

ASP.NET GridView Control Pocket Guide

*This free book is provided by courtesy of C# Corner and Mindcracker Network and its authors. Feel free to share this book with your friends and co-workers. **Please do not reproduce, republish, edit or copy this book.***

Vincent Maverick

(C# Corner MVP)

About Author

Vincent Maverick is a Microsoft ASP.NET MVP since 2009, C# Corner MVP and DZone MVB. He works as a Technical Lead Developer in a research and development company. He works on ASP.NET, C#, MS SQL, Entity Framework, LINQ, AJAX, JavaScript, JQuery, HTML5, CSS, and other technologies.



Vincent Maverick

(C# Corner MVP)

ASP.NET GridView Control Pocket Guide

Many years ago, I wrote a series of articles in my [old blog](#) about how to use GridView to perform CRUD, and a bunch of [tips and tricks](#) related to GridView control. This book will cover the most frequently-asked ‘how to’s’ into one for an easy reference.

If you are new to ASP.NET WebForms and wanted to use GridView control for some of your projects, this book might be a good lookup reference for you. Keep in mind that this book doesn’t cover everything about GridView, and does not cover complex scenarios such as styling, reporting and drilldown support.

TABLE OF CONTENTS

1.	Overview.....	1
2.	Performing CRUD Operations in GridView with BoundFields.....	1
	2.1 Creating a Database.....	1
	2.2 Creating the Project.....	2
	2.3 Setting Up the UI.....	2
	2.4 Web.Config.....	2
	2.5 Binding GridView with Data.....	3
	2.6 Testing the App.....	4
	2.7 Show Header and Footer When No Data Returned.....	5
	2.8 Testing the App.....	6
	2.9 Adding Rows to GridView.....	7
	2.9.1 Testing the App.....	8
	2.9.2 Implementing the Insert Functionality.....	9
	2.9.3 Testing the App.....	10
	2.10 Editing and Updating GridView Row.....	13
	2.10.1 Testing the App.....	16
	2.11 Deleting GridView Rows.....	17
	2.11.1 Testing the App.....	18
3.	Performing CRUD Operations in GridView with TemplateFields.....	19
	3.1 Binding GridView with Data.....	19
	3.1.1 Testing the App.....	21
	3.2 Adding Rows to GridView.....	22
	3.2.1 Testing the App.....	23

3.3 Editing and Updating GridView Rows.....	25
3.3.1 Testing the App.....	27
3.4 Deleting GridView Rows.....	29
3.4.1 Testing the App.....	29
4. Display Confirmation Message on Delete.....	31
4.1 Using the ShowDeleteButton CommandField.....	31
4.2 Using TemplateField with LinkButton.....	33
4.2.1 The Declarative Approach.....	33
4.3.2 The Code behind Approach.....	33
4.3 Testing the App.....	34
5. GridView Multiple Delete with CheckBox and Confirm.....	35
5.1 Testing the App.....	37
6. Sorting GridView Manually With TemplateFields.....	38
6.1 Testing the App.....	41
7. Implementing Custom Paging in GridView With LINQ.....	42
8. Implementing Cascading DropDownList on Edit Mode.....	49
8.1 Scenario.....	49
8.2 The HTML Markup.....	49
8.3 The DataSource.....	51
8.4 The Implementation.....	54
8.5 Binding the GridView.....	55
8.6 Testing the App.....	57
9. Move Multiple Rows between GridViews.....	59
9.1 Testing the App.....	63
10. Pivot Data in GridView - A Generic Pivot Method with DataTable.....	66

11. Highlight Row in GridView with Colored Columns.....	69
11.1 The HTML Markup.....	69
11.2 The Code Behind.....	69
11.3 The JavaScript Code.....	72
12. Highlight GridView Row On Click And Retain Selected Row On Postback.....	73
13. Using Radio Button in GridView With JavaScript Validation.....	76
14. How To: Do Calculations in GridView.....	81
14.1 Server-Side Approach.....	81
14.2 The Client-Side Approach with JavaScript.....	83
15. Inserting and Deleting Sub Rows in GridView.....	88
16. Dynamically Adding and Deleting Rows in GridView and Saving All Rows at Once.....	92
16.1 The HTML Markup.....	93
16.2 The Code Behind.....	94
16.3 Method Definitions	99
16.4 The Events.....	99
16.5 The Output.....	99
16.6 Saving All Data at Once.....	102
16.7 Output.....	104
17. Tips and Tricks.....	107
17.1 Accessing AutoGenerated Columns in GridView.....	107
17.2 Dynamic Cascading DropDownList using JavaScript.....	108
17.3 MasterPage, WebUserControl, ModalPopup and UpdatePanel – Passing GridView values to parent pag.....	110
17.3.1 The Page Markup.....	110
17.3.2 The WebUserControl Markup.....	111

