Store Documents in on-line Briefcase (Part 2)

By Bipin Joshi
www.binaryintellect.net

In the Part 1 of this series we kicked off developing the briefcase application. We created database tables and stored procedures. We also created two classes namely Files and Folders to manipulate files and folders respectively. Continuing further we will now develop the user interface of the briefcase. The ASP.NET 2.0 TreeView control will be a natural choice for displaying folder hierarchy. The files will be displayed in a GridView control.

The briefcase user interface

Before you develop the user interface of the briefcase application it would be worthwhile to see how the application is going to look and function. The Figure 1 shows the user interface of the briefcase.

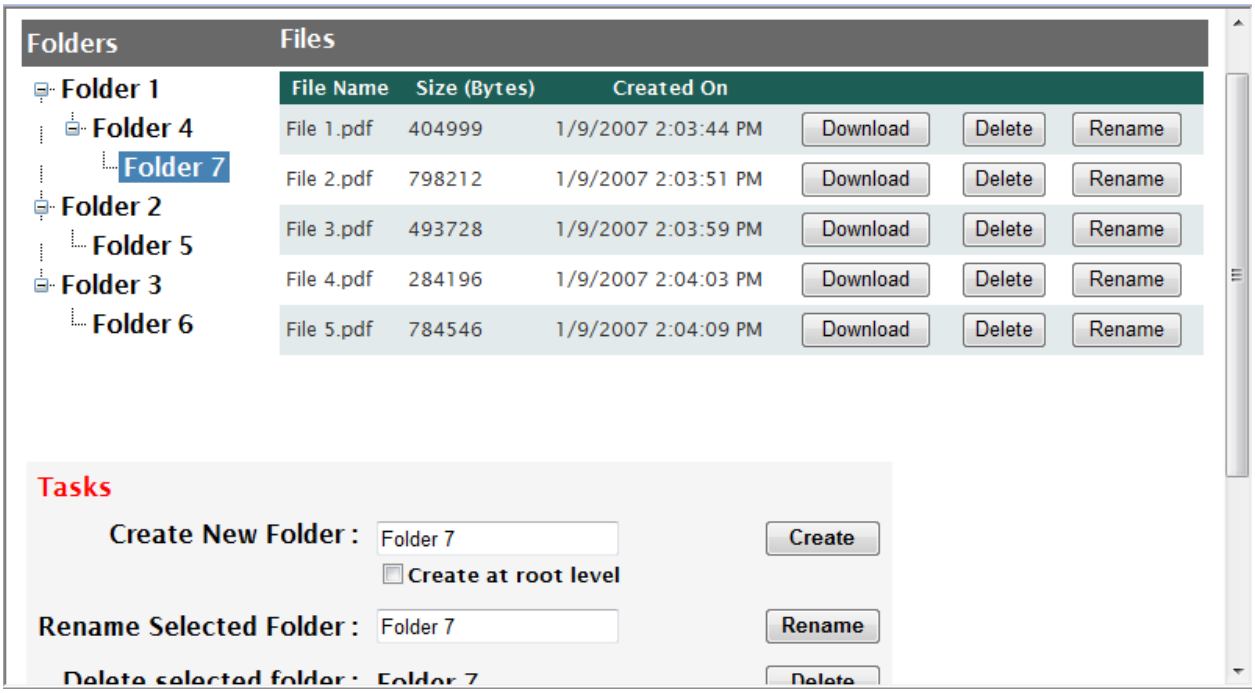

Figure 1: User interface of briefcase

On the left hand side of the page we have a TreeView control that displays list of folders. Upon selecting a folder all the files contained in it are shown on the right hand side in a GridView control. The files can then be downloaded, deleted or renamed with the help of appropriate buttons. Below the folder and file listing there is a task pane that allows us to perform tasks such as creating folder, deleting a folder and uploading files to a folder.

The base table

To begin designing the user interface, add an HTML table with four rows and two columns. Drag and drop a Label control in the topmost row. This Label is used to display any error messages while performing various operations. Drag and drop one Label control in both the cells of the second row. Set their Text property to “Folders” and “Files” respectively.

Designing the TreeView

Now drag and drop a Panel control in the first cell of the third row. This panel will house the TreeView. We use Panel control so that we can have horizontal and vertical scrollbars to this region. Set Height and Width property of the Panel to 300px and 150px respectively. Also, set ScrollBars property of the Panel to to Auto. Setting the ScrollBars property to Auto indicates that the scrollbars will be displayed only when the context overflows the boundaries of the Panel.

Next, drag and drop a TreeView control inside the Panel control. The TreeView control is available in the Navigation node of the toolbox (Figure 2).

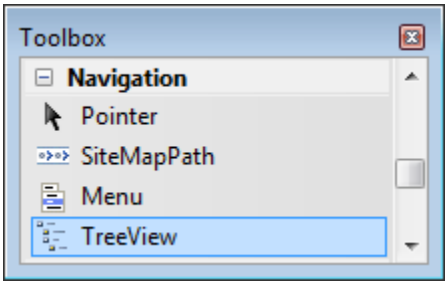

Figure 2: TreeView control on the toolbox

Set the Width property of TreeView to 100% and Font-Name property to Lucida Sans. Also, set ShowLines property to True. This will cause the TreeView to show dotted lines showing the nesting levels between the tree nodes. Set SelectedNodeStyle of the TreeView in such a way that the selected folder is shows with some highlighted color. The Figure 3 shows the complete markup of the Panel and TreeView.

```
<asp:Panel ID="Panel1" runat="server"
ScrollBars="Auto" Width="200px" Height="300px">
<asp:TreeView ID="TreeView1" runat="server"
Font-Bold="True" Width="100%" Font-Names="Lucida Sans"
ForeColor="Black" ShowLines="True">
<SelectedNodeStyle BackColor="SteelBlue" ForeColor="White" />
<NodeStyle ChildNodesPadding="5px" HorizontalPadding="3px" />
</asp:TreeView>
</asp:Panel>
```

Figure 3: Markup of Panel and TreeView

Designing the GridView

Now drag and drop a GridView control in the second cell of third row. Right click on it and select "Auto Format…" from the shortcut menu. In the Auto Format dialog (Figure 4) select some formatting scheme and click OK.

