Working with Data in ASP.NET 2.0 :: Including a File Upload Option When Adding a New Record

Introduction

In the previous two tutorials we explored techniques for storing binary data that is associated with the application's data model, looked at how to use the FileUpload control to send files from the client to the web server, and saw how to present this binary data in a data Web control. We've yet to talk about how to associate uploaded data with the data model, though.

In this tutorial we will create a web page to add a new category. In addition to TextBoxes for the category's name and description, this page will need to include two FileUpload controls – one for the new category's picture and one for the brochure. The uploaded picture will be stored directly in the new record's `Picture` column, whereas the brochure will be saved to the `~/Brochures` folder with the path to the file saved in the new record's `BrochurePath` column.

Before creating this new web page, we'll need to update the architecture. The `CategoriesTableAdapter`'s main query does not retrieve the `Picture` column. Consequently, the auto-generated `Insert` method only has inputs for the `CategoryName`, `Description`, and `BrochurePath` fields. Therefore, we need to create an additional method in the TableAdapter that prompts for all four `Categories` fields. The `CategoriesBLL` class in the Business Logic Layer will also need to be updated.

Step 1: Adding an `InsertWithPicture` Method to the `CategoriesTableAdapter`

When we created the `CategoriesTableAdapter` back in the Creating a Data Access Layer tutorial, we configured it to automatically generate `INSERT`, `UPDATE`, and `DELETE` statements based on the main query. Moreover, we instructed the TableAdapter to employ the DB Direct approach, which created the methods `Insert`, `Update`, and `Delete`. These methods execute the auto-generated `INSERT`, `UPDATE`, and `DELETE` statements and, consequently, accept input parameters based on the columns returned by the main query. In the Uploading Files tutorial we augmented the `CategoriesTableAdapter`'s main query to use the `BrochurePath` column.

Since the `CategoriesTableAdapter`'s main query does not reference the `Picture` column, we can neither add a new record nor update an existing record with a value for the `Picture` column. In order to capture this information, we can either create a new method in the TableAdapter that is used specifically to insert a record with binary data or we can customize the auto-generated `INSERT` statement. The problem with customizing the auto-generated `INSERT` statement is that we risk having our customizations overwritten by the wizard. For example, imagine that we customized the `INSERT` statement to include use of the `Picture` column. This would update the TableAdapter's `Insert` method to include an additional input parameter for the category's picture's binary data. We could then create a method in the Business Logic Layer to use this DAL method and invoke this BLL method through the Presentation Layer, and everything would work wonderfully. That is, until the next time we configured the TableAdapter through the TableAdapter Configuration wizard. As soon as the wizard completed, our customizations to the `INSERT` statement would be overwritten, the `Insert` method would revert to its old form, and our code would no longer compile!

Note: This annoyance is a non-issue when using stored procedures instead of ad-hoc SQL statements. A future tutorial will explore using stored procedures in lieu of ad-hoc SQL statements in the Data Access Layer.

To avoid this potential headache, rather than customizing the auto-generated SQL statements let's instead create a new method for the TableAdapter. This method, named `InsertWithPicture`, will accept values for the `CategoryName`, `Description`, `BrochurePath`, and `Picture` columns and execute an `INSERT` statement that stores all four values in a new record.

Open the Typed DataSet and, from the Designer, right-click on the `CategoriesTableAdapter`'s header and choose Add Query from the context menu. This launches the TableAdapter Query Configuration Wizard, which begins by asking us how the TableAdapter query should access the database. Choose "Use SQL statements" and click Next. The next step prompts for the type of query to be generated. Since we're creating a query to add a new record to the `Categories` table, choose "INSERT" and click Next.

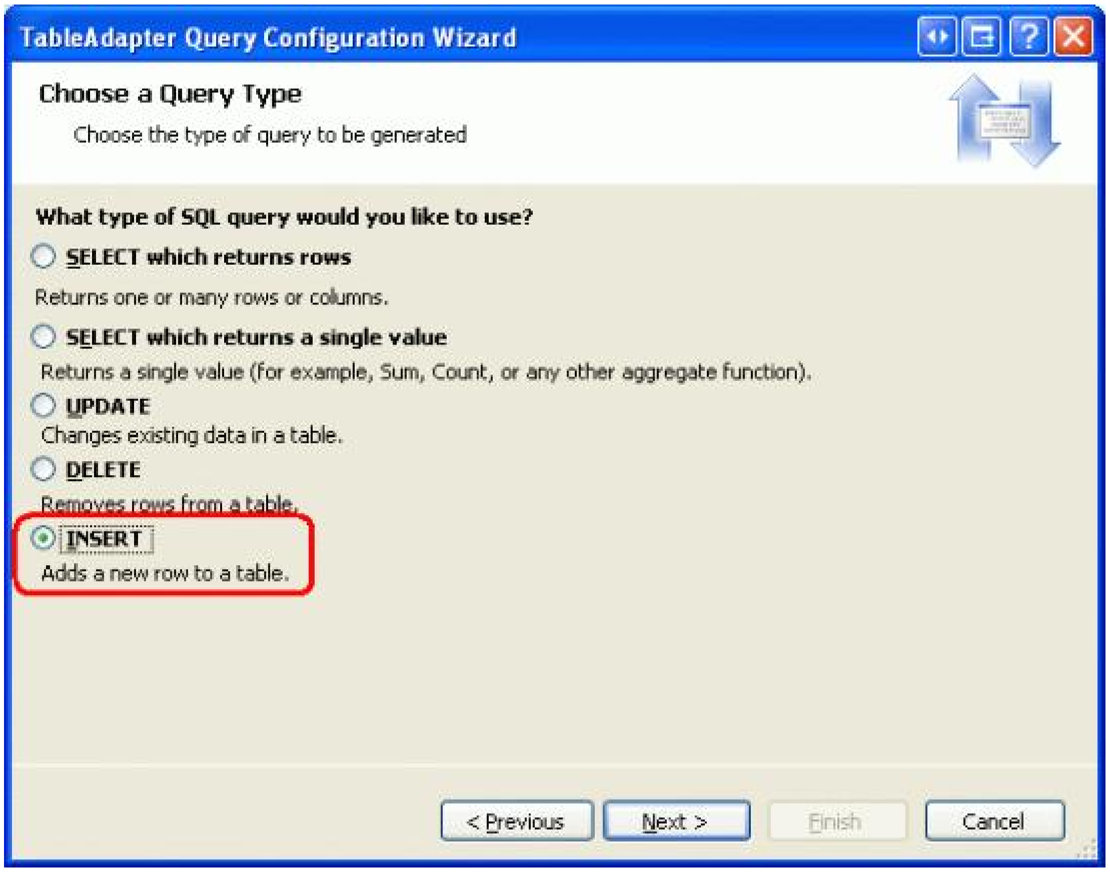
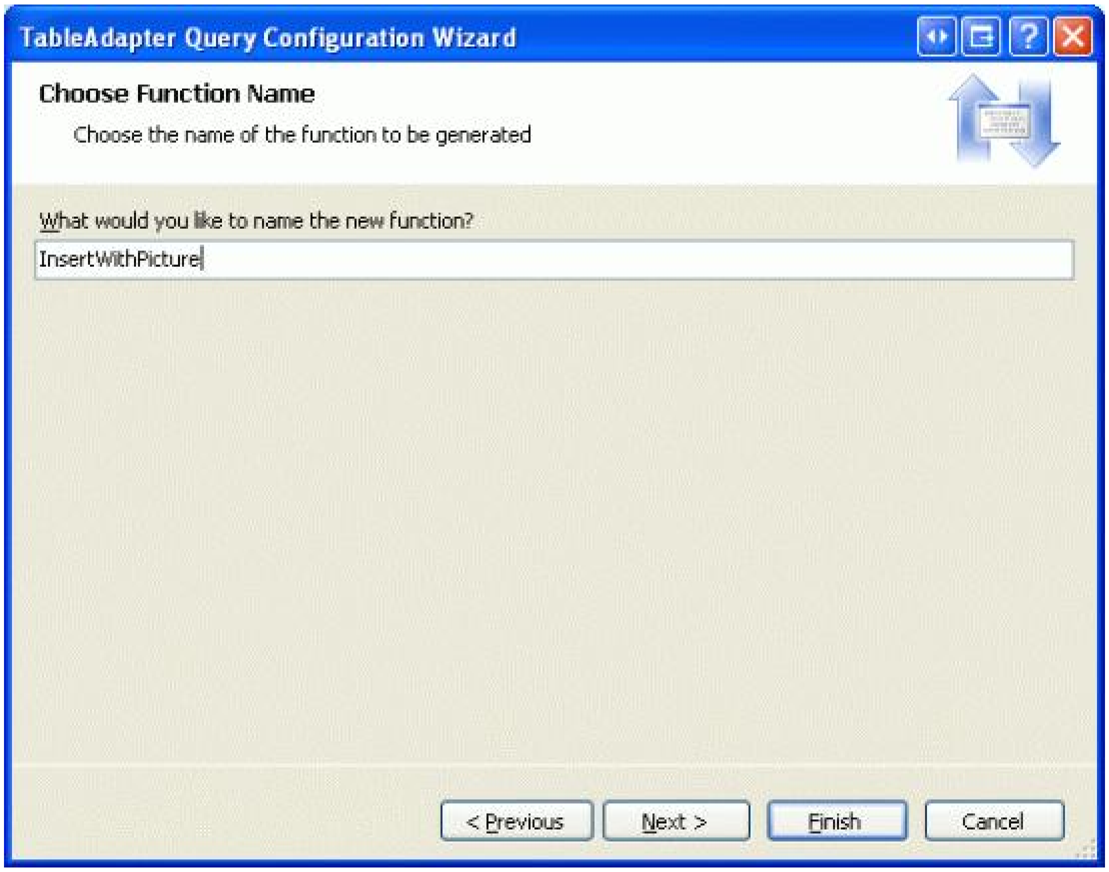