17.3.3 The Code Behind at the backend.....	112
17.3.4 Output.....	114
17.4 Highlighting GridView Rows on Mouseover.....	115
17.5 Remove GridView Row Highlighting on Edit Mode.....	115
17.6 Ways On Hiding AutoGenerateColumns in GridView.....	116
17.6.1 Using the Cells Index.....	116
17.6.2 Looping through GridView Row Controls Collections.....	116
17.6.3 Looping through GridView Cells.....	117
17.7 Wrap Particular Column Data in GridView.....	117
17.8 Accessing Controls in GridView TemplateFields on GridView Edit Mode.....	118
17.9 Limiting the Data Being displayed in GridView and Display as Tooltip.....	118
17.10 Move AutoGenerate Columns at the LeftMost Part of the GridView Columns.....	119
17.10.1 The Solution.....	121
17.11 How to Get Hidden Column Values in GridView.....	122
17.11.1 The workarounds.....	123
17.12 Copy All Selected GridView Rows to an Array.....	124
18. Summary.....	125

Overview

From the documentation, GridView is one of the data representation controls, which are available in ASP.NET WebForms. It basically displays the values of a data source in a table, where each column represents a field and each row represents a record. GridView control enables you to select, sort and edit these items.

Throughout this book, we'll be using ADO.NET as our data-access to feed GridView with the data. To put it in other words, we be seeing DataTable as our DataSource. Using ADO.NET does not mean that GridView is limited to it. Depending on your ASP.NET version, of course you could definitely use LINQ to SQL, Entity Framework or any other object relational mapper as your data access. You can then use custom entities, which implements either the System.Collections.IEnumerable interface (such as System.Data.DataView, System.Collections.ArrayList, or System.Collections.Generic.List<T>) or the IListSource interface as your DataSource, if you prefer.

Performing CRUD Operations in GridView with BoundFields

Let's start by implementing simple Add, Edit, Update and Delete operations in GridView with BoundFields. Before doing it, let's create a simple database, which we can use for this demo. If you have an existing database to work on, then you can skip the database creation, if you want.

Creating a Database

Open SQL Management Studio Express and open a New Query Window or just press CTRL + N to launch the query Window and then run the scripts, given below.

```
CREATEDatabase DemoDB

GO

CREATETABLE[dbo].[Employee](

    [Id] INTIDENTITY(1, 1) NOTNULL, [FirstName] NVARCHAR(50) NULL, [LastName]
    NVARCHAR(50) NULL, [Position] NVARCHAR(30) NULL, [Team] NVARCHAR(30) NULL

);
```

Notice, we set the Id to auto increment, so that the Id will be automatically generated for every new added row.

Creating the Project

At this point, we now have a sample database to work on. Now, it's time for us to create the Application.

Open Visual Studio and create a new WebForm's Web Application project. Add a new WebForm file and name it, as per your desire.

Setting Up the UI

Now, grab a GridView control from Visual Studio ToolBox and place it in the form. Afterwards, configure the GridView columns to display the data. ASXP HTML markup should look, as given below.

```
<asp:GridViewID="gvEmployee"runat="server"AutoGenerateColumns="false"ShowFooter="true"ShowHeader="true">
<Columns>
<asp:BoundFieldDataField="FirstName"HeaderText="First Name"/>
<asp:BoundFieldDataField="LastName"HeaderText="Last Name"/>
<asp:BoundFieldDataField="Position"HeaderText="Position"/>
<asp:BoundFieldDataField="Team"HeaderText="Team"/>
</Columns>
</asp:GridView>
```

The markup, shown above is composed of a few BoundField columns. Each DataField property represents the FieldName from our Database that we want to display. By setting AutoGenerateColumns to false, we tell the ASP.NET engine to render only the fields, based on what we defined in BoundFields. Note, GridView AutoGenerateColumns is true by default. This means that GridView control will generate all the columns, based on the DataSource assigned to it. The ShowFooter and ShowHeader attributes indicate a flag to show/hide the Header and Footer rows.

