayed while the compilation is taking place. You can precompile your Web site to remove this delay using the Publish Web Site tool.

The Publish Web Site tool has several options that you should be aware of:

- Allow this Precompiled Site to be Updatable – doesn't build the content of ASPX pages so that you can change the HTML markup after the Web site has been precompiled.
- Use Fixed Naming and Single Page Assemblies – turns off batch builds so that assemblies are generated with fixed names.
- Enable Strong Naming on Precompiled Assemblies – allows you to attach a key file or key container to strongly name the precompiled assemblies.

Using the Publish Web Site tool, you can specify the location where you want the precompiled site to be placed. You can then copy this directory to the remote server.

Optimize and Troubleshoot a Web Application

Customize Event-Level Analysis

It is possible to log events raised by ASP.NET using Event Providers. By default, only "All Errors" and "Failure Audits" are logged to the Event Log. It is possible to log a lot more than this.

Event Providers

There are five ASP.NET Event Providers:

Name	Description
EventLogWebEventProvider	Logs to the Event Log. By default, this logs "All Errors" and "Failure Audits".
MailWebEventProvider	Logs the event via mail. You will use one of the derived classes - either *SimpleMailWebEventProvider* or *TemplatedMailWebEventProvider* – to do the logging.
SqlWebEventProvider	Logs the event to an SQL Server.
TraceWebEventProvider	Logs the raised events to the ASP.NET tracing system.
WmiWebEventProvider	Logs events to the WMI system.

It is also possible to build your own Event Provider by inheriting from *WebEventProvider* or *BufferedWebEventProvider*.

Event Providers are configured in the *<healthMonitoring><providers>* section of the configuration files. Machine.config has both the *EventLogWebEventProvider* and a *SqlWebEventProvider* (to the LocalSqlServer connection) configured. It is possible to override these settings in Web.config.

Web Events

The Event Providers can log any of the events raised by ASP.NET. These can be broadly split into five categories, derived from the following base classes:

Name	Description
WebApplicationLifecycleEvent	Any significant event in the lifecycle of the Web site.
WebAuditEvent	Any login attempts, security related events or any failures related to View State occur.
WebBaseErrorEvent	Any errors that occur on the site.
WebHeartbeatEvent	Any heartbeat events that are raised at specific intervals.
WebRequestEvent	Any events providing request information.

You can specify the events to capture in the <*healthMonitoring*><*eventMappings*> section of the configuration files. You then need to tie the <*eventMappings*> section to the <*providers*> section by adding a link in the <*healthMonitoring*><*rules*> section.

Use Performance Counters

Performance counters can be used to monitor your application. The .NET Framework 2.0 makes it easy to use performance counters in your application.

You can create an instance of a performance counter by passing the category name and counter name to the constructor of the *PerformanceCounter* class. This creates a read-only counter that you can then query for its values. If you want to write values, you must pass *false* as a third parameter to the constructor:

```
PerformanceCounter pcRestarts = new PerformanceCounter ("ASP.NET", ~CCC
 "Application Restarts", false)
```

You can now use the *Decrement*, *Increment*, and *IncrementBy* methods to modify the value of the performance counter, or the *RawValue* property to set the counter to a specific value.

ASP.NET Tracing

Tracing allows you to view the execution of a Web Form. Tracing is disabled by default but it can be enabled in Web.config by setting the *enabled* attribute of <*system.web*><*trace*> to true. You can also set the *localOnly* attribute to *true* to enable trace requests to only be viewed from the local machine.

When tracing is enabled, you have a section at the end of every Web Form that contains the tracing information. There are several sections to the tracing information that allow you to view the cookie and session values that the page is using. The two that you'll use most often are:

- **Trace Information** – provides details related to lifecycle events and timings of the events. You can write information to this section using the *Trace.Write* and *Trace.Warn* methods of the *Page* and *UserControl* classes – the only difference being that Trace.Warn calls are shown in red.
- **Control Tree** – shows the entire tree hierarchy for the page. This includes the number of bytes that the control uses in both the ViewState and ControlState sent to the client.

It is also possible to view all of the tracing information for a Web site by navigating to the trace.axd file in the root of the Web site. This is an ASP.NET redirect that displays the items held in the trace cache (set using the *requestLimit* attribute of <*trace*> in Web.config) and allows you to view the tracing information separately from the pages. By default the trace cache holds the earliest entries and ignores any new entries. You will need to set the *mostRecent* attribute to *true* to always show the latest trace details.

Caching

Caching allows you to store frequently accessed data in memory rather than retrieving it on every page request. If the data is slow to retrieve and doesn't change very often, then caching the data in memory may be quicker as future queries can be made against the cached data.

There are two ways that you can cache data:

- **Application caching** – stores any object in memory using the *Cache* object. The object can be automatically removed if memory limts, time limits or other dependencies are met.
- **Output caching** – rather than storing individual objects in memory, it is possible to keep the rendered Web Form or User Control for future requests.

Application Caching

A single *Cache* object exists for the entire application. Both the *Page* and *UserControl* classes provide a *Cache* property that provides access to it.

Before you use an object from the cache, you need to ensure that the object exists (it may have been automatically removed) by checking that the object isn't *null*:

```
if (Cache["cacheKey"] == null)
  // must create again
else
  // cast cached object to correct type
```

You can place any object into the cache but you must cast it to the correct type when returning it from the *Cache* object.

You can use the *Cache* object as you would with any collection, as there are *Add* and *Remove* methods to manipulate the cache.