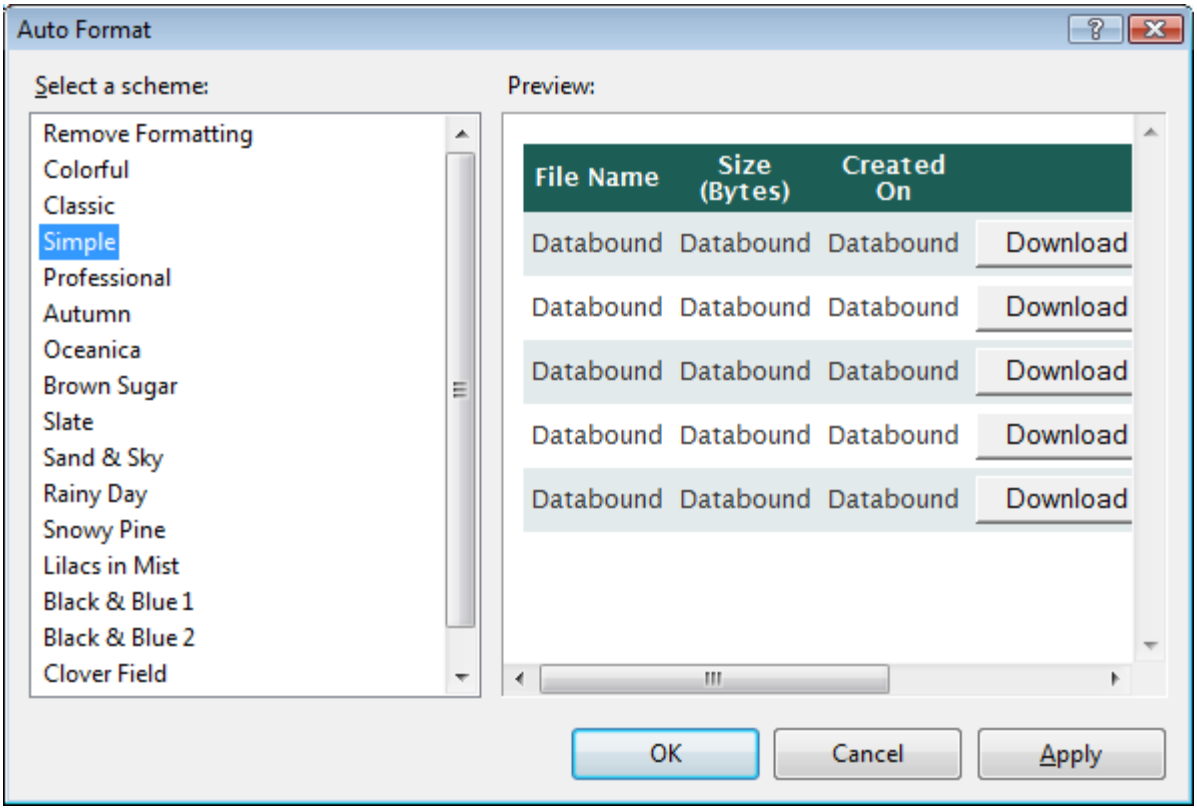

Figure 4: Auto format dialog of GridView

Set the Width property to 100% and Font-Name property to Lucida Sans. Now its time to add columns to the GridView. To do so, open the smart tags of the GridView and select Fields option. This will open Fields dialog as shown in Figure 5.

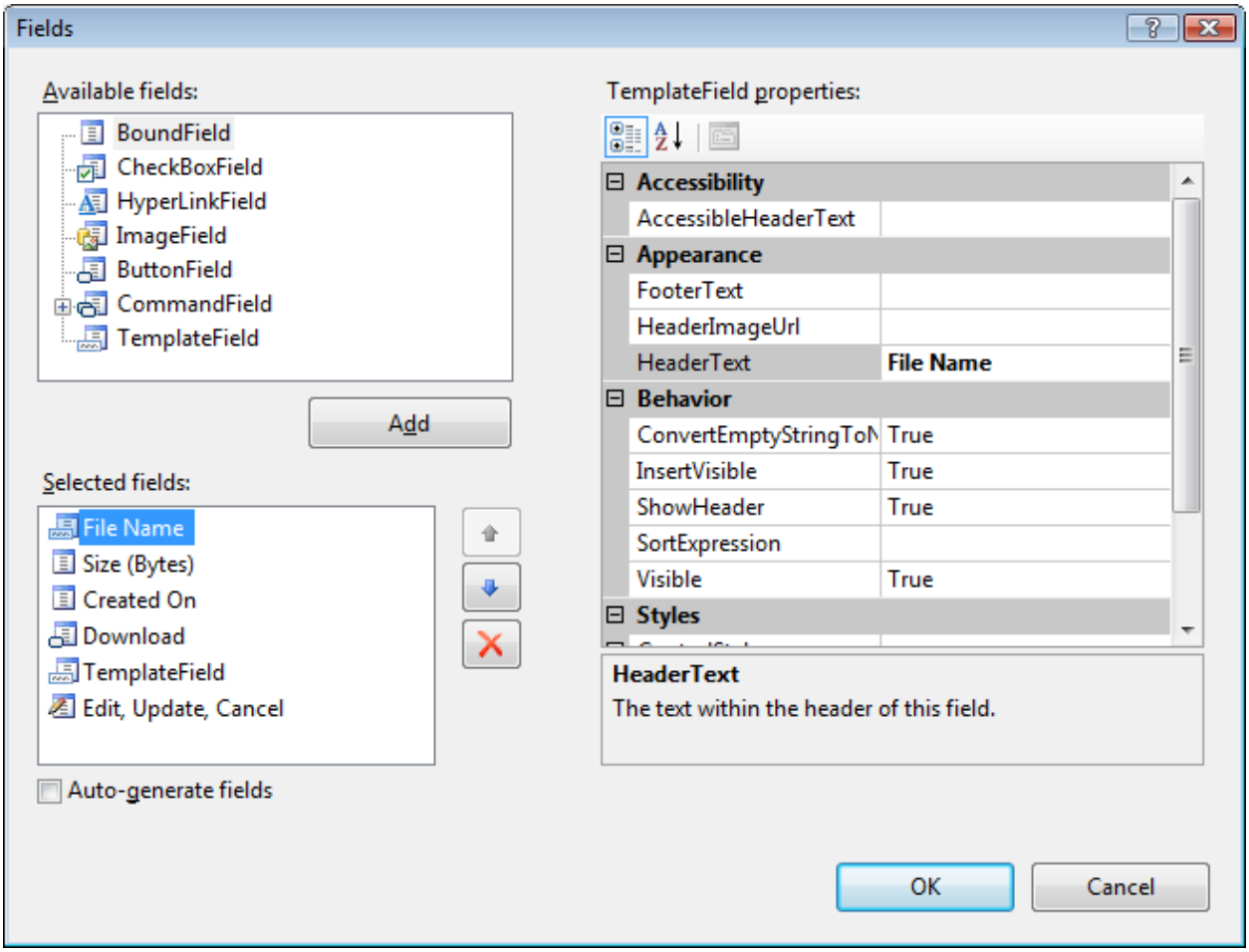

Figure 5: Fields dialog of GridView control

Using Fields dialog add two template fields, one button field, one command field and two bound fields. The columns of the GridView and their properties are listed in Figure 6.