Web.Config

Now, we need to define the connection string for us to communicate with our database. Here, an example of a connection string configuration, defined within <configuration> node in web.config file.

```
<connectionStrings>
<addname="DBConnectionString"
connectionString="Data Source=(localdb)\MSSQLLocalDB;
Initial Catalog=DemoDB;
Integrated Security=True;
Connect Timeout=30;
```

```

        Encrypt=False;
        TrustServerCertificate=True;
        ApplicationIntent=ReadWrite;
        MultiSubnetFailover=False"
providerName="System.Data.SqlClient"/>
</connectionStrings>

```

Few things to note are The “DBConnectionString” is the value, which we are going to use for referencing the connection string, we defined above. You need to change the value of the DataSource and Initial Catalog, based on your Server name and the database name.

Also notice, we are using LocalDB, which is included in Visual Studio. If you are using SQL Management Studio Express, your connectionString might look, as given below.

```

<addname="DBConnectionString" connectionString="Data Source=YourDatabaseServerName;
        Initial Catalog=DemoDB;
        Integrated Security=SSPI;"
providerName="System.Data.SqlClient"/>

```

Binding GridView with Data

Now, we have everything. We need setup. It’s time for us to bind our GridView with the data. Switch to the code at the backend file of the form (.ASPX.CS) and copy the code, given below.

```

private DataTable GetAllEmployee()
{
    DataTable dt = new DataTable();
    using (SqlConnection sqlConn = new SqlConnection
        (ConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString))
    {
        string sql = "SELECT * FROM dbo.[Employee]";
        using (SqlCommand sqlCmd = new SqlCommand(sql, sqlConn))
        {
            sqlConn.Open();
            using (SqlDataAdapter sqlAdapter = new SqlDataAdapter(sqlCmd))
            {
                sqlAdapter.Fill(dt);
            }
        }
    }

    return dt;
}

private void BindGridView(DataTable source)
{
    if (source.Rows.Count > 0)
    {
        gvEmployee.DataSource = source;
        gvEmployee.DataBind();
    }
}

```

```
protectedvoid Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        DataTable dt = GetAllEmployee();
        BindGridView(dt);
    }
}
```

GetAllEmployee() method returns a DataTable object. This method is where we actually fetch the data from the database, using ADO.NET. Notice, we wrapped the SqlConnection, SqlCommand and SqlDataAdapter objects within the using statement. Since these objects implement IDisposable, putting them within the using statement will automatically dispose and close the connection of the object after it is being used. In other words, if we place the code within the using statement, we don't need to explicitly dispose the object in the code because the using statement will take care of it. As an additional note, a using statement uses a try and finally block under the hood, which disposes an IDisposable object in the finally block. We have also referenced the connectionString, which we defined earlier, using the System.Configuration.ConfigurationManager class.

The BindGridView() method is responsible for setting the DataSource and invoking Control.DataBind().

Before running the app, make sure to add the namespaces, given below.

```
using System.Data;
using System.Data.SqlClient;
using System.Configuration;
```

System.Data namespace enables us to utilize and access classes, which represents the ADO.NET architecture such as DataTable, DataRow, DataColumn and many more. The System.Data.SqlClient namespace is the .NET framework data provider for SQL Server. System.Configuration namespace enables us to gain access to the web.config elements such as <connectionString> element.

Testing the App

Now, set your WebForm page as "Start Up Page" and hit F5 to run the Application. You should be getting the output, given below.

