CHAPTER 12

Useful Techniques

The past eleven chapters have covered quite a lot of ground and provided a good foundation in creating data-driven Web sites. This chapter covers several useful techniques to improve your Web site's use of databases.

When updating or deleting from the database, there is always the possibility that another user has managed to change the data in the database before you've had the chance to make your own changes. This potentially leaves the database in a bit of a mess, as the first set of changes will be lost and overwritten with the second set of changes, resulting in a data *concurrency* problem. We'll look at one solution to this problem, which prevents changes being made to the database if changes to the same data have already been made.

Next, we'll explore caching. In previous examples, you've always needed to make a round trip to the database to retrieve query results from the database. For frequently changing data, this is the only way that you can show up-to-date data. But for data that doesn't change that often, you can use the caching functionality of ASP.NET to store that data on the Web server and remove the relatively expensive trips to the database.

Our next topic is transactions. When you need to execute multiple related queries against the database (such as inserting, updating, or deleting a Player), you want all of the queries to succeed or you want all of the queries to fail. By default, each query you execute exists in its own little world, and as soon as it's executed, the database is updated with the changes. If a related query fails, then the database is left in an inconsistent state—is the data correct or is it incorrect? Transactions allow you to combine queries together so that they either all succeed or all fail.

Finally, we'll look at using multiple result sets. In previous chapters, you've looked at query batches, but they have either not returned any results or returned only one set of results. ASP.NET allows you to combine multiple `SELECT` queries into one query batch and deal with the results through a single DataReader.

This chapter covers the following topics:

- How to ensure data concurrency
- How to cache data on the Web server to avoid round trips to the database
- How to work with transactions to ensure that a set of queries either completely succeeds or completely fails
- How to combine multiple `SELECT` queries in a query batch

Concurrency

When we looked at modifying the database in Chapters 8 and 9, you saw that you can encounter problems in data concurrency when the data you are modifying has already been modified by someone else. Suppose both Alice and Bob are updating the same Manufacturer, say Apple, at the same time to change the e-mail address. At the start of the update, both Alice and Bob see the same e-mail address of `lackey@apple.com`. If Alice changes the Manufacturer e-mail address to `someguy@apple.com` and Bob changes the Manufacturer e-mail address to `hello@apple.com`, what value does the database show? It all depends on who saved the changes last. If Alice updated the e-mail address before Bob did, then Bob's changes would be in the database and any changes made by Alice would be lost.

In cases where only one person manages the content of the database, this may be the correct behavior. But what happens if you have several people who manage the content of the database? Do you want to take the "last-in" approach and overwrite any changes that are made? Or do you want to prevent the second change from taking effect—a sort of "first-in" approach?

As you've seen, by using the primary key value of the table that you're updating, the details in the table will always be updated (the last-in approach), as the only part of the `WHERE` clause matching the row to be updated is the primary key.

Consider the `UPDATE` query that we've looked at for Manufacturers:

```
UPDATE Manufacturer SET
  ManufacturerName = @ManufacturerName,
  ManufacturerCounty = @ManufacturerCountry,
  ManufacturerEmail = @ManufacturerEmail,
  ManufacturerWebsite = @ManufacturerWebsite
WHERE ManufacturerID = @ManufacturerID
```

Every time you run this query, the Manufacturer details will be updated. To ensure that the query succeeds only whenever none of the details for the Manufacturer have changed during the course of the query, you need to modify the `WHERE` clause to perform the update only if all the details match what you think they should be, based on your knowledge before you ran the query:

```
UPDATE Manufacturer SET
  ManufacturerName = @ManufacturerName,
  ManufacturerCountry = @ManufacturerCountry,
  ManufacturerEmail = @ManufacturerEmail,
  ManufacturerWebsite = @ManufacturerWebsite
WHERE ManufacturerID = @ManufacturerID
  AND ManufacturerName = @originalManufacturerName
  AND ManufacturerCountry = @originalManufacturerCountry
  AND ManufacturerEmail = @originalManufacturerEmail
  AND ManufacturerWebsite = @originalManufacturerWebsite
```

With the modified query, the Manufacturer will be updated only if all of the details for the Manufacturer match what they were before you started updating. So, in the Alice and Bob example, the second update, made by Bob, won't be applied, as the ManufacturerEmail column will be `someguy@apple.com`, not the original `lackey@apple.com`.

So, the update won't be applied, and you need to inform the user that the update hasn't been made. If you're using a Command object to make the update, you can check the return from the `ExecuteNonQuery()` method. If it returns zero, there have been no updates to the database, and you can assume that the details have already been changed. If you're making the update in a `GridView`, you can check the `AffectedRows` property of the `GridViewUpdatedEventArgs` in the `RowUpdated` event (the same is also true for the `DetailsView` and `FormView`, except that the event is `ItemUpdated`).

We'll look at handling concurrency errors using a Command object to make the update, and then you'll see that you can also handle concurrency errors when using a `SqlDataSource` to populate a `DetailsView`.

■ **Note** Before you can follow the examples in this chapter, you need to download the code for this chapter from the Apress Web site (`http://www.apress.com`). In the code download, you'll find a folder called `original`, which contains several pages that you'll use as you work through the chapter. These are pages that we've already looked at in earlier chapters. Here, you'll add the functionality discussed in this chapter. You could rebuild these pages from scratch in this chapter, but I'm sure that you would rather concentrate on the new information, without needing to rebuild the same pages over and over again.

Try It Out: Handling Concurrency Using Command Objects

We'll first look at handling concurrency issues when using a Command object to connect to the database.

1. In Visual Web Developer, create a new Web site at `C:\BAND\Chapter12` and delete the auto-generated `Default.aspx` file.

2. Add a new `Web.config` file to the Web site and add a new setting to the `<connectionStrings />` element:

```
<add name="SqlConnectionString"
  connectionString="Data Source=localhost\BAND;Initial Catalog=Players;
    Persist Security Info=True;User ID=band;Password=letmein"
  providerName="System.Data.SqlClient" />
```

3. Copy `Manufacturers_DataSource.aspx` from the `original` folder in the code download to the root of the `Chapter12` Web site.

4. Add a new Web Form to the Web site called `Manufacturers_Edit_Command.aspx`. Make sure that the Place Code in Separate File check box is unchecked.

5. In the Source view, find the `<title>` tag and change the page title to **Edit Manufacturer Using Command**.

6. Add the correct `Import` statement to the top of the page:

```
<%@ Import Namespace="System.Data.SqlClient" %>
```

7. Switch to the Design view and add a `Label` to the page. Set its `ID` to `lblError`, `ForeColor` to Red, and `Visible` to `false`. Remove the value from the `Text` property.

8. Switch to the Source view and add the following markup after the `Label`:

```
<table>
  <tr>
    <td>Name:</td>
    <td><asp:TextBox ID="txtName" runat="server"></asp:TextBox></td>
  </tr>
  <tr>
    <td>Country:</td>
    <td><asp:TextBox ID="txtCountry" runat="server"></asp:TextBox></td>
  </tr>
  <tr>
    <td>Email:</td>
    <td><asp:TextBox ID="txtEmail" runat="server"></asp:TextBox></td>
  </tr>
  <tr>
    <td>Website:</td>
    <td><asp:TextBox ID="txtWebsite" runat="server"></asp:TextBox></td>
  </tr>
</table>
```

9. Switch back to the Design view and add two `Button` controls after the table. The first `Button` should have its `ID` set to `btnUpdate` and it `Text` property set to `Update`. The second should have its `ID` set to `btnCancel` and its `Text` property set to `Cancel`. Your page should look similar to Figure 12-1.

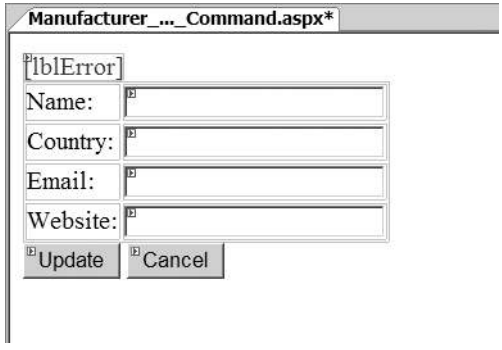

Figure 12-1. *Page design to add a new Manufacturer*

10. Add a `Load` event to the page and add the following code to the `Page_Load` event handler:

```
protected void Page_Load(object sender, EventArgs e)
{
  lblError.Visible = false;
```

```
  if (Page.IsPostBack == false)
  {
    LoadManufacturer();
  }
}
```

11. Add the `LoadManufacturer()` method:

```
private void LoadManufacturer()
{
  // create the connection
  string strConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  SqlConnection myConnection = new SqlConnection(strConnectionString);

  try
  {
    // create the SELECT command
    string strQuery = "SELECT ManufacturerName, ManufacturerCountry, ➥
      ManufacturerEmail, ManufacturerWebsite ➥
      FROM Manufacturer WHERE ManufacturerID = @ManufacturerID";
    SqlCommand myCommand = new SqlCommand(strQuery, myConnection);
    myCommand.Parameters.AddWithValue("@ManufacturerID",
      Request.QueryString["ManufacturerID"]);

    // open the connection
    myConnection.Open();

    // execute the query
    SqlDataReader myReader = myCommand.ExecuteReader();

    // if we have results then we need to parse them
    if (myReader.Read() == true)
    {
      txtName.Text = myReader.GetString(
        myReader.GetOrdinal("ManufacturerName"));
      txtCountry.Text = myReader.GetString(
        myReader.GetOrdinal("ManufacturerCountry"));
      txtEmail.Text = myReader.GetString(
        myReader.GetOrdinal("ManufacturerEmail"));
      txtWebsite.Text = myReader.GetString(
        myReader.GetOrdinal("ManufacturerWebsite"));

      // save values into viewstate
      ViewState["ManufacturerName"] = txtName.Text;
      ViewState["ManufacturerCountry"] = txtCountry.Text;
      ViewState["ManufacturerEmail"] = txtEmail.Text;
      ViewState["ManufacturerWebsite"] = txtWebsite.Text;
    }
```

```
      // close the reader
      myReader.Close();
    }
    catch (Exception ex)
    {
      lblError.Text = ex.Message;
      lblError.Visible = true;
    }
    finally
    {
      // always close the connection
      myConnection.Close();
    }
  }
```

12. Add a `Click` event to the Update button and add the following code to the `btnUpdate_Click` event handler:

```
protected void btnUpdate_Click(object sender, EventArgs e)
{
  if (Page.IsValid == true)
  {
    SaveManufacturer();
  }
}
```