By default, an object will stay in the cache until it is removed (i.e. it never becomes invalid). There is also an *Insert* method (with several overloads) that allows you to specify when the cached object becomes invalid. These extra parameters are as follows:

- CacheDependency – allows you to specify a file, cache key, or another *CacheDependency* object. When the dependency changes, the cached object becomes invalid and is removed from the cache.
- DateTime – allows an absolute expiration to be applied to the cached object.
- TimeSpan – allows a sliding expiration (since the last access of the object) to be added to the cached object.
- CacheItemPriority – priority compared to other items in the cache. Objects with a lower priority are removed first.
- CacheItemRemovedCallback – a method to call when the cached object is removed from the cache.

Output Caching

If you build an ASP.NET Web Form or User Control that doesn't change very often, you can use Output Caching to store a copy of the rendered Web Form or User Control on the server. When it is requested, the cached version is used and it can be returned almost instantly. This can significantly improve performance on pages that have multiple queries.

Output Caching is controlled using the *@OutputCache* directive and can be applied to both Web Forms and User Controls.

There are several properties that you can configure, giving you full control over the caching. The ones that you'll use the most often are:

- Duration – the number of seconds to cache the page. This is the only required property.
- VaryByParam – if your page is parameterized (e.g. a product page), then this is a comma-separated list of Query String parameters. The output cache stores a different version of the page for each combination of the specified parameters. If you do not want to cache the page by parameter, then set this to none.
- VaryByControl – a comma-separated list of User Control IDs that are used to vary the page caching.
- Shared – applies only to User Controls but specifies whether the same cached content can be used on different Web Forms.

One problem with using Output Caching is that there will be several cases where most of the page can be cached but there are areas of the page that shouldn't be cached. In these cases you can use a *Substitution* control to replace portions of the cached page.

The *Substitution* control is similar to a *Literal* but the control is exempt from Output Caching. You must specify a method (using the *MethodName* property) that accepts an *HttpContext* object and returns a *String*. The returned *String* is then rendered in place of the *Substitution* control.

Customizing and Personalizing a Web Application

Implement a Consistent Page Design Using Master Pages

Master pages allow you to implement a consistent page design for all the pages in your application. The master page defines the look and feel for your Web site. You then create individual content pages that are merged with the master page when the page is requested.

There are two separate types of page that can be created when using master pages:

- **Master Page** – a page with a .master extension. Instead of a *@Page* directive, the master page has a *@Master* directive. A master page is the same as a standard Web Form and can contain anything that a Web Form can. What makes a master page different is the *ContentPlaceHolder* controls that are replaced by content form the content pages at runtime.
- **Content Page** – a normal Web Form that is merged with the master page specified in the *MasterPageFile* attribute of the *@Page* directive. Each of the *ContentPlaceHolder* controls in the master page has a corresponding *Content* control that is displayed at runtime.

You can bind a master page to a content page in three ways:

- **Application level** – by setting the *masterPageFile* attribute of the *<system.web><pages>* element in the root Web.config. The master page is applied to all content pages unless the content page doesn't contain any *Content* controls.
- **Folder level** – by setting the master page in a Web.config file that isn't at the root of the site.
- **Page level** – by setting the *MasterPageFile* attribute in the *@Page* directive of a Web Form.

Default Content

If you don't include the *Content* control for the *ContentPlaceHolder* in the master page, then nothing is output in the place reserved for the *ContentPlaceHolder*. It is possible to add default content to a *ContentPlaceHolder* by adding the content to the master page. If the *Content* control is not defined (an empty *Content* control is still defined) then the default content will be displayed.

Referencing the Master Page

There are times when you will need to reference properties or controls of the master page from the content page. A Web Form has a *Master* property that returns the instance of the master page that is being used. If you're not using a master page, then this property returns *null*.

This is of the *MasterPage* type and may not provide everything that we need. We can use the *@MasterType* directive to specify the file containing the master page using the *VirtualPath* attribute (set to the same value as the *MasterPageFile* attribute of the *@Page* directive). At runtime, the *Master* property will be cast to the correct type (i.e. the specific master page) and you will have access to all of the properties and controls of the master page.

Master Page Events

As a master page can contain Web Server Controls, it needs to be able to handle the events raised from those controls. Events can also be raised in content pages as the content page can also contain Web Server Controls. You should respond to events in the page that contains the Web Server Control.

The only thing to be aware of is the order that page level events are raised. This is not consistent for each of the events.

For the *Init* and *Unload* events, the order is as follows:

- User Controls contained on Content Page
- User Control contained on Master Page
- Master Page
- Content Page

For the *Load* and *PreRender* events, the ordering is different:

- Content Page
- Master Page

- User Controls contained on Content Page
- User Control contained on Master Page

Nested Master Pages

It is also possible to nest master pages, although Visual Studio 2005 does not support modifying nested master pages in Design view. Nested master pages are ideal when you have a general site structure in the top-level master page and different sections with different layouts defining their own master page.

A nested master page is the same as a normal master page, except that it also has a *MasterPageFile* attribute added to the *@MasterPage* directive.

Within each *Content* control on the child master page, you are free to add whatever controls you require to define the layout of the nested master page. The only caveat is that only *ContentPlaceHolder* controls of the child master page are available to the content page. So, if you want a *ContentPlaceHolder* from the parent master page to be available to the content page, you must define a new *ContentPlaceHolder* in the *Content* control on the child master page.

Changing Master Pages Dynamically

You can also change the master page that is being used programmatically. The *MasterPageFile* property of a Web Form allows you to change the master page that is being used. You must, however, do this in the *PreInit* event, before any control population takes place (which occurs during the *Init* event).