TableAdapter Query Configuration Wizard
Choose a Query Type
Choose the type of query to be generated
What type of SQL query would you like to use?
SELECT which returns rows
Returns one or many rows or columns.
SELECT which returns a single value
Returns a single value (for example, Sum, Count, or any other aggregate function).
UPDATE
Changes existing data in a table.
DELETE
Removes rows from a table.
INSERT
Adds a new row to a table.
< Previous
Next >
Finish
Cancel

Figure 1: Select the "INSERT" Option

We now need to specify the `INSERT` SQL statement. The wizard automatically suggests an `INSERT` statement corresponding to the TableAdapter's main query. In this case, it's an `INSERT` statement that inserts the `CategoryName`, `Description`, and `BrochurePath` values. Update the statement so that the `Picture` column is included along with a `@Picture` parameter, like so:

```
INSERT INTO [Categories]
    ([CategoryName], [Description], [BrochurePath], [Picture])
VALUES
    (@CategoryName, @Description, @BrochurePath, @Picture)
```

The final screen of the wizard asks us to name the new TableAdapter method. Enter `InsertWithPicture` and click Finish.


TableAdapter Query Configuration Wizard
Choose Function Name
Choose the name of the function to be generated
< Previous
Next >
Finish
Cancel

What would you like to name the new function?

InsertWithPicture

Figure 2: Name the New TableAdapter Method `InsertWithPicture`

Step 2: Updating the Business Logic Layer

Since the Presentation Layer should only interface with the Business Logic Layer rather than bypassing it to go directly to the Data Access Layer, we need to create a BLL method that invokes the DAL method we just created (`InsertWithPicture`). For this tutorial, create a method in the `CategoriesBLL` class named `InsertWithPicture` that accepts as input three `string`s and a `byte` array. The `string` input parameters are for the category's name, description, and brochure file path, while the `byte` array is for the binary contents of the category's picture. As the following code shows, this BLL method invokes the corresponding DAL method:

```
[System.ComponentModel.DataObjectMethodAttribute
    (System.ComponentModel.DataObjectMethodType.Insert, false)]
public void InsertWithPicture(string categoryName, string description,
    string brochurePath, byte[] picture)
{
    Adapter.InsertWithPicture(categoryName, description, brochurePath, picture);
}
```

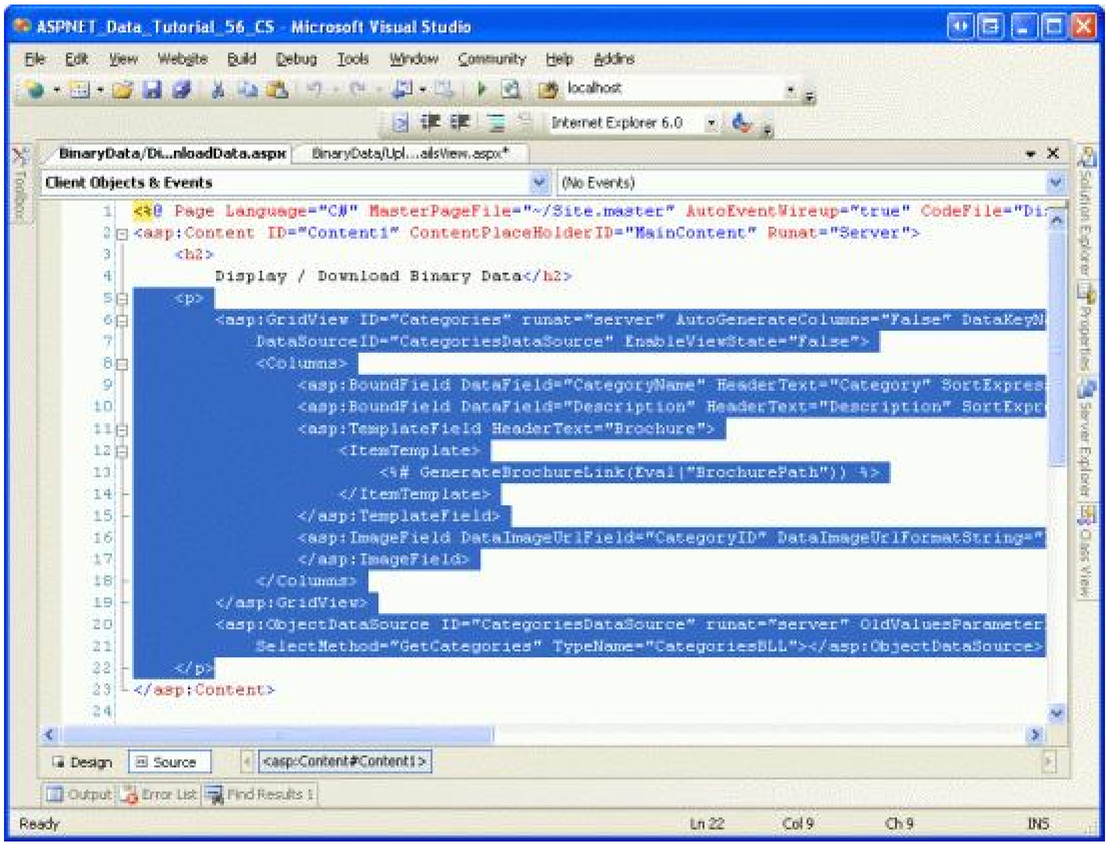
Note: Make sure that you have saved the Typed DataSet before adding the `InsertWithPicture` method to the BLL. Since the `CategoriesTableAdapter` class code is auto-generated based on the Typed DataSet, if you don't first save your changes to the Typed DataSet the `Adapter` property won't know about the

`InsertWithPicture` method.

Step 3: Listing the Existing Categories and their Binary Data

In this tutorial we will create a page that allows an end user to add a new category to the system, providing a picture and brochure for the new category. In the [preceding tutorial](#) we used a GridView with a TemplateField and ImageField to display each category's name, description, picture, and a link to download its brochure. Let's replicate that functionality for this tutorial, creating a page that both lists all existing categories and allows for new ones to be created.