13. Add the `SaveManufacturer()` method:

```
private void SaveManufacturer()
{
  // create the connection
  string strConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  SqlConnection myConnection = new SqlConnection(strConnectionString);

  try
  {
    // create the UPDATE command
    string strQuery = "UPDATE Manufacturer SET ➥
      ManufacturerName = @ManufacturerName, ➥
      ManufacturerCountry = @ManufacturerCountry, ➥
      ManufacturerEmail = @ManufacturerEmail, ➥
      ManufacturerWebsite = @ManufacturerWebsite ➥
      WHERE ManufacturerID = @ManufacturerID ➥
      AND ManufacturerName = @originalManufacturerName ➥
      AND ManufacturerCountry = @originalManufacturerCountry ➥
      AND ManufacturerEmail = @originalManufacturerEmail ➥
      AND ManufacturerWebsite = @originalManufacturerWebsite";
    SqlCommand myCommand = new SqlCommand(strQuery, myConnection);
```

```
// add the parameters
myCommand.Parameters.AddWithValue("@ManufacturerID",
  Request.QueryString["ManufacturerID"]);
myCommand.Parameters.AddWithValue("@ManufacturerName",
  txtName.Text);
myCommand.Parameters.AddWithValue("@ManufacturerCountry",
  txtCountry.Text);
myCommand.Parameters.AddWithValue("@ManufacturerEmail",
  txtEmail.Text);
myCommand.Parameters.AddWithValue("@ManufacturerWebsite",
  txtWebsite.Text);
myCommand.Parameters.AddWithValue("@originalManufacturerName",
  ViewState["ManufacturerName"]);
myCommand.Parameters.AddWithValue("@originalManufacturerCountry",
  ViewState["ManufacturerCountry"]);
myCommand.Parameters.AddWithValue("@originalManufacturerEmail",
  ViewState["ManufacturerEmail"]);
myCommand.Parameters.AddWithValue("@originalManufacturerWebsite",
  ViewState["ManufacturerWebsite"]);

// open the connection
myConnection.Open();

// execute the query
int intCount = myCommand.ExecuteNonQuery();

// no records affected is error
if (intCount == 0)
{
  lblError.Text = "No update was made.  Concurrency problem.";
  lblError.Visible = true;
}
else
{
  // disable controls
  txtName.Enabled = false;
  txtCountry.Enabled = false;
  txtEmail.Enabled = false;
  txtWebsite.Enabled = false;
  btnUpdate.Enabled = false;
```

```
        // change the cancel to continue
        btnCancel.Text = "Continue";
      }
    }
    catch (Exception ex)
    {
      lblError.Text = ex.Message;
      lblError.Visible = true;
    }
    finally
    {
      // always close the connection
      myConnection.Close();
    }
  }
```

14. Add a `Click` event to the Cancel button and add the following code to the `btnCancel_Click` event handler:

```
protected void btnCancel_Click(object sender, EventArgs e)
{
  if (Request.QueryString["Type"] == "DS")
  {
    Response.Redirect("./Manufacturers_DataSource.aspx");
  }
  else if (Request.QueryString["Type"] == "DR")
  {
    Response.Redirect("./Manufacturers_DataReader.aspx");
  }
}
```

15. Save the page, and then open `Manufacturers.aspx` in your browser. Click the Edit Command button for a Manufacturer and make some changes to the Manufacturer. You'll see that the update works as you would expect.

16. Open another instance of Internet Explorer by selecting File ➤ New ➤ Window from the Internet Explorer menu. You'll now have two instances of Internet Explorer viewing the list of Manufacturers.

17. Click to edit the same Manufacturer in both instances of Internet Explorer. In one of the instances, you'll be able to make changes, and these will be saved, as shown in Figure 12-2.

18. Try to modify the Manufacturer in the other instance of Internet Explorer. This time, the change will be rejected and a warning will be shown, as you can see in Figure 12-3.

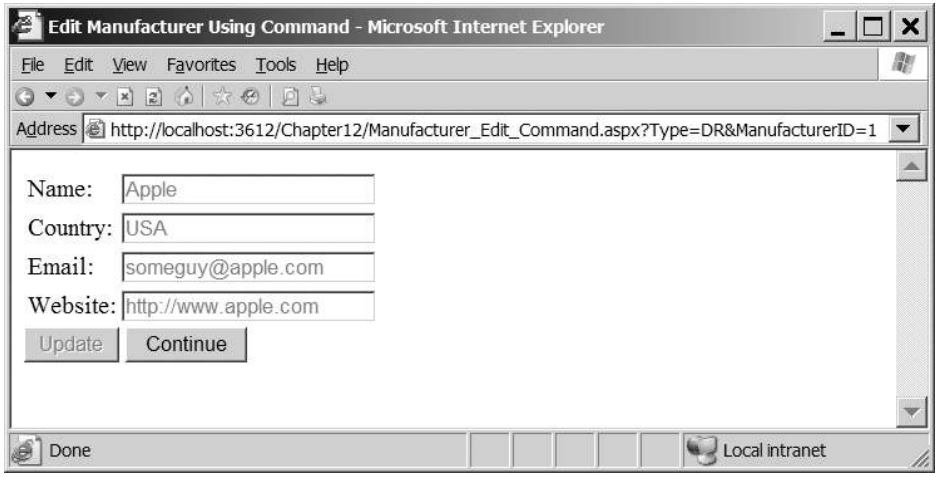

Figure 12-2. *You can make changes in one instance of Internet Explorer.*

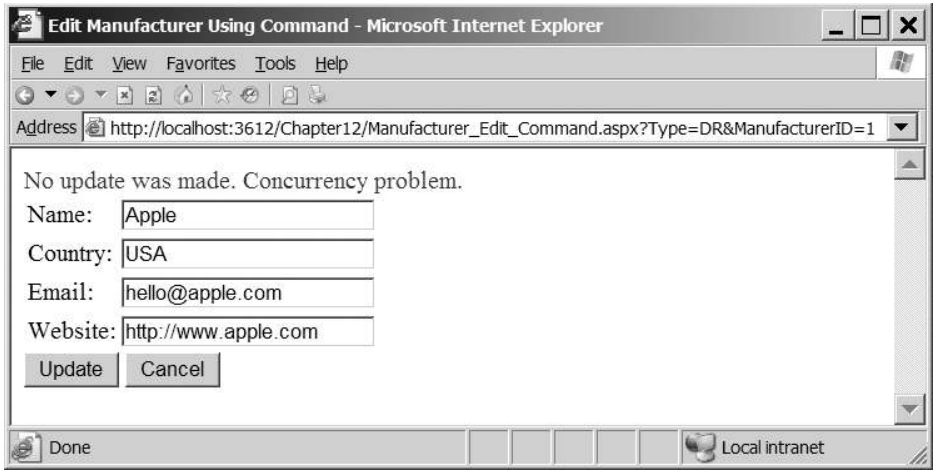

Figure 12-3. *You cannot make changes if they cause a data concurrency problem.*

How It Works

In Chapter 8, you saw how to send an `UPDATE` query to the database, and that's effectively all you're doing here. Granted, it's a more complex `UPDATE` query, but it's still just an `UPDATE` query:

```
UPDATE Manufacturer SET
  ManufacturerName = @ManufacturerName,
  ManufacturerCountry = @ManufacturerCountry,
  ManufacturerEmail = @ManufacturerEmail,
  ManufacturerWebsite = @ManufacturerWebsite
WHERE ManufacturerID = @ManufacturerID
  AND ManufacturerName = @originalManufacturerName
```

```
  AND ManufacturerCountry = @originalManufacturerCountry
  AND ManufacturerEmail = @originalManufacturerEmail
  AND ManufacturerWebsite = @originalManufacturerWebsite
```

The key to the handling of concurrency is remembering the original values so that you can use them to populate the `WHERE` clause correctly. You can store them quite easily in `ViewState`, and they'll always be available to the page. So, in the `LoadManufacturer()` method, you retrieve the values that you need from the database and set the four `TextBox` controls' `Text` properties. Rather than accessing the DataReader twice, you then use the `TextBox` control `Text` properties to save the correct values into `ViewState`:

```
// save values into viewstate
ViewState["ManufacturerName"] = txtName.Text;
ViewState["ManufacturerCountry"] = txtCountry.Text;
ViewState["ManufacturerEmail"] = txtEmail.Text;
ViewState["ManufacturerWebsite"] = txtWebsite.Text;
```

You can then use the values stored in `ViewState` when you add the parameters to the Command object:

```
myCommand.Parameters.AddWithValue("@originalManufacturerName",
  ViewState["ManufacturerName"]);
myCommand.Parameters.AddWithValue("@originalManufacturerCountry",
  ViewState["ManufacturerCountry"]);
myCommand.Parameters.AddWithValue("@originalManufacturerEmail",
  ViewState["ManufacturerEmail"]);
myCommand.Parameters.AddWithValue("@originalManufacturerWebsite",
  ViewState["ManufacturerWebsite"]);
```

Once you've added all the parameters correctly, you can then execute the `UPDATE` query. If the query fails, the `ExecuteNonQuery()` method will return zero, indicating that no rows were affected by the `UPDATE`. You tell the user about the problem, as shown in Figure 12-3.

The Cancel button has its text changed so you can use it for both the cancel and continue purposes. Rather than having two buttons with two `Click` event handlers, you have only one.

Within the event handler, you have slightly more code than you might expect:

```
if (Request.QueryString["Type"] == "DS")
{
  Response.Redirect("./Manufacturers_DataSource.aspx");
}
else if (Request.QueryString["Type"] == "DR")
{
  Response.Redirect("./Manufacturers_DataReader.aspx");
}
```

Don't worry about it! If you look at the URL for the page, you'll see that it has a type as part of the query string. You're actually going to use these edit pages a little later, and you need the Cancel/Continue button to be able to return to the correct page. You've added the `Type` parameter so that you can control which of the Manufacturers pages you're returned to.

■ **Note** In the examples here, you'll work with only the `UPDATE` query, and not the `DELETE` query. The process for handling concurrency problems with the `DELETE` query is the same as with the `UPDATE` query.

Try It Out: Handling Concurrency Using the SqlDataSource

The `SqlDataSource` also handles data concurrency and will generate the correct `UPDATE` query to force concurrency.

1. Add a new Web Form to the Web site called `Manufacturer_Edit_DataSource.aspx`. Make sure that the Place Code in Separate File check box is unchecked.
2. In the Source view, find the `<title>` tag and change the page title to **Edit Manufacturer Using DataSource**.
3. Switch to the Design view and add a `SqlDataSource` to the page (which will be called `SqlDataSource1`). From the Tasks menu, select Configure Data Source.
4. Select `SqlConnectionString` as the data connection to use and click the Next button.
5. Create a query that selects the ManufacturerID, ManufacturerName, ManufacturerCountry, ManufacturerEmail, and ManufacturerWebsite columns from the Manufacturer table.
6. Click the WHERE button and add a `WHERE` clause for ManufacturerID that uses the ManufacturerID `QueryString` value, as shown in Figure 12-4. Click OK to close the Add WHERE Clause dialog box.
7. Click the Advanced button. In the Advanced SQL Generation Options dialog box, click both the Generate INSERT, UPDATE, and DELETE Statements check box and the Use Optimistic Concurrency check box, as shown in Figure 12-5. Click OK to close the Advanced SQL Generation Options dialog box.

