CHAPTER 14

Additional ASP.NET Features

LEARNING OBJECTIVES

- **To be able to upload image data to a web application**
- **To be able to manage image data in a database**
- **To be able to use database transactions**
- **To be able to use web parts to create a portal-style web application**

INTRODUCTION

In this chapter we look at a number of features of ASP.NET that might be considered advanced, but can nonetheless be very useful. These features include managing image data in a web application (including storing images in a database), caching, transaction management, and web parts. The first part of the chapter uses an example feature from a web application, uploading digital images, and uses this feature to explore different strategies for managing image data. In the second part of the chapter, the ASP.NET WebPart controls are covered, and an example page developed to illustrate how easily a simple web portal can be created by making use of their rich feature set. We also create another ASP.NET web user control to include in the portal.

14.1 Managing image data in a web application

In this section we will develop some web application components to upload and display digital images or photographs. Key decisions involved in this development include whether to store the images in the file system, or in a database, and how to format them for display. The application gives an opportunity to explore ASP.NET database access in more detail, and illustrate the use of database transactions and binary (BLOB) fields. We then explore the mechanisms provided by ASP.NET for caching web pages and data in order to boost performance.

NOTE	Some of the examples in this chapter do not function correctly using Visual Web Developer's internal web browser. If you notice any problems, you are advised to test the pages instead using the 'File' → 'Browse With . . . ' command and then selecting Internet Explorer as the browser to use.

14.1.1 The image upload example

A common feature of many websites is the ability to post and share digital images or photographs. Indeed, this is one of the 'killer applications' driving the growth of Web 2.0 sites such as Facebook. Whereas email, forums and web blogs allow users to communicate and share information in textual form, media sharing seems to be more effective in attracting a larger community. Popular digital media include music, still images and video. Each of these has its own issues and concerns, not just technical ones but also a range of legal and social considerations. From a technical perspective, the problems include how to store what are often rather large files, and how to index them to support effective and efficient retrieval. Legal and social issues include copyright and privacy.

The insurance company we are considering might have two main business reasons for including images or videos on its website: firstly to host adverts or promotional material, and secondly to allow policy holders to augment the claims they submit. We have already covered the ASP.NET FileUpload control in Chapter 8, which allows you to upload files to a folder, so in this chapter we further develop the scenario here in the context of WebHomeCover user stories, whereby claimants can upload digital photographs in support of their claim. There are two main user stories involved: firstly the upload itself, and secondly a facility whereby claim processing staff can view the uploaded images associated with a specific claim. In addition, media sharing sites also need to include some house-keeping operations, but we do not consider those here. Indeed, for this application, it is likely the insurance company will choose to retain the submitted images for an extended period, and possibly permanently. Image retention allows the company to go back and review claims for auditing purposes (auditing is an important aspect of any business, of course). Other aspects of house-keeping include managing the storage required and the need for regular and adequate back-ups. As indicated above, however, we are not covering those operations here.

Instead our concern is the two use case features described in Table 14.1.

We have considered in earlier chapters how to manage users, allow them to log in, and navigate to a web page having selected policy and claim IDs from a drop down list. The focus here will therefore be on implementing two web forms, one for upload, and the other to view the uploaded images.

14.1.2 Image Upload

This section deals with the image upload page.

The main design decision we must consider is where to store the images. There are three obvious approaches: 1) store them in the file system, 2) in the database, or 3) a hybrid

TABLE 14.1 Use cases for Image Upload and View

Upload Image	*View Images*
Pre-requisite: a valid policy and claim	Pre-requisite: claim has images to view
User logs in and selects an open claim User clicks to agree disclaimer User supplies file name, title and description for image file User uploads photograph	Employee logs in Employee selects policy and claim ID
Post-condition: image URL is in database	Post-condition: All associated images are displayed, together with their title and description

approach in which the images are stored in the file system, but their URLs and other associated information are stored in the database.

The first approach is deceptively simple. We have already shown how image files can be uploaded directly, and only a modest amount of code is required. However, in this use case, we also require a title and description to be supplied and stored. It is possible to find a workaround such as incorporating them into the file name in some ingenious way, or to store them in some other file, possibly in XML format, with the same name as the image, but a different file extension, for example, 'image123.jpg' and 'image123.xml'. Either approach requires additional programming, so there is now little reason not to use a database. One advantage of using a database is that it supports pre-programmed and ad hoc queries. If we want to discover whether users in a certain part of the country are more likely to upload images, this only takes a few lines of SQL if we are using a database. If the images and associated metadata are all stored in the file system, however, rather more code is required for this or any other ad hoc query. A final disadvantage is that in this approach the user uploading the image typically has fairly direct access to the web server's file store, a potential security hazard. For example, can they see or over-write any critical files, or images from other users?

The second approach would be to store both the images and their associated metadata in our database somehow. This would require an additional table or tables to be defined. Storing images directly in a database requires the use of a blob (Binary Large OBject) or similar column. Modern databases such as SQL Server have good support for blobs, but additional programming is required to insert and retrieve them. For example, to display an image file directly from a database requires additional code both to extract the binary data from the database and also to return it to the web browser, typically using the Response.BinaryWrite method. Table 14.2 lists some of the advantages and disadvantages of these approaches.

As you can see, there are many advantages in storing the image files directly in the database, but some disadvantages also. In particular, we need to consider carefully whether it is best to allocate the additional storage in the database or web server. In addition, careful analysis is needed to understand the potential impact on traffic between the web and database servers.

The third or hybrid solution attempts to combine the advantages of the two approaches by storing the image files in the web server file system and the associated metadata such as title and description in the database, together with the image filename or URL. With this

TABLE 14.2 Comparing storing images in the database and storing images in the file system

Storing Images in the Database	*Storing Images in the File System*
Can only be managed by database administration software	Can be managed by file-system oriented utilities
Easy to code ad hoc queries using SQL	Ad hoc queries are more difficult to code
Database can manage referential integrity	External utilities required for image file house-keeping
Image upload can be included in an ACID (*Atomic, Consistent, Isolated, Durable*) database transaction	Transactional updates require additional programming
Standard security considerations only	Allowing users write access to the web server file system may create additional security issues
Database tables may become very large	Need to manage web server file space
Increases the traffic sent from the database server to the web server	Image files are not sent between the web server and database servers

approach, traffic between the database and web servers is minimized, but it is still possible to obtain some of the benefits of the previous approach such as the ability to write ad hoc queries in SQL, and images can be managed by file-oriented utilities. The hybrid approach leaves unresolved the issues of referential integrity and transactional updates, however.

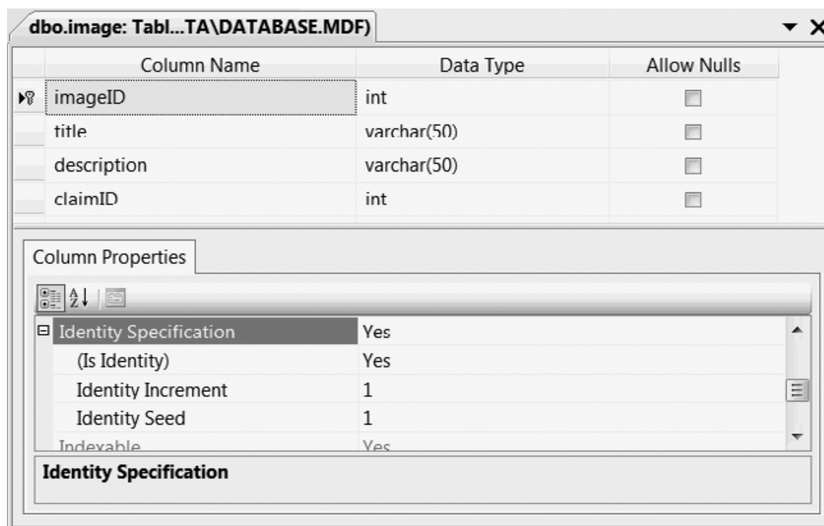
Balancing the benefits and disadvantages, it seems the third approach has a modest advantage over the second one in our case.