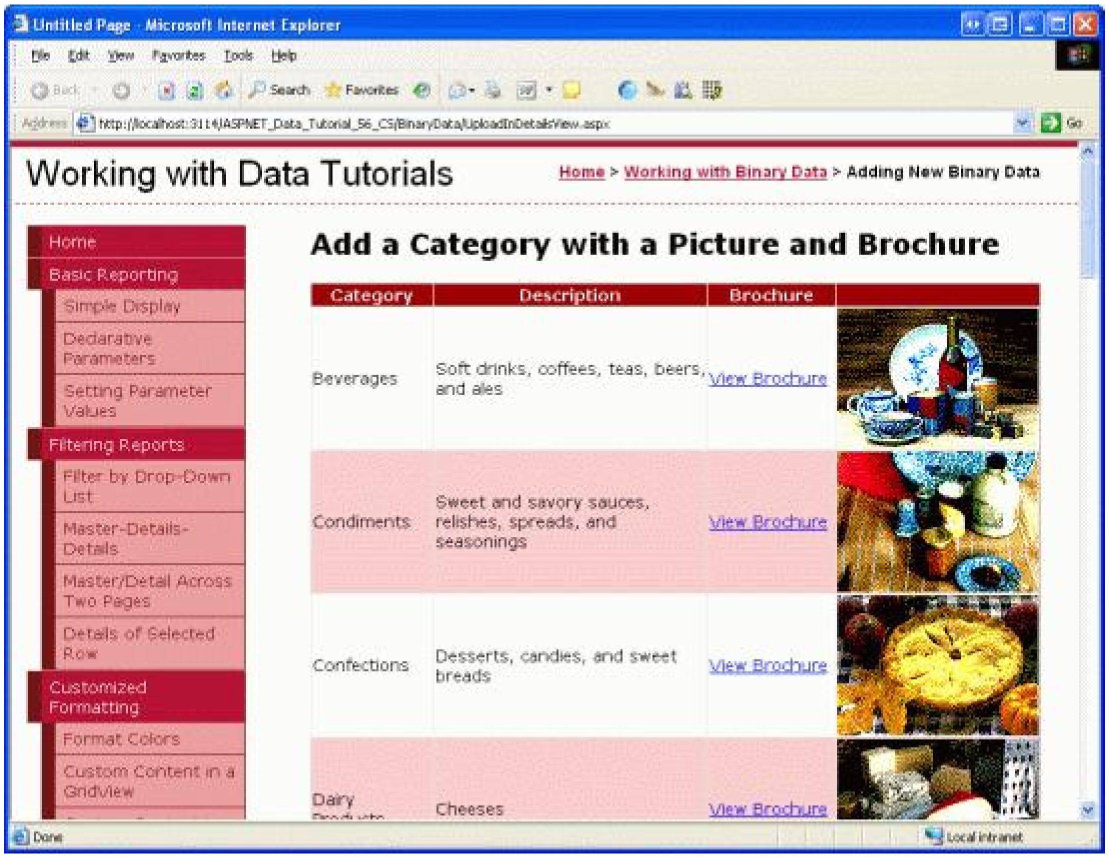
Start by opening the `DisplayOrDownload.aspx` page from the `BinaryData` folder. Go to the Source view and copy the GridView and ObjectDataSource's declarative syntax, pasting it within the `<asp:Content>` element in `UploadInDetailsView.aspx`. Also, don't forget to copy over the `GenerateBrochureLink` method from the code-behind class of `DisplayOrDownload.aspx` to `UploadInDetailsView.aspx`.



```
<%@ Page Language="C#" MasterPageFile="~/Site.master" AutoEventWireup="true" CodeFile="Di
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" Runat="Server">
    <h2>
        Display / Download Binary Data</h2>
    <p>
        <asp:GridView ID="Categories" runat="server" AutoGenerateColumns="False" DataKeyN
            DataSourceID="CategoriesDataSource" EnableViewState="False">
            <Columns>
                <asp:BoundField DataField="CategoryName" HeaderText="Category" SortExpres
                <asp:BoundField DataField="Description" HeaderText="Description" SortExpr
                <asp:TemplateField HeaderText="Brochure">
                    <ItemTemplate>
                        <%# GenerateBrochureLink(Eval("BrochurePath")) %>
                    </ItemTemplate>
                </asp:TemplateField>
                <asp:ImageField DataImageUrlField="CategoryID" DataImageUrlFormatString="
                </asp:ImageField>
            </Columns>
        </asp:GridView>
        <asp:ObjectDataSource ID="CategoriesDataSource" runat="server" OldValuesParameter
            SelectMethod="GetCategories" TypeName="CategoriesBLL"></asp:ObjectDataSource>
    </p>
</asp:Content>
```

Figure 3: Copy and Paste the Declarative Syntax from `DisplayOrDownload.aspx` to `UploadInDetailsView.aspx`

After copying the declarative syntax and `GenerateBrochureLink` method over to the `UploadInDetailsView.aspx` page, view the page through a browser to ensure that everything was copied over correctly. You should see a GridView listing the eight categories that includes a link to download the brochure as well as the category's picture.



Working with Data Tutorials

Home > Working with Binary Data > Adding New Binary Data

Add a Category with a Picture and Brochure

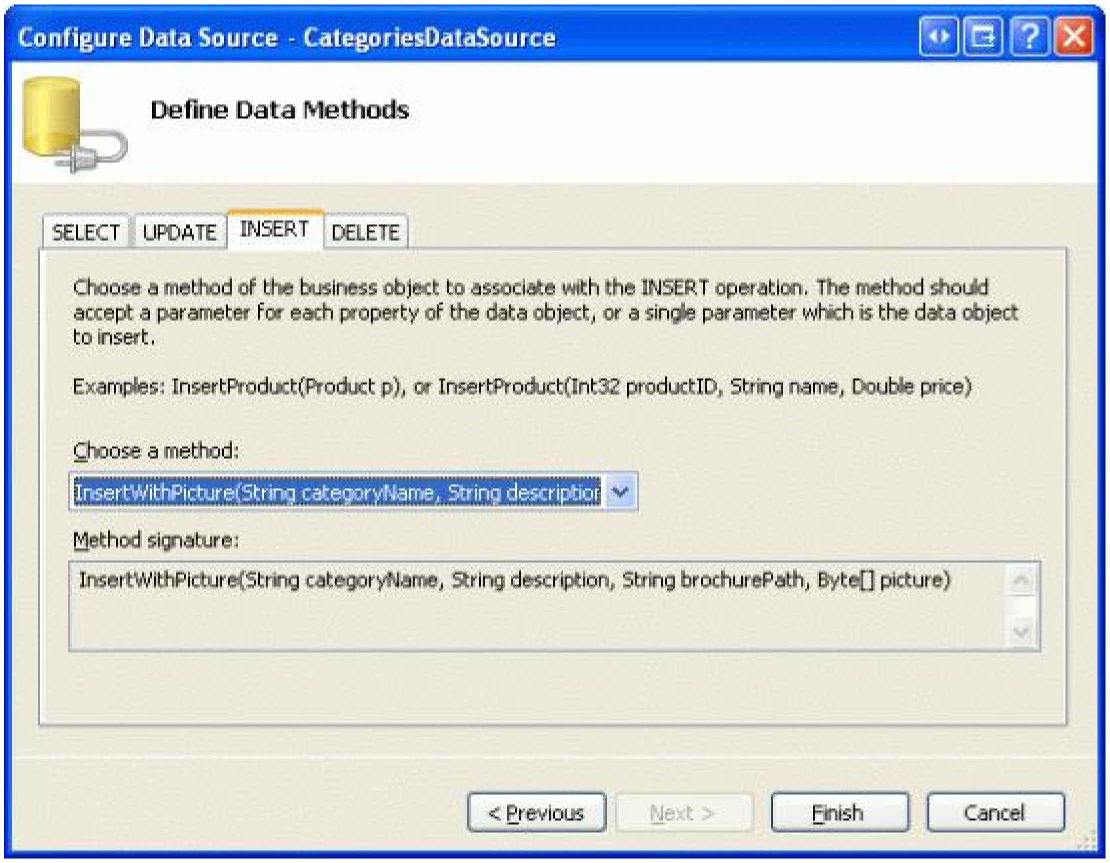
Category	Description	Brochure	
Beverages	Soft drinks, coffees, teas, beers, and ales	View Brochure	
Condiments	Sweet and savory sauces, relishes, spreads, and seasonings	View Brochure	
Confections	Desserts, candies, and sweet breads	View Brochure	
Dairy Produ	Cheeses	View Brochure	

Figure 4: You Should Now See Each Category Along with Its Binary Data

Step 4: Configuring the `CategoriesDataSource` to Support Inserting

The `CategoriesDataSource` ObjectDataSource used by the `Categories` GridView currently does not provide the ability to insert data. In order to support inserting through this data source control, we need to map its `Insert` method to a method in its underlying object, `CategoriesBLL`. In particular, we want to map it to the `CategoriesBLL` method we added back in Step 2, `InsertWithPicture`.