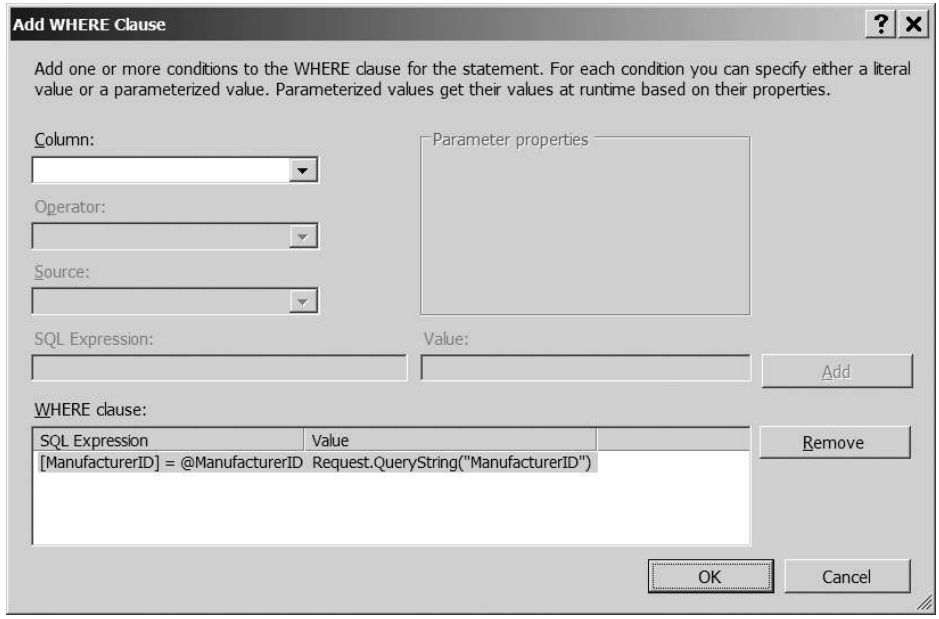

Figure 12-4. *Using the QueryString value to constrain the SqlDataSource*

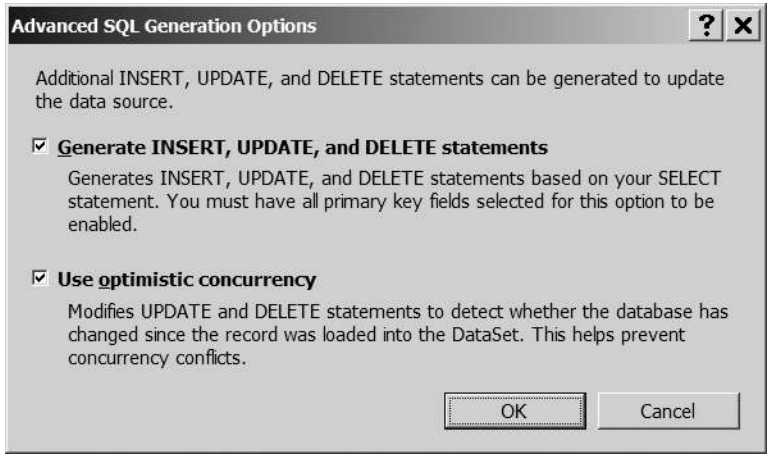

Figure 12-5. *Auto-generating concurrency-proof UPDATE and DELETE queries*

8. Click Next, and then click Finish to close the Configure Data Source wizard.

9. Add a `Label` to the page. Set its `ID` to `lblError`, `ForeColor` to Red, and `Visible` to `false`. Remove the value from the `Text` property.

10. Add a `DetailsView` to the page and use `SqlDataSource1` as the DataSource from the Tasks menu.

11. Switch to the Source view and remove the first `BoundField` from the `Fields` collection of the `DetailsView`.

12. Add the following `FooterTemplate` to the `DetailsView`:

```
<FooterTemplate>
  <asp:Button ID="btnUpdate" CommandName="Update"
    runat="server" Text="Update" />
  <asp:Button ID="btnCancel" CommandName="Cancel"
    runat="server" Text="Cancel" />
</FooterTemplate>
```

13. Add a `Load` event to the page and add the following code to the `Page_Load` event handler:

```
protected void Page_Load(object sender, EventArgs e)
{
  lblError.Visible = false;

  if (Page.IsPostBack == false)
  {
    DetailsView1.ChangeMode(DetailsViewMode.Edit);
  }
}
```

14. Add the `ItemUpdated` event to the `DetailsView` and add the following code to the `DetailsView1_ItemUpdated` event handler:

```
protected void DetailsView1_ItemUpdated(object sender,
  DetailsViewUpdatedEventArgs e)
{
  if (e.Exception != null)
  {
    lblError.Text = e.Exception.Message;
    lblError.Visible = true;
    e.ExceptionHandled = true;
    e.KeepInEditMode = true;
  }
  else if (e.AffectedRows == 0)
  {
    lblError.Text = "No update was made.  Concurrency problem.";
    lblError.Visible = true;
    e.KeepInEditMode = true;
  }
}
```

15. Add the `ItemCommand` event to the `DetailsView` and add the following code to the `DetailsView1_ItemCommand` event handler:

```
protected void DetailsView1_ItemCommand(object sender,
  DetailsViewCommandEventArgs e)
{
  if (e.CommandName == "Cancel")
  {
    if (Request.QueryString["Type"] == "DS")
    {
      Response.Redirect("./Manufacturers_DataSource.aspx");
    }
    else if (Request.QueryString["Type"] == "DR")
    {
      Response.Redirect("./Manufacturers_DataReader.aspx");
    }
  }
}
```

16. Add the `DataBound` event to the `DetailsView` and add the following code to the `DetailsView_DataBound` event handler:

```
protected void DetailsView1_DataBound(object sender, EventArgs e)
{
  // set the buttons correctly
  if (DetailsView1.CurrentMode == DetailsViewMode.ReadOnly)
  {
    ((Button)DetailsView1.FooterRow.FindControl("btnUpdate")).
      Enabled = false;
    ((Button)DetailsView1.FindControl("btnCancel")).Text = "Continue";
  }
}
```

17. Save the page, and then open `Manufacturers_DataSource.aspx` in your browser. Click the Edit DataSource button for a Manufacturer and make some changes to the Manufacturer. You'll see that the update works as you would expect.

18. Open another instance of Internet Explorer by selecting File ➤ New ➤ Window from the Internet Explorer menu. You'll now have two instances of Internet Explorer viewing the list of Manufacturers.

19. Click to edit the same Manufacturer in both instances of Internet Explorer. In one of the instances, you'll be able to make changes, and these will be saved, as shown in Figure 12-6.

20. Try to modify the Manufacturer in the other instance of Internet Explorer. This time, the change will be rejected and a warning will be displayed, as shown in Figure 12-7.

Figure 12-6. *Using a DataSource, changes can be made as you would expect.*

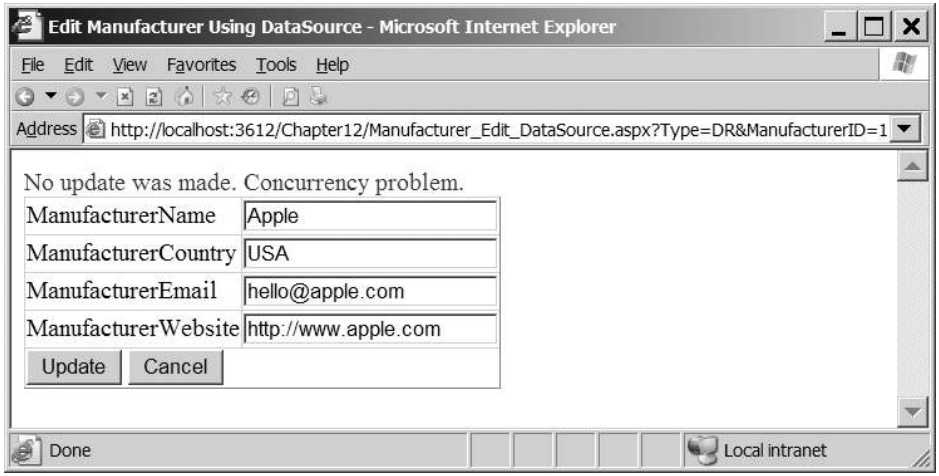

Figure 12-7. *You cannot make changes if they cause a data concurrency problem.*

How It Works

As you've just seen, enabling concurrency protection for the `SqlDataSource` is as simple as checking a box in the Configure Data Source wizard. It may have looked as though there was more than that, but a lot of the code that you added was to control the `DetailsView` and make it work in a "solo" way when it's not being used in a master-detail situation with a `GridView`.

When configuring the `SqlDataSource`, you have the option of enabling the control for concurrency. You've already seen how the `SqlDataSource` will generate `INSERT`, `UPDATE`, and `DELETE` queries if you tell it to do so. By checking the Use Optimistic Concurrency check box in the Advanced SQL Generation Options dialog box, you've also told the `SqlDataSource` that you want to use concurrency. You'll see that the `UpdateCommand` for the `SqlDataSource` is no longer a simple `UPDATE` query, but is now the query to support concurrency that you've already seen:

```
UPDATE Manufacturer SET
  ManufacturerName = @ManufacturerName,
  ManufacturerCountry = @ManufacturerCountry,
  ManufacturerEmail = @ManufacturerEmail,
  ManufacturerWebsite = @ManufacturerWebsite
WHERE ManufacturerID = @original_ManufacturerID
  AND ManufacturerName = @original_ManufacturerName
  AND ManufacturerCountry = @original_ManufacturerCountry
  AND ManufacturerEmail = @original_ManufacturerEmail
  AND ManufacturerWebsite = @original_ManufacturerWebsite
```

And the `UpdateParameters` collection also has the extra parameters specified:

```
<UpdateParameters>
  <asp:Parameter Name="ManufacturerName" Type="String" />
  <asp:Parameter Name="ManufacturerCountry" Type="String" />
  <asp:Parameter Name="ManufacturerEmail" Type="String" />
  <asp:Parameter Name="ManufacturerWebsite" Type="String" />
  <asp:Parameter Name="original_ManufacturerID" Type="Int32" />
  <asp:Parameter Name="original_ManufacturerName" Type="String" />
  <asp:Parameter Name="original_ManufacturerCountry" Type="String" />
  <asp:Parameter Name="original_ManufacturerEmail" Type="String" />
  <asp:Parameter Name="original_ManufacturerWebsite" Type="String" />
</UpdateParameters>
```

The `SqlDataSource` and `DetailsView` manage the concurrency checks and the necessary parameter values automatically. All that you need to deal with is the control of the user interface.

Because you're using the `DetailsView` on its own page, you want it only in `Edit` mode. You've chosen to edit the Manufacturer on the previous page so, in `Page_Load`, you always switch the `DetailsView` into `Edit` mode:

```
if (Page.IsPostBack == false)
{
  DetailsView1.ChangeMode(DetailsViewMode.Edit);
}
```

You're also don't want to rely on the automatic buttons that the `DetailsView` offers. These don't quite fit the purposes here, so you add your own buttons into the `FooterTemplate`:

```
<FooterTemplate>
  <asp:Button ID="btnUpdate" CommandName="Update"
    runat="server" Text="Update" />
  <asp:Button ID="btnCancel" CommandName="Cancel"
    runat="server" Text="Cancel" />
</FooterTemplate>
```

As you'll recall from Chapter 7, by using a `CommandName` of `Update`, you can force the `DetailsView` to perform an update. By using `Cancel`, you can force the `DetailsView` to cancel the current operation.

When canceling, you want to return to the list of Manufacturers, so in the `ItemCommand` event handler, you check that it was the Cancel button that was clicked. If it was, you return to the correct list of Manufacturers:

```
if (e.CommandName == "Cancel")
{
  if (Request.QueryString["Type"] == "DS")
  {
    Response.Redirect("./Manufacturers_DataSource.aspx");
  }
  else if (Request.QueryString["Type"] == "DR")
  {
    Response.Redirect("./Manufacturers_DataReader.aspx");
  }
}
```

If the user clicks the Update button, the `DetailsView` will perform the update and then raise the `ItemUpdated` event. As you'll recall from Chapter 9, this event allows you to check if there were any exceptions raised during the update and to also check how many rows were updated by the query. You need to handle both of these conditions and gracefully let the user know what is happening.