Column for…	Column Type	HeaderText	DataField
Displaying file name	TemplateField	File Name	FileName
Displaying size of the file	BoundField	Size (Bytes)	FileSize
Displaying time stamp of the file	BoundField	Created On	DateCreated
Displaying download buttons	ButtonField	-	-
Displaying delete buttons	TemplateField	-	-
Displaying rename buttons	CommandField	-	-

Figure 6: Columns of the GridView

Set ButtonType property of ButtonField and CommandField to Button so that those columns will display push buttons instead of link buttons. Also, set the Text and CommandName properties of the ButtonField to Download. Further, set EditText and UpdateText property of CommandField

to Rename and Change respectively. Uncheck the Auto generate fields checkbox at the bottom of the dialog and close the dialog.

You might be wondering as to why we create File Name column as template field instead of bound field. The reason we created it as a template field is that doing so will allow us to attach validation controls to it in the edit mode. This way we can easily ensure that the user has entered a file name. Let's design the File Name template column now.

Right click on the GridView and select Edit Template menu option. Further select File Name template column. Doing so will open the GridView template designer as shown in Figure 7.

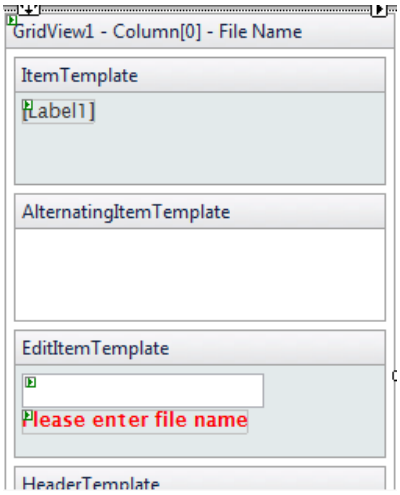

Figure 7: Designing File Name template

Drag and drop a Label control inside the ItemTemplate region. This Label will display the file name in read only mode of the GridView. Also, drag and drop a TextBox and a RequiredFieldValidator control inside the EditItemTemplate.

Now open the smart tag of the Label and choose "Edit Databindings…" option to open data bindings editor as shown in Figure 8.

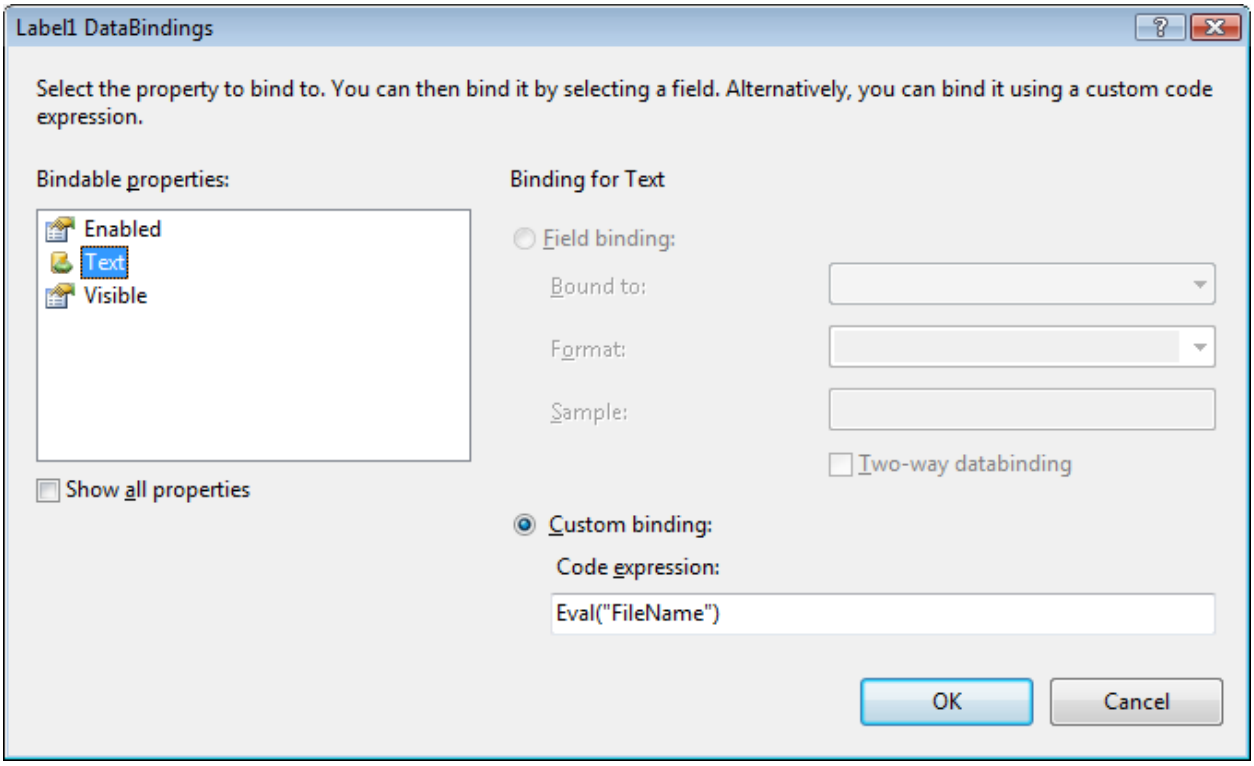

Figure 8: Data binding Label with FileName column

Select the Text property under "Bindable Properties" section and under "Custom binding" section enter Eval("FileName"). The Eval() function is one way data binding expression of ASP.NET and displays data from the data source into the control.