Start by clicking the Configure Data Source link from the ObjectDataSource's smart tag. The first screen shows the object the data source is configured to work with, `CategoriesBLL`. Leave this setting as-is and click Next to advance to the Define Data Methods screen. Move to the INSERT tab and pick the `InsertWithPicture` method from the drop-down list. Click Finish to complete the wizard.



Choose a method of the business object to associate with the INSERT operation. The method should accept a parameter for each property of the data object, or a single parameter which is the data object to insert.

Examples: InsertProduct(Product p), or InsertProduct(Int32 productID, String name, Double price)

Choose a method:

InsertWithPicture(String categoryName, String descriptior

Method signature:

InsertWithPicture(String categoryName, String description, String brochurePath, Byte[] picture)

Figure 5: Configure the ObjectDataSource to use the `InsertWithPicture` Method

Note: Upon completing the wizard, Visual Studio may ask if you want to "Refresh Fields and Keys," which will regenerate the data Web controls fields. Choose No, because choosing Yes will overwrite any field customizations you may have made.

After completing the wizard, the ObjectDataSource will now include a value for its `InsertMethod` property as well as `InsertParameters` for the four category columns, as the following declarative markup illustrates:

```
<asp:ObjectDataSource ID="CategoriesDataSource" runat="server"
    OldValuesParameterFormatString="original_{0}" SelectMethod="GetCategories"
    TypeName="CategoriesBLL" InsertMethod="InsertWithPicture">
    <InsertParameters>
        <asp:Parameter Name="categoryName" Type="String" />
        <asp:Parameter Name="description" Type="String" />
        <asp:Parameter Name="brochurePath" Type="String" />
        <asp:Parameter Name="picture" Type="Object" />
    </InsertParameters>
</asp:ObjectDataSource>
```

Step 5: Creating the Inserting Interface

As first covered in the An Overview of Inserting, Updating, and Deleting Data, the DetailsView control provides a built-in inserting interface that can be utilized when working with a data source control that supports inserting. Let's add a DetailsView control to this page above the GridView that will permanently render its inserting interface, allowing a user to quickly add a new category. Upon adding a new category in the DetailsView, the

GridView beneath it will automatically refresh and display the new category.

Start by dragging a DetailsView from the Toolbox onto the Designer above the GridView, setting its `ID` property to `NewCategory` and clearing out the `Height` and `Width` property values. From the DetailsView's smart tag, bind it to the existing `CategoriesDataSource` and then check the "Enable Inserting" checkbox.

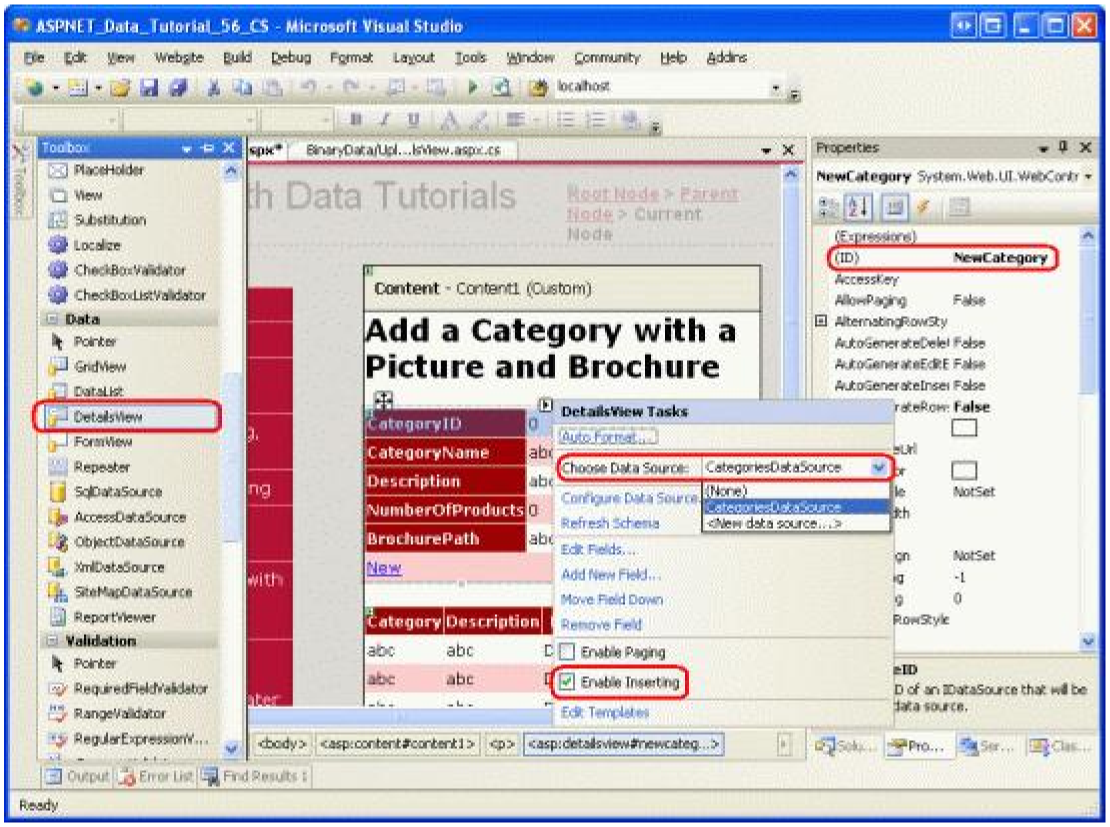

ASPNET_Data_Tutorial_56_CS - Microsoft Visual Studio
DetailsView Tasks
Choose Data Source: CategoriesDataSource
Enable Inserting
(ID) NewCategory

Figure 6: Bind the DetailsView to the `CategoriesDataSource` and Enable Inserting

To permanently render the DetailsView in its inserting interface, set its `DefaultMode` property to `Insert`.

Note that the DetailsView has five BoundFields – `CategoryID`, `CategoryName`, `Description`, `NumberOfProducts`, and `BrochurePath` – although the `CategoryID` BoundField is not rendered in the inserting interface because its `InsertVisible` property is set to `false`. These BoundFields exists because they are the columns returned by the `GetCategories()` method, which is what the ObjectDataSource invokes to retrieve its data. For inserting, however, we don't want to let the user specify a value for `NumberOfProducts`. Moreover, we need to allow them to upload a picture for the new category as well as upload a PDF for the brochure.

Remove the `NumberOfProducts` BoundField from the DetailsView altogether and then update the `HeaderText` properties of the `CategoryName` and `BrochurePath` BoundFields to "Category" and "Brochure", respectively. Next, convert the `BrochurePath` BoundField into a TemplateField and add a new TemplateField for the picture, giving this new TemplateField a `HeaderText` value of "Picture". Move the `Picture` TemplateField so that it is between the `BrochurePath` TemplateField and CommandField.