When changing master pages programmatically, you need to ensure that all the master pages that you may use have the same *ContentPlaceHolder* controls. Any changes that you make to the *ContentPlaceHolder* controls in one master page must be applied to all the other master page controls.

Customize a Web Page Using Themes

Themes are collections of properties that define the appearance of Web Forms and User Controls within a Web site. Themes allow you to give your Web site a consistent appearance.

All themes are stored in the *App_Themes* folder and each them has its own folder corresponding to the name of the Theme. A theme can contain several different elements:

- **Style sheets** – any CSS style sheets defined in the theme folder are automatically linked to any pages using the theme.
- **Images/resources** – any images or resources that are specific to the theme can be stored within the theme folder.
- **Skins** – skins allow you to define custom layouts for Web Server Controls.

Themes can be attached to pages in several different ways:

- **Page level Theme** – by attaching a theme using the *Theme* attribute of the *@Page* directive.
- **Folder level Theme** – by specifying a theme using the *Theme* attribute of the <*pages*> element in Web.config.

- **Page level StylesheetTheme** – by attaching a style sheet using the *StylesheetTheme* attribute of the *@Page* directive.
- **Folder level StylesheetTheme** – by specifying a theme using the *StylesheetTheme* attribute of the <*pages*> element in Web.config.

Define the Appearance of Controls Using Skins

Skin files change the appearance of controls and allow you to define default settings for controls. A skin file is created within the specific theme folder in *App_Themes* and has a .skin extension.

A skin file can contain several different control skins. A control skin is the same as a definition for a normal Web Server Control, minus the ID tag.

There are two types of control skin that you can define:

- **Default control skin** – a default control is applied to all controls within a page. A default control skin is a control skin that does not have a *SkinID* attribute defined. You can have only one default control skin defined per theme.
- **Named control skin** – a named control skin has a *SkinID* attribute defined and is only applied to controls that have a matching *SkinID* attribute.

All skin files belonging to a theme are automatically attached to a Web Form when it makes use of the theme. Named control skin properties override those defined in a default control skin.

The way the theme is attached also impacts the way that properties defined on the control instance on the Web Form are handled. If the theme is attached using the *Theme* attribute, then any properties defined on the control in the Web Form are overridden. *StylesheetTheme* attached themes are overridden by properties defined on the control in the Web Form.

User Profiles

User profiles provide a standard way of storing user specific information in the database. Once you define the profile properties, these are automatically made available to your Web Form and are automatically populated with values for the current user.

User profiles make use of the provider model and, by default, store all their information in an SQL Server database using the *SqlProfileProvider* class. Before you can use profiles, you need to configure the database using the *aspnet_regsql.exe* command line tool.

You configure the profile provider by modifying Web.config. You need to add a <*profile*> element to <*system.web*> that specifies the *defaultProvider* that you want to use:

```
<profile defaultProvider="sqlProvider">
```

Within the <*providers*> element you then define the provider itself, pointing at the correct connection string to use:

```
<providers>
 <add name="sqlProvider"
  type="System.Web.Profile.SqlProfileProvider"
  connectionString="connString" />
```

Configuring Profile Properties

You must define profile properties before you can use them in code. Within the <*properties*> element of the <*profile*>, you define the properties that you want available:

```
<properties>
  <add name="Theme" />
  <add name="Age" type="System.Int32" />
```

Each of these properties is then made available to your code via the *Profile* property of the *Context* object of a Web Form and User Control. The properties defined in Web.config are correctly typed (via the *type* attribute).

Anonymous User Profiles

It is also possible for anonymous users (those that are unauthenticated) to have profile information, although this will not be persisted between visits to the site. You must specify that you want anonymous user profiles in <*system.web*>:

```
<anonymousIdentification enabled="true" />
```

You must also specify which properties of the profile are available. By default, the anonymous user profile has no properties. You must specify the *allowAnonymous* attribute on each property that you want to allow:

```
<properties>
  <add name="Theme" allowAnonymous="true" />
```

If you're using anonymous user profiles and then allow the user to authenticate, the anonymous profile information will be lost. If you wish to keep any information in the anonymous profile, you must respond to the *MigrateAnonymous* event of the Profile in Global.asax:

```
public void Profile_OnMigrateAnonymous(object sender, ~CCC
  ProfileMigrateEventsArgs args)
{
  ProfileCommon anonProfile = Profile.GetProfile(args.AnonymousID);

  // copy the profile information

  ProfileManager.DeleteProfile(args.AnonymousID);
  AnonymousIdentificationModule.ClearAnonymousIdentifier();
  Membership.DeleteUser(args.AnonymousID, true);
}
```

Dynamically Adding and Removing Child Controls

You can add and remove controls at runtime. You can do this directly on the Web Form or User Control but it is much easier if you make use of a *PlaceHolder* control.

In your code, you create an instance of the control you require and add this to the *Controls* collection of the *PlaceHolder*. You can do this at any stage of the page lifecycle but the earlier the better. If you're relying on the View State of the control being persisted across post backs, you must do this in the *Init* event handler.

Implement Web Parts

Web parts are components of a Web Form that the user can move, display and hide. You can think of them as gadgets that the user can use to individualize the Web Form for their requirements.
Web Parts can be created in two ways:

- By defining a custom control that inherits from the *WebPart* abstract base class. This method provides the most functionality to the Web Part.
- By using existing controls (Web Server Controls, Custom Controls or User Controls). These are automatically wrapped by a *GenericWebPart* class which provides a reduced subset of functionality.