It is also worth considering the disclaimer text which the policy holders must agree to before they upload an image to the insurance company website. Obviously, they must consent to the company using the image for business purposes such as processing the claim and auditing. By doing so, the policy holder should agree that they will not undertake any copyright claim or charge against the company. Of course, this requires that the policy holder warrants that they are the ones who own the image, and that the image does not contain any obscene or illegal material. Clearly, the legal department must write the disclaimer to ensure it serves the required purpose and is legally binding. As web developers, our responsibility is to ensure the user reads the disclaimer, or more realistically that they click a button to agree that they have read the disclaimer and agree to be bound to it. As disclaimers are often quite long, it is increasingly common to include a hyperlink rather than the disclaimer itself. Occasionally sites monitor the user to ensure they actually follow this link before they are allowed to agree to the terms and conditions, although this can be annoying for some users.

14.1.3 Image upload

Before this solution can be implemented, a new database table is required to store the image IDs and metadata. For simplicity, each image will have a file name such as 1.jpg, 2.jpg, and so on (it would be a relatively simple matter to allow different types of image file by storing this as an additional field in the database). The image files are also located

FIGURE 14.1 Defining the new image table in the database. The imageID is an automatically incremented identity field



in a single directory which the web server has write access to. Thus the file names on the server are likely to be different from the ones the user has chosen. This avoids the potential security hazard of attempting to access a file whose name the user has specified, and the need to check their input for special characters.

As you can see in Figure 14.1, the 'imageID' is specified as a database identity field, with automatically generated values, starting from 1 and incrementing by one each time a new image is uploaded. The title and description are simple strings, and the 'claimID' is a foreign key relating the image to the relevant claim.

Once the table has been created, an image upload web form can be built and tested, as shown in Figure 14.2. This form is a variant of the file upload page described in Chapter 7. In addition to the file name, it also includes fields whereby the claim reference, title and description can be entered. There are also a hyperlink and check box for the terms and conditions, as described above. The file upload browse dialog can be used to select the required image file, and the Upload button is used to submit the selected image file. In the screen shot below, assume that the user has clicked on the hyperlink, read the terms and conditions, clicked on the check box to agree to them, typed in the required information and selected an image file for upload.

The code behind this web form validates the input and handles the file upload, recording the metadata in the database. First is the C# code:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;
using System.Linq;
```

```
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;

public partial class ImageUpload : System.Web.UI.Page
{
 protected void Page_Load(object sender, EventArgs e)
 {
  if (IsPostBack && FileUpload1.HasFile && CheckBox1.Checked)
  {
   System.Configuration.Configuration rootWebConfig =
System.Web.Configuration.WebConfigurationManager.OpenWebConfiguration
("~/Final");
   String myConnString =
rootWebConfig.ConnectionStrings.ConnectionStrings["ConnectionString"]
.ToString();
   SqlConnection myConn = new SqlConnection(myConnString);
   myConn.Open();
   SqlCommand myQuery = new SqlCommand
    ("SELECT claimID FROM claim WHERE reference=@reference", myConn);
   myQuery.Parameters.AddWithValue("@reference", TextBox1.Text);
   SqlDataReader myDR;
   myDR = myQuery.ExecuteReader();
   if (!myDR.Read())
   {
    Label1.Text = "Invalid claim reference";
    return;
   }
   int claimID = myDR.GetInt32(0);
   myDR.Close();
   SqlCommand myInsert = new SqlCommand("INSERT INTO image
    VALUES(@title,@description,@claimID)", myConn);
   myInsert.Parameters.AddWithValue("@title", TextBox2.Text);
   myInsert.Parameters.AddWithValue("@description", TextBox3.Text);
   myInsert.Parameters.AddWithValue("@claimID", claimID);
   int rows = myInsert.ExecuteNonQuery();
   if (rows != 1)
    throw new Exception("Unexpected Result for Insert");
   SqlCommand myCount = new SqlCommand
     ("SELECT MAX(imageID) AS IMAGEID FROM image", myConn);
   myDR = myCount.ExecuteReader();
   myDR.Read();
   int imageID = myDR.GetInt32(0);
   myDR.Close();
   String path = Server.MapPath("~/Uploads/");
   try
   {
    FileUpload1.PostedFile.SaveAs
```

```
        (path + imageID.ToString() + ".jpg");
      Label1.Text = "Upload successful";
     }
     catch(Exception myEx)
     {
      Label1.Text = "Upload failed";
     }
    }
   }
  }
```

Next is the Visual Basic code:

```
Imports System.Data
Imports System.Data.SqlClient
Imports System.Web.UI
Imports System.Web.UI.HtmlControls
Imports System.Web.UI.WebControls

Partial Class ImageUpload Inherits System.Web.UI.Page

 Protected Sub Page_Load(ByVal sender As Object, _
   ByVal e As EventArgs)
  If IsPostBack = True And FileUpload1.HasFile = True And _
   CheckBox1.Checked = True Then
    Dim rootWebConfig As System.Configuration.Configuration = _
    System.Web.Configuration.WebConfigurationManager.
    OpenWebConfiguration("~/Final")
   Dim myConnString As String = _
    rootWebConfig.ConnectionStrings.
    ConnectionStrings("ConnectionString").ToString()
   Dim myConn As SqlConnection = New SqlConnection(myConnString)
   myConn.Open()
   Dim myQuery As SqlCommand = New SqlCommand("SELECT claimID _
    FROM claim WHERE reference=@reference", myConn) _
    myQuery.Parameters.AddWithValue("@reference", _
    TextBox1.Text)
   Dim myDr As SqlDataReader
   myDr = myQuery.ExecuteReader()
   If myDr.Read() = False Then
    Label1.Text = "Invalid claim reference"
    Return
   End If
   Dim claimID As Integer = myDr.GetInt32(0)
   myDr.Close()
   Dim myInsert As SqlCommand = New SqlCommand("INSERT INTO _
    image VALUES(@title,@description,@claimID)", myConn)
   myInsert.Parameters.AddWithValue("@title", TextBox2.Text)
   myInsert.Parameters.AddWithValue("@description", _
     TextBox3.Text)
   myInsert.Parameters.AddWithValue("@claimID", claimID)
   Dim rows As Integer = myInsert.ExecuteNonQuery()
```

14

```
      If rows <> 1 Then
       Throw New Exception("Unexpected Result for Insert")
      End If
      Dim myCount As SqlCommand = New SqlCommand("SELECT _
        MAX(imageID) AS IMAGEID FROM image", myConn)
      myDr = myCount.ExecuteReader()
      myDr.Read()
      Dim imageID As Integer = myDr.GetInt32(0)
      myDr.Close()
      Dim path As String = Server.MapPath("~/Uploads/")
      Try
       FileUpload1.PostedFile.SaveAs(path + imageID.ToString() _
        + ".jpg")
        Label1.Text = "Upload successful"
      Catch myEx As Exception
       Label1.Text = "Upload failed"
      End Try
     End If
   End Sub
  End Class
```

This code uses a connection string which must be defined in the application's web.config file using a 'connectionStrings' element similar to the following (as seen in Chapter 11):