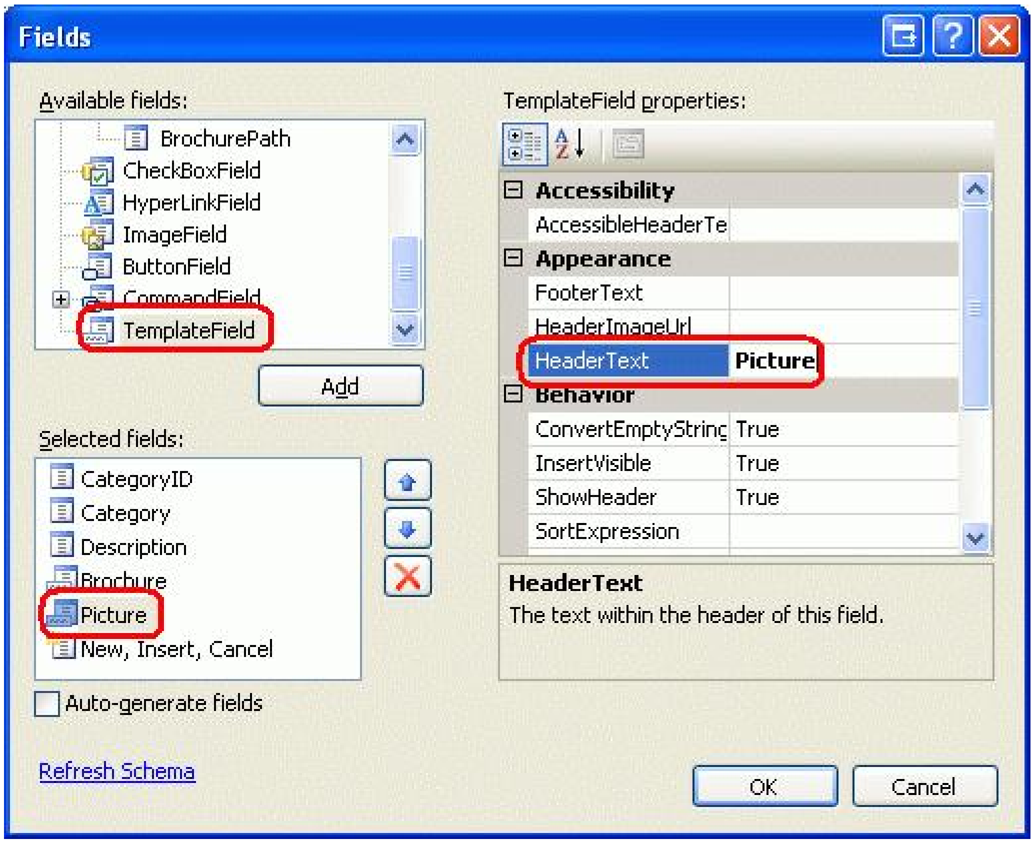


Figure 7: Bind the DetailsView to the `CategoriesDataSource` and Enable Inserting

If you converted the `BrochurePath` BoundField into a TemplateField through the Edit Fields dialog box, the TemplateField includes an `ItemTemplate`, `EditItemTemplate`, and `InsertItemTemplate`. Only the `InsertItemTemplate` is needed, however, so feel free to remove the other two templates. At this point your DetailsView's declarative syntax should look like the following:

```
<asp:DetailsView ID="NewCategory" runat="server" AutoGenerateRows="False"
    DataKeyNames="CategoryID" DataSourceID="CategoriesDataSource"
    DefaultMode="Insert">
    <Fields>
        <asp:BoundField DataField="CategoryID" HeaderText="CategoryID"
            InsertVisible="False" ReadOnly="True"
            SortExpression="CategoryID" />
        <asp:BoundField DataField="CategoryName" HeaderText="Category"
            SortExpression="CategoryName" />
        <asp:BoundField DataField="Description" HeaderText="Description"
            SortExpression="Description" />
        <asp:TemplateField HeaderText="Brochure" SortExpression="BrochurePath">
            <InsertItemTemplate>
                <asp:TextBox ID="TextBox1" runat="server"
                    Text='<%# Bind("BrochurePath") %>'></asp:TextBox>
            </InsertItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Picture"></asp:TemplateField>
        <asp:CommandField ShowInsertButton="True" />
    </Fields>
</asp:DetailsView>
```

Adding FileUpload Controls for the Brochure and Picture Fields

Presently, the `BrochurePath` TemplateField's `InsertItemTemplate` contains a TextBox, while the `Picture` TemplateField does not contain any templates. We need to update these two TemplateField's `InsertItemTemplate`s to use FileUpload controls.

From the DetailsView's smart tag, choose the "Edit Templates" option and then select the `BrochurePath` TemplateField's `InsertItemTemplate` from the drop-down list. Remove the TextBox and then drag a FileUpload control from the Toolbox into the template. Set the FileUpload control's `ID` to `BrochureUpload`. Similarly, add a FileUpload control to the `Picture` TemplateField's `InsertItemTemplate`. Set this FileUpload control's `ID` to `PictureUpload`.

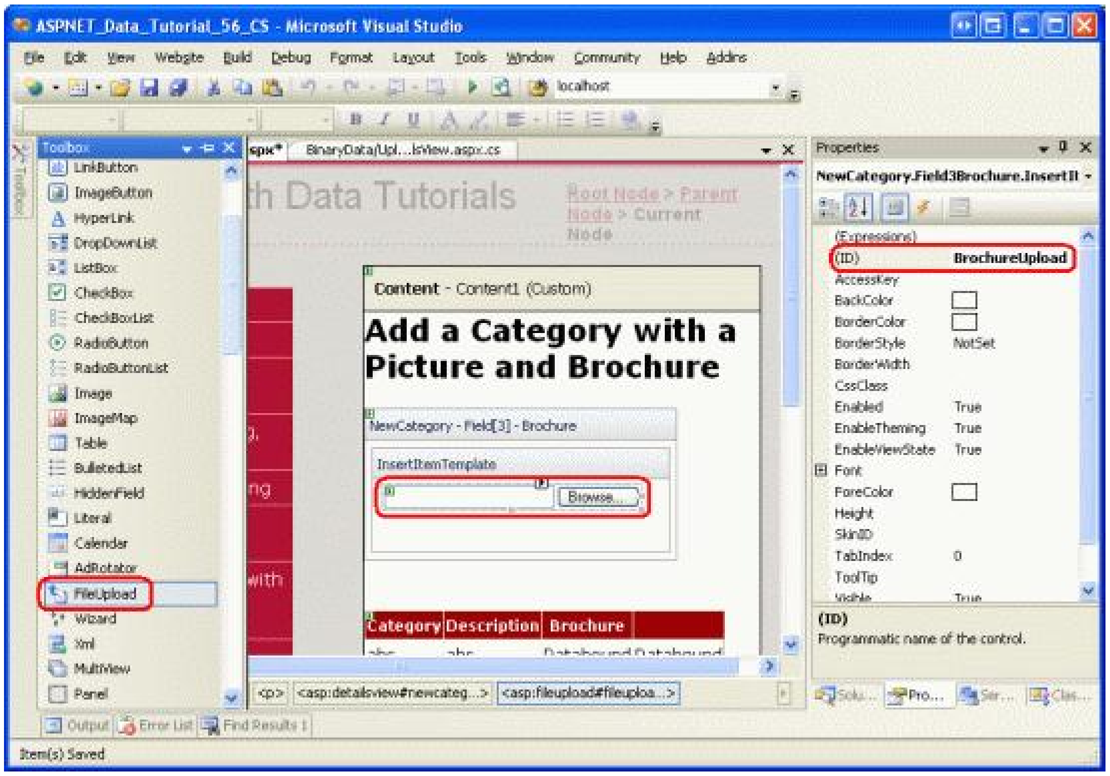

ASPNET_Data_Tutorial_56_CS - Microsoft Visual Studio
Toolbox
FileUpload
Add a Category with a Picture and Brochure
NewCategory - Field[3] - Brochure
InsertItemTemplate
Browse...
Properties
(ID) BrochureUpload

Figure 8: Add a FileUpload Control to the `InsertItemTemplate`

After making these additions, the two TemplateField's declarative syntax will be:

```
<asp:TemplateField HeaderText="Brochure" SortExpression="BrochurePath">
    <InsertItemTemplate>
        <asp:FileUpload ID="BrochureUpload" runat="server" />
    </InsertItemTemplate>
</asp:TemplateField>
<asp:TemplateField HeaderText="Picture">
    <InsertItemTemplate>
        <asp:FileUpload ID="PictureUpload" runat="server" />
    </InsertItemTemplate>
</asp:TemplateField>
```

When a user adds a new category, we want to ensure that the brochure and picture are of the correct file type. For the brochure, the user must supply a PDF. For the picture, we need the user to upload an image file, but do we allow *any* image file or only image files of a particular type, such as GIFs or JPGs? In order to allow for different file types, we'd need to extend the `Categories` schema to include a column that captures the file type so that this type can be sent to the client through `Response.ContentType` in `DisplayCategoryPicture.aspx`. Since we don't have such a column, it would be prudent to restrict users to only providing a specific image file type. The `Categories` table's existing images are bitmaps, but JPGs are a more appropriate file format for images served over the web.