To enable the use Web Parts on your page, you need to follow several steps:

- Add a *WebPartManager* control to the page before any other Web Part controls.
- Add a *WebPartZone* control to the page which will contain the Web Part controls. If you add more than one *WebPartZone* to the page, it will allow users to move the Web Part controls between zones.
- Add Web Parts to your *WebPartZone* control (either a *WebPart* derived control or a control that will be automatically wrapped by a *GenericWebPart*).

The default view of a Web Part enabled page is *BrowseDisplayMode*. When in *BrowseDisplayMode* you have limited control over what you can do to the Web Parts on the page – limited to minimizing them (to only display their title) or closing them completely.

You can change the mode of the Web Parts by setting the *DisplayMode* property of the *WebPartManager* instance to one of the static properties defined on *WebPartManager*. Each of these different modes allows slightly different functionality.
Once you're finished with a particular mode, the *DisplayMode* property should be reset to *BrowseDisplayMode*.

Arranging and Editing Web Parts

There are two modes that you can use to allow users to personalize their site experience:

- **DesignDisplayMode** – in addition to the functionality provided by *BrowseDisplayMode*, the Web Parts can also be moved. This is either within a single *WebPartZone* or to another *WebPartZone*.
- **EditDisplayMode** – in addition to the functionality provided by *DesignDisplayMode*, *EditDisplayMode* also allows the title, direction and appearance of Web Parts to be configured. You can also delete Web Part controls when in *EditDisplayMode*. This requires the addition of an *EditorZone* control to the page to hold editing tools and the selection of which editing tools (*AppearanceEditorPart* and *LayoutEditorPart*) you wish to use.

Adding New Web Parts

In order to allow the user to add new Web Part controls to a page, you need to be in *CatalogDisplayMode*. This requires a *CatalogZone* to be defined that contains the different types of control that the user can add. There are three types of Web Parts that can be added using a specific *CatalogZone* Web Part:

- **DeclarativeCatalogPart** – allows you to declare Web Parts within the page and these are always available for the user to add.
- **ImportCatalogPart** – allows Web Part descriptions to be imported from file. These are simply settings that are applied to a specific type of Web Part.
- **PageCatalogPart** – this keeps track of the Web Parts that the user has chosen to close, allowing them to be re-added to the page.

Connecting Web Parts

It is also possible to connect Web Parts together so that they can share information between themselves. Web Parts are connected using a consumer-provider model.

To enable connections, the underlying control (whether it is a *WebPart* derived control or a *GenericWebPart* wrapped control) must be configured correctly.

To be a provider, you must define a method that returns the value to be passed to the consumer. This method must be annotated with the *ConnectionProvider* attribute:

```
[ConnectionProvider("ProviderName","ValueProvider")]
public string GetValue()
{
        // return the value
}
```

To be a consumer, you must define a method that accepts the value passed by the provider. This property must be annotated with the *ConnectionConsumer* attribute:

```
[ConnectionConsumer("ConsumerName","ValueConsumer")]
public void SetValue(string strValue)
{
        // use the value
}
```

You can then define static connections within the Web Form or allow the user to create their own dynamic connections.

Static Connections

Static connections are configured using a *WebPartConnection* control. Once configured, it is always available. This is added to the <*StaticConnections*> element for the WebPartManager:

```
<StaticConnections>
  <asp:WebPartConnection ID="wpConn1"
    ProviderID="ProviderControlID"
    ProviderConnectionPointID="ValueProvider"
    ConsumerID="ConsumerControlID"
    ConsumerConnectionPointID="ValueConsumer" />
</StaticConnections>
```

Dynamic Connections
Users can also create their own connections. In order to support this, you must add a *ConnectionZone* control to the page and switch to *ConnectDisplayMode*. When in this mode, you can break existing connections and configure new ones. The *ConnectionZone* control will interrogate all the Web Parts visible and only allow you to connect compatible Web Parts.

Implementing Authentication and Authorization

Authentication is the process of deciding whether a user is allowed to access your Web site. Authorization is the process of deciding what rights a user has within your Web site.

By default, your Web site is configured to authenticate access to any restricted pages using Windows Authentication. This presents the user with a browser generated login dialog when a user requests a restricted page.

This is fine when your application is an intranet site and you can integrate with Active Directory, but when you need to build an internet site you need a different mechanism.

Configuring Forms Authentication

With Forms Authentication, you present the user with an HTML login page and manage the entire user database as part of your Web site.

To enable Forms Authentication, you need to modify the <*authentication*> element in Web.config:

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

This will redirect all unauthenticated users to the login.aspx Web Form when they request a restricted page.

Forms Authentication, by default, uses cookies to store the authentication token for the user. You can also choose to store the authentication token as part of the URL by setting the value of the *cookieless* attribute to one of the following values:

- **UseCookies** – will always use cookies to store the authentication token.
- **UseUri** – will always use the URL to store the authentication token.
- **AutoDetect** – will use cookies or the URL depending upon the specific browsers configuration.
- **UseDeviceProfile** – this is the default setting and uses cookies if the browser's profile (not what the specific browser indicates) indicates that it supports cookies.