Similarly, open data bindings editor of the TextBox and bind its Text property with FileName column. Since the TextBox will be updating the file name we need to use Bind() function instead of Eval().

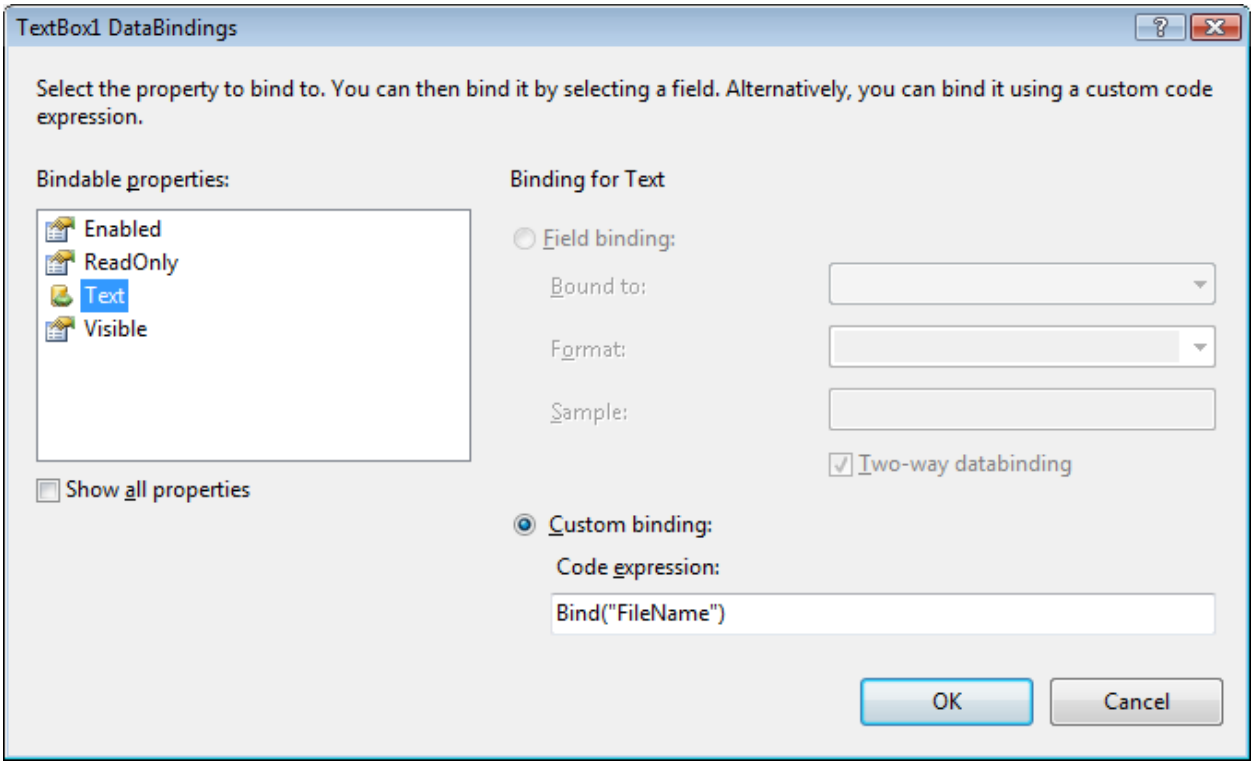

Figure 9 : Data binding TextBox with FileName column

Finally, select the RequiredFieldValidation control and set its ControlToValidate property to the ID of the TextBox. Also, set its ErrorMessage property to “Please enter file name”. This completes the design of FileName template column. Right click at the top of the designer and choose “End template designer”.

In order to design the delete template field, drag and drop a Button control into its ItemTemplate and set its Text property to Delete. Also, set its OnClientClick property to “return ConfirmDelete();”. We will be writing the ConfirmDelete() JavaScript function later that will ask for confirmation from the end user. If the user confirms the delete then the ConfirmDelete() function returns true otherwise it returns false. Accordingly the click event of the Delete button is raised or cancelled. Now, open the data bindings editor and bind CommandArguments property to Id column (Figure 10).