```
<connectionStrings>
 <add name="ConnectionString"
   connectionString="DataSource=.\SQLEXPRESS;AttachDbFilename=
   |DataDirectory|\Database.mdf;
   Integrated Security=True;User Instance=True"
  providerName="System.Data.SqlClient"/>
</connectionStrings>
```

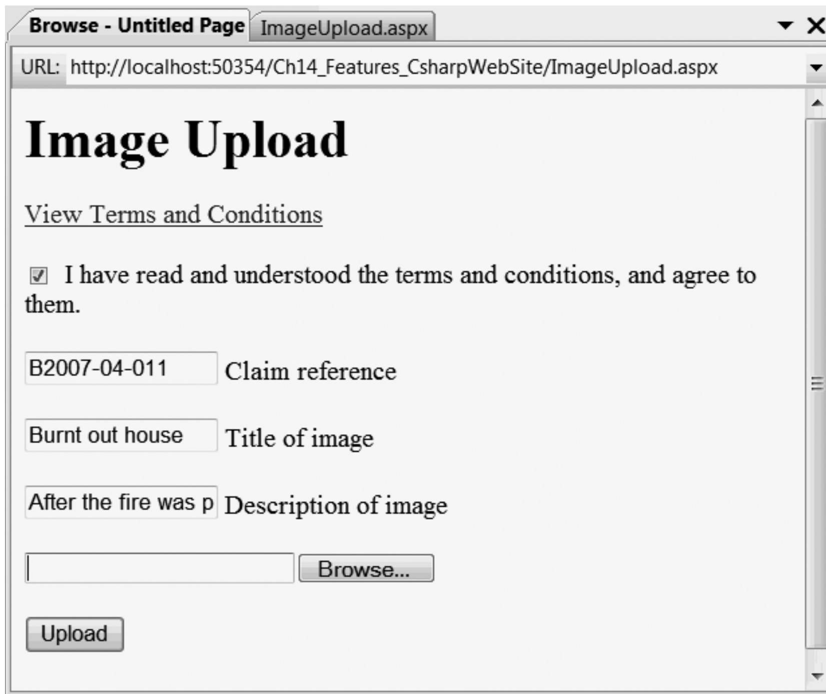

The claim reference supplied by the user is validated before the data is inserted into the database. The new image ID is retrieved, and used as the file name to create the uploaded file.

A few comments can be made on this code. Although it is already a page or so in length, error handling is somewhat rudimentary. More feedback to the user is required if the upload fails, or indeed if it does not even start because they have failed to check the required check box. More seriously, the code is not transactional. It is possible that the claim could for some reason be deleted just after its reference has been validated; or that the image ID retrieved might in fact belong to some other image being uploaded 'at the same time'. To avoid these potential problems, ACID database transactions should be used, as explored later in this chapter.

14.1.4 Viewing the images

In this application, it is assumed the user, an authorized insurance company employee, wishes to view all images associated with a single claim. It is also assumed that the number of such images will be relatively small, from perhaps only one up to a couple of dozen at most. Identifying the images to retrieve is as simple as typing in the claim reference, or

FIGURE 14.2 The Image Upload web form viewed in a browser



selecting it from a drop down list as in Chapter 10. There is no need in this case to provide a means of navigating around and selecting from the large number of images you might find in a typical photo-sharing application, typically using scaled down thumbnails of the original images.

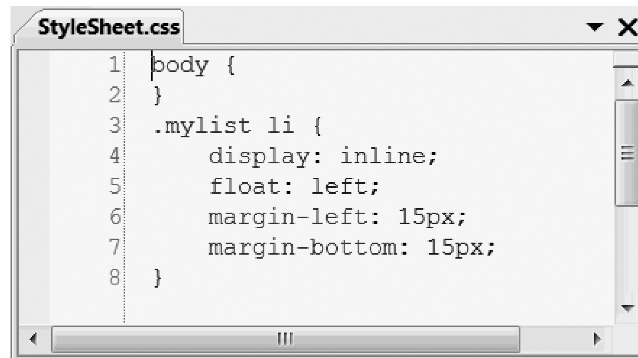
The main issues therefore concern the layout of the images. A common approach is to use a table with one of more columns, and fixed size cells. This has the advantage of providing a regular layout. It is difficult to know, however, how many columns to use. The optimal number depends on the width of the actual images, and the window used to display them. Both these quantities can vary, so it is not possible to determine the correct answer in advance. Instead, the calculation should be done at run-time, and preferably by the browser, as this has access to the current window geometry. This effect can be achieved by making each image a left-floating element so that as many images as can fit are displayed on each line before the next one is started.

This effect can be achieved with a stylesheet such as the one in Figure 14.3 on the following page.

To make use of this stylesheet, our images must be displayed using a list rather than a table. To do this, we need to make use of a different data control than the ones we have seen so far, all of which output HTML tables. Instead, we can use the ListView control, which was introduced in ASP.NET 3.5 to provide a more flexible and precisely defined output. The .aspx file shown below indicates how this control can be used.

In this web form there are two controls, an SqlDataSource which retrieves the records associated with a specific claim from the database, and a ListView control which displays the

FIGURE 14.3 A CSS stylesheet which specifies that list items are floated in-line with a 15 pixel gap to the left and below



data retrieved by the first control. The ListView consists of a number of templates each of which can be defined graphically or, as here, using ASP.NET mark-up. The Empty DataTemplate defines what is to be displayed when the data source finds no records to return. Otherwise, the LayoutTemplate defines the mark-up to be generated at the start and end of the displaying the selected records. In this case, our LayoutTemplate consists of an unordered list matching the CSS class defined above. Inside the list is a placeholder control:

```
<asp:PlaceHolder ID="itemPlaceholder" runat="server"/>
```

At run-time, this control is replaced by the ItemTemplate control which, as you can see below, consists of an HTML list item ('li') element containing two labels, one for the title and one for the description, and an 'asp:image' control to display the uploaded images one at a time.

Some comments on this mark-up are in order. Firstly, note the use of the data binding syntax:

```
Text='<%# Eval("title") %>'
```

We have seen this syntax before when editing data control templates, but not entered it directly ourselves. At run-time the data-binding text is replaced by the correspondingly named field from the current database record. Note also the alternative form of the data-binding syntax in which a format specifier {0} is used to indicate where the database field is to be inserted. Secondly, note the images have specified widths and heights. In practice, any fixed width and height is unlikely to be appropriate for the given set of images, of course. Finally, note that the template mechanism described here is flexible enough to allow ListView controls to output lists, tables, or other HTML structures.

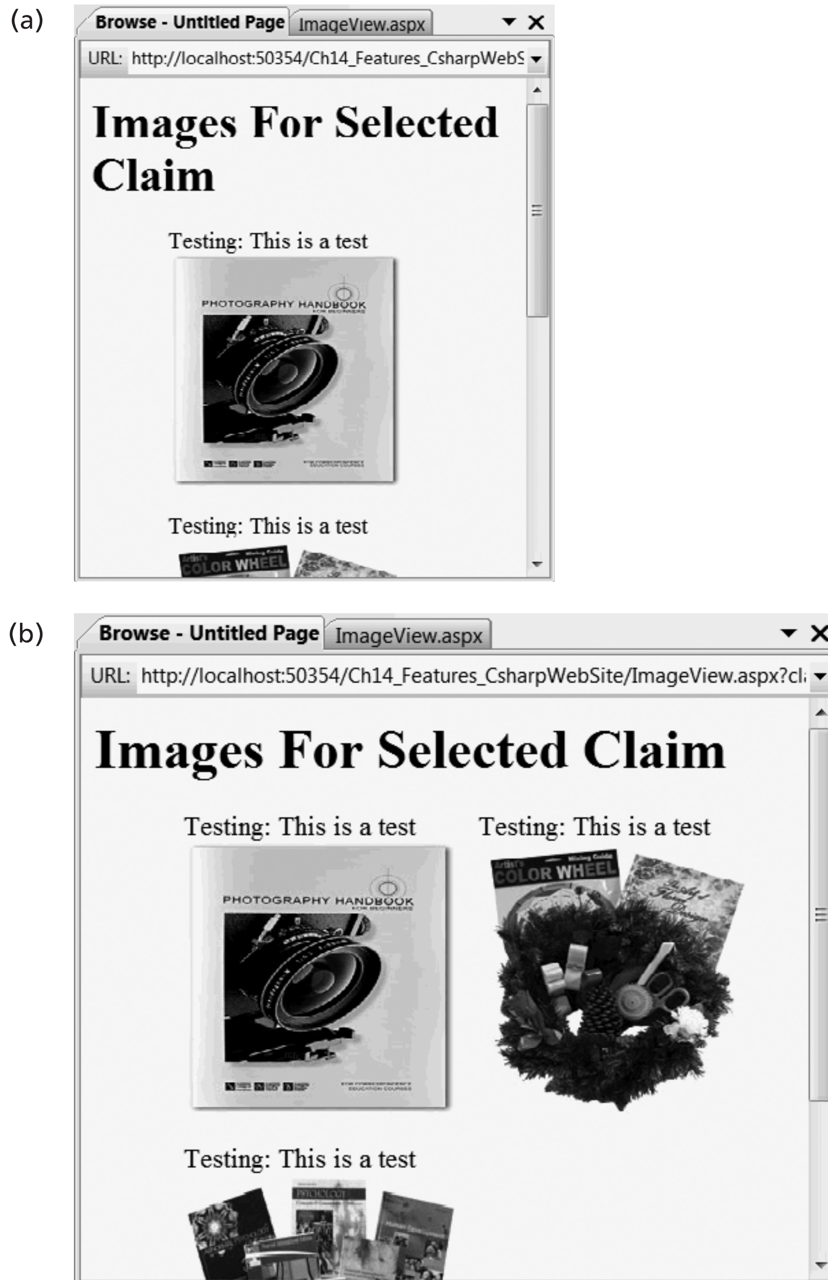
In addition to the templates described here, the ListView control moreover supports others which are used when the data is to be edited, which requires that two-way data binding is used:

```
<html xmlns="http://www.w3.org/1999/xhtml">
 <head runat="server">
```

```
    <title>Untitled Page</title>
    <link type="text/css" rel="stylesheet" href="StyleSheet.css">
   </head>
   <body>
    <form id="form1" runat="server">
     <h1>Images For Selected Claim</h1>
     <p>
      <asp:SqlDataSource ID="SqlDataSource1" runat="server"
        ConnectionString="<%$ ConnectionStrings:ConnectionString %>"
        SelectCommand="SELECT [imageID], [title], [description]
         FROM [image] WHERE ([claimID] = @claimID)">
      <SelectParameters>
        <asp:QueryStringParameter
          Name="claimID" QueryStringField="claimID" Type="Int32" />
       </SelectParameters>
      </asp:SqlDataSource>
     </p>
     <p>
      <asp:ListView ID="ListView1" runat="server"
         DataKeyNames="imageID"
        DataSourceID="SqlDataSource1">
        <EmptyDataTemplate>
         <p>No images found</p>
        </EmptyDataTemplate>
        <LayoutTemplate>
         <ul class="mylist">
         <asp:PlaceHolder ID="itemPlaceholder" runat="server"/>
        </ul>
       </LayoutTemplate>
       <ItemTemplate>
        <li>
         <asp:Label ID="titleLabel" runat="server"
            Text='<%# Eval("title") %>' />:
        <asp:Label ID="descriptionLabel" runat="server"
            Text='<%# Eval("description") %>' />
        <br />
       <asp:Image ID="Image1" runat="server" Width=200px Height=200px
         ImageUrl='<%# Eval("imageID", "~/Uploads/{0}.jpg") %>' />
        </li>
       </ItemTemplate>
      </asp:ListView>
     </p>
    </form>
   </body>
  </html>
```

The screen shots in Figure 14.4 show this web form in operation. Note the way that the number of images on each line varies as the window is re-sized: as the window is widened, the page shows first one, then two, images per row, and so on.

FIGURE 14.4 This image gallery uses floating elements to fit as many images as possible on one line



14.2 Image database storage and transactions

Having considered the hybrid solution, it is now time to look at solution two, in which the images themselves are stored, together with their associated metadata, in the database itself. In this section, we create variants called ImageUpload2.aspx and ImageView2.aspx of the

original web forms (ImageUpload.aspx and ImageView.aspx). In addition, a new image table must be defined in the database, which we will call image2. This will have the same fields as the original image table, but with one additional field, called 'image', of type varbinary(MAX).

The web form for uploading an image does not need to change, but the code behind (ImageUpload2.aspx.cs) is now rather different, as shown below, first in C#:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;

public partial class ImageUpload : System.Web.UI.Page
{
 protected void Page_Load(object sender, EventArgs e)
 {
  if (IsPostBack && FileUpload1.HasFile && CheckBox1.Checked)
  {
   System.Configuration.Configuration rootWebConfig =
System.Web.Configuration.WebConfigurationManager.OpenWebConfiguration
("~/Final");
   String myConnString =
rootWebConfig.ConnectionStrings.ConnectionStrings["ConnectionString"]
.ToString();
   SqlConnection myConn = new SqlConnection(myConnString);
   myConn.Open();
   SqlTransaction myTrans = myConn.BeginTransaction();
   try
   {
    SqlCommand myQuery = new SqlCommand
     ("SELECT claimID FROM claim WHERE reference=@reference",
        myConn);
    myQuery.Transaction = myTrans;
    myQuery.Parameters.AddWithValue("@reference", TextBox1.Text);
    SqlDataReader myDR;
    myDR = myQuery.ExecuteReader();
    if (!myDR.Read())
    {
     Label1.Text = "Invalid claim reference";
     return;
    }
    int claimID = myDR.GetInt32(0);
```

```
        myDR.Close();
        Byte[] imageBytes = new
    Byte[FileUpload1.PostedFile.InputStream.Length];
        FileUpload1.PostedFile.InputStream.Read(imageBytes, 0,
    imageBytes.Length);
        SqlCommand myInsert = new SqlCommand
        ("INSERT INTO image2
     VALUES(@title,@description,@claimID,@image)",
        myConn);
        myInsert.Transaction = myTrans;
        myInsert.Parameters.AddWithValue("@title", TextBox2.Text);
        myInsert.Parameters.AddWithValue("@description", TextBox3.Text);
        myInsert.Parameters.AddWithValue("@claimID", claimID);
        myInsert.Parameters.AddWithValue("@image", imageBytes);
        int rows = myInsert.ExecuteNonQuery();
        if (rows != 1) throw new Exception("Unexpected Result for
    Insert");
        myTrans.Commit();
        TextBox1.Text = "";
        TextBox2.Text = "";
        TextBox3.Text = "";
       }
       catch (Exception myEx)
       {
        myTrans.Rollback();
       }
      }
     }
    }
```

The equivalent VB code is as follows:

```
Imports System.Data
Imports System.Data.SqlClient
Imports System.Web.UI
Imports System.Web.UI.HtmlControls
Imports System.Web.UI.WebControls

Partial Class ImageUpload2 Inherits System.Web.UI.Page

 Protected Sub Page_Load(ByVal sender As Object,
   ByVal e As EventArgs)
  If IsPostBack = True And FileUpload1.HasFile = True And _
   CheckBox1.Checked = True Then
   Dim rootWebConfig As System.Configuration.Configuration = _
    System.Web.Configuration.WebConfigurationManager.
    OpenWebConfiguration("~/Final")

   Dim myConnString As String = _
    rootWebConfig.ConnectionStrings.
    ConnectionStrings("ConnectionString").ToString()

   Dim myConn As SqlConnection = New SqlConnection(myConnString)
   myConn.Open()
```

```
        Dim myTrans As SqlTransaction = myConn.BeginTransaction()

        Try
         Dim myQuery As SqlCommand = New SqlCommand("SELECT claimID _
           FROM claim WHERE reference=@reference", myConn)
         myQuery.Transaction = myTrans
         myQuery.Parameters.AddWithValue("@reference", TextBox1.Text)
         Dim myDR As SqlDataReader
         myDR = myQuery.ExecuteReader()
         If myDR.Read() = False Then
          Label1.Text = "Invalid claim reference"
          Return
         End If
         Dim claimID As Integer = myDR.GetInt32(0)
         myDR.Close()
         Dim imageBytes As Byte()
         ReDim imageBytes(FileUpload1.PostedFile.InputStream.Length)
         FileUpload1.PostedFile.InputStream.Read(imageBytes, 0, _
           FileUpload1.PostedFile.InputStream.Length)
         Dim myInsert As SqlCommand = New SqlCommand("INSERT INTO _
           image2 VALUES(@title,@description,@claimID,@image)", _
           myConn)
         myInsert.Transaction = myTrans
         myInsert.Parameters.AddWithValue("@title", TextBox2.Text)
         myInsert.Parameters.AddWithValue("@description", _
           TextBox3.Text)
         myInsert.Parameters.AddWithValue("@claimID", claimID)
         myInsert.Parameters.AddWithValue("@image", imageBytes)
         Dim rows As Integer = myInsert.ExecuteNonQuery()
         If rows <> 1 Then
          Throw New Exception("Unexpected Result for Insert")
         End If
         myTrans.Commit()

         Label1.Text = ""
         TextBox1.Text = ""
         TextBox2.Text = ""
         TextBox3.Text = ""

        Catch myEx As Exception
         Label1.Text = myEx.Message
         myTrans.Rollback()
        End Try
       End If
      End Sub
     End Class
```

Here an ACID database transaction is begun once the database connection has been opened. This transaction is used to ensure that the SQL query used to validate the claim reference is synchronized with the SQL command used to insert the image into the database. This guards against the (admittedly somewhat unlikely) situation where the claim is deleted after it has been validated but before the image is inserted. If all goes well, the

transaction is committed at the end of the normal flow of events. Alternatively, if there are any errors, the transaction is rolled back, leaving the database unchanged. As you can see, it only takes a few extra lines of code to make use of this powerful database facility.

The code to insert the image into the database is somewhat different from the code to save it to the file system, but hopefully self-explanatory.

Finally, if all goes well, the textboxes are cleared so that the user can, if they wish, enter new data and insert another image into the database. The check box is not cleared as it is only necessary for the user to read the disclaimer once.