If you have an exception during the update, the `Exception` property of the `DetailsViewUpdatedEventArgs` will be set to the exception that was raised. You'll show the details of the problem to the user and let the `DetailsView` know that you've handled the exception. You're also setting the `KeepInEditMode` property to `true`, since the update hasn't been made and you don't want to leave `Edit` mode:

```
if (e.Exception != null)
{
  lblError.Text = e.Exception.Message;
  lblError.Visible = true;
  e.ExceptionHandled = true;
  e.KeepInEditMode = true;
}
```

If no exception was raised, there may be concurrency issues to handle. The `AffectedRows` property will return zero if no rows were updated. If this happens, you let the user know that a concurrency problem has occurred:

```
else if (e.AffectedRows == 0)
{
  lblError.Text = "No update was made.  Concurrency problem.";
  lblError.Visible = true;
  e.KeepInEditMode = true;
}
```

The final piece of code is again concerned solely with the user interface and how you use a `DetailsView` on its own. If the `DetailsView` is in `ReadOnly` mode, it means that the update has been successful. You don't want to allow the user to click the Update button again, so you

disable it. The user can no longer cancel the update, so you change the text of the Cancel button to Continue. Both of these occur in the DataBound event:

```
protected void DetailsView1_DataBound(object sender, EventArgs e)
{
  // set the buttons correctly
  if (DetailsView1.CurrentMode == DetailsViewMode.ReadOnly)
  {
    ((Button)DetailsView1.FooterRow.FindControl("btnUpdate")).
      Enabled = false;
    ((Button)DetailsView1.FindControl("btnCancel")).Text = "Continue";
  }
}
```

As you can see, the SqlDataSource handles all of the logic for concurrency issues automatically. All you have to do to enable concurrency protection is check the appropriate check box in the wizard. You had to add a little code within the ItemUpdated event handler to deal with any exceptions or concurrency problems that arose, but the tasks of generating the UPDATE query, handling the original values, and passing these values to the query are handled automatically.

Caching

As you know by now, actually accessing the database is an expensive operation. Making the connection and then executing a query takes time. If there are instances where you don't need to query the database, consider caching results on the Web server so you can avoid that round trip to the database.

As you're aware, ASP.NET provides a cache for you to store information that will be used several times. The ideal types of objects to store on the cache are those that take a long time to create, so you don't need to re-create them.

So what results should be stored in the cache? Although you can cache anything that you want, storing regularly changing data there is a little pointless, since it will soon be invalidated. Data that changes infrequently is suitable for caching. In the sample database, new Manufacturers won't be added that often, so the list of Manufacturers is an ideal candidate for caching. However, you need to ensure that the data that is cached doesn't become invalid. If a new Manufacturer is added to the database, then the cached version of the list of Manufacturers will be incorrect.

Adding objects to the cache is very simple. The Page object provides direct access to the cache using the Cache property. You can add data to the cache using either the Add() or Insert() method. For the Insert() method, as a minimum, you need to specify a key for the object and the object itself:

```
dsManufacturers.RemotingFormat = SerializationFormat.Binary;
Cache.Insert("Manufacturers", dsManufacturers);
```

This adds a DataSet, dsManufacturers, to the cache indexed on the Manufacturers key. It sets the RemotingFormat of the DataSet to Binary, as this is the most efficient format in which you can store the DataSet. (You're storing this in the memory of the Web server, so you need to

use as little space as possible.) It uses the `Insert()` method, which will automatically overwrite an existing item with the same key; the `Add()` method would throw an error.

You can access objects in the cache by using the key as the index to the `Cache` object itself. The cached object is returned as an `Object`, so it will need to be cast to the correct type:

```
DataSet dsManufacturers = (DataSet)Cache["Manufacturers"];
```

But that's not the end of the story. After you've added things to the cache, do they stay there forever? Certainly not! There are several instances where a cached object may be removed, including the following:

- ASP.NET removes the object automatically because there are resourcing issues with the Web site. As the memory of the Web server is used, ASP.NET will remove objects from the cache as required to ensure that the Web server operates correctly.
- You can specify that the objects can be cached for a fixed period of time or that the object is removed if it isn't used for a specified period of time.
- You can manually remove objects from the cache using the `Remove()` method.

So, if the item isn't in the cache, how do you detect that it isn't there? You can't perform a check on the cache directly to see if the object is there (it has no `Contains()` method as you would expect from a collection). Instead, you need to check that you actually have an object returned. For example, to test if the list of Manufacturers is in the cache, perform the following check:

```
DataSet dsManufacturers = (DataSet)Cache["Manufacturers"];
if (dsManufacturers == null)
{
  // rebuild DataSet
  // add DataSet to cache
}
```

If the `DataSet` isn't in the cache, the object returned will be equal to null. In that case, you would need to rebuild the `DataSet` and then make sure that it's added to the cache to be reused next time.

We'll now look at the two options for specifying the period of time that an object will remain cached. Then you'll work through an example where you cache the results from a DataReader and see that the queries to the database are reduced. Caching a `DataSet` is broadly similar, so we'll just look at the differences. Finally, you'll see how to use the `Remove()` method to ensure that when the database is modified, there are no cached objects that are invalid.

■ **Note** Caching is a massive topic. MSDN provides information and examples covering all the different types of caching at `http://msdn.microsoft.com/en-us/library/726btaeh.aspx`. ASP.NET provides a series of Quickstart tutorials at `http://www.asp.net/QuickStart/aspnet/doc/caching/default.aspx`.

Specifying the Life Span of a Cached Object

When caching objects, you can allow the object to exist on the server until ASP.NET decides that it needs to be removed, but this isn't ideal. Although you're caching only data that changes infrequently, there aren't any guarantees that the cached data will be reused, and storing it in memory on the Web server when it isn't used isn't necessary.

You therefore need some way of specifying that the object that you're caching is valid for only a specified period of time. ASP.NET supports two types of time-based expiration:

- With *absolute expiration*, you specify a specific date and time that the data in the cache will become invalid.
- *Sliding expiration* allows you to specify a period of time that the object will remain in the cache since it was last accessed.

For example, to add a list of Manufacturers, stored in a `colManufacturers` object, to the cache, use the following:

```
Cache.Insert("Manufacturers", colManufacturers, null,
  DateTime.Now.AddMinutes(5), Cache.NoSlidingExpiration);
```

The fourth parameter to this overload of the `Insert()` method allows you to specify the absolute time that the cached object expires. This example takes the current time and adds 5 minutes to it. Once those 5 minutes are up (from the moment you called the `Insert()` method), the object is removed from the cache. Because you're using absolute expiration, you set the final parameter to `Cache.NoSlidingExpiration`.

To use sliding expiration, you use a similar call to the `Insert()` method:

```
Cache.Insert("Manufacturers", colManufacturers, null,
  Cache.NoAbsoluteExpiration, TimeSpan.FromMinutes(5));
```

This version sets the absolute expiration to `Cache.NoAbsoluteExpiration` and specifies a sliding expiration. In this case, `TimeSpan` is set as 5 minutes in the future. The list of Manufacturers will be cached for 5 minutes after it was last used and then discarded. If the list was accessed before it was invalidated, even if that access takes place after 4 minutes and 59 seconds, the object would remain in the cache for a further 5 minutes. And if it was accessed again? It has another 5 minutes before it is invalidated. The expiration slides!

■ **Note** The third parameter to the `Insert()` method (passed as `null` in the examples here) allows you to add cache dependencies to the added object, so that it can be invalidated automatically if the dependent object is invalidated as well. For some good examples of cache dependencies, see the MSDN documentation referenced earlier, at `http://msdn.microsoft.com/en-us/library/726btaeh.aspx`.

Try It Out: Caching the Manufacturers

In this first example of caching, you'll build a new page that retrieves the list of Manufacturers using a DataReader and show the results in a `GridView`. You'll start with a basic page from the code download and modify this to add caching to reduce the number of queries made against the database.

1. Copy `Manufacturers_DataReader.aspx` from the `original` folder in the code download to the root of the `Chapter12` Web site.

2. Open `Manufacturers_DataReader.aspx` and modify the `Page_Load` event as follows:

```
protected void Page_Load(object sender, EventArgs e)
{
  if (Page.IsPostBack == false)
  {
    // retrieve the Manufacturers from cache
    ArrayList colManufacturers = (ArayList)Cache["Manufacturers"];

    // only load if not cached
    if (colManufacturers == null)
    {
      // create the connection
      string strConnectionString = ConfigurationManager.
        ConnectionStrings["SqlConnectionString"].ConnectionString;
      SqlConnection myConnection = new SqlConnection(strConnectionString);

      try
      {
        // query to execute
        string strQuery = "SELECT ManufacturerID, ManufacturerName, ➥
          ManufacturerCountry, ManufacturerEmail, ManufacturerWebsite ➥
          FROM Manufacturer ORDER BY ManufacturerName";

        // create the command
        SqlCommand myCommand = new SqlCommand(strQuery, myConnection);

        // open the database connection
        myConnection.Open();

        // run query
        SqlDataReader myReader = myCommand.ExecuteReader();
```

```
            // create a new collection
            colManufacturers = new ArrayList();
            foreach (System.Data.Common.DbDataRecord objRecord in myReader)
            {
              colManufacturers.Add(objRecord);
            }

            // close the reader
            myReader.Close();

            // cache the collection
            Cache.Insert("Manufacturers", colManufacturers, null,
              Cache.NoAbsoluteExpiration, TimeSpan.FromMinutes(5));
          }
          finally
          {
            // always close the database connection
            myConnection.Close();
          }
        }

        // set the data source and bind
        GridView1.DataSource = colManufacturers;
        GridView1.DataBind();
      }
    }
```

3. Save the page and set `Manufacturers_DataReader.aspx` as the start page for the Web site.

4. Add a breakpoint to the first line of the `Page_Load` event handler and start debugging the application. When the page first loads, the collection of Manufacturers will not be present in the cache, as shown in Figure 12-8, so the database will be queried. If you continue to step through the code by pressing F10, you'll see this happening.

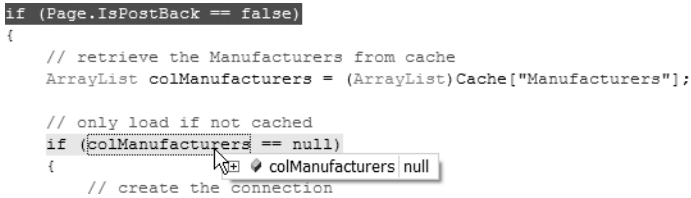

Figure 12-8. *On first load, the list of Manufacturers is not in the cache.*

5. Once the page has loaded and the list of Manufacturers is displayed in the `GridView`, press F5 to reload the page.

6. This time, when the breakpoint is hit, step through the code. You'll see that the list of Manufacturers has been retrieved from the cache, as shown in Figure 12-9. If you continue to step through the code, the database will not be queried.

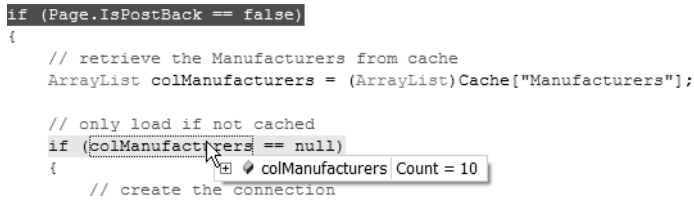

Figure 12-9. *Subsequent loads retrieve the list of Manufacturers from the cache.*

7. Wait 5 minutes (go and make a cup of tea!), and then refresh the page. Step through the code, and you'll see that the list of Manufacturers has been removed from the cache and the database will be requeried.

How It Works

This example demonstrated that caching data is quite simple, as is dealing with the cache, as you'll see shortly. First, though, let's look at what you're actually storing on the cache.

As you know, the DataReader object is connected to the database while it is being used, and it provides forward-only, read-only access to the data. Because it is connected at all times, there is no way you can cache it without massaging it beforehand. You can't directly cache the DataReader; instead, you cache the results that it has returned.