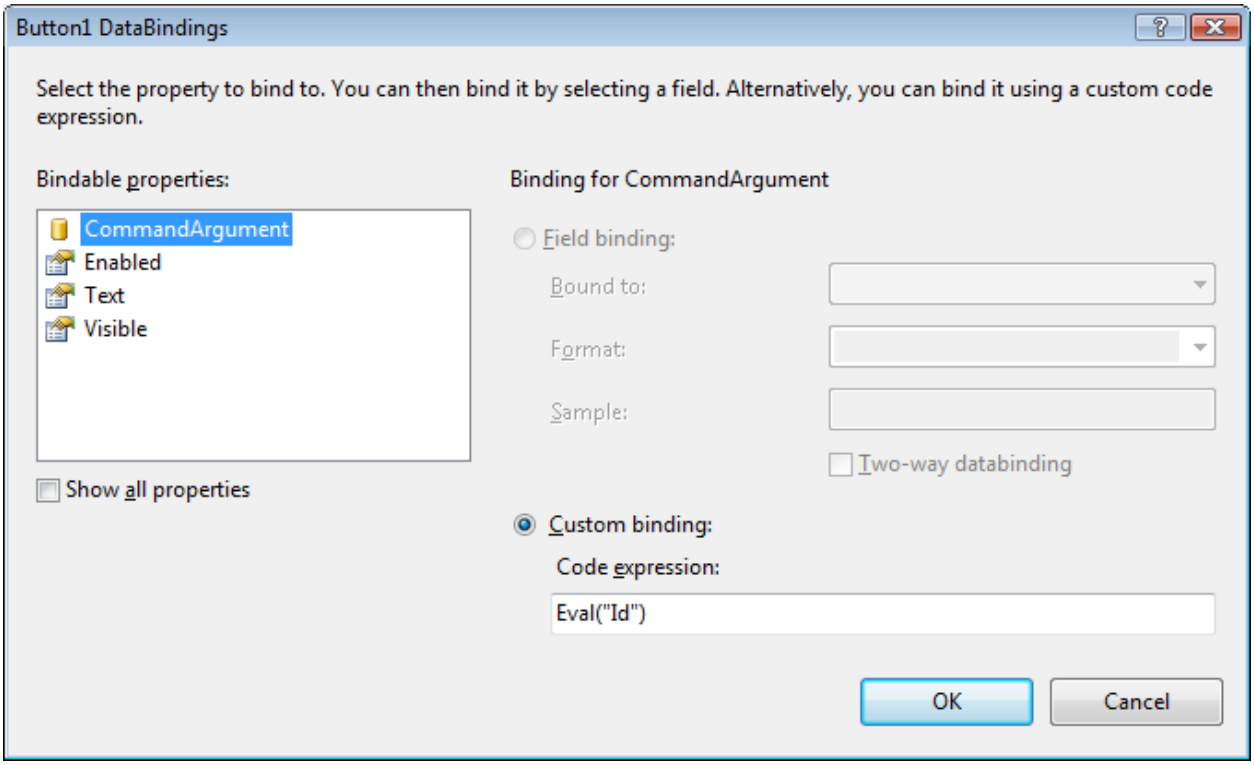

Figure 10: Data binding CommandArgument property with Id column

The CommandArgument is used later to delete a file with Id equal to the value of the command argument property.

Your GridView should now resemble Figure 11. Figure 12 shows the complete markup of the GridView.

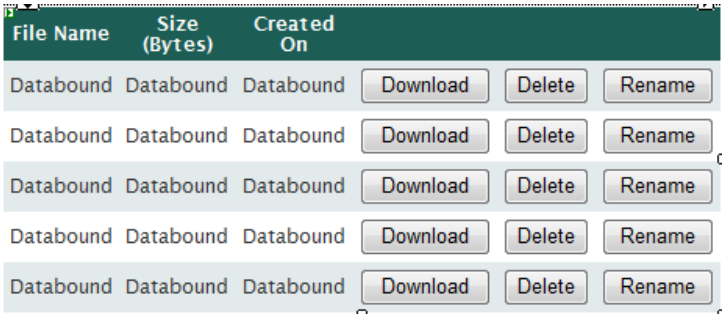

Figure 11: The GridView after completing the design

```
<asp:GridView ID="GridView1" runat="server"
AutoGenerateColumns="False" CellPadding="4"
DataKeyNames="Id" ForeColor="#333333" GridLines="None"
Width="100%" Font-Names="Lucida Sans" Font-Size="12px">
<FooterStyle BackColor="#1C5E55" Font-Bold="True"
```

```
ForeColor="White" />
<Columns>
<asp:TemplateField HeaderText="File Name">
<EditItemTemplate>
<asp:TextBox ID="TextBox1" runat="server"
Text='<%# Bind("FileName") %>'>
</asp:TextBox>
<asp:RequiredFieldValidator ID="RequiredFieldValidator1"
runat="server" ControlToValidate="TextBox1"
Display="Dynamic" ErrorMessage="Please enter file name" Font-
Bold="True">
</asp:RequiredFieldValidator>
</EditItemTemplate>
<ItemTemplate>
<asp:Label ID="Label1" runat="server" Text='<%# Eval("FileName")
%>'>
</asp:Label>
</ItemTemplate>
</asp:TemplateField>
<asp:BoundField DataField="FileSize" HeaderText="Size (Bytes)"
ReadOnly="True" />
<asp:BoundField DataField="DateCreated" HeaderText="Created On"
ReadOnly="True" />
<asp:ButtonField ButtonType="Button" CommandName="DownloadFile"
Text="Download" />
<asp:TemplateField ShowHeader="False">
<ItemTemplate>
<asp:Button ID="Button1" runat="server" CausesValidation="False"
CommandArgument='<%# Eval("Id") %>'
CommandName="DeleteFile" OnClick="Button1_Click1"
OnClientClick="return ConfirmDelete();"
Text="Delete" />
</ItemTemplate>
</asp:TemplateField>
<asp:CommandField ButtonType="Button" ShowEditButton="True"
EditText="Rename" UpdateText="Change" />
</Columns>
<RowStyle BackColor="#E3EAEB" />
<SelectedRowStyle BackColor="#C5BBAF" Font-Bold="True"
ForeColor="#333333" />
<PagerStyle BackColor="#666666" ForeColor="White"
HorizontalAlign="Center" />
<HeaderStyle BackColor="#1C5E55" Font-Bold="True"
ForeColor="White" />
<AlternatingRowStyle BackColor="White" />
</asp:GridView>
```

Figure 12: Markup of GridView control

Before deleting any file the user is prompted for confirming the action. This is done with the help of a small JavaScript function as shown in Figure 13.

```
<script language="javascript">
function ConfirmDelete()
{
if(confirm('Do you wish to delete this file/folder?'))
{
event.returnValue=true;
}
else
{
event.returnValue=false;
}
}
</script>
```

Figure 13: Confirming the delete operation

The ConfirmDelete() function is written in a <script> block in the <HEAD> section of the web form. It displays a JavaScript confirm dialog to the end user. If the user clicks on OK then the returnValue property of event JavaScript object is set to true indicating that the event is to be raised. Otherwise the returnValue property is set to false indicating that the event is to be cancelled.

Designing the task panel

Now that we have completed designing the TreeView and GridView, let's move on to the tasks panel at the bottom. The task panel looks as shown in Figure 14.

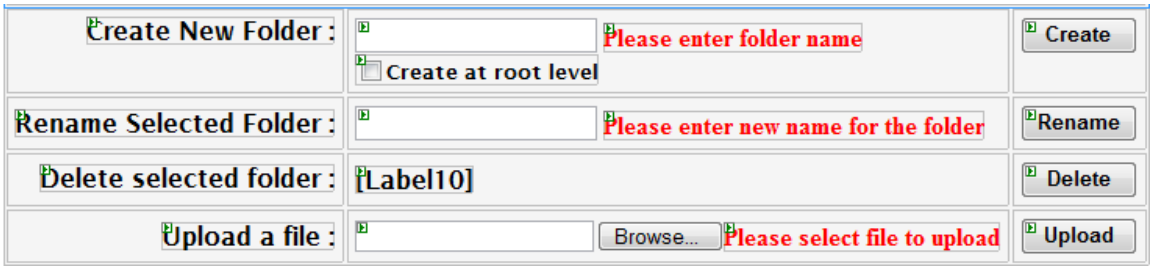

Figure 14: The task panel

The task panel allows you to create, rename and delete folders. It also allows you to upload a file to selected folder. In order to create a new folder you need to specify its name in the relevant TextBox and click on Create button. A RequiredFieldValidator ensures that the folder name is textbox is not empty. The "Create at root level" CheckBox indicates if the folder is a sub folder of the selected folder or a top level folder. In order to rename a folder you need to select it and

enter its new name in the relevant TextBox and click on Rename button. Another RequiredFieldValidator control ensures that the new name is not empty. A folder can be deleted by selecting it and then clicking on Delete button. Finally, a file can be uploaded into the selected folder by selecting it FileUpload control and then clicking on the Upload button. The Figure 15 shows various controls and their properties.