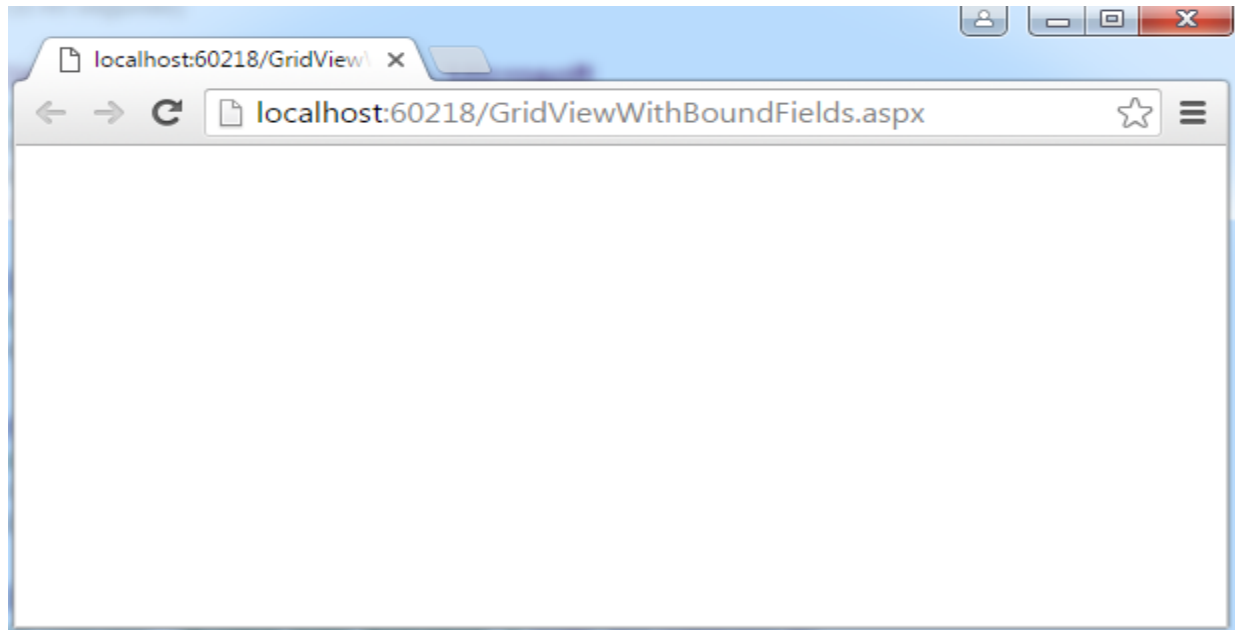


Figure 1: Blank Output

As you may already guess, it renders a blank page because we don't have any data from our database yet. By default, GridView control will not be rendered in the page, if there is no data associated with it.

Show Header and Footer When No Data Returns

To avoid the user's confusion with the blank screen, the easiest way to notify them is to show a message in the page, using a Label control but we wanted to have a more decent way of notifying the end-users and to do it, show GridView Header and Footer with a message.

Keep in mind, there are many ways to show a message, when there is no data present in GridView. Some uses the EmptyDataText attribute, some setup an <EmptyDataTemplate>, some uses the ShowHeaderWhenEmpty attribute too and some use CSS to display a message. The technique, which we are going to use is different, since we need to show both the Header and the Footer, when there is no data present.

The trick here is to add a new blank row to the DataTable, which we have used as the DataSource, if there is no data returned from the query. Here's the method, given below.

```
private void ShowNoResultFound(DataTable source, GridView gv)
{
    // create a new blank row to the DataTable
    source.Rows.Add(source.NewRow());
}
```

```
// Bind the DataTable which contain a blank row to the GridView
gv.DataSource = source;
gv.DataBind();

// Get the total number of columns in the GridView
//to know what the Column Span should be
int columnsCount = gv.Columns.Count;
gv.Rows[0].Cells.Clear();// clear all the cells in the row
gv.Rows[0].Cells.Add(new TableCell()); //add a new blank cell

//set the column span to the new added cell
gv.Rows[0].Cells[0].ColumnSpan = columnsCount;

//You can set the styles here
gv.Rows[0].Cells[0].HorizontalAlign = HorizontalAlign.Center;
gv.Rows[0].Cells[0].ForeColor = System.Drawing.Color.Red;
gv.Rows[0].Cells[0].Font.Bold = true;
//set No Results found to the new added cell
gv.Rows[0].Cells[0].Text = "NO RESULT FOUND!";
}
```

The method, given above takes two parameters- the DataTable, which we used as the DataSource, while the other is the ID of the GridView.

All we need to do is call the method ShowNoResultFound(), when our DataSource (the DataTable) returns nothing. The updated BindGridView() method is given below.

```
private void BindGridView(DataTable source)
{
    if(source.Rows.Count > 0)
    {
        gvEmployee.DataSource = source;
        gvEmployee.DataBind();
    }
    else
    {
        ShowNoResultFound(source, gvEmployee);
    }
}
```

Testing the App

Running the page shows the result, as given below.