14.2.1 Solution Two: Image View

With the images stored in the database, we need to change the different forms and code used to view the images. This is now more complex to implement. The 'img' element used in XHTML to display images expects them to be identified by a URL, but in our system the images are in fact identified by an imageID, which is an identity field in our image2 database table. How may we reconcile these two rather different notions of identity?

The solution is create another web form, ShowImage.aspx say, which will display the image whose integer ID is passed to it via the query string. Now image 3, for example, can be displayed using the URL ShowImage.aspx?imageID=3, and so on.

The ShowImage.aspx page has no XHTML mark-up, but just an ASP.NET header:

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="ShowImage.aspx.cs"
 Inherits="ShowImage" %>
```

The code behind file, ShowImage.aspx.cs, does all the work. Here is the C# code:


```
using System;
using System.Collections;
using System.Collections.Specialized;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;

public partial class ShowImage : System.Web.UI.Page
{
 protected void Page_Load(object sender, EventArgs e)
 {
```

```
 System.Configuration.Configuration rootWebConfig =
System.Web.Configuration.WebConfigurationManager.OpenWebConfiguration
("~/Final");
  String myConnString =
rootWebConfig.ConnectionStrings.ConnectionStrings["ConnectionString"]
.ToString();
  SqlConnection myConn = new SqlConnection(myConnString);
  myConn.Open();
  SqlCommand myQuery = new SqlCommand("SELECT image FROM image2 WHERE
 imageID=@imageID", myConn);
  NameValueCollection myColl = Request.QueryString;
/* assumed to be imageID = n */
  string[] myVals = myColl.GetValues(0);
  int imageID = Convert.ToInt32(myVals[0]);
  myQuery.Parameters.AddWithValue("@imageID", imageID);
  Response.ContentType = "image/jpeg";
  SqlDataReader myDR = myQuery.ExecuteReader();
  myDR.Read();
  byte[] myBytes = (byte[])myDR.GetValue(0);
  Response.OutputStream.Write(myBytes,0,myBytes.Length);
  myConn.Close();
  }
}
```

Here is the Visual Basic code:

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As EventArgs)
 Dim rootWebConfig As System.Configuration.Configuration = _
   System.Web.Configuration.WebConfigurationManager.
   OpenWebConfiguration("~/Final")
 Dim myConnString As String = _
   rootWebConfig.ConnectionStrings.
   ConnectionStrings("ConnectionString").ToString()

 Dim myConn As SqlConnection = New SqlConnection(myConnString)
 myConn.Open()

 Dim myQuery As SqlCommand = New SqlCommand("SELECT image FROM _
 image2 WHERE imageID=@imageID", myConn)

 Dim myColl As NameValueCollection = Request.QueryString
' assumed to be imageID = n
 Dim myVals As String() = myColl.GetValues(0)
 Dim imageID As Integer = Convert.ToInt32(myVals(0))
 myQuery.Parameters.AddWithValue("@imageID", imageID)
 Response.ContentType = "image/jpeg"
 Dim myDR As SqlDataReader = myQuery.ExecuteReader()
 myDR.Read()
 Dim myBytes As Byte() = CType(myDR.GetValue(0), Byte())
 Response.OutputStream.Write(myBytes, 0, myBytes.Length)
 myConn.Close()
End Sub
```

The database is connected to as usual, and the image retrieved using an SQL query such as the ones we have seen before. Now, however, the data being retrieved is in binary format, so a byte array (or other similar structure) is the best way to store it. Finally, the data is written to the HTTP response, whose content type is set to be a JPEG image ('image/jpeg'). The OutputStream.Write() method is used here to provide binary output.

Note that this form does not function correctly using Visual Web Developer's internal browser, but works happily with external browsers such as Internet Explorer.

Only modest changes are needed to convert the original ImageView.aspx into the required ImageView2.aspx form. The Image control used in the ListView control is easily changed to make use of these URLs for displaying images directly from our image2 database table. That is to say, the original line:

```
<asp:Image ID="Image1" runat="server" Width=200px Height=200px
  ImageUrl='<%# Eval("imageID", "~/Uploads/{0}.jpg") %>' />
```

should be replaced with:

```
<asp:Image ID="Image1" runat="server" Width=200px Height=200px
  ImageUrl='<%# Eval("imageID", "~/ShowImage.aspx?imageID={0}") %>'
/>
```

And the select command must be updated to refer to the image2 table:

```
SelectCommand="SELECT [imageID], [title], [description] FROM [image2]
WHERE ([claimID] = @claimID)">
```

This page now functions correctly, giving similar results to those shown in Figure 14.4, when used with Internet Explorer (rather than VWD's internal web browser).

Note, however, that the ShowImage.aspx page is, as it stands, a security hole. By supplying this arbitrary 'imageIDs', users can view any image in the database, hardly a desirable situation. To close this security hole, it is necessary to make use of the user membership database and APIs discussed in Chapter 12. For example, a check could be made that the user is in a certain role such as claimsApprover. Alternatively, if the user has logged in as a policy holder, they should only be able to view images relating to claims being made on one of their own policies. This requires only a modest amount of code, but it is necessary to wrap up the pages created in this chapter in a login system similar to the one developed in Chapter 12. The most important lesson, however, is that 'just adding another page' to a safe system can easily compromise security, so careful review is required for each such change.

Finally, it is interesting to note that SQL Server 2008 provides a hybrid solution of its own. Binary data can be marked with the FILESTREAM attribute, which means the data item is managed by SQL Server but actually stored in the file system.

14.3 ASP.NET caching

With pages such as the ones we have just developed, it is easy to imagine that performance could become an issue. If we have a lot of users accessing pages with multiple images and other rich content that is pulled from the database, response times may suffer. Fortunately,

ASP.NET includes a number of features which make it quite simple for you to boost the performance of your application using caching, a technique whereby commonly accessed data is stored in memory rather than being fetched or re-created each time. Introducing caching into your ASP.NET application can be as simple as adding a few extra lines of ASP.NET mark-up to your .aspx web forms.

Once an item has been added to the cache, it remains there unless or until a) it is explicitly removed, b) memory runs low and is needed by other items, c) the item expires, meaning that it reaches a pre-defined time for removal, or d) some other item on which it depends changes, so that the cached copy is no longer reliable.

To cache a page simply add the ‘OutputCache’ directive to the page header, for example this directive, which declares that the page should be cached for the next ten seconds:

```
<%@ OutputCache Duration="10">
```

Other attributes of this directive allow you to define the location of the cache, and whether to store multiple versions of the page depending on the supplied HTTP parameters (using the ‘VaryByParam’ attribute).

In addition to caching the whole page, you can ask ASP.NET to cache parts of a page. To do this, you should construct the page from user controls, then include an OutputCache directive in each user control you want to be cached. In this way, different fragments of the page can have different caching policies applied to them. Finally, note the Label control, which displays text on a web page. This is never cached, so it can be used for personalized information such as the user ID, or name, or address.

In addition to the simple, code-free, approaches to caching just described, ASP.NET provides classes whose methods allow you to control application and database caching using only a few lines of code. The Cache object allows you to Add, Insert and Remove items from the cache. Alternatively, rather than deleting cached items using method calls, you can specify an expiration policy, for example that the object is to be removed from the cache five minutes after being inserted, or at a specific time of day. Moreover, you can declare dependencies whereby if an object changes, cached objects which depend on it are automatically disposed of. The SqlCacheDependency class provides similar methods and facilities which allow you to cache the results of SQL select queries.

14.4 Web parts

The WebParts group of controls allows you to create a web portal or intranet site with only a modest amount of coding. A number of controls are available which work together to provide a sophisticated customizable web-based user interface. In this section we illustrate the use of the majority of these controls by putting together a skeleton web portal based on a table of WebPartZone controls.