If a user uploads an incorrect file type, we need to cancel the insert and display a message indicating the problem. Add a Label Web control beneath the DetailsView. Set its `ID` property to `UploadWarning`, clear out its `Text` property, set the `CssClass` property to "Warning", and the `Visible` and `EnableViewState` properties to `false`. The `Warning` CSS class is defined in `Styles.css` and renders the text in a large, red, italicized, bold font.

> **Note:** Ideally, the `CategoryName` and `Description` BoundFields would be converted to TemplateFields and their inserting interfaces customized. The `Description` inserting interface, for example, would likely be better suited through a multi-line textbox. And since the `CategoryName` column does not accept `NULL` values, a RequiredFieldValidator should be added to ensure the user provides a value for the new category's name. These steps are left as an exercise to the reader. Refer back to Customizing the Data Modification Interface for an in-depth look at augmenting the data modification interfaces.

Step 6: Saving the Uploaded Brochure to the Web Server's File System

When the user enters the values for a new category and clicks the Insert button, a postback occurs and the inserting workflow unfolds. First, the DetailsView's `ItemInserting` event fires. Next, the ObjectDataSource's `Insert()` method is invoked, which results in a new record being added to the `Categories` table. After that, the DetailsView's `ItemInserted` event fires.

Before the ObjectDataSource's `Insert()` method is invoked, we must first ensure that the appropriate file types were uploaded by the user and then save the brochure PDF to the web server's file system. Create an event handler for the DetailsView's `ItemInserting` event and add the following code:

```
// Reference the FileUpload control
FileUpload BrochureUpload =
    (FileUpload)NewCategory.FindControl("BrochureUpload");

if (BrochureUpload.HasFile)
{
    // Make sure that a PDF has been uploaded
    if (string.Compare(System.IO.Path.GetExtension
        (BrochureUpload.FileName), ".pdf", true) != 0)
    {
        UploadWarning.Text =
            "Only PDF documents may be used for a category's brochure.";
        UploadWarning.Visible = true;
        e.Cancel = true;
        return;
    }
}
```

The event handler starts by referencing the `BrochureUpload` FileUpload control from the DetailsView's templates. Then, if a brochure has been uploaded, the uploaded file's extension is examined. If the extension is not ".PDF", then a warning is displayed, the insert is cancelled, and the execution of the event handler ends.

Note: Relying on the uploaded file's extension is not a sure-fire technique for ensuring that the uploaded file is a PDF document. The user could have a valid PDF document with the extension `.Brochure`, or could have taken a non-PDF document and given it a `.pdf` extension. The file's binary content would need to be programmatically examined in order to more conclusively verify the file type. Such thorough approaches, though, are often overkill; checking the extension is sufficient for most scenarios.

As discussed in the Uploading Files tutorial, care must be taken when saving files to the file system so that one user's upload does not overwrite another's. For this tutorial we will attempt to use the same name as the uploaded file. If there already exists a file in the `~/Brochures` directory with that same file name, however, we'll append a number at the end until a unique name is found. For example, if the user uploads a brochure file named `Meats.pdf`, but there is already a file named `Meats.pdf` in the `~/Brochures` folder, we'll change the saved file name to `Meats-1.pdf`. If that exists, we'll try `Meats-2.pdf`, and so on, until a unique file name is found.

The following code uses the `File.Exists(`*`path`*`)` method to determine if a file already exists with the specified file name. If so, it continues to try new file names for the brochure until no conflict is found.

```
const string BrochureDirectory = "~/Brochures/";
string brochurePath = BrochureDirectory + BrochureUpload.FileName;
string fileNameWithoutExtension =
    System.IO.Path.GetFileNameWithoutExtension(BrochureUpload.FileName);

int iteration = 1;

while (System.IO.File.Exists(Server.MapPath(brochurePath)))
{
    brochurePath = string.Concat(BrochureDirectory,
        fileNameWithoutExtension, "-", iteration, ".pdf");
    iteration++;
}
```

Once a valid file name has been found, the file needs to be saved to the file system and the ObjectDataSource's `brochurePath InsertParameter` value needs to be updated so that this file name is written to the database. As we saw back in the *Uploading Files* tutorial, the file can be saved using the FileUpload control's `SaveAs(`*`path`*`)` method. To update the ObjectDataSource's `brochurePath` parameter, use the `e.Values` collection.

```
// Save the file to disk and set the value of the brochurePath parameter
BrochureUpload.SaveAs(Server.MapPath(brochurePath));
e.Values["brochurePath"] = brochurePath;
```

Step 7: Saving the Uploaded Picture to the Database

To store the uploaded picture in the new `Categories` record, we need to assign the uploaded binary content to the ObjectDataSource's `picture` parameter in the DetailsView's `ItemInserting` event. Before we make this assignment, however, we need to first make sure that the uploaded picture is a JPG and not some other image type. As in Step 6, let's use the uploaded picture's file extension to ascertain its type.

While the `Categories` table allows `NULL` values for the `Picture` column, all categories currently have a picture. Let's force the user to provide a picture when adding a new category through this page. The following code checks to ensure that a picture has been uploaded and that it has an appropriate extension.

```
// Reference the FileUpload controls
FileUpload PictureUpload = (FileUpload)NewCategory.FindControl("PictureUpload");
if (PictureUpload.HasFile)
{
    // Make sure that a JPG has been uploaded
    if (string.Compare(System.IO.Path.GetExtension(PictureUpload.FileName),
            ".jpg", true) != 0 &&
        string.Compare(System.IO.Path.GetExtension(PictureUpload.FileName),
            ".jpeg", true) != 0)
    {
        UploadWarning.Text =
            "Only JPG documents may be used for a category's picture.";
        UploadWarning.Visible = true;
        e.Cancel = true;
        return;
    }
}
else
{
    // No picture uploaded!
    UploadWarning.Text =
        "You must provide a picture for the new category.";
    UploadWarning.Visible = true;
    e.Cancel = true;
    return;
}
```

This code should be placed *before* the code from Step 6 so that if there is a problem with the picture upload, the event handler will terminate before the brochure file is saved to the file system.

Assuming that an appropriate file has been uploaded, assign the uploaded binary content to the picture parameter's value with the following line of code:

```
    // Set the value of the picture parameter
    e.Values["picture"] = PictureUpload.FileBytes;
```