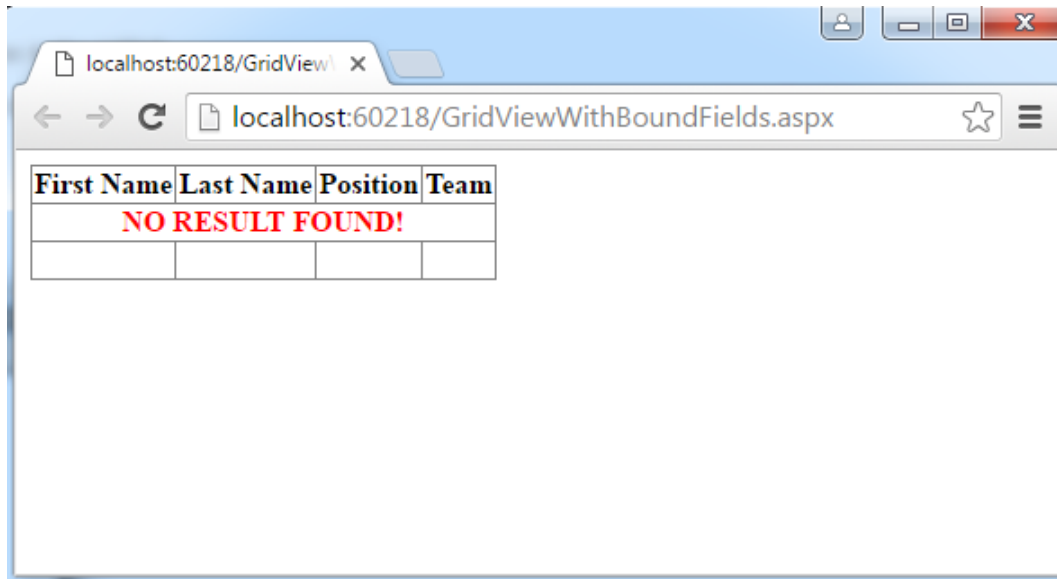


Figure 2: No Result Found

Adding Rows to GridView

Now, it's time for us to add the new data to our GridView. Keep in mind, the GridView control doesn't support built-in an Insert functionality. This means, we need to configure our own implementation to insert the data to the GridView. The simplest way to add new data is setup some TextBox controls in the form outside the GridView itself. This way, we won't be dealing with finding the TextBox controls, when getting its values.

For this particular demo, we won't be doing it. Instead, we will be creating some TextBox controls inside GridView's Footer row, which would serve as our data entry to add new rows to GridView. This approach is a bit challenging because we need to do some extra work on creating and extracting the controls dynamically.

Let's go ahead and switch to the designer form of our page and do the steps, given below.

1. Click on the GridView control.
2. Navigate to the Properties Window.
3. Click the Events button (you can see it as a lightning icon from the properties).
4. Double-click the RowCreated event to let Visual Studio generate the event for you.

Now, copy the code, given below at RowCreated event.

```
protectedvoid gvEmployee_RowCreated(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.Footer)
```

```

{
for (int i = 0; i < e.Row.Cells.Count; i++)
{
//insert TextBox for each column
TextBox tb = new TextBox();
tb.ID = string.Format("txtEntry{0}", i);

e.Row.Cells[i].Controls.Add(tb);

//insert the Add Button
if (i == e.Row.Cells.Count - 1)
{
Button btn = new Button();
btn.ID = "btnAddNew";
btn.Text = "Add New";

e.Row.Cells[i].Controls.Add(btn);
}
}
}

```

Let's see, what we just did there.

We choose the RowCreated event because it's the perfect event to generate dynamic controls within GridView. The first thing, we did from the code above is we checked for the DataControlRowType.Footer. This line will ensure, where we will run the specific code, when the condition is true. We then loop through the Footer's Row.Cells and add the controls we need. In this case, it is some TextBox controls and a Button control.

Testing the app

Running the code should result, as given below.