Each web form that uses web parts must include a WebPartManager control, as shown in the Design view in Figure 14.5. In our very simple example we have added a table with two rows and two columns, and in each cell of the table have added a WebPartZone control from the WebParts group of controls. Inside each WebPartZone, a standard Label control

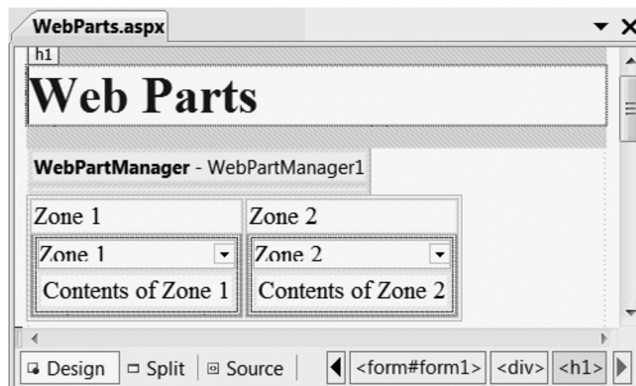
has been added. To start with, the WebPartZone will show as ‘untitled’ so you should add a ‘HeaderText’ attribute to each WebPartZone and also add a ‘Title’ attribute to each Label, which you must do in Source View, rather than Design View. In Source view, each of your cells should have mark-up similar to the following:

```
<asp:WebPartZone ID="WebPartZone1" runat="server"
    HeaderText="Zone 1">
 <ZoneTemplate>
  <asp:Label ID="Label1" runat="server" Title="Zone 1"
    Text="Contents of Zone 1">
  </asp:Label>
 </ZoneTemplate>
</asp:WebPartZone>
```

Viewing the page in the browser will then display the page with its table of controls as shown in Figure 14.6. The drop down arrow at the top of each zone gives access to its ‘verbs’ menu In this case, only the ‘Minimize’ and ‘Close’ verbs are available in this menu.

The user can interact with the web parts by minimizing or restoring them using the ‘verbs’ present in the menu associated zone that contains the web part. The state of the web parts

FIGURE 14.5 A web form with a WebPartManager control and two WebPartZone controls



14

FIGURE 14.6 The WebPartManager and WebPartZone controls viewed in a browser

is recorded in the personalization database, details of which were covered in Chapter 12. If, for example, you minimize one of the zones, close down the web page, and start it up again, you will see that it remains minimized, as shown for example in Figure 14.7.

If the user chooses to close one of the web parts, however, there is no easy way for them to restore it. The database administrator can rectify the problem by removing the appropriate record from the aspnet_PersonalizationPerUser table, but this is hardly a satisfactory solution. The simplest way to avoid the problem is to hide the Close verb, which can be done in the Page_Load event hander by including assignments such as:

```
WebPartZone1.CloseVerb.Visible = false
```

The Web Parts controls include several which allow users to change the appearance and behavior of their view of the system. To give a flavor of this in action, we will add an EditorZone control to the bottom of the web form, and include inside this control an AppearanceEditorPart. Finally, we will add a button below the EditorZone, with text such as 'Switch mode' and a click event handler. In Design view the ASP.NET mark-up should be similar to the following:

```
<asp:EditorZone ID="EditorZone1" runat="server">
 <ZoneTemplate>
  <asp:AppearanceEditorPart ID="AppearanceEditorPart1"
     runat="server" />
 </ZoneTemplate>
</asp:EditorZone>
<asp:Button ID="Button1" runat="server" onclick="Button1_Click"
     Text="Switch Mode" />
```

Note that the button has a click event handler. This should be programmed to toggle between web part display and edit modes using the following code:

```
protected void Button1_Click(object sender, EventArgs e)
{
 if (WebPartManager1.DisplayMode == WebPartManager.BrowseDisplayMode)
  WebPartManager1.DisplayMode = WebPartManager.EditDisplayMode;
 else
  WebPartManager1.DisplayMode = WebPartManager.BrowseDisplayMode;
}
```

and similarly in Visual Basic:

```
Protected Sub Button1_Click(ByVal sender As Object, _
  ByVal e As EventArgs)
    If WebPartManager1.DisplayMode Is _
      WebPartManager.BrowseDisplayMode Then
        WebPartManager1.DisplayMode = _
           WebPartManager.EditDisplayMode
    Else
      WebPartManager1.DisplayMode = _
        WebPartManager.BrowseDisplayMode
    End If
End Sub
```

The effect of these additions is shown in Figure 14.7. This shows the page being viewed in the browser. Clicking the Switch Mode button switches to Edit mode as shown in the right-hand pane. Each web part zone now has an additional verb in its menu, called simply Edit. Clicking this option brings up the appearance editor, which is shown in the final screen shot, and which allows the user to change the Title and appearance of the Zone.

In edit mode, each web part zone has an additional frame, and their zone menu now includes an 'Edit' verb. Clicking this brings up the web part Editor Zone, as shown in the screen shot, which allows the user to change the properties such as its Title. Changes are saved in the personalization database, so they will still apply the next time the web page is visited. A final feature of EditDisplayMode is that web parts may be dragged between zones. To see this feature in operation, you should view the page using Internet Explorer rather than Visual Web Developer's internal browser, then switch to Edit mode. If you now move your mouse over a web part, you will see that the mouse point turns into a four-way pointer, signifying that you can now drag and drop the web part. Move it across to another web zone, and drop it there. In Figure 14.8, for example, a web part is being moved back from zone 2 to zone 1. Note that zone 1 automatically displays the message 'Add a Web Part to this zone by dropping it here' when it contains no web parts. (Internet Explorer is used for this example rather than Visual Web Developer's internal web browser.)

FIGURE 14.7 The Web Parts example with an Editor Zone

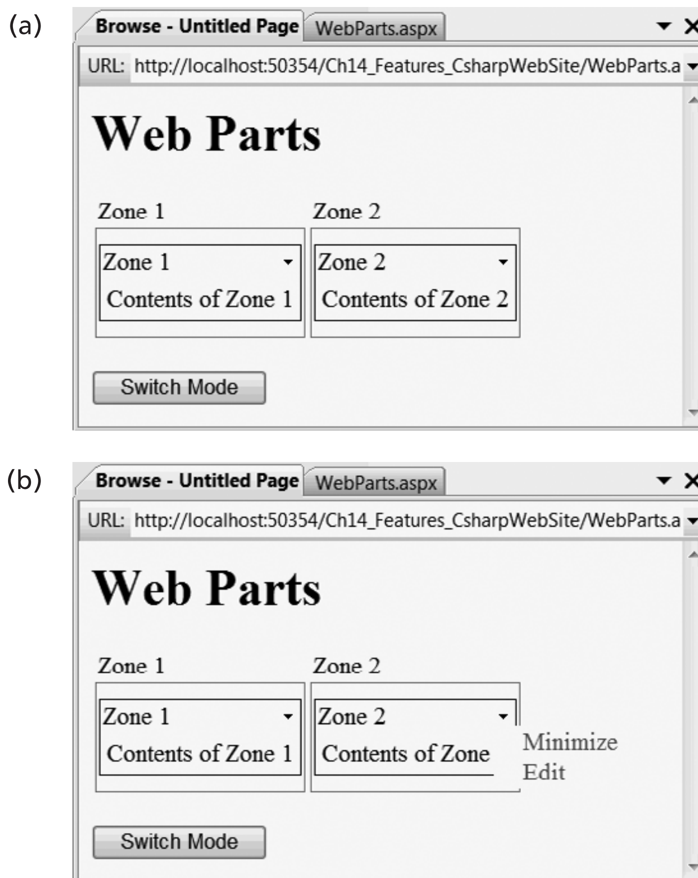


FIGURE 14.7 *Continued*

(c)

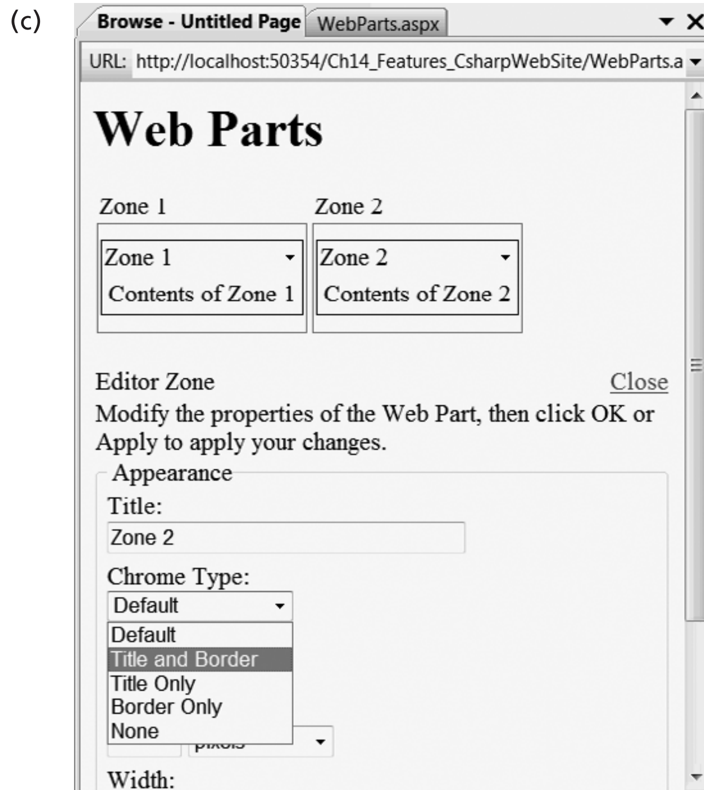


FIGURE 14.8 Dragging and dropping web parts between zones in web part Edit mode

14.5 Extending web parts with a web user control

So far, our web parts have been rather trivial, consisting of just a label and title. A web part can be any ASP.NET control so that you can choose to make a calendar, for example, into a web part. This also, however, does not meet the typical expectations of a web portal whereby the portal elements, or portlets, are expected to provide rich application-specific functionality. ASP.NET allows you to define your own web parts in two different ways, either as web user controls, or as custom built web parts. In this section, we consider the first of these two alternatives, the ASP.NET web user control.