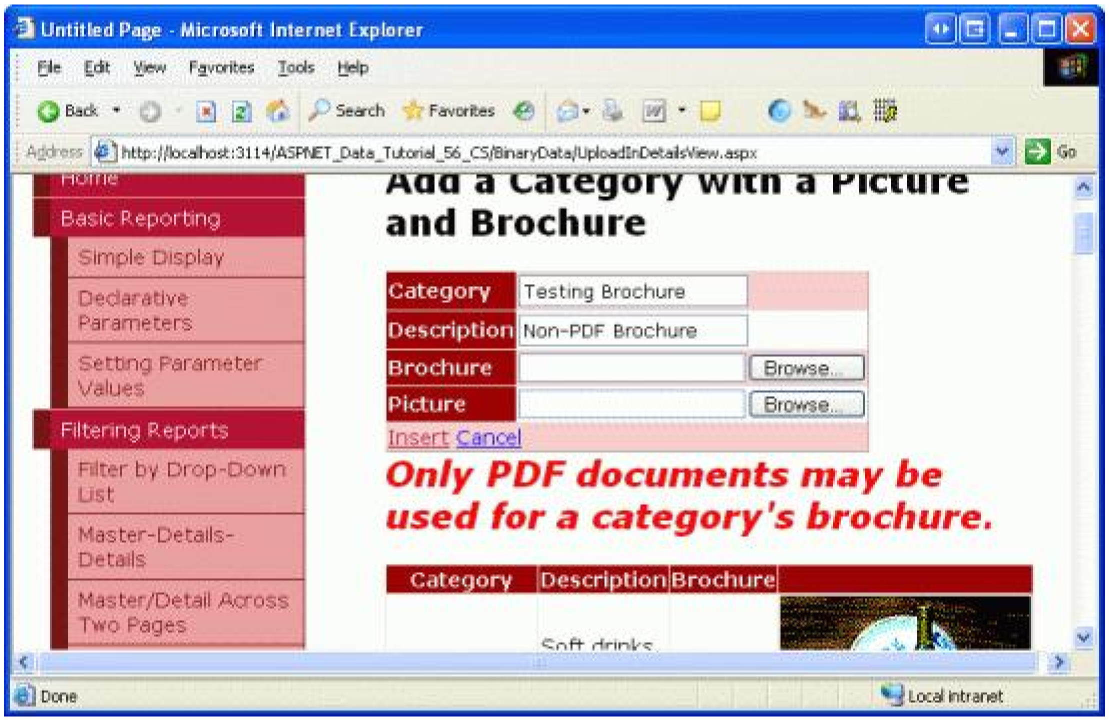
The Complete `ItemInserting` Event Handler

For completeness, here is the `ItemInserting` event handler in its entirety:

```
    protected void NewCategory_ItemInserting(object sender, DetailsViewInsertEventArgs e)
    {
        // Reference the FileUpload controls
        FileUpload PictureUpload = (FileUpload)NewCategory.FindControl("PictureUpload");
        if (PictureUpload.HasFile)
        {
            // Make sure that a JPG has been uploaded
            if (string.Compare(System.IO.Path.GetExtension(PictureUpload.FileName),
                    ".jpg", true) != 0 &&
                string.Compare(System.IO.Path.GetExtension(PictureUpload.FileName),
                    ".jpeg", true) != 0)
            {
                UploadWarning.Text =
                    "Only JPG documents may be used for a category's picture.";
                UploadWarning.Visible = true;
                e.Cancel = true;
                return;
            }
        }
        else
        {
            // No picture uploaded!
            UploadWarning.Text =
                "You must provide a picture for the new category.";
            UploadWarning.Visible = true;
            e.Cancel = true;
            return;
        }

        // Set the value of the picture parameter
        e.Values["picture"] = PictureUpload.FileBytes;


        // Reference the FileUpload controls
        FileUpload BrochureUpload =
            (FileUpload)NewCategory.FindControl("BrochureUpload");
        if (BrochureUpload.HasFile)
        {
            // Make sure that a PDF has been uploaded
            if (string.Compare(System.IO.Path.GetExtension(BrochureUpload.FileName),
                ".pdf", true) != 0)
            {
                UploadWarning.Text =
                    "Only PDF documents may be used for a category's brochure.";
                UploadWarning.Visible = true;
                e.Cancel = true;
                return;
            }

            const string BrochureDirectory = "~/Brochures/";
            string brochurePath = BrochureDirectory + BrochureUpload.FileName;
            string fileNameWithoutExtension =
                System.IO.Path.GetFileNameWithoutExtension(BrochureUpload.FileName);

            int iteration = 1;

            while (System.IO.File.Exists(Server.MapPath(brochurePath)))
            {
                brochurePath = string.Concat(BrochureDirectory, fileNameWithoutExtension,
                    "-", iteration, ".pdf");
                iteration++;
            }

            // Save the file to disk and set the value of the brochurePath parameter
            BrochureUpload.SaveAs(Server.MapPath(brochurePath));
            e.Values["brochurePath"] = brochurePath;
        }
    }
```

Step 8: Fixing the `DisplayCategoryPicture.aspx` Page

Let's take a moment to test out the inserting interface and `ItemInserting` event handler that was created over the last few steps. Visit the `UploadInDetailsView.aspx` page through a browser and attempt to add a category, but omit the picture, or specify a non-JPG picture or a non-PDF brochure. In any of these cases, an error message will be displayed and the insert workflow cancelled.


Untitled Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:3114/ASPNET_Data_Tutorial_56_CS/BinaryData/UploadInDetailsView.aspx
Go
Home
Basic Reporting
Simple Display
Declarative Parameters
Setting Parameter Values
Filtering Reports
Filter by Drop-Down List
Master-Details-Details
Master/Detail Across Two Pages
Done
Local intranet

Add a Category with a Picture and Brochure

Category	Testing Brochure	
Description	Non-PDF Brochure	
Brochure		Browse...
Picture		Browse...

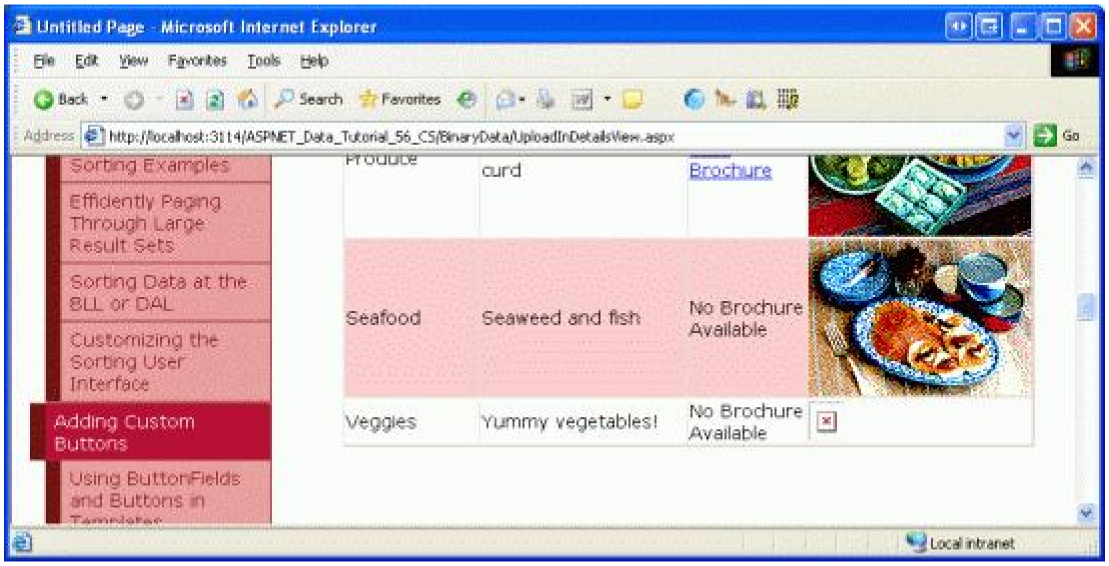
Insert Cancel

Only PDF documents may be used for a category's brochure.

Category	Description	Brochure	
	Soft drinks		

Figure 9: A Warning Message is Displayed If an Invalid File Type is Uploaded

Once you have verified that the page requires a picture to be uploaded and won't accept non-PDF or non-JPG files, add a new category with a valid JPG picture, leaving the Brochure field empty. After clicking the Insert button, the page will postback and a new record will be added to the `Categories` table with the uploaded image's binary contents stored directly in the database. The GridView is updated and shows a row for the newly added category, but, as Figure 10 shows, the new category's picture is not rendered correctly.