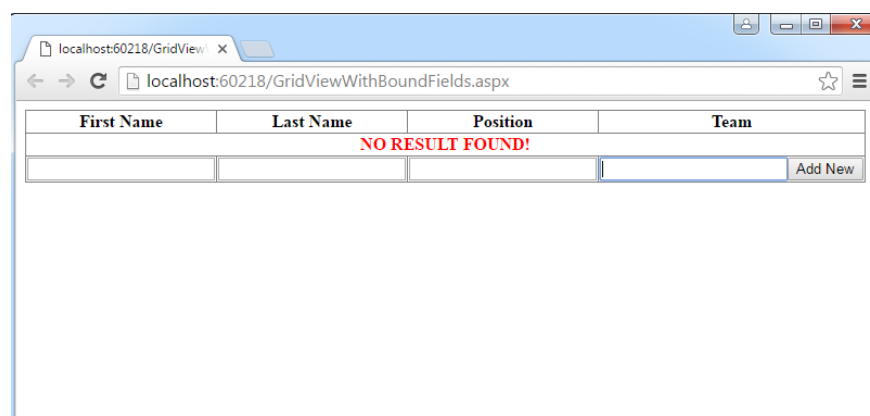


Figure 3: Default Data Entry

Implementing the Insert Functionality

Now, we're ready to implement the insert functionality for our GridView. The first thing we need to do is to update our RowCreated event to wire up a Click event for AddNew Button. The updated code is given below (see the highlighted code).

```
//insert the Add Button
if (i == e.Row.Cells.Count - 1)
{
    Button btn = new Button();
    btn.ID = "btnAddNew";
    btn.Text = "Add New";

    btn.Click += btnAddNew_Click;

    e.Row.Cells[i].Controls.Add(btn);
}
```

We need TO create the event handler, as given below.

```
protected void btnAddNew_Click(object sender, EventArgs e)
{
    Button btn = (Button)sender;
    GridViewRow row = (GridViewRow)btn.NamingContainer;
    if (row != null)
    {
        if (InsertNewEmployee(row))
            ClientScript.RegisterClientScriptBlock(typeof(Page), "ALERT", "alert('New record added.');" , true);
        else
            ClientScript.RegisterClientScriptBlock(typeof(Page), "ALERT", "alert('All fields cannot be empty.');" , true);
    }

    BindGridView(GetAllEmployee());
}
```

Let's take a quick look of what we just did there.

The first thing, we did from the code, given above is, we casted the sender of the object to a Button type to determine, which control triggers the event. We then casted the btn.NamingContainer to a GridViewRow type to determine which row has the control sender located? Notice, we implemented a simple validation check, based on the value returned from the InsertNewEmployee() method. If it's true, we will display a simple popup message to notify the user that the records were added, otherwise display all the message, which cannot be empty. InsertNewEmployee() method is responsible for validating the inputs and inserting the values to the database. We'll be seeing how this method is implemented soon enough. Notice, at bottom line, we called the BindGridView(GetAllEmployee()) method. This will ensure that the GridView will contain the newly added data on PostBacks.

The code for the InsertNewEmployee() method is given below.

```
private bool InsertNewEmployee(GridViewRow row)
{
    string firstName = ((TextBox)row.FindControl("txtEntry0")).Text;
    string lastName = ((TextBox)row.FindControl("txtEntry1")).Text;
    string position = ((TextBox)row.FindControl("txtEntry2")).Text;
    string team = ((TextBox)row.FindControl("txtEntry3")).Text;

    if (!string.IsNullOrEmpty(firstName) && !string.IsNullOrEmpty(lastName)
        && !string.IsNullOrEmpty(position) && !string.IsNullOrEmpty(team))
    {
        using (SqlConnection sqlConn = new SqlConnection
            (ConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString))
        {
            string sql = @"INSERT INTO dbo.[Employee] (FirstName,LastName,Position,Team)
                VALUES (@FirstName,@LastName,@Position,@Team)";

            using (SqlCommand sqlCmd = new SqlCommand(sql, sqlConn))
            {
                sqlConn.Open();
                sqlCmd.Parameters.AddWithValue("@FirstName", firstName);
                sqlCmd.Parameters.AddWithValue("@LastName", lastName);
                sqlCmd.Parameters.AddWithValue("@Position", position);
                sqlCmd.Parameters.AddWithValue("@Team", team);

                sqlCmd.ExecuteNonQuery();
            }
        }

        return true;
    }

    return false;
}
```

In the first few lines, we extracted the value from TextBox controls. We did it by using FindControl() method and passed the specific ID for each control. If you remember, the ID was generated during the creation of TextBox controls dynamically from RowCreated event. Subsequently, we will check for the empty values from each string variable, using string.IsNullOrEmpty(). If the if-statement condition returns true, we issue a query to our database, using ADO.NET, else we simply return false.