Setting up the Database

By default, Forms Authentication uses an SQLEXPRESS database located in the *App_Data* folder. To use an external database you must first use the *aspnet_regsql.exe* command line tool to provision the database and then configure the Membership Provider in Web.config. You need to add a <*membership*> element to <*system.web*> that specified the *defaultProvider* that you want to use:

```
<membership defaultProvider="sqlMembershipProvider">
```

Within the <*providers*> element you then define the provider itself, pointing at the correct connection string to use:

```
<providers>
 <add name="sqlMembershipProvider"
  type="System.Web.Security.SqlMembershipProvider"
  connectionStringName="connString">
```

The Membership API

You have direct access to the database storing the Membership information via the *System.Web.Security.Membership* class. This has static methods that allow you to perform several actions against the database, the most common of which are as follows:

- **CreateUser**, **DeleteUser**, **UpdateUser** – allow you to manage individual users.
- **GetUser**, **GetAllUsers** – allow you to return either a specific user from the database or a collection containing all the users.
- **FindUsersByEmail**, **FindUsersByName** – allow you to return users that match the specified email address or username.
- **ValidateUser** – allows you to manually authenticate users against the database.

Anonymous Identification

If you know that your Web site is not going to use authentication, you can turn authentication off completely in Web.config:

```
<authentication mode="None">
```

Use Authorization to Establish Rights

Once authenticated, you need to establish the rights of your user. This is accomplished using roles. By default, all users are granted access to all Web Forms within the Web site and you need to restrict access to specific locations.

Setting up the Database

By default, Forms Authentication uses a SQLEXPRESS database attached to the *App_Data* folder. To use an external database, you must first use the *aspnet_regsql.exe* command line tool to provision the database and then configure the Roles Provider in Web.config. You need to add a <*roleManager*> element to <*system.web*> that specified the *defaultProvider* that you want to use:

```
<roleManager defaultProvider="sqlRoleProvider">
```

Within the <*providers*> element you then define the provider itself, pointing at the correct connection string to use:

```
<providers>
 <add name=" sqlRoleProvider"
  type="System.Web.Security.SqlRoleProvider"
  connectionStringName="connString">
```

The Roles API

You have direct access to the database storing the Roles information via the *System.Web.Security.Roles* class. This has static methods that allow you to perform several actions against the database, the most common of which are as follows:

- **CreateRole**, **DeleteRole** – allow you to manage individual roles.
- **GetAllRoles** – returns a collection containing all the roles.
- **FindUsersInRole** – returns a collection of users in a specific role.
- **AddUserToRole, AddUsersToRole, AddUserToRoles, AddUsersToRoles** – add a user (or collection of users) to a role (or collection of roles).
- **DeleteUserFromRole, DeleteUsersFromRole, DeleteUserFromRoles, AddUsersFromRoles** – delete a user (or collection of users) from a role (or collection of roles).
- **IsUserInRole** – checks whether a user is in a specified role.

Checking For Specific Roles

There are two ways you can check for a user belonging to a specific role. You can use the *Roles.IsInRole* method, passing the username and role, to check if the specified user is in the given role.

The *Page* and *UserControl* classes also provide a *User* property that contains the details of the current user. You can use the *User.IsInRole* method, passing the role, to check if the current user is in the specified role.

However, by default, the *Roles* collection for the current user in Forms Authentication is empty and never populated. You need to add the following hack to the application's *AuthenticateRequest* event in Global.asax:

```
if (Context.Request.IsAuthenticated == true)
 Context.User = new System.Security.Principal.GenericPrincipal( ~CCC
  User.Identity, Roles.GetRolesForUser(User.Identity.Name));
```

Restricting Access

You can restrict access to Web Forms by manually checking that a user is in a given role in code by using one of the *IsInRole* methods in a Web Form.

You can also configure access to files and directories in Web.config. Within <*system.web*> you can define an <*authorization*> element that has <*allow*> and <*deny*> children that specify the user or roles that are allowed to access the specified resource. You specify users using the *users* attribute on <*allow*> or <*deny*> and the roles using the *roles* attribute. Multiple users or roles are comma separated and you can use the special value of "*" to mean all users and "?" to mean unauthenticated or anonymous users. It is important to remember that the order in which <*allow*> or <*deny*> statements appear is the order in which those rules are applied.

There are two ways to specify the <*authorization*> element within Web.config:

- <*system.web*><*authorization*> - this applies the specified roles to the directory, and all sub-directories, containing Web.config.
- <*location*><*system.web*><*authorization*> - this allows more fine grained control.

The <*location*> element allows you to configure settings for specific directories or files using the *path* attribute. Setting *path* to a directory causes the authorization rules to apply to that directory and all sub-directories, while specifying a file applies the authorization rules just to that file.

Use Windows Authentication

Windows Authentication is the default method of authentication and is configured in Web.config in the <*authentication*> element of <*system.web*>:

```
<authentication mode="Windows" />
```

Windows Authentication replaces the Membership Provider and retrieves all the user information directly from Active Directory. Any user groups the user belongs to are applied as roles (available from the *User.IsInRole* method). You must not have a Role Provider configured when using Windows Authentication. If you do, the Windows user groups will not be applied correctly to the user.

Impersonating Users

When accessing files or network resources, a Web site uses the ASPNET (IIS5) or Network Service (IIS6) account. This is actually configured in the <*processModel*> element of Machine.config. You can override this behavior by modifying Machine.config. This changes the setting for all Web sites on the machine:

```
<processModel userName="user" password="pass" />
```

If you need to configure this on a Web site basis, you will need to use impersonation. This can be configured in Machine.config (for all Web sites) or in Web.config by modifying the <*identity*> element:

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

Impersonation causes all file or network resource requests to use the current user's Windows account. For users that aren't authenticated (either they're not logged in or the Web site is using Forms Authentication), the *IUSR_machine* account is used.