The DataReader object returns the results from the query as a series of `DbDataRecord` objects. A `DbDataRecord`, once created by the DataReader, is not connected to the database and you can store it in the cache and reuse it as required.

As you're going to use the collection for data binding to the `GridView`, you use an `ArrayList`. The `ArrayList` supports the `ICollection` interface, so you can directly use it for data binding:

```
GridView1.DataSource = colManufacturers;
GridView1.DataBind();
```

To populate the `ArrayList`, you can iterate through the DataReader and simply add each `DbDataRecord` to the collection:

```
colManufacturers = new ArrayList();
foreach (System.Data.Common.DbDataRecord objRecord in myReader)
{
  colManufacturers.Add(objRecord);
}
```

After you've constructed the `ArrayList` that contains your massaged results, you can cache it and reuse it whenever you need it.

Adding the `ArrayList` to the `Cache` is accomplished in one line of code:

```
Cache.Insert("Manufacturers", colManufacturers, null,
  Cache.NoAbsoluteExpiration, TimeSpan.FromMinutes(5));
```

As you've already seen, you can cache data until a certain date and time using absolute expiration, or you can use sliding expiration to cache the data for a set period of time. In this case, you cache the list of Manufacturers for 5 minutes since it was last used.

You can check if the list of Manufacturers is present in the cache by using the key for the object you're retrieving, and cast the object to the correct type:

```
ArrayList colManufacturers = (ArrayList)Cache["Manufacturers"];
```

If there is a matching object in the cache, it will be returned and cast to the correct type. If there is no matching object—either because this is the first load of the page and you've never cached the `ArrayList` or it has been removed (for whatever reason)—the `Cache` object will return null. If you don't have a cached `ArrayList`, you query the database and construct the collection and cache, so that it's available in the future.

Caching a DataSet

Caching a `DataSet` is even easier than caching a DataReader. Because the `DataSet` is disconnected from the database, you can add it directly to the cache without any of the extra coding that was required to massage the DataReader into a cacheable collection.

So, in the previous example, the code to retrieve the list of Manufacturers would now look like this:

```
// retrieve the Manufacturers from cache
DataSet dsManufacturers = (DataSet)Cache["Manufacturers"];

// only load if not cached
if (dsManufacturers == null)
{
  // fill the DataSet as you'd normally do

  // set to Binary serialization
  dsManufacturers.RemotingFormat = SerializationFormat.Binary;

  // cache the DataSet
  Cache.Insert("Manufacturers", dsManufacturers, null,
    Cache.NoAbsoluteExpiration, TimeSpan.FromMinutes(5));
}

// now populate the GridView
GridView1.DataSource = dsManufacturers;
GridView1.DataBind();
```

As you can see, the code is very similar to the code that you used for the DataReader earlier. You construct the object to cache, in this case a `DataSet`, and then add it to the cache.

The actual cache statement again specifies a 5-minute sliding expiration, so it will be removed from the cache 5 minutes after the last access (or earlier if ASP.NET decides it needs to be removed or you remove it manually).

When caching a DataSet, you need to use the most efficient method of storing it. By default, the DataSet is serialized as XML, and this isn't very efficient. By setting the RemotingFormat to SerializationFormat.Binary, you use the most efficient serialization method available.

Removing Objects from the Cache

When using time-based cache expiry, there is always the chance that the data that you've cached isn't the same as the data in the database. If you've cached a list of Manufacturers and another user has added a new Manufacturer, then the database and the cache no longer contain the same list of Manufacturers. In these cases, you need to remove the cached object so that the results are requeried from the database.

Removing an item from the cache manually is simplicity itself. As you know the key of the object that you want to remove, you simply call the Remove() method:

```
Cache.Remove("Manufacturers");
```

This will remove the specified object from the cache. An attempt to remove an object that isn't in the cache won't cause any problems; it will just be ignored. ASP.NET may remove cached objects automatically, so the object may have been removed without your knowledge.

Try It Out: Removing Cached Objects Manually

In this example, you'll add code to the pages for editing a Manufacturer to remove the list of Manufacturers if you modify a Manufacturer. The process for removing an object from the cache is the same whether you're using a Command object or a SqlDataSource. The only difference is when you call the Remove() method.

1. Open Manufacturers_Edit_Command.aspx and switch to the Source view.
2. Modify the SaveManufacturers() method as follows:

```
        // change the cancel to continue
        btnCancel.Text = "Continue";

        // remove from the Cache
        Cache.Remove("Manufacturers");
      }
    }
    catch (Exception ex)
    {
      lblError.Text = ex.Message;
      lblError.Visible = true;
    }
```

3. Open Manufacturers_Edit_DataSource.aspx and switch to the Source view.

4. Modify the `ItemUpdate` event for the `DetailsView` as follows:

```
else if (e.AffectedRows == 0)
{
  lblError.Text = "No update was made.  Concurrency problem.";
  lblError.Visible = true;
  e.KeepInEditMode = true;
}
else
{
  // remove from the Cache
  Cache.Remove("Manufacturers");
}
```

5. Open `Manufacturers_DataReader.aspx` and, if you don't already have one, add a breakpoint to the first line of code in the `Page_Load` event handler.

6. Save all three pages and set `Manufacturers_DataReader.aspx` as the start page for the Web site.

7. Press F5 to start debugging. When the code hits the breakpoint for the first time, step through the code, and you'll see that the database is queried for the list of Manufacturers.

8. Let the page finish loading and press F5 to refresh it. Step through the code, and you'll see that the cached version of the results is used.

9. Click the Edit Command button for one of the Manufacturers. On the next page, click Update to make a change (even though nothing has actually changed) to the database. Click the Continue button to return to the list of Manufacturers. The breakpoint will be hit again. This time, the cache won't contain the list of Manufacturers, because the list was removed when you updated the Manufacturer on the previous page.

10. Let the page finish loading, and then press F5 to refresh it and confirm that the list of Manufacturers has again been cached.

11. Click the Edit DataSource button for a Manufacturer. On the next page, click Update to force a database update. Click the Continue button to return to the list of Manufacturers.

12. Step through the code again. You'll see that the list of Manufacturers has again been removed from the cache.

How It Works

Although there were quite a few steps to the example, you've made only two code changes (well, actually the same change in two different places). The majority of the example was proving that the list of Manufacturers had indeed been removed from the cache.

Of interest is this line of code:

```
// remove from the Cache
Cache.Remove("Manufacturers");
```

You're removing the object with the key of `Manufacturers` from the cache. In this case, that's the list of Manufacturers.

For the Command object version of the edit page, you remove the object from the cache if you've updated the database. In this example, you're checking for any concurrency errors when updating, so you remove the list from the cache only when you've modified a row in the database:

```
// no records affected is error
if (intCount == 0)
{
  // ...
}
else
{
  // ...

  // remove from the Cache
  Cache.Remove("Manufacturers");
}
```

The `SqlDataSource` version of the page is similar in that you remove the list from the cache only if the update has been successful. In this case, you need to check if you don't have an exception raised and you've modified a row in the `ItemUpdated` event handler:

```
if (e.Exception != null)
{
  // ...
}
else if (e.AffectedRows == 0)
{
  // ...
}
else
{
  // remove from the Cache
  Cache.Remove("Manufacturers");
}
```

Once you've removed the object from the cache, the list of Manufacturers is repopulated when the `GridView` is displayed.

If you actually wait long enough (5 minutes), the list of Manufacturers will already have been removed from the cache before you call the `Remove()` method. This won't cause a problem. As stated earlier, if the object that you're trying to remove from the cache doesn't exist, `Remove()` is simply ignored.

Transactions

When you looked at inserting, updating, and deleting Players in Chapter 8, you saw that these tasks require two queries: one to handle the Player table and another to handle the

WhatPlaysWhatFormat table. In fact, you saw that updating an existing Player actually requires three queries to be executed.

When inserting or updating, you trap any errors that occur and continue the save only if the previous query executed correctly. Consider the `Click` event handler for the Save button:

```
// only save if valid
if (Page.IsValid == true)
{
  // save the player to the database
  bool blnPlayerError = SavePlayer();

  // did an error occur?
  if (blnPlayerError == true)
  {
    QueryResult.Text = "An error has occurred!";
  }
  else
  {
    // save the formats for the player
    bool blnFormatError = SaveFormats();

    // did an error occur?
    if (blnFormatError == true)
    {
      QueryResult.Text = "An error has occurred!";
    }
    else
    {
      // show the result
      QueryResult.Text = "Save of player '" +
        Request.QueryString["PlayerID"].ToString() +
        "' was successful";

      // disable the submit button
      SubmitButton.Enabled = false;
    }
  }
}
```

If the `SavePlayer()` method fails, you'll tell the user that there is an error and not attempt to execute the `SaveFormats()` method. If the `SavePlayer()` method succeeds, then the `SaveFormats()` method is called.

But what if the `SaveFormats()` method fails? The `SavePlayer()` method has already made its changes to the database, and only the second set of changes fail. The result is that the Player stored in the database is now incorrect.

What you need to happen is for all queries to the database to succeed. If any of the queries that are being executed fail, then they must all fail. What you need is a transaction.

Transactions are a way to group different queries so that they all complete or they all fail—it's all or nothing.

To use the correct terminology, you *begin*, or *start*, a transaction, and then perform whatever steps you need to take as part of the transaction. If everything went as you wanted it to, you then *commit* the transaction, and all modifications are saved to the database. If something has gone wrong, a *rollback* of the transaction takes place, and the database isn't modified—it will appear as though the SQL queries you executed never took place.

Transactions, as with most things in the computing world, have their own acronym: ACID. It's not some throwback to the 1960s, but the first letters of the four properties that all transactions must exhibit:

Atomic: All queries within the transaction should succeed or fail. The archetypal example that's always used is a bank transfer scenario. If money is being transferred between two bank accounts, the tasks of taking the money from the source account and putting the money in the destination account both must succeed or both must fail. You can't have money removed from the source account but not added to the destination account, and you can't have money appearing in the destination account without the money being removed from the source account.

Consistent: A transaction transforms the database from one consistent state to another consistent state.

Isolated: A transaction that's currently executing will not see the results of any other transaction until the other transaction has completed.

Durable: A committed transaction should remain committed in the database, even if a failure occurs after the transaction has been committed.

Transactions can be handled in the following three places:

- You can handle transactions in the database. With SQL Server 2005 and MySQL 5.0, it is possible to implement transactions within stored procedures and ensure that all the queries that are to be executed are indeed executed.
- You can handle transactions in your code. Using ADO.NET, you can enlist different Command objects using the same connection to be part of a transaction using a Transaction object (`SqlTransaction`, `OdbcTransaction`, or `OleDbTransaction`). This allows you to execute several different queries or stored procedures and commit the results to the database only if they all execute correctly.
- You can use an External Transaction Manager. .NET makes it possible to run transactions across several different database connections (which may be on one machine or across several machines) and to also include nondatabase resources. In previous versions of .NET, you could perform transactions using COM+ using the `System.EnterpriseServices` namespace, but this is quite complex. .NET 2.0 introduces the `System.Transactions` namespace, which makes implementing distributed transactions a lot easier.

One thing to bear in mind when using transactions is that they impart a performance penalty on execution. During the lifetime of the transaction, any resources that are used are locked until the transaction is completed or rolled back. Any other queries trying to access

those resources will be blocked and will have to wait until the transaction is over before the resource can be used.

Think carefully about whether to use transactions. Obviously, sometimes you must use transactions to ensure that the data is correct and can't be left in a state that you don't want it in. Don't, however, assume that every SQL query you're executing must be explicitly defined within a transaction. Transactions reduce the performance of the database, so if you don't need a transaction, don't use one.