As we saw in Chapter 7, when we created a web user control to use in the side bar of the master page, you can create a web user control using the 'File' → 'New File . . .' dialog. One of the file types this dialog offers you (assuming you currently have a website open) is the Web User Control. It is recommended you place your web user controls in their own folder or sub-directory of your website such as MyUserControls. Each user control has a name such as MyUserControl.ascx, and an associated code behind file such as MyUserControl.ascx.cs or .vb, just like an ASP.NET web form. As with web forms, you can include any ASP.NET control, and can set properties and add event handlers for each control in the same way. Thus, apart from the file extension, there is little difference between creating an ASP.NET web user control and a web form. As with web forms, you can edit user controls in Design view, or Source view, or a combination of these two approaches. For example, editing a simple web user control is shown below in Figure 14.9 immediately followed by the equivalent ASP.NET mark-up. Note that, apart from the .ascx extension, this is almost identical to editing an ASP.NET web form.

The mark-up which you see for this control in Source view is as follows:

```
<%@ Control Language="C#" AutoEventWireup="true"
CodeFile="MyUserControl.ascx.cs"
 Inherits="UserControls_MyUserControl" %>
<h1>Company Information</h1>
<p>
 <asp:Image ID="Image1" runat="server" AlternateText="Company Logo"
   ImageUrl="~/UserControls/marketing.jpg" />
</p>
```

14

FIGURE 14.9 Editing an ASP.NET web user control using Design view

As you can see, this is very similar to a web form. The main differences here are that the Inherits clause is somewhat different, and the <asp:form> element is missing.

The great thing about web user controls is that they can be used in a web form more or less as if they were ASP.NET web controls. To enable this, you must first add a Register directive to your web form. The following directive, for example, declares that the form will make use of the user control above, using an XHTML element of the form <abc:Logo />:

```
<%@ Register Tagprefix="abc" Tagname="Logo"
Src="~/UserControls/MyUserControl.ascx" %>
```

We can now extend our skeleton web portal by making use of the web user control we have just created. The ASP.NET mark-up below shows how this can be accomplished.

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="WebParts2.aspx.cs"
 Inherits="WebParts2" %>
<%@ Register Tagprefix="abc" Tagname="Logo"
Src="~/UserControls/MyUserControl.ascx" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
 <head runat="server">
  <title>Untitled Page</title>
 </head>
 <body>
  <form id="form1" runat="server">
   <div>
    <h1>Web Parts Example</h1>
    <asp:XmlDataSource ID="XmlDataSource1" runat="server"
       DataFile="Ads.xml" />
    <asp:WebPartManager ID="WebPartManager1" runat="server">
    </asp:WebPartManager>
    <table>
     <tr>
      <td>
       <asp:WebPartZone ID="WebPartZone1" runat="server">
        <ZoneTemplate>
         <asp:Calendar ID="Calendar1" runat="server"
           Title="My Calendar" />
        </ZoneTemplate>
       </asp:WebPartZone>
      </td>
      <td>
       <asp:WebPartZone ID="WebPartZone2" runat="server">
        <ZoneTemplate>
         <abc:Logo ID="Logo" runat="server" Title="Company Page" />
        </ZoneTemplate>
       </asp:WebPartZone>
      </td>
     </tr>
    </table>
```

```
<table>
 <tr>
  <td>
   <asp:CatalogZone ID="CatalogZone1" runat="server">
    <ZoneTemplate>
     <asp:PageCatalogPart id="PageCatalogPart1" runat="server"
         Title="My Page Catalog"/>
     <asp:DeclarativeCatalogPart id="DeclarativeCatalogPart1"
         runat="server">
      <WebPartsTemplate>
       <asp:Calendar runat="server" ID="Calendar1"
          Title="My Calendar"/>
       <asp:Label runat="server" ID="Label1"
          Title="My First Label" Text="A simple web part"/>
       <asp:Label runat="server" ID="Label2"
          Title="My Second Label" Text="Another web part"/>
       <abc:Logo runat="server" ID="Logo1" Title="Company Page"/>
      </WebPartsTemplate>
    </asp:DeclarativeCatalogPart>
   </ZoneTemplate>
  </asp:CatalogZone>
 </td>
</tr>
<tr>
 <td>
  <asp:EditorZone ID="EditorZone1" runat="server">
   <ZoneTemplate>
    <asp:AppearanceEditorPart id="AppearanceEditorPart1"
        runat="server" />
    <asp:LayoutEditorPart id="LayoutEditorPart1"
        runat="server" />
    </ZoneTemplate>
  </asp:EditorZone>
 </td>
</tr>
</table>
<p>
 <asp:Label ID="Label1" runat="server"></asp:Label>
</p>
<p>
 <asp:Button ID="Button1" runat="server" onclick="Button1_Click"
     Text="Switch Mode" />
 </p>
</div>
</form>
</body>
</html>
```

This page adds our simple user control to the second web part zone. In addition it includes this control, and a number of others, in a DeclarativeCatalogPart web control. The declarative web part lists web parts which the user may choose to appear on the page. The

declarative web part is included in a CatalogZone web part container, together with a PageCatalogPart whose use will be described below. Finally, the editor zone now includes a LayoutEditorPart web control as well as the AppearanceEditorPart.

The event handler for our 'Switch Mode' button now cycles through four (out of five) possible web part display modes, and the label is used to show to the user which mode is currently active.

The code in C# is as follows:

```
protected void Button1_Click(object sender, EventArgs e)
{
 if (WebPartManager1.DisplayMode == WebPartManager.BrowseDisplayMode)
 {
  WebPartManager1.DisplayMode = WebPartManager.EditDisplayMode;
  Label1.Text = "Web parts are currently in edit mode";
 }
 else if (WebPartManager1.DisplayMode ==
    WebPartManager.EditDisplayMode)
 {
  WebPartManager1.DisplayMode = WebPartManager.DesignDisplayMode;
  Label1.Text = "Web parts are currently in design mode";
 }
 else if (WebPartManager1.DisplayMode ==
    WebPartManager.DesignDisplayMode)
 {
  WebPartManager1.DisplayMode = WebPartManager.CatalogDisplayMode;
  Label1.Text = "Web parts are currently in catalog mode";
  }
  else if (WebPartManager1.DisplayMode ==
    WebPartManager.CatalogDisplayMode)
  {
  WebPartManager1.DisplayMode = WebPartManager.BrowseDisplayMode;
  Label1.Text = "Web parts are currently in browse mode";
 }
}
```

and similarly in Visual Basic:

```
Protected Sub Button1_Click(ByVal sender As Object, _
   ByVal e As EventArgs)
 If WebPartManager1.DisplayMode Is _
  WebPartManager.BrowseDisplayMode Then
  WebPartManager1.DisplayMode = WebPartManager.EditDisplayMode
  Label1.Text = "Web parts are currently in edit mode"
 ElseIf WebPartManager1.DisplayMode Is _
     WebPartManager.EditDisplayMode Then
  WebPartManager1.DisplayMode = _
  WebPartManager.DesignDisplayMode
  Label1.Text = "Web parts are currently in design mode"
  ElseIf WebPartManager1.DisplayMode Is _
```