You can also impersonate a specific user account by adding those details to the <*identity*> element:

```
<identity impersonate="true" userName="user" password="pass" />
```

Login Controls

When using Forms Authentication, you need to build all your own login pages (login, registration, etc.). This can be very tiresome and is repeated on every Web site you build. The ASP.NET Login Controls are tied directly to the Membership Provider and simplify much of this work for you:

- **Login** – displays username and password text boxes and a checkbox to indicate that login information is to be remembered and authentication to be automatic next time.
- **LoginView** – allows you to display different information to logged-in and anonymous users. You define a *LoggedInTemplate* and an *AnonymousTemplate* that are shown to the user as necessary.
- **LoginStatus** – displays a login link for anonymous users and a logout link for logged-in users. You can specify the text displayed using the *LoginText* and *LogoutText* properties, or, if you want to use images, you can use the *LoginImageUrl* and *LogoutImageUrl* properties.
- **LoginName** – displays the username of the current logged in user.
- **PasswordRecovery** – allows users to retrieve their password. Depending upon the Forms Authentication settings, this will either create a new password or send the existing password to the user.
- **CreateUserWizard** – allows users to register on the site. There is a default set of information (username, password, email address) but it can be extended to gather whatever information is required.
- **ChangePassword** – allows a logged in user to change their password.

Configuring Security Information

The default configuration for the Membership Provider is to require the user to enter a username, password and email address as the minimum information that is required. You can also specify that each user must enter a secret question and answer by setting the *requiresQuestionAndAnswer* attribute to true when defining the Membership Provider being used.

Configuring the Mail Server

Several of the login controls send emails to the user. In order to send mail you need to configure the

```
<mailSettings> element in <system.net>
<mailSettings>
 <smtp deliveryMethod="network" from="noreply@noreply.com">
  <network host="localhost"></network>
 </smtp>
</mailSettings>
```

You need to set the *from* attribute to the from address, otherwise you will get a runtime error when using the Login Controls. You also need to set *host* to the address of the mail server. If the mail server requires authentication, you can set the *username* and *password* attributes.

The Login Control

Simply putting a *Login* control on a Web Form provides all the application logic you need to allow people to login to your site. The control takes care of authenticating the user against the database and redirecting to the correct location once login is complete.

If you need to provide your own authentication logic, then you can override the control's *Authenticate* event to add your own authentication. You must set the *Authenticated* property of the *AuthenticateEventsArgs* to true, or false, indicating whether the user was authenticated correctly.

The PasswordRecovery Control

The *PasswordRecovery* control sends a password to the email address specified for the user. If necessary, it will also ask for the answer for the user's secret question before emailing the password.

Which password is sent depends on the settings of the Membership Provider. If the Membership Provider is configured to store one-way hashed passwords, a new password will be emailed to the user. If clear-text password is stored in the database, the existing password will be emailed to the user.

The email to send is configured using a *MailDefinition* element of the control. You can specify a text file to use as the basis of the email using the *BodyFileName* attribute and when this is parsed the *<% username %>* and *<% password %>* strings are replaced with the correct values.

If you need further customization of the email you can catch the *SendingMail* event which is raised just before the mail is sent. The *Message* property of the passed event arguments contains the message that is to be sent.

The CreateUserWizard Control

The *CreateUserWizard* control allows users to register with your site. It asks all of the information required for the Membership Provider configuration (username, password, email address) as well as the optional secret question and answer if necessary.

You have full control over which steps are shown in the Wizard and you can define extra steps and add them to the *WizardSteps* collection for the control. If you need to, you can also override the two default steps – *CreateUserWizardStep* and *CompleteWizardStep* – and replace them with custom content.

In order to save the information entered in custom steps, you should handle the *CreatedUser* event (this is fired after a call to *Membership.CreateUser*) so you can save any profile information to the Membership Provider.

Set the *CancelDestinationPageUrl* to specify a page for the user to be taken to if they click the Cancel button.

Once the wizard is complete, the user is shown the *CompleteWizardStep* and is shown a Continue button. You can either specify the *ContinueDestinationPageUrl* property for the *CreateUserWizard* control or override the *ContinueButtonClick* event to transfer the user to the next page.

You can configure the email sent when a user registers using a *MailDefinition* element in the same way as for the *PasswordRecovery* control.

The ChangePassword Control

The *ChangePassword* control allows a user to change their password by entering their old password and a new password. If necessary, it will also ask for the answer for the user's secret question before changing the password.

You can configure the email sent when the password is changed using a *MailDefinition* element in the same way as for the *PasswordRecovery* control.

Creating ASP.NET Mobile Applications

Create a Mobile Web Application Project

A mobile Web Application is the same as a standard ASP.NET Web Application except that it uses Mobile Web Forms rather than a standard Web Form.

Whereas a Web Form inherits from *System.Web.UI.Page*, a Mobile Web Form inherits from *System.Web.UI.MobileControls.MobilePage*. The same applies to User Controls – a User Control inherits from *System.Web.UI.UserControl*, whereas a Mobile User Control inherits from *System.Web.UI.MobileControls.MobileUserControl*.