Control	Property	Value
Label1	Text	Create New Folder :
Label2	Text	Rename Selected Folder :
Label3	Text	Delete Selected Folder :
Label9	Text	Upload a File :
Label10	Text	-- empty string --
RequiredFieldValidator1	ControlToValidate	TextBox1
	ErrorMessage	Please enter folder name
	VaidationGroup	createfolder
RequiredFieldValidator2	ControlToValidate	TextBox2
	ErrorMessage	Please enter new name for the folder
	VaidationGroup	renamefolder
RequiredFieldValidator3	ControlToValidate	FileUpload1
	ErrorMessage	Please select file to upload
	VaidationGroup	uploadfile
Button1	Text	Create
	VaidationGroup	createfolder
Button2	Text	Rename
	VaidationGroup	renamefolder
Button3	Text	Delete
	OnClientClick	return ConfirmDelete();
Button4	Text	Upload
	VaidationGroup	uploadfile

Figure 14: Setting control properties for the task panel controls

Notice the use of ValidationGroup property in the Figure 14. We want to validate the textboxes only if associated buttons are clicked. Using the ValidationGroup property of validation controls and Button controls we group them in logical groups ensuring that button from a group triggers only the associated validation. Also, notice that the OnClientClick property of Delete button is calling the same ConfirmDelete() JavaScript function that we created earlier.

This completes the design of the user interface. Now we will write some code that makes the web form functional.

Writing the code

Our code consists of event handlers and some helper methods. We need to code for the following operations:

- Populating the folder tree
- Creating a folder
- Renaming a folder
- Deleting a folder
- Displaying list of files from selected folder
- Uploading a file
- Downloading a file
- Deleting a file
- Renaming a file

In the next sections we will write code for each of the above operations.

Populating the folder tree

Throughout our code we will often need to populate the TreeView control again and again. Hence we need to have a method that encapsulates that task. The Figure 15 shows this function.

```
private void FillFolderTree(TreeNode parent)
{
int parentfolderid = 0;
if (parent != null)
{
parentfolderid = int.Parse(parent.Value.ToString());
}
DataTable folders = Folders.GetSubFolders(parentfolderid);
if (parent != null)
{
parent.ChildNodes.Clear();
}
else
{
TreeView1.Nodes.Clear();
}
foreach (DataRow folder in folders.Rows)
{
TreeNode node = new TreeNode();
node.Text = folder["FolderName"].ToString();
node.Value = folder["Id"].ToString();
if (parent == null)
{
TreeView1.Nodes.Add(node);
}
else
{
parent.ChildNodes.Add(node);
}
}
```

```
if (TreeView1.SelectedNode != null)
{
TreeView1.SelectedNode.Expand();
}
}
```

Figure 15: Filling the TreeView

The FillFolderTree() method accepts a reference to a TreeNode object whose child nodes are to be populated. If the parent parameter is null then the root node will be populated. Inside it stores the folder Id of the supplied TreeNode in an integer variable. This is done using the Value property of TreeNode class. It then retrieves a DataTable containing list of subfolders by calling GetSubFolders() method Folders class. The next if-else block clears Child nodes of the TreeNode so that there are no duplicate nodes after filling the tree. Then the code iterates through the DataTable and with each iteration a new TreeNode is added to the parent node. The Text property of TreeNode displays the name of the folder whereas its Value property contains the folder Id. If the newly created TreeNode is to be added directly to the TreeView then it is added to the Nodes collection. Otherwise it is added to the ChildNodes collection of the parent TreeNode. Finally, the selected node of the TreeView is expanded to show the newly added TreeNodes with the help of Expand() method. The FillFolderTree() method is called when a new folder is added or deleted.

Creating new folders

In order to create a new folder the user need to enter its name in the textbox and click on the Create button. The Click event handler of Create button is shown in Figure 16.

```
protected void Button1_Click(object sender, EventArgs e)
{
try
{
if (TreeView1.SelectedNode != null && !CheckBox1.Checked)
{
Folders.Create(TextBox1.Text,
int.Parse(TreeView1.SelectedNode.Value.ToString()),
DateTime.Now);
FillFolderTree(TreeView1.SelectedNode);
}
else
{
Folders.Create(TextBox1.Text, 0, DateTime.Now);
FillFolderTree(null);
}
}
catch (Exception ex)
{
Label7.Text = ex.Message;
```

```
}
}
```

Figure 16: Creating a new folder

The code checks if the new folder is subfolder of selected folder or to be added at the top level. This is done by the first if condition. If some TreeNode is selected and the CheckBox is unchecked i.e. the folder is not a top level folder then the new folder is a subfolder of the selected folder. The Create() method of Folders class is then called and folder name, parent folder Id and time stamp are passed to it. Notice that parent folder Id is obtained via the Value property of the SelectedNode. In order to reflect the new folder in the tree FillFolderTree() method is called passing the reference of selected node. If the folder is a top level folder then its parent Id will be 0 as shown in the else block. If there is any exception while creating the folder such as duplicate folder name then the detailed error message is displayed in a Label control.

Renaming folders

In order to rename a folder it must be selected first. Its new name must be specified in the relevant textbox and Rename button must be clicked. The Figure 17 shows the Click event handler of Rename button.

```
protected void Button2_Click(object sender, EventArgs e)
{
try
{
int id = int.Parse(TreeView1.SelectedNode.Value.ToString());
Folders.Rename(id, TextBox2.Text);
TreeView1.SelectedNode.Text = TextBox2.Text;
}
catch (Exception ex)
{
Label7.Text = ex.Message;
}
}
```

Figure 17: Renaming a folder

The code retrieves Id of the selected folder using SelectedNode property. The Rename() method of Folders is then called by passing folder Id and new name. To reflect the change the Text property of SelectedNode is changed to the new name.

Deleting folders

A folder can be deleted by selecting it and clicking on the Delete button. Clicking on the Delete button fill first ask for confirmation as shown in Figure 18.