Defining Database Transactions

Every query that you execute in the database will have an implicit transaction associated with it. As it's only a single query, you're never aware that it is running as a transaction, and you can, effectively, forget the fact that it is a transaction.

The simplest transaction that you will define is one that is "complete" within one stored procedure—either everything you're trying to do is committed to the database or it's all rolled back.

Consider the example of deleting a Player from the database. You need to make sure that the data is deleted from both the Player and WhatPlaysWhatFormat tables or from neither of them.

In SQL Server 2005, transactions are controlled using the `BEGIN TRANSACTION`, `COMMIT TRANSACTION`, and `ROLLBACK TRANSACTION` queries. In MySQL 5.0, you use the corresponding `START TRANSACTION`, `COMMIT`, and `ROLLBACK` queries.

Deleting a Player in SQL Server 2005 is as simple as executing the following queries:

```
-- start the transaction
BEGIN TRANSACTION

-- first delete
DELETE FROM WhatPlaysWhatFormat WHERE WPWFPlayerID = @PlayerID

-- second delete
DELETE FROM Player WHERE PlayerID = @PlayerID

-- commit the transaction
COMMIT TRANSACTION
```

First, you use the `BEGIN TRANSACTION` query to instruct the database that you want to start a transaction. After you execute the two `DELETE` queries, you then call `COMMIT TRANSACTION` to commit the changes to the database. It's only at this point that the data is actually deleted from the database.

You'll notice that you don't have a `ROLLBACK TRANSACTION` in the stored procedure. If a transaction is started and an error is raised, the `ROLLBACK TRANSACTION` is executed automatically by the database, which causes any changes to be rolled back. In this case, you wouldn't have an element that was partially deleted.

Although you don't need a `ROLLBACK TRANSACTION` if you have an error, you do need the `COMMIT TRANSACTION` at the end of the stored procedure. If the stored procedure reaches the `COMMIT TRANSACTION`, everything has gone correctly and you can commit the transaction. Although the transaction is rolled back automatically if an error has occurred, it won't be committed automatically; you must call `COMMIT TRANSACTION`. Failure to commit or roll back a

transaction that has been started will result in an error being raised and the transaction rolled back, which will not be what you wanted if you forgot to call `COMMIT TRANSACTION`.

■ **Note** The rollback of the transaction will not always be done by the database and may sometimes be handled by your code or ADO.NET. One example is adding a null value into a `NOT NULL` column in SQL Server 2005. This is not a fatal error as far as SQL Server is concerned and the rollback will be performed by ADO.NET.

Using a Transaction Object

Database transactions are handled in code using a Transaction object (either a `SqlTransaction`, an `OdbcTransaction`, or an `OleDbTransaction`). To use a Transaction object, you just need to tell the Command object that it's part of the transaction. As explained in Chapter 4, one of the Command object constructors takes a Transaction object as a parameter. For example, you can create a `SqlCommand` object as follows:

```
SqlCommand(string, SqlConnection, SqlTransaction)
```

Once you have a Transaction object, it's simple to enlist a Command object in the transaction by passing the object to the Command object constructor, or you can set the `Transaction` property after you've created the Command object. However, starting the transaction is not as simple as creating a new Transaction object.

You cannot create a Transaction object directly; you must use the `BeginTransaction()` method of the Connection object. Calling this method creates the necessary Transaction object and tells the Connection object that it needs to be transactional.

You can then use the created Transaction object with all the Command objects that you want to include in the transaction. Every Command object that uses the Connection object must also use the same Transaction object—once a Connection object is transactional, every related Command object must also be transactional. If they're not, an error will be raised when you try to execute a query on the nontransactional Command object.

When the transaction is complete, you call `Commit()` on the Transaction object to commit the transaction to the database or call `Rollback()` to abort the transaction.

The process for using transactions in code is simple and can be broken down into the following six steps:

1. Open the connection to the database.
2. Call the `BeginTransaction()` method on the Connection object to start the transaction and store the Transaction object for later use.
3. Create a Command object, and then specify the Transaction object that you want to use.
4. Use the Command object as you normally would.
5. Loop steps 3 and 4 as often as required.
6. Either commit or roll back the transaction by calling the `Commit()` or `Rollback()` method on the Transaction object.

You may have noticed the limitation with using the Transaction object: it works only across a single connection, and you cannot use a Transaction object with a Command object that uses a different connection. If you attempt to use the same Transaction object across different connections, you'll receive an error. If you want to run a transaction across different connections, you need to use the `System.Transactions` method of handling distributed transactions, as described in the "Implementing Distributed Transactions" section later in this chapter.

Try It Out: Using a Transaction Object

In this example, you'll modify the INSERT Player page from Chapter 8 to support transactions. You'll first modify one of the queries to force an error so that you can see the problems that occur when you don't have having transactions. By modifying the pages to use transactions, you'll show that the changes are rolled back when the error occurs.

1. Open Visual Web Developer and copy `Players.aspx` and `Players_Insert.aspx` from the `original` folder in the code download to the root of the `Chapter12` Web site.
2. From the Solution Explorer, set `Players.aspx` as the start page for the Web site.
3. Open `Players_Insert.aspx` and modify the `INSERT` query in the `SaveFormats()` method as follows:

```
// query to execute
string strQuery = "INSERT WhatPlaysWhatFormats ➥
  (WPWFPlayerID, WPWFFormatID) VALUES (@PlayerID, @FormatID)";
```

4. Save the page, and then start debugging for the Web site.
5. Click the Add Player link and fill in the details for a new Player. Select some Formats, and then click the Insert Player button. You know that there's going to be an error (as you've forced an incorrect `INSERT` query in the `SaveFormats()` method), and the error is handled, as shown in Figure 12-10.
6. Open SQL Server Management Studio and connect to the `localhost\BAND` database server. Navigate to the Tables node from the Players database.
7. Open the Players table. You'll see that the new Player has been added, as shown in Figure 12-11 (Pear, in this example).
8. Open the WhatPlaysWhatFormat table. You will not see the Formats that you selected. The error has prevented the Formats from being added, but the Player has still been added. You need a transaction to avoid the problem.
9. Close Internet Explorer and switch back to Visual Web Developer. Open `Players_Insert.aspx`.

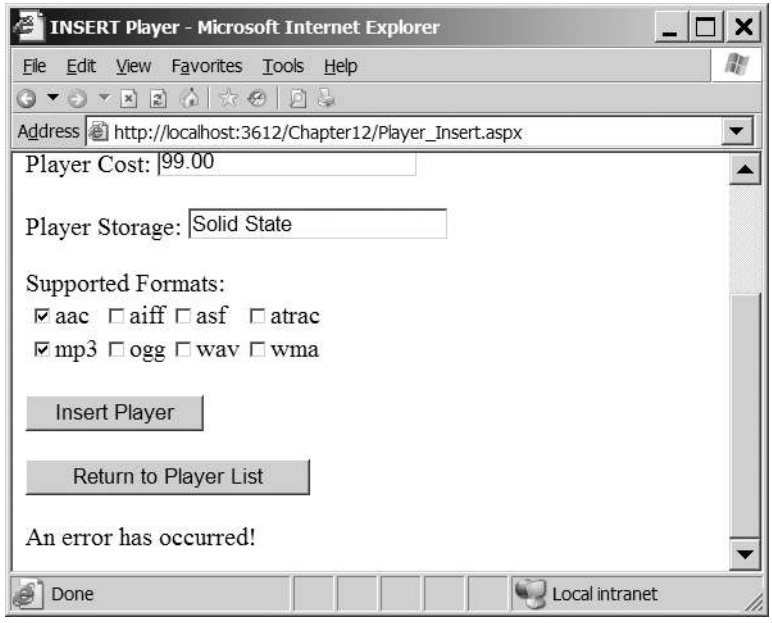

Figure 12-10. *The error is trapped but it isn't accurate.*

	16	H10	3	189.00	Hard Disk
	17	H300 Series	3	319.00	Hard Disk
	18	Carbon	5	169.00	Hard Disk
	19	Napster YH-920	10	179.00	Hard Disk
	20	Network Walkman NW-HD3	7	215.00	Hard Disk
	21	Pear	1	99.00	Solid State

Figure 12-11. *The Player has been added.*

10. Replace the `Click` event handler for the `SubmitButton` with the following:

```
protected void SubmitButton_Click(object sender, EventArgs e)
{
  // only save if valid
  if (Page.IsValid == true)
  {
    // create the connection
    string strConnectionString = ConfigurationManager.
      ConnectionStrings["SqlConnectionString"].ConnectionString;
    SqlConnection myConnection = new SqlConnection(strConnectionString);

    try
    {
      // open the connection
      myConnection.Open();
```

```
        // begin the transaction
        SqlTransaction myTransaction = myConnection.BeginTransaction();

        // save the player
        int intPlayerID = SavePlayer(myConnection, myTransaction);

        // save the formats
        SaveFormats(intPlayerID, myConnection, myTransaction);

        // commit the transaction
        myTransaction.Commit();

        // show the result
        QueryResult.Text = "Save of player '" +
          intPlayerID.ToString() + "' was successful";

        // disable the submit button
        SubmitButton.Enabled = false;
      }
      catch
      {
        // show the error
        QueryResult.Text = "An error has occurred!";
      }
      finally
      {
        // always close the connection
        myConnection.Close();
      }
    }
  }
```

11. Replace the `SavePlayer()` method with the following:

```
  private int SavePlayer(SqlConnection myConnection,
    SqlTransaction myTransaction)
  {
    // query to execute
    string strQuery = "INSERT Player (PlayerName, PlayerManufacturerID, ➥
      PlayerCost, PlayerStorage) VALUES (@Name, @ManufacturerID, @Cost, ➥
      @Storage); SELECT SCOPE_IDENTITY();";

    // create the command
    SqlCommand myCommand = new SqlCommand(strQuery, myConnection,
      myTransaction);
```

```
  // add the four parameters
  myCommand.Parameters.AddWithValue("@Name", PlayerName.Text);
  myCommand.Parameters.AddWithValue("@ManufacturerID",
    ManufacturerList.SelectedValue);
  myCommand.Parameters.AddWithValue("@Cost", PlayerCost.Text);
  myCommand.Parameters.AddWithValue("@Storage", PlayerStorage.Text);

  // execute the query
  int intPlayerID = Convert.ToInt32(myCommand.ExecuteScalar());

  // return the ID
  return (intPlayerID);
}
```

12. Replace the `SaveFormats()` method with the following (notice that it still has the error in the `INSERT` query):

```
private void SaveFormats(int intPlayerID, SqlConnection myConnection,
  SqlTransaction myTransaction)
{
  // query to execute
  string strQuery = "INSERT WhatPlaysWhatFormats ➥
    (WPWFPlayerID, WPWFFormatID) VALUES (@PlayerID, @FormatID)";

  // create the command object
  SqlCommand myCommand = new SqlCommand(strQuery, myConnection,
    myTransaction);

  // add the two parameters
  myCommand.Parameters.AddWithValue("@PlayerID", intPlayerID);
  myCommand.Parameters.Add("@FormatID", System.Data.SqlDbType.Int);

  // loop through each of the formats
  foreach (ListItem objFormat in FormatList.Items)
  {
    // save if selected
    if (objFormat.Selected == true)
    {
      // set the parameter value
      myCommand.Parameters["@FormatID"].Value = objFormat.Value;

      // execute the query
      myCommand.ExecuteNonQuery();
    }
  }
}
```

13. Save the page and open `Players.aspx`. Again choose to add a new Player to the database.

14. Enter the details for a new Player and click the Insert Player button. Again, the error will be trapped (Figure 12-10). However, if you query the database, you'll see that neither the Player or WhatPlaysWhatFormat table shows any details for the failed insertion.