A Mobile Web Form uses Mobile Controls to handle rendering correctly to mobile devices. These controls are all contained within the *System.Web.UI.MobileControls* namespace and use the *mobile* prefix (rather than *asp*). This prefix is not registered by default, so at the top of all Mobile Web Forms and Mobile User Controls you'll see the following *@Register* directive:

```
<%@ Register TagPrefix="mobile" Assembly="System.Web.Mobile" ~CCC
  Namespace="System.Web.UI.MobileControls" %>
```

Session State

Most mobile devices don't allow cookies, so you should make sessions cookieless by setting the *sessionState* element in Web.config:

```
<sessionState cookieless="true" />
```

Multiple Forms

Unlike a Web Form, a Mobile Web Form does not contain a *<Form>* element. Mobile Web Forms use a *<mobile:Form>* to contain the controls that are displayed and each Mobile Web Form can contain several *<mobile:Form>* controls. Only one *<mobile:Form>* can be displayed at a time and this is controlled with the *ActiveForm* property of the Mobile Web Form.

Creating Mobile Web Forms and Mobile User Controls

Creating a Mobile Web Form, or Mobile User Control, is similar to creating a normal Web Form or User Control. From the Add New Item dialog you can choose the correct item to add as shown in **Figure 13**.

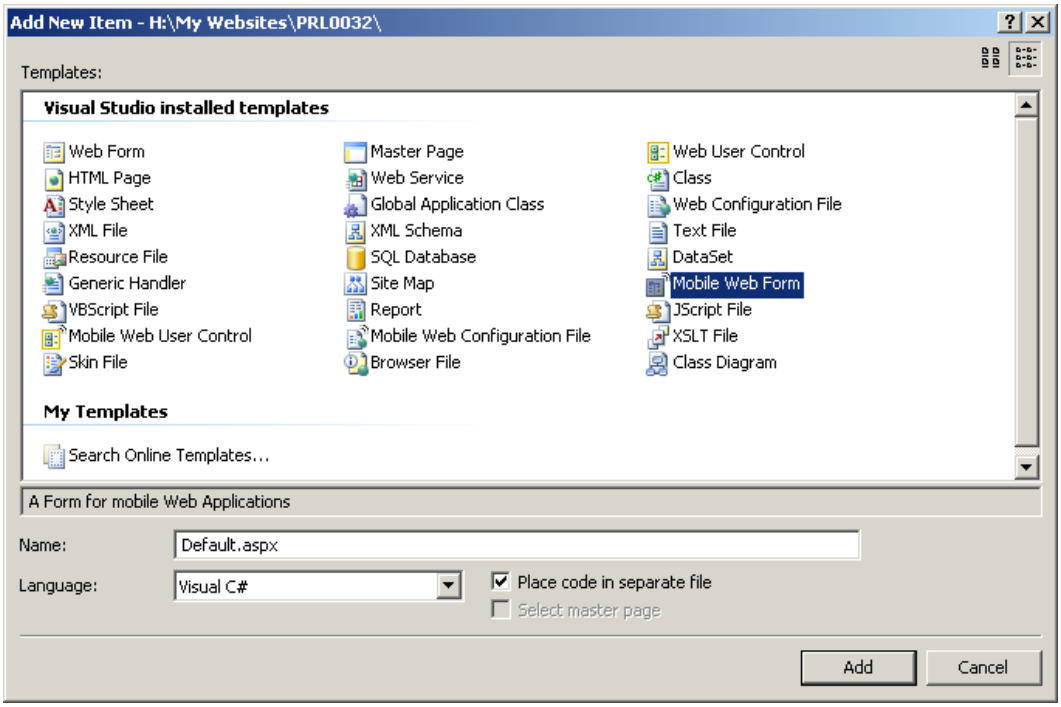

Figure 13 – Adding a Mobile Web Form using the Add New Item dialog

Use Mobile Web Controls

Mobile Web Controls are contained in the *System.Web.UI.MobileControls* namespace. They provide a subset of the functionality available through normal Web Controls. Some of the common Mobile Web Controls and their Web Control equivalents are shown in the following table:

Control	Description
Command	Provides the functionality to post user input from UI elements back to the server, similar to the *Button*, *ImageButton* and *LinkButton* Web Controls.
Image	Used to display an image from a set of device-specific images.
Label	Equivalent to the *Label* Web Control.
Link	Equivalent to a *HyperLink* Web Control and displays a text link. An image link should be created using the *Image* Mobile Web Control and setting the *NavigateUrl* property.
List	Similar functionality to the *DataList* and *Repeater* Web Controls.
ObjectList	Similar functionality to the *DataGrid* Web Control.
SelectionList	Combines the functionality of the *CheckBox*, *CheckBoxList*, *DropDownList*, *ListBox*, *RadioButton* and *RadioButtonList* Web Controls. The *SelectType* property is used to specify which type of list is displayed.
TextBox	Equivalent to a *TextBox* Web Control, except there is no automatic post back, read-only or multiline functionality.
ValidationSummary	Equivalent to the *ValidationSummary* Web Control. The *CompareValidator*, *CustomValidator*, *RangeValidator*, *RegularExpressionValidator* and *RequiredFieldValidator* also have Mobile Web Form equivalents.

Using Styles

Mobile Web Forms do not use CSS style sheets. Instead, they make use of the *StyleSheet* Mobile Web Control. Within the *StyleSheet*, you define several *Style* Mobile Web Controls that define the styles to be applied. These styles are then applied to Mobile Web Controls using the *StyleReference* property. The *Style* control only supports a limited subset of styles – *BackColor*, *Font* and *ForeColor*.

Use Adaptive Rendering

Adaptive rendering is the process a control uses to render differently based upon the browser that is requesting the Web Form. The User-Agent header, passed as part of the request, is used to identify the browser.