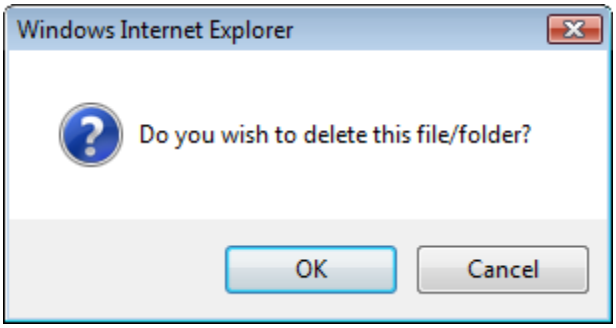

Figure 18: Confirming folder delete

If OK is selected post back happens and Click event handler of Delete button is executed. Otherwise the delete operation is cancelled. The Click event handler of Delete button is shown in Figure 19.

```
protected void Button3_Click(object sender, EventArgs e)
{
if (TreeView1.SelectedNode != null)
{
Folders.Delete(int.Parse
(TreeView1.SelectedNode.Value.ToString()));
Files.DeleteFromFolder(int.Parse(TreeView1.SelectedNode.Value.
ToString()));
foreach (TreeNode node in TreeView1.SelectedNode.ChildNodes)
{
Folders.Delete(int.Parse(node.Value.ToString()));
Files.DeleteFromFolder(int.Parse(node.Value.ToString()));
}
if (TreeView1.SelectedNode.Parent != null)
{
FillFolderTree(TreeView1.SelectedNode.Parent);
}
else
{
FillFolderTree(null);
}
GridView1.DataSource = null;
GridView1.DataBind();
}
else
{
Label7.Text = "Please select folder to be deleted";
}
}
```

Figure 19: Deleting a folder

The code calls Delete() method of Folders class by passing the folder Id of SelectedNode. When you delete a folder two additional things must happen:

- All the subfolders of the folder being deleted must also be deleted
- All the files from that folder and all its subfolders must be deleted

Hence the code calls DeleteFromFolder() method of Folders class and passed the folder Id as the parameter. This deletes all the files from the selected folder. Further a for loop iterates through the child nodes of the selected node. With each iteration Delete() and DeleteFromFolder() methods are called. Once the delete operation is over FillFolderTree() method is called to repopulate the tree. Finally the GridView is bound by setting its DataSource property to null. This ensures that the GridView doesn't show the files that are just deleted.

Displaying list of files

When user selects a folder all the files from that folder need to be displayed in the GridView we designed earlier. This is done by handling the SelectedNodeChanged event of TreeView control. This event is raised when user changes the node selection of TreeView. Figure 20 shows this event handler.

```
protected void TreeView1_SelectedNodeChanged(object sender,
EventArgs e)
{
FillFolderTree(TreeView1.SelectedNode);
TextBox2.Text = TreeView1.SelectedNode.Text;
Label10.Text = TreeView1.SelectedNode.Text;
BindGrid();
}
```

Figure 20: Handling SelectedNodeChanged event of TreeView

The code calls FillFolderTree() method so that subfolders of the selected folder are populated. We fill the subfolders as and when required rather than in advance so as to improve the performance. The textbox that allows us to rename the folder is populated with the current name. Similarly the Label associated with Delete button shows the name of the folder that can be deleted. Finally, BindGrid() function is called that binds the GridView with the list of files. The BindGrid() function is shown in Figure 21.

```
private void BindGrid()
{
DataTable files = Files.GetFilesFromFolder
(int.Parse(TreeView1.SelectedNode.Value.ToString()));
GridView1.DataSource = files;
GridView1.DataBind();
}
```

Figure 21: Binding the GridView

The code calls GetFilesFromFolder() method of Files class by passing folder Id. The files are returned as DataTable which is then bound with the GridView.

Uploading a new file

In order to upload a file we need to select it using FileUpload control and then click on the Upload button. A folder must be selected prior to uploading a file. Otherwise an error message must be shown. The Click event handler of Upload button is shown in Figure 22.

```
protected void Button4_Click(object sender, EventArgs e)
{
try
{
if (TreeView1.SelectedNode != null)
{
int folderid = int.Parse(TreeView1.SelectedNode.Value);
string filename =
Path.GetFileName(FileUpload1.PostedFile.FileName);
int filesize = 0;
if (FileUpload1.HasFile)
{
Stream stream = FileUpload1.PostedFile.InputStream;
filesize = FileUpload1.PostedFile.ContentLength;
byte[] filedata = new byte[filesize];
stream.Read(filedata, 0, filesize);
Files.Create(filename, filedata, filesize, folderid,
DateTime.Now);
}
BindGrid();
}
else
{
Label7.Text = "Please select destination folder";
}
}
catch (Exception ex)
{
Label7.Text = ex.Message;
}
}
```

Figure 22: Uploading a file

The code retrieves the folder Id of the selected folder. This is the target folder in which the file is to be uploaded. The FileUpload control gives us complete client side path and file name of the

file being uploaded. We need only the file name and hence we used GetFileName() method of Path class. The GetFileName() method accepts file system path and returns just the file name with extension. The code then decides if the file has been uploaded using HasFile property of FileUpload control. Recollect that we need to store file contents in Files table. Hence, the code gets hold of InputStream of the posted file. The InputStream property is a reference to a Stream object that contains the uploaded file. The ContentLength property of PostedFile object gives the size of the file being uploaded. The data from InputStream is then read into a byte array. The Create() method of Files class is then called that stores the file name, parent folder Id, file content, size and time stamp to the Files table. Finally BindGrid() method is called so that the GridView displays the file we just uploaded. The else part of the if condition displays an error message if no folder is selected prior to uploading a file.