Untitled Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:3114/ASPNET_Data_Tutorial_56_CS/BinaryData/UploadInDetailsView.aspx
Go
Sorting Examples
Efficiently Paging Through Large Result Sets
Sorting Data at the BLL or DAL
Customizing the Sorting User Interface
Adding Custom Buttons
Using ButtonFields and Buttons in Templates
Local intranet

Produce	curd	Brochure	
Seafood	Seaweed and fish	No Brochure Available	
Veggies	Yummy vegetables!	No Brochure Available	

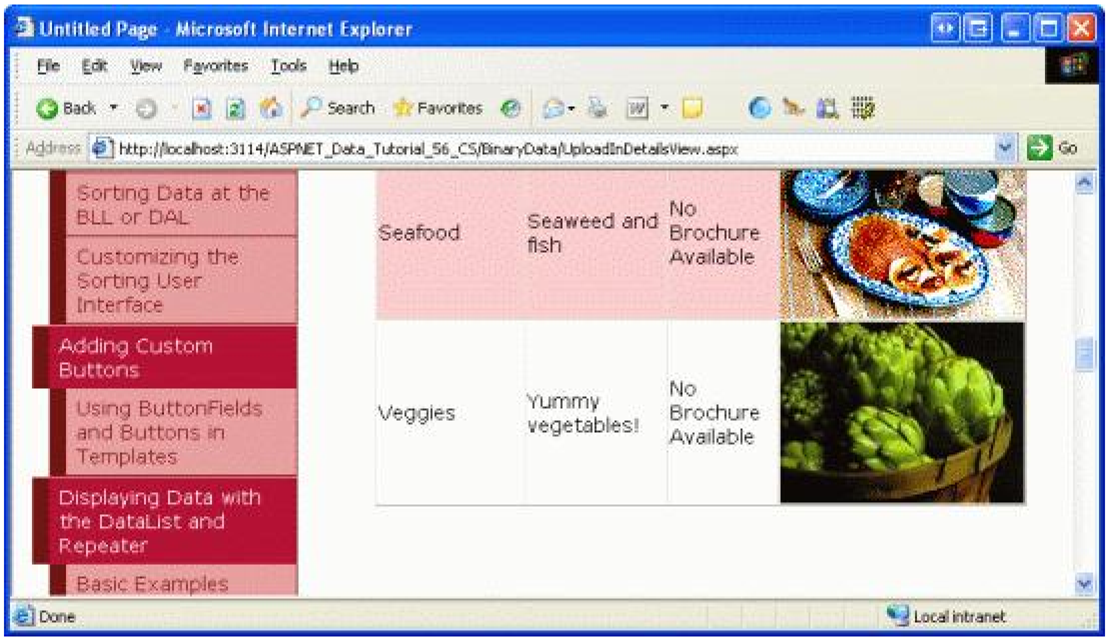
Figure 10: The New Category's Picture is not Displayed

The reason the new picture is not displayed is because the `DisplayCategoryPicture.aspx` page that returns a specified category's picture is configured to process bitmaps that have an OLE header. This 78 byte header is stripped from the `Picture` column's binary contents before they are sent back to the client. But the JPG file we just uploaded for the new category does not have this OLE header; therefore, valid, necessary bytes are being removed from the image's binary data.

Since there are now both bitmaps with OLE headers and JPGs in the `Categories` table, we need to update `DisplayCategoryPicture.aspx` so that it does the OLE header stripping for the original eight categories and bypasses this stripping for the newer category records. In our next tutorial we'll examine how to update an existing record's image, and we'll update all of the old category pictures so that they are JPGs. For now, though, use the following code in `DisplayCategoryPicture.aspx` to strip the OLE headers only for those original eight categories:

```
protected void Page_Load(object sender, EventArgs e)
{
    int categoryID = Convert.ToInt32(Request.QueryString["CategoryID"]);

    // Get information about the specified category
    CategoriesBLL categoryAPI = new CategoriesBLL();
    Northwind.CategoriesDataTable categories =
        categoryAPI.GetCategoryWithBinaryDataByCategoryID(categoryID);
    Northwind.CategoriesRow category = categories[0];

    if (categoryID <= 8)
    {
        // For older categories, we must strip the OLE header... images are bitmaps

        // Output HTTP headers providing information about the binary data
        Response.ContentType = "image/bmp";

        // Output the binary data
        // But first we need to strip out the OLE header
        const int OleHeaderLength = 78;
        int strippedImageLength = category.Picture.Length - OleHeaderLength;
        byte[] strippedImageData = new byte[strippedImageLength];
        Array.Copy(category.Picture, OleHeaderLength, strippedImageData,
            0, strippedImageLength);

        Response.BinaryWrite(strippedImageData);
    }
    else
    {
        // For new categories, images are JPGs...

        // Output HTTP headers providing information about the binary data
        Response.ContentType = "image/jpeg";

        // Output the binary data
        Response.BinaryWrite(category.Picture);
    }
}
```

With this change, the JPG image is now rendered correctly in the GridView.


Untitled Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:3114/ASPNET_Data_Tutorial_56_CS/BinaryData/UploadInDetailsView.aspx
Go
Sorting Data at the BLL or DAL
Customizing the Sorting User Interface
Adding Custom Buttons
Using ButtonFields and Buttons in Templates
Displaying Data with the DataList and Repeater
Basic Examples
Done
Local intranet

Seafood	Seaweed and fish	No Brochure Available	
Veggies	Yummy vegetables!	No Brochure Available	

Figure 11: The JPG Images for New Categories are Correctly Rendered

Step 9: Deleting the Brochure in the Face of an Exception

One of the challenges of storing binary data on the web server's file system is that it introduces a disconnect between the data model and its binary data. Therefore, whenever a record is deleted, the corresponding binary data on the file system must also be removed. This can come into play when inserting, as well. Consider the following scenario: a user adds a new category, specifying a valid picture and brochure. Upon clicking the Insert button, a postback occurs and the DetailsView's `ItemInserting` event fires, saving the brochure to the web server's file system. Next, the ObjectDataSource's `Insert()` method is invoked, which calls the `CategoriesBLL` class's `InsertWithPicture` method, which calls the `CategoriesTableAdapter`'s `InsertWithPicture` method.

Now, what happens if the database is offline, or if there is an error in the `INSERT` SQL statement? Clearly the INSERT will fail, so no new category row will be added to the database. But we still have the uploaded brochure file sitting on the web server's file system! This file needs to be deleted in the face of an exception during the inserting workflow.

As discussed previously in the Handling BLL- and DAL-Level Exceptions in an ASP.NET Page tutorial, when an exception is thrown from within the depths of the architecture it is bubbled up through the various layers. At the Presentation Layer, we can determine if an exception has occurred from the DetailsView's `ItemInserted` event. This event handler also provides the values of the ObjectDataSource's `InsertParameters`. Therefore, we can create an event handler for the `ItemInserted` event that checks if there was an exception and, if so, deletes the file specified by the ObjectDataSource's `brochurePath` parameter:

```
protected void NewCategory_ItemInserted
    (object sender, DetailsViewInsertedEventArgs e)
{
    if (e.Exception != null)
    {
        // Need to delete brochure file, if it exists
        if (e.Values["brochurePath"] != null)
            System.IO.File.Delete(Server.MapPath(
                e.Values["brochurePath"].ToString()));
    }
}
```

Summary

There are a number of steps that must be performed in order to provide a web-based interface for adding records that include binary data. If the binary data is being stored directly into the database, chances are you'll need to update the architecture, adding specific methods to handle the case where binary data is being inserted. Once the architecture has been updated, the next step is creating the inserting interface, which can be accomplished using a DetailsView that has been customized to include a FileUpload control for each binary data field. The uploaded data can then be saved to the web server's file system or assigned to a data source parameter in the DetailsView's `ItemInserting` event handler.

Saving binary data to the file system requires more planning than saving data directly into the database. A naming scheme must be chosen in order to avoid one user's upload overwriting another's. Also, extra steps must be taken to delete the uploaded file if the database insert fails.

We now have the ability to add new categories to the system with a brochure and picture, but we've yet to look at how to update an existing category's binary data or how to correctly remove the binary data for a deleted category. We'll explore these two topics in the next tutorial.

Happy Programming!

About the Author

Scott Mitchell, author of seven ASP/ASP.NET books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since 1998. Scott works as an independent consultant, trainer, and writer. His latest book is *Sams Teach Yourself ASP.NET 2.0 in 24 Hours*. He can be reached at mitchell@4GuysFromRolla.com. or via his blog, which can be found at http://ScottOnWriting.NET.

Special Thanks To…

This tutorial series was reviewed by many helpful reviewers. Lead reviewers for this tutorial were Dave Gardner, Teresa Murphy, and Bernadette Leigh. Interested in reviewing my upcoming MSDN articles? If so, drop me a line at mitchell@4GuysFromRolla.com.