15. Go back to Visual Web Developer and fix the broken `INSERT` query in `SaveFormats()` as follows:

```
// query to execute
string strQuery = "INSERT WhatPlaysWhatFormat ➥
  (WPWFPlayerID, WPWFFormatID) VALUES (@PlayerID, @FormatID)";
```

How It Works

Although it looked like you modified quite a lot of code in this example, that really wasn't the case! Most of the changes in the `SavePlayer()` and `SaveFormats()` methods were to remove code, and we'll look at these shortly. The really interesting code, at least from a transactional point of view, is in the `SubmitButton_Click` event handler.

Handling the Transaction

When using a Transaction object, all of the queries need to operate over the same connection. But before you create that connection, you check that the page is valid (that is, all the validators that you may have added have passed):

```
// only save if valid
if (Page.IsValid == true)
{
  // create the connection
  string strConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  SqlConnection myConnection = new SqlConnection(strConnectionString);

  try
  {
    // ...
  }
  catch
  {
    // show the error
    QueryResult.Text = "An error has occurred!";
  }
  finally
  {
    // always close the connection
    myConnection.Close();
  }
}
```

You've also added the error-handling code that you've come to expect. If you have an error anywhere while using the Connection object, you display an error message to the user (in the `catch` block) before always closing the connection (in the `finally` block).

A transaction can only be created on an open connection, so the first thing you need to do is open the connection:

```
// open the connection
myConnection.Open();
```

Once you have an open connection, you can create a transaction by calling the `BeginTransaction()` method of the Connection object:

```
// begin the transaction
SqlTransaction myTransaction = myConnection.BeginTransaction();
```

The `BeginTransaction()` method returns a Transaction object that you can use to enlist Command objects into the transaction, as you'll see when we look at the `SavePlayer()` and `SaveFormats()` methods shortly.

The next two lines are the calls to the two methods to save the Player to the database. In both cases, you need to pass the Connection and Transaction objects to the methods:

```
// save the player
int intPlayerID = SavePlayer(myConnection, myTransaction);

// save the formats
SaveFormats(intPlayerID, myConnection, myTransaction);
```

As all the queries need to use the same connection and same transaction, you need the two methods to be able to see the Connection and Transaction objects in order to use them. You could have made them available as global variables to the page, but they should be available only where they're needed, so you pass them to the two methods.

If both methods execute correctly, then you want to commit the transaction by calling the `Commit()` method of the Transaction object:

```
// commit the transaction
myTransaction.Commit();
```

This will make the changes to the database. Then you proceed to close the connection in the `finally` block.

But, where's the call to the `Rollback()` method if there's an error? There isn't one! If an error occurs anywhere within the `try` block, execution jumps to the `catch` block, and the `Commit()` method is never called. The Transaction object is pessimistic. If it isn't committed, then when it is disposed of (as it will be when it goes out of scope with the jump to the `catch` block), the `Rollback()` method is automatically called.

So, if there's an error, the changes are rolled back automatically. This is exactly what you saw happen in the example.

Saving the Player and the Formats

You've removed all error handling from the `SavePlayer()` method. You've moved all of the necessary error handling to the `SubmitButton_Click` event handler, so that you can handle the commit and rollback of the transaction from a central location.

Apart from the lack of error handling, you've made only two changes to the method. You've removed the creation of a connection to the database (as you're now passed the Connection object to use), and you've changed the way that the Command object is constructed:

```
// create the command
SqlCommand myCommand = new SqlCommand(strQuery, myConnection,
  myTransaction);
```

As well as specifying the query to execute and the existing Connection object, you're also passing in the Transaction object that you want to use. Any queries executed by this Command object will now be enlisted in the transaction and will be committed or rolled back with all the other queries enlisted in the transaction.

The changes to the `SaveFormats()` method follow the same pattern as the `SavePlayer()` method. You've removed all the error handling, you no longer create the connection to the database, and when creating the Command object, you enlist it in the provided transaction:

```
// create the command object
SqlCommand myCommand = new SqlCommand(strQuery, myConnection,
    myTransaction);
```

Whenever the query is executed (which it will be once for each Format that the Player supports), it will be enlisted in the transaction and committed only if all of the other queries execute successfully.

Implementing Distributed Transactions

As you've just seen, using a Transaction object to control transactions is ideal and will allow you to perform multiple queries across the same connection. However, not all transactions occur to the same database or across the same connection. Indeed, the `Players_Insert.aspx` page originally used two different connections to the same database. You modified the page to use the same connection so that you could use a Transaction object.

However, you may not always be able to modify the code to allow the use of the Transaction object. In this case, you need to enable a distributed transaction.

Put simply, a *distributed transaction* is a transaction that needs to run across more than one database connection. The different connections may be to the same database, they may be to different databases on the same server, or they may be connections to different database servers altogether. The key is that as soon as you use a different Connection object, the transaction becomes distributed.

In ASP.NET 1.1, you would need to write code to deal with the `System.EnterpriseServices` namespace in order to implement a distributed transaction. In ASP.NET 2.0, you can now use the `System.Transactions` namespace, which provides various objects to make distributed transactions quite simple. In order to use a distributed transaction, all you need to do is create a `TransactionScope` object:

```
using (TransactionScope objTransScope = new TransactionScope())
{
  // any database interaction here is transactional

  // must commit the transaction
  objTransScope.Complete();
}
```

Any code that executes against a database connection within the `using` statement is automatically enlisted in the transaction. If an error occurs and the `Complete()` method isn't called, the transaction will be rolled back automatically.

You don't always need to use the `using` statement to control what is enlisted in the transaction. You can also use a normal `try..catch..finally` statement to handle the transaction:

```
TransactionScope objTransScope = new TransactionScope()

try
{
  // any database interaction here is transactional

  // must commit the transaction
  objTransScope.Complete();
}
finally
{
  objTransScope.Dispose();
}
```

This is equivalent to the preceding code. Any errors in the `try` block will cause the `Complete()` method to be skipped and the transaction to be rolled back when the `Dispose()` method is called.

■ **Note** Currently, only SQL Server 2005 is designed to be used with the `System.Transactions` namespace. Neither Microsoft Access nor MySQL 5.0, at the time of printing, can be used in conjunction with the `System.Transactions` namespace. Microsoft Access will throw an error if you try it, and MySQL 5.0 will simply ignore the transaction scope and execute each query individually. With these databases, you're stuck with using the `OleDbTransaction` and `OdbcTransaction` objects to enforce transactions across the same Connection object. But in the future, you can expect the namespace to be used in a lot more cases. For more details about using `System.Transactions`, see `http://msdn.microsoft.com/en-us/library/0abf6ykb.aspx`.

Although the `System.Transactions` namespace is designed to support distributed transactions, not every transaction that it handles is distributed. A transaction under `System.Transactions` starts its life under the Lightweight Transaction Manager (LTM) and, if the transaction is not distributed, will remain under the control of the LTM. If the transaction needs to

be distributed, it must be under the control of the Distributed Transaction Coordinator (DTC), and using this adds an overhead to the transaction.

Thankfully, .NET 2.0 manages the transaction for you. It decides when a transaction needs to be under the control of the DTC and promotes the transaction from being a local transaction to a full-blown distributed transaction.

So when does a transaction require the DTC to control it? For our purposes, this occurs whenever you use a second Connection object. When a connection is first used within the `TransactionScope`, it is initially under the control of the LTM. Since it's using only one connection, there is no need to add the overhead of making the transaction distributed. If no other connections are used, then all the queries will be executed under the control of the LTM; the DTC is never used.

As soon as a different Connection object tries to enlist in the transaction, it is promoted to being a distributed transaction under the control of the DTC. However this requires that you have the Distributed Transaction Coordinator service running; otherwise, the transaction cannot be promoted and an error will be generated.

We'll now look at both of these scenarios: local and distributed transactions.

Try It Out: Using a Local Transaction

In order for a transaction to be under the control of the LTM, you need to use the same connection for all queries within the transaction. In this example, you'll modify the DELETE Player page from Chapter 8 to use the `System.Transactions` namespace and implement a local transaction.

1. Open the Administrative Tools folder of Control Panel and open the Services application. Find the Distributed Transaction Coordinator in the list of services. If it is running, stop it (either from the context menu or from the toolbar).

2. Open the `Chapter12` Web site in Visual Web Developer and a reference to `System.Transactions` to the Web site.

3. Copy `Players_Delete.aspx` from the `original` folder in the code download to the root of the Web site.

4. Open `Players_Delete.aspx` in the root of the Web site and add the correct `Import` statement to the top of the page:

```
<%@ Import Namespace="System.Transactions" %>
```

5. Replace the `SubmitButton_Click` event handler with the following (there is an intentional error in the second `DELETE` query):

```
protected void SubmitButton_Click(object sender, EventArgs e)
{
  // create the connection
  string strConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  SqlConnection myConnection = new SqlConnection(strConnectionString);
```

```
  try
  {
    using (TransactionScope objTransScope = new TransactionScope())
    {
      // create the first query
      string strQuery1 = "DELETE FROM WhatPlaysWhatFormat ➥
        WHERE WPWFPlayerID = @PlayerID;";
      SqlCommand myCommand1 = new SqlCommand(strQuery1, myConnection);
      myCommand1.Parameters.AddWithValue("@PlayerID",
        Request.QueryString["PlayerID"]);

      // create the second query
      string strQuery2 = "DELETE FROM Players WHERE PlayerID = @PlayerID;";
      SqlCommand myCommand2 = new SqlCommand(strQuery2, myConnection);
      myCommand2.Parameters.AddWithValue("@PlayerID",
        Request.QueryString["PlayerID"]);

      // open the connection
      myConnection.Open();

      // execute the queries
      myCommand1.ExecuteNonQuery();
      myCommand2.ExecuteNonQuery();

      // show the result
      QueryResult.Text = "Delete of player '" +
        Request.QueryString["PlayerID"] + "' was successful";

      // disable the submit button
      SubmitButton.Enabled = false;

      // must commit the transaction
      objTransScope.Complete();
    }
  }
  catch (Exception ex)
  {
    // show the error
    QueryResult.Text = "An error has occurred: " + ex.Message;
  }
  finally
  {
    // close the connection
    myConnection.Close();
  }
}
```

6. Save the page, and then start debugging to load `Players.aspx`.

7. Click the Add Player link. Add a new Player to the database, and then click Continue to return to the list of Players.

8. Find the Player that you've just added and click the Delete button. When `Players_Delete.aspx` is loaded, click the Delete Player button to confirm the deletion. As shown in Figure 12-12, an error has occurred.

Figure 12-12. *The Player cannot be deleted.*

9. Open SQL Server Management Studio. You'll see that the Player is still present in the Players table and also in the WhatPlaysWhatFormat table, as shown in Figure 12-13.

	22	3
	22	4
	22	5
	22	6

Figure 12-13. *The Player is still in the database.*

10. Stop debugging. Modify the second query in the `SubmitButton_Click` event handler so that it is now correct:

```
// create the second query
string strQuery2 = "DELETE FROM Player WHERE PlayerID = @PlayerID;";
```

11. Load the Web site again. You'll be able to delete the Player, and the transaction will commit to the database. If you check the Player and WhatPlaysWhatFormat tables, you'll see that the Player has indeed been deleted.