Downloading a file

Each row inside the GridView has Download button. Clicking on this button starts the file download. To handle click of the Download button we use RowCommand event of the GridView. The compete code of the RowCommand event is shown in Figure 23.

```
protected void GridView1_RowCommand(object sender,
GridViewCommandEventArgs e)
{
int fileid =
int.Parse(GridView1.DataKeys[int.Parse(e.CommandArgument.ToStrin
g())].Value.ToString());
if (e.CommandName == "Download")
{
DataTable file = Files.GetFile(fileid);
byte[] filedata = (byte[])file.Rows[0]["filedata"];
Response.Clear();
Response.AddHeader("Content-Disposition", "attachment;filename="
+ file.Rows[0]["filename"]);
Response.BinaryWrite(filedata);
Response.End();
}
}
```

Figure 23: Downloading a file

Recollect that the DataKeyNames property of the GridView is set to Id. This causes the DataKeys collection of GridView to be populated with the file Ids. The CommandArgument property of GridViewCommandEventArgs class gives the index of the row whose Download button has been clicked. The Id of the file to be downloaded is then retrieved from the DataKeys collection. Based on the file Id its contents are retrieved from the Files table. This is done by calling GetFile() method of Files class. The value of filedata column is stored in a byte array. The response stream is then cleared by calling Clear() method of Response object. The Content-Disposition HTTP header is then added to the response stream with the help of AddHeader() method of Response object. Notice the use of Content-Disposition header. The value of

“attachment” makes the browser show the download dialog instead of opening the file with associated program or plug in. The default name of the file being downloaded as shown in the download dialog is indicated using filename attribute value. The actual file contents are written to the response stream using BinaryWrite() method of Response object. Finally the response is ended by calling End() method of Response object.

Renaming a file

In order to rename a file we use the in-line editing feature of GridView. Since our GridView is not bound with ant data source control as such we need to handle three events namely RowEditing, RowCancelingEdit and RowUpdating of the GridView. The event handlers of these three buttons are shown in Figure 24.

```
protected void GridView1_RowEditing(object sender,
GridViewEditEventArgs e)
{
GridView1.EditIndex = e.NewEditIndex;
BindGrid();
}
protected void GridView1_RowCancelingEdit(object sender,
GridViewCancelEditEventArgs e)
{
GridView1.EditIndex = -1;
BindGrid();
}

protected void GridView1_RowUpdating(object sender,
GridViewUpdateEventArgs e)
{
int fileid =
int.Parse(GridView1.DataKeys[e.RowIndex].Value.ToString());
GridViewRow row = GridView1.Rows[e.RowIndex];
Files.Rename(fileid,
((TextBox)row.Cells[0].FindControl("TextBox1")).Text);
GridView1.EditIndex = -1;
BindGrid();
}
```

Figure 24: Renaming a file

The RowEditing event is raised with we click the Rename button. The RowCancelingEdit event is raised when we click on Cancel button. Similarly, RowUpdating event is raised when we click on Change button.

The RowEditing event handler sets EditIndex property of GridView to the NewEditIndex property of GridViewEditEventArgs class. The NewEditIndex property index indicates the index

of the row whose Rename button is clicked. The GridView is then bound using BindGrid() method. Figure 25 shows the GridView in edit mode.

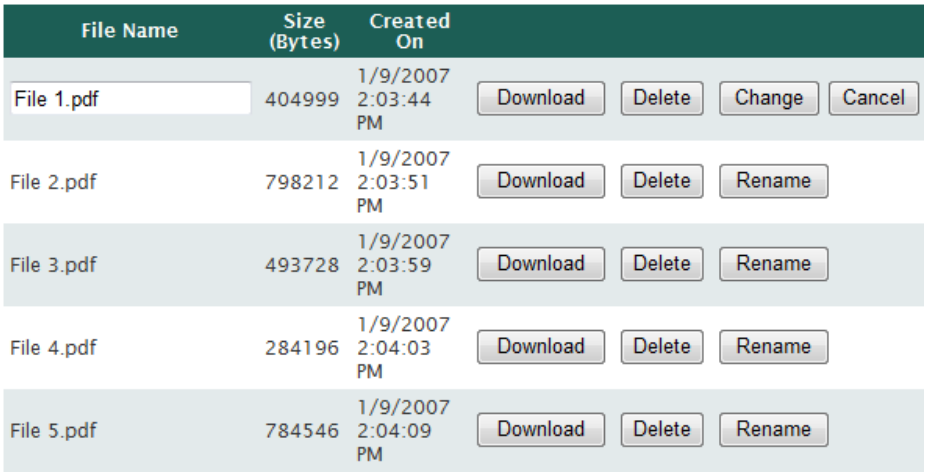

Figure 25: GridView in edit mode

The RowCancelingEdit event handler simply sets the EditIndex to -1. Setting the EditIndex to -1 indicates that no row is editable. The GridView is bound again using BindGrid() method.

The RowUpdating event handler does the core work of renaming the file. It retrieves the file Id from the DataKeys collection. The new file name entered in the TextBox is obtained from the first cell (file name is the first column in the GridView) using FindControl() method. The FindControl() method accepts the ID of a control to look for and returns its reference. The return value of FindControl() is Control. Hence we type cast it into TextBox and get the new file name from its Text property. This new name is supplied to the Rename() method of Files class. Once the renaming is over the EditIndex of the GridView is set to -1 and BindGrid() method is called.

Deleting a file

A file can be deleted by click on the Delete button for its row in the GridView. Clicking on the Delete button prompts for confirmation (see Figure 18). Recollect that Delete button is placed inside a TemplateField. The Figure 26 shows Click event handler of Delete button.

```
protected void Button1_Click1(object sender, EventArgs e)
{
int fileid = int.Parse(((Button)sender).CommandArgument);
Files.Delete(fileid);
BindGrid();
}
```

Figure 26: Deleting a file

The code retrieves the file Id from the CommandArgument property of the Button. Recollect that we have bound the CommandArgument property of Delete button with Id column of Files table.

The file is then deleted by calling Delete() method of Files class. Finally, BindGrid() method is called so that the GridView displays only the available files.

That's it! We just finished coding our briefcase application. Run it and test various pieces of functionality. Your briefcase should look similar to Figure 1.

Summary

In this second part of the two part series we designed and developed the user interface of the briefcase application. The TreeView control is natural choice for displaying any heretical structure. In our case we used it to display folder tree. The GridView control was used to display list of files. We used the GridView without binding it with data source controls. If used in this manner it reminds of the DataGrid control of ASP.NET 1.x. Our briefcase is complete with file upload, download, delete and rename functionality. Here are some suggestions for improving it further:

- You may add some security features such as membership and roles
- You may add multiuser functionality so that files and folders are maintained per user basis
- You may improve the exception handling by providing more friendly messages or even custom error pages
- You may allow the user to upload multiple files at a time
- You may allow to specify the file name and description while uploading the files
- You may restrict the uploaded files to certain size and extension
- You may maintain a history of activities