Adaptive rendering can be used by all controls; a few of the normal Web Controls use adaptive rendering, too. Mobile Web Controls, however, make extensive use of adaptive rendering. If you look at the *System.Web.UI.MobileControls.Adapters* namespace, you'll see that there are several classes that can be broadly grouped into three different Adapter Sets:

- CHTML – these adapters are used to render the control to Compact HTML and are prefixed with Chtml.
- HTML – these adapters are used to render the control to standard HTML and are prefixed with Html.
- WML – these adapters are used to render the control to Wireless Markup Language and are prefixed with Wml.

Which adapter set is chosen depends on the capabilities of the calling browser. If the browser supports HTML 3.2 and JavaScript, the HTML Adapter Set is used. The CHTML Adapter Set is used if the browser does support HTML 3.2 but doesn't support JavaScript. The WML Adapter Set is used if the browse supports WML 1.1.

You'll see that nearly all the Mobile Web Controls have an adapter – there is a *ChtmlLinkAdapter*, an *HtmlLinkAdapter* and a *WmlLinkAdapter*. However you will notice that not all Mobile Web Controls have a CHTML adapter – there is an *HtmlTextViewAdapter* but there isn't an equivalent CHTML adapter.

The CHTML Adapter Set is derived from the HTML Adapter Set. Only HTML Adapter Set controls that use scripting have an equivalent CHTML Adapter. Otherwise, the HTML Adapter is used.

Selecting Which Adaptive Rendering to Use

Control Adapters are attached to Mobile Controls automatically by the ASP.NET runtime. Depending on the specific browser's functionality, a different Adapter Set is applied.

The different Adapter Sets are specified in the global *Web.config* file (usually stored in the *C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\CONFIG* folder).

Under the <*system.web*><*mobileControls*> element, there is a <*device*> element for each Adapter Set. The <*device*> elements can also inherit from each other and you can see that the CHTML Adapter Set (*ChtmlDeviceAdapters*) inherits from the HTML Adapter Set (*HtmlDeviceAdapters*).

Within each Adapter Set definition there are <*control*> elements that specify the *name* of the control and the *adapter* that is to be used.

Overriding Adaptive Rendering Settings

It is also possible to override the Control Adapter used in specific instances by defining a .browser file in the *App_Browsers* folder of the Web site. You can use this method of specifying Control Adapters for normal Web Controls and Mobile Web Controls. Any settings defined in the *App_Browsers* folder override any specific Mobile Control settings and also any settings defined in the global browsers list (at *C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\CONFIG\Browsers*).

A .browser file contains a <*browsers*> element which contains individual <*browser*> elements that map to a specific browser using the *refID* attribute (pointing at a specific <*browser*> definition in the global browsers list). Within this <*browser*> element, you can specify a Control Adapter to use for a given control in an <*adapter*> element:

```
<controlAdapters>
  <adapter controlType="System.Web.UI.WebControls.Menu"
     adapterType="System.Web.UI.WebControls.Adapters.MenuAdapter" />
</controlAdapters>
```

In this case we're specifying a *MenuAdapter* Control Adapter for the *Menu* Web Control. This will override any other Control Adapter specified for the *Menu* Web Control.

Use Device Specific Rendering

Device specific rendering is a method whereby you can determine what is output depending upon the functionality of the browser. Adaptive rendering is one form of this but you can also control what is rendered both in HTML markup and also directly in code.

Device Specific Rendering in Markup

You can provide device specific rendering in markup using the *DeviceSpecific* Mobile Web Control. This can be applied as a child control within all Mobile Web Controls (this functionality is inherited from the base *MobileControl* class).

This control is tied to the <*deviceFilters*> element in a mobile Web.config and there are several predefined filters, including *isHTML32*, *isPocketIE*, *supportsColor* and *supportsJavaScript*. These are compare filters and you can define your own in Web.Config in the <*system.web*><*deviceFilters*> element by specifying a <*filter*> with a *name*, *compare* and *argument*. For instance the *isHTML32* filter is defined as follows:

```
<filter name="isHTML32" compare="PreferredRenderingType"
  argument="html32" />
```

You can also define an evaluator filter that calls a method (returning a boolean) on a specific class by specifying a filter with a *name*, *type* and *method*. For instance:

```
<filter name="isAfternoon" type="className" method="methodName" />
```

Both compare and evaluator filters can be used as the *Filter* attribute to a <*Choice*> element of the *DeviceSpecific* control:

```
<mobile:Label ID="Label1" runat="server>
  <DeviceSpecific>
    <Choice Filter="isPocketIE" Text="Is running Pocket ID" />
    <Choice Filter="isHTML32" Text="Supports HTML 3.2" />
    <Choice Text="Not Pocket IE and no support for HTML 3.2" />
  </DeviceSpecific>
</mobile:Label>
```

The <*Choice*> elements are checked in order and the first one that matches sets the *Text* property of the containing control. If you don't specify a Filter attribute to a <*Choice*> element, then that will be used as the default value.

Device Specific Rendering in Code

You can also check the capabilities of the browser when rendering the page using the *Request.Browser* property. This returns an *HttpBrowserCapabilities* class that you can interrogate to determine what to display.

The list of properties for the *HttpBrowserCapabilities* class is extensive. You can check if the browser is a mobile device (*IsMobileDevice*), whether JavaScript is supported (*JavaScript*), the version of the browser (*MajorVersion* and *MinorVersion*), whether Java and ActiveX are supported (*JavaApplets* and *ActiveXControls* respectively), and return the User-Agent string (*Browser*).

Using the properties available you can show or hide certain controls, change the text or images displayed, and so on.