```
        WebPartManager.DesignDisplayMode Then
      WebPartManager1.DisplayMode = _
        WebPartManager.CatalogDisplayMode
      Label1.Text = "Web parts are currently in catalog mode"
      ElseIf WebPartManager1.DisplayMode Is _
        WebPartManager.CatalogDisplayMode Then
       WebPartManager1.DisplayMode = _
         WebPartManager.BrowseDisplayMode
        Label1.Text = "Web parts are currently in browse mode"
    End If
  End Sub
```

The new modes here are DesignDisplayMode, which offers the user a subset of the facilities of edit mode whereby they can move web parts around but not edit them in other ways, and CatalogDisplayMode, which is illustrated in Figure 14.10. The Catalog Zone has both a Page Catalog and a Declarative Catalog. At present the Declarative Catalog is selected, and offers the user four web parts to choose from. By selecting the associated check boxes, and one of the available web part zones, the web parts can be added. The page catalog, which is not selected, allows the user to restore web parts they have previously deleted.

In CatalogDisplayMode, the user can select web parts from the available catalogs and add them to any of the web part zones on the page. In our case we have two catalogs, the

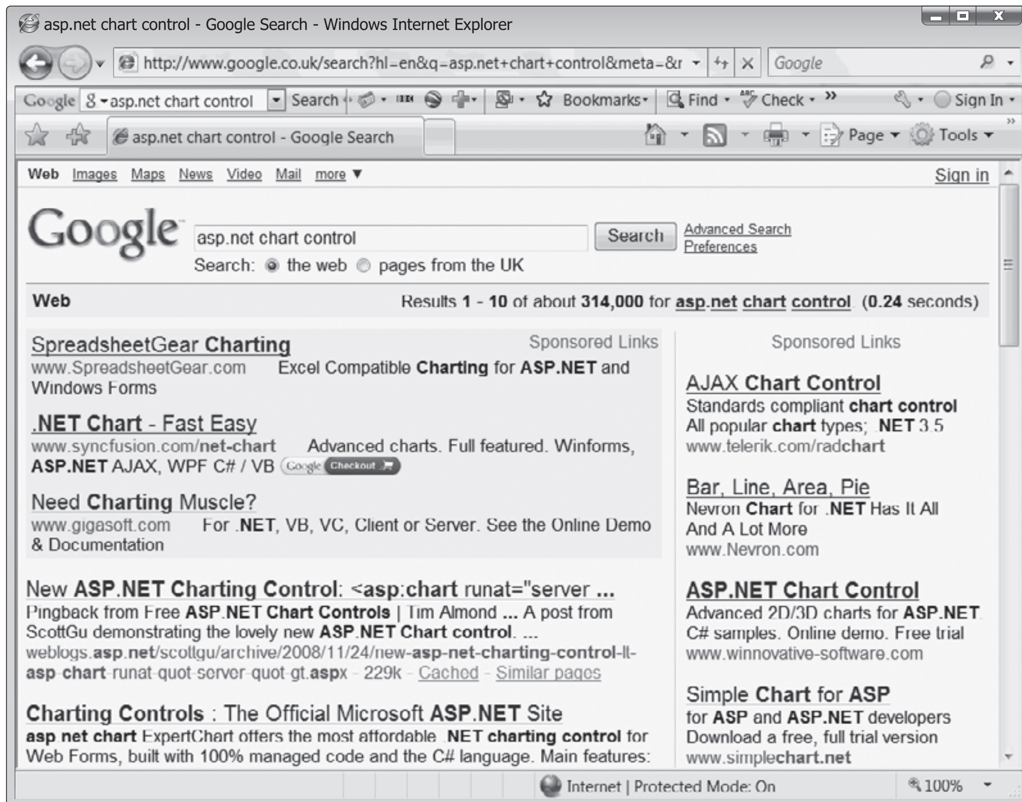
FIGURE 14.10 The web part Catalog display mode

declarative catalog which offers the user the choice of web parts listed on the web page, and the page catalog, which keeps track of web parts the user has closed, and offers them the chance to retrieve them again. Note that now we are using a page catalog, there is no longer any reason to prevent the user from closing web parts as described in the previous section.

14.6 Further Information about Web Parts

In the previous section, we showed how to create a simple web user part containing a heading and company logo. In practice we would want our web user controls to include dynamic database-driven content. For example, we could have a web user control which queried the claims database to count the number of outstanding claims, and report this, or else the number of policies with outstanding premiums. Often, companies develop informational 'dashboards' for their workers which are driven by such statistics, displaying them in a graphical or visual format. You have already seen the ASP.NET controls which allow you to display database content in tabular form. There are also numerous graphical controls from third parties, as you will see if you enter a phrase such as 'asp.net chart control' into your favorite search engine, as illustrated in Figure 14.11. Developing a dynamic database-driven web user control is therefore left as an exercise for you.

FIGURE 14.11 Some of the many ASP.NET chart controls which are available from third parties over the web



It has so far been implicitly understood that changes to web parts made using Edit, Design or Catalog mode will, by default, affect only the current user. Interestingly, however, there is an additional facility of the web parts framework whereby any changes are shared by all users. In effect, this allows customization of the web portal. This mode of operation must be enabled by executing the WebPartManager1.Personalization.ToggleScope() method. This method can only be executed successfully if the current user has been granted permission to enter shared scope, which requires the addition of lines such as the following in the site's web.config file:

```
<webParts>
 <personalization>
  <authorization>
   <allow roles="admin, site_designer" verbs="enterSharedScope" />
  </authorization>
 </personalization>
</webParts>
```

There are many other aspects and features of web parts, more than can be covered in the space available here. Indeed, whole books have been written on this subject, for example Neimke (2006). In particular, you can program your own custom web parts, rather than developing web user controls. Moreover, there is the ability to define connections between web parts so that they can share information. These and other features are discussed in a number of on-line help pages and tutorials which will allow you to exploit their full potential. In fact there is a wealth of help and on-line information which will allow you to exploit their full potential. You should note however that some time and coding effort will be required for this. You should also be aware that some of the available material covers the use of web parts in Microsoft's SharePoint system and these web parts extend yet further their use in ASP.NET.

Self Study Questions

1. What method is used to save an uploaded file to disk?
2. What CSS attribute is used to float an HTML element such as an ordered list?
3. What method is used to indicate the start of a database transaction?
4. What method is used to transmit binary data into the HTTP output stream?
5. What directive is used to indicate that an ASP.NET page should be cached?
6. What control is required before any other web part control may be used?
7. What directive is used to declare the web user controls to be used by a web form?
8. Name two features Visual Studio provides which VWD does not.
9. Name one feature which VWD provides which Visual Studio does not.
10. What utility must you run to set up the membership database for use by the full edition of SQL Server?

Exercises

14.1 As noted previously, the code for UploadImage.aspx should be wrapped in an ACID database transaction to guard against interference between two users uploading files simultaneously. Taking the code from UploadImage2.aspx as a model, make the first web form transactional.

14.2 Modify UploadImage.asp so that it stores a thumbnail as well as the original image. To implement this, you can use the Microsoft .Net System.Drawing.Bitmap and System.Drawing.Image classes.

14.3 Develop a web user control which displays the number of open (unapproved) claims. Add this web part to the skeleton web portal developed above.

14.4 If you have access to a full edition of SQL Server, port the insurance database and associated pages we developed using the Express Edition to the full edition.

SUMMARY

In this chapter, you have looked at and practiced using a number of advanced ASP.NET features including:

- Uploading binary image files both to the file system, and also directly to the database
- Displaying binary image files from the file system and from the database
- Using the ListView control
- Displaying database tables using lists rather than tables
- Programming database transactions
- Using Web Parts controls
- Creating and using a web user control
- Writing code to change the WebPartManager's DisplayMode

You also read about a number of other ASP.NET features such as:

- Page output and database caching
- Third party controls for charting
- Web part customization and connections

References and further reading

ASP.NET Web Parts Overview, Microsoft Developers Network, (http://msdn.microsoft.com/en-us/library/hhy9ewf1.aspx).
Connection Strings (http://www.connectionstrings.com).
Mitchell, Scott 2006 *Storing Binary Files Directly in the Database Using ASP.NET 2.0*, (http://aspnet.4guysfromrolla.com/articles/120606-1.aspx).
Neimke, D., 2006. *ASP.NET 2.0 Web Parts in Action.* Manning.