How It Works

Once again, you've used an existing page to reduce the work that you need to do in order to see the desired results.

Within the `SubmitButton_Click` event handler, you're executing two `DELETE` queries against the database. The first thing you need is a connection to the database, so you create a Connection object, `myConnection`, as you've done in previous examples. You then wrap the remainder of the database access code in a `try..catch..finally` block so that you can handle any errors that occur.

To make all the queries executed against the database transactional, you wrap all of the database interactions within a `using` statement (only the pertinent parts of the code are shown here):

```
using (TransactionScope objTransScope = new TransactionScope())
{
  // create the first query
  SqlCommand myCommand1 = new SqlCommand(strQuery1, myConnection);

  // create the second query
  SqlCommand myCommand2 = new SqlCommand(strQuery2, myConnection);

  // open the connection
  myConnection.Open();

  // execute the queries
  myCommand1.ExecuteNonQuery();
  myCommand2.ExecuteNonQuery();

  // must commit the transaction
  objTransScope.Complete();
}
```

Once you have a `TransactionScope` object, you can create the two Command objects that you want to execute. Notice that they're both using the same Connection object, `myConnection`, so this transaction will remain under the control of the LTM, rather than being promoted to a distributed transaction under the control of the DTC.

After you've created the two Command objects, you open the connection to the database, execute the queries, and then `Complete()` the transaction.

If all goes well, the changes to the database will be made when you call `Complete()`. However, if there's an error, as in this case, the call to the `Complete()` method will be skipped, and the transaction will be rolled back automatically.

Let's backtrack to the very beginning of this example. The first thing that you did was turn off the Distributed Transaction Coordinator service, which effectively turns off distributed transactions. This proves that the transaction that you created remained a local transaction under the control of the LTM. If the transaction had needed to be promoted to a distributed transaction, an error would have been thrown. You'll see this in the next example.

Try It Out: Using a Distributed Transaction

You're now going to build an example that requires a distributed transaction. You'll see how the transaction is promoted to distributed only when a second Connection object is used.

1. Open Visual Web Developer and copy `Players_Update.aspx` from the `original` folder in the code download to the root of the `Chapter12` Web site.

2. Open `Players_Update.aspx` in the root of the Web site and add the correct `Import` statement to the top of the page:

```
<%@ Import Namespace="System.Transactions" %>
```

3. Modify the `SubmitButton_Click` event handler as follows (the changed code is in bold):

```
protected void SubmitButton_Click(object sender, EventArgs e)
{
  // only save if valid
  if (Page.IsValid == true)
  {
    using (TransactionScope objTransScope = new TransactionScope())
    {
      // save the player to the database
      bool blnPlayerError = SavePlayer();

      // did an error occur?
      if (blnPlayerError == true)
      {
        QueryResult.Text = "An error has occurred!";
      }
      else
      {
        // save the formats for the player
        bool blnFormatError = SaveFormats();

        // did an error occur?
        if (blnFormatError == true)
        {
          QueryResult.Text = "An error has occurred!";
        }
        else
        {
          // show the result
          QueryResult.Text = "Save of player '" +
            Request.QueryString["PlayerID"].ToString() + "' was successful";

          // disable the submit button
          SubmitButton.Enabled = false;
```

```
            // must commit the transaction
            objTransScope.Complete();
          }
        }
      }
    }
  }
```

4. Add a breakpoint to the `using` statement that you've added to `SubmitButton_Click`.
5. Save the page, and then start debugging to load `Players.aspx`.
6. Click the Edit button for one of the Players in the list. Click the Update Player button to start the update process and hit the breakpoint you've added.
7. Step through the code and step into the `SavePlayer()` and `SaveFormats()` methods. You'll be able to step through `SavePlayer()` without any problems, and the `UPDATE` query in that method will execute without any problems. However the `SaveFormats()` method isn't as cooperative. As soon as you try to open the new `SqlConnection` object, an error is thrown, as shown in Figure 12-14.

Figure 12-14. *The transaction can't be promoted to a distributed transaction.*

8. Press F5 to continue execution, and `Players_Update.aspx` will show an error. If you use SQL Server Management Studio, none of the changes that you've made will have been committed to the database.
9. Open the Services application from the Administrative Tools folder of the Control Panel and start the Distributed Transaction Coordinator service.
10. Save the changes to the Player. You'll be able to step through the code without any problems, and the transaction will be committed to the database.

How It Works

As explained earlier, a transaction is promoted to being distributed whenever it uses more than one database connection, and a distributed transaction is controlled by the DTC. This example has shown both of these features.

In the previous example, you stopped the Distributed Transaction Coordinator service to demonstrate that if you use the same connection to the database, you don't create a distributed transaction; it remains a local transaction. By starting this example with the service still stopped, you've seen that the transaction that you're executing does indeed need to be promoted.

You've wrapped all of the code to access the database inside a `using` statement, and any queries that you execute will be automatically enlisted within the `TransactionScope` you specified. This is true for code that runs directly within the `using` statement (as you saw in the previous example) or any code that runs in methods that are called from within the `using` statement.

Both `SavePlayer()` and `SaveFormats()` create their own Connection object and, even though these are to the same database, because you have more than one Connection object, it will automatically become a distributed transaction. The interesting thing to notice is that it doesn't become a distributed transaction until the second Connection object is required. The `SavePlayer()` method actually runs as a local transaction. It isn't until the `SaveFormats()` method tries to use its own Connection object that the transaction needs to be distributed. At that point, the transaction is automatically promoted from a local transaction to a distributed transaction and, as the Distributed Transaction Coordinator service isn't running, an error is thrown.

Multiple Result Sets

As you've learned in previous chapters you can execute several different queries as part of the same query batch to the database. For example, when you add a new Player to the database, you're executing an `INSERT` query to add to the Player table and then a `SELECT` query to return the PlayerID of the newly added Player:

```
INSERT Player (PlayerName, PlayerManufacturerID,
  PlayerCost, PlayerStorage)
VALUES (@Name, @ManufacturerID, @Cost, @Storage);
SELECT SCOPE_IDENTITY();
```

One thing that all of the examples that we've looked at so far have in common is that they contain only one `SELECT` query. When you've needed to execute two `SELECT` queries, you've executed these using two different `SqlCommand` objects. However, as noted in Chapter 8, you can actually execute multiple `SELECT` queries as part of the same query batch. For example, to retrieve all of the details for a Player at the same time, you could execute the following query batch:

```
SELECT PlayerName, PlayerManufacturerID, PlayerCost, PlayerStorage
  FROM Player WHERE PlayerID=@PlayerID;
SELECT WPWFFormatID FROM WhatPlaysWhatFormat
  WHERE WPWFPlayerID = @PlayerID;
```

If you executed this query batch through the `ExecuteReader()` method, the results of both queries would be returned within the same DataReader. You would need to use the `NextResult()` method to access the results of the second query.

■ **Note** Only SQL Server 2005 allows you to use a query batch to execute multiple SELECT queries. Neither Microsoft Access nor MySQL 5.0 supports query batches. With those databases, you need to use separate SELECT queries.

Try It Out: Executing Two SELECT Queries in a Query Batch

In this example, you'll update the Players_Update.aspx page to query the database only once when retrieving an existing player. You will use a query batch to execute the two SELECT queries, and then use the NextResult() method to access the results of the second query.

1. Open Visual Web Developer and open Players_Update.aspx in the root of the Web site.
2. Replace the RetrieveExistingPlayer() method with the following:

```
private void RetrieveExistingPlayer()
{
  // create the connection
  string strConnectionString = ConfigurationManager.
    ConnectionStrings["SqlConnectionString"].ConnectionString;
  SqlConnection myConnection = new SqlConnection(strConnectionString);

  try
  {
    // create the query batch
    string strQuery = "SELECT PlayerName, PlayerManufacturerID, ➥
      PlayerCost, PlayerStorage FROM Player WHERE PlayerID = ➥
      @PlayerID; SELECT WPWFFormatID FROM WhatPlaysWhatFormat ➥
      WHERE WPWFPlayerID = @PlayerID;";
    SqlCommand myCommand = new SqlCommand(strQuery, myConnection);
    myCommand.Parameters.AddWithValue("@PlayerID",
      Request.QueryString["PlayerID"]);

    // open the connection
    myConnection.Open();

    // execute the query batch
    SqlDataReader myReader = myCommand.ExecuteReader();

    // if we have results then we need to parse them
    if (myReader.Read() == true)
    {
      PlayerName.Text = myReader.GetString(
        myReader.GetOrdinal("PlayerName"));
      ManufacturerList.SelectedValue = myReader.GetInt32(
        myReader.GetOrdinal("PlayerManufacturerID")).ToString();
```

```
            PlayerCost.Text = myReader.GetDecimal(
              myReader.GetOrdinal("PlayerCost")).ToString();
            PlayerStorage.Text = myReader.GetString(
              myReader.GetOrdinal("PlayerStorage"));
          }

          // get the next results
          myReader.NextResult();

          // if we have results then we need to parse them
          while (myReader.Read() == true)
          {
            foreach (ListItem objFormat in FormatList.Items)
            {
              if (objFormat.Value == myReader.GetInt32(
                myReader.GetOrdinal("WPWFFormatID")).ToString())
              {
                objFormat.Selected = true;
                break;
              }
            }
          }

          // close the reader
          myReader.Close();
        }
        finally
        {
          // close the connection
          myConnection.Close();
        }
      }
```

3. Save the page, and then start debugging the Web site. Click to edit one of the Players, and you'll see that the details of the existing Player are returned as expected.

How It Works

In this example, you've modified the `RetrieveManufacturer()` method to execute a query batch containing two `SELECT` queries:

```
SELECT PlayerName, PlayerManufacturerID, PlayerCost,
  PlayerStorage FROM Player WHERE PlayerID = @PlayerID;
SELECT WPWFFormatID FROM WhatPlaysWhatFormat
  WHERE WPWFPlayerID = @PlayerID;";
```

When executing a query batch containing several `SELECT` queries, the DataReader is initially connected to the results of the first `SELECT` query.

So, initially, you have access to the details for the individual Player. You need to first check that you have results for this query using the `Read()` method (the `HasRows` property would work equally as well) before you parse the results and set the controls on the page:

```
// if we have results then we need to parse them
if (myReader.Read() == true)
{
  // ...
}
```

Once you're finished with the first set of results, you can move on to the results for the second `SELECT` query by calling the `NextResult()` method:

```
// get the next results
myReader.NextResult();
```

The `NextResult()` method advances to the next set of results in the DataReader. In this case, you're returning the media Formats that the Player supports, and you can parse through the rows returned:

```
// if we have results then we need to parse them
while (myReader.Read() == true)
{
  // ...
}
```

That's all there is to it. For every `SELECT` query, there is a result set in the DataReader that you can access. Even if the `SELECT` query doesn't return any results, it will still have a result set that has the `HasRows` property set to `false`.

Summary

In this chapter, we've looked at a few topics that will broaden your knowledge and help you build better Web sites. Here, you learned the following:

- With a few changes to the `UPDATE` query, you can prevent changes from being made to the database if the data in the database is different from what you were expecting.
- By caching data that changes infrequently, you can improve performance by reducing the number of queries made against the database.
- By placing several queries in the same transaction, you can commit or roll back the changes as a whole to ensure that the database isn't left in an inconsistent state.
- By placing several `SELECT` queries in the same query batch, you can return several sets of results using the same DataReader.

This chapter completes this book's coverage of specific techniques for building data-driven Web sites. The next and final chapter provides some guidance on how to put it all together into a well-designed and well-implemented application.