Testing the App

Running the page will result, as given below.

When a user leaves some of the fields empty, it is displayed, as shown below.

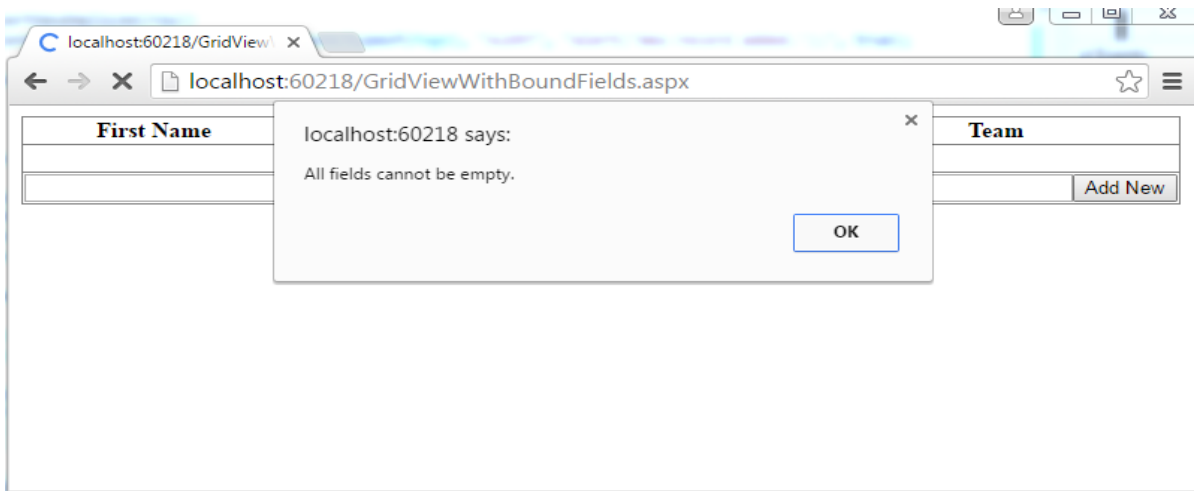


Figure 4: Output

Before hitting the “Add New” button, the output is given below.

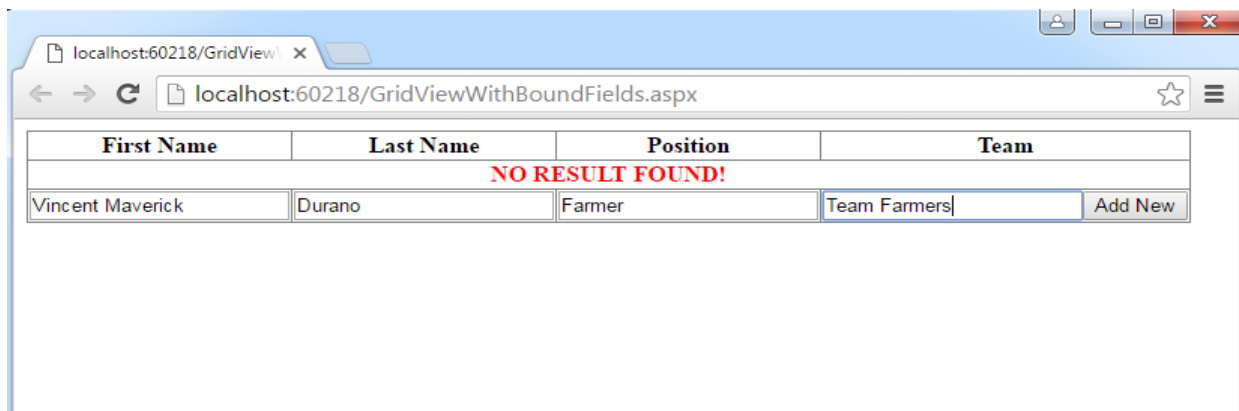


Figure 5: Output

A popup message indicates that the records were added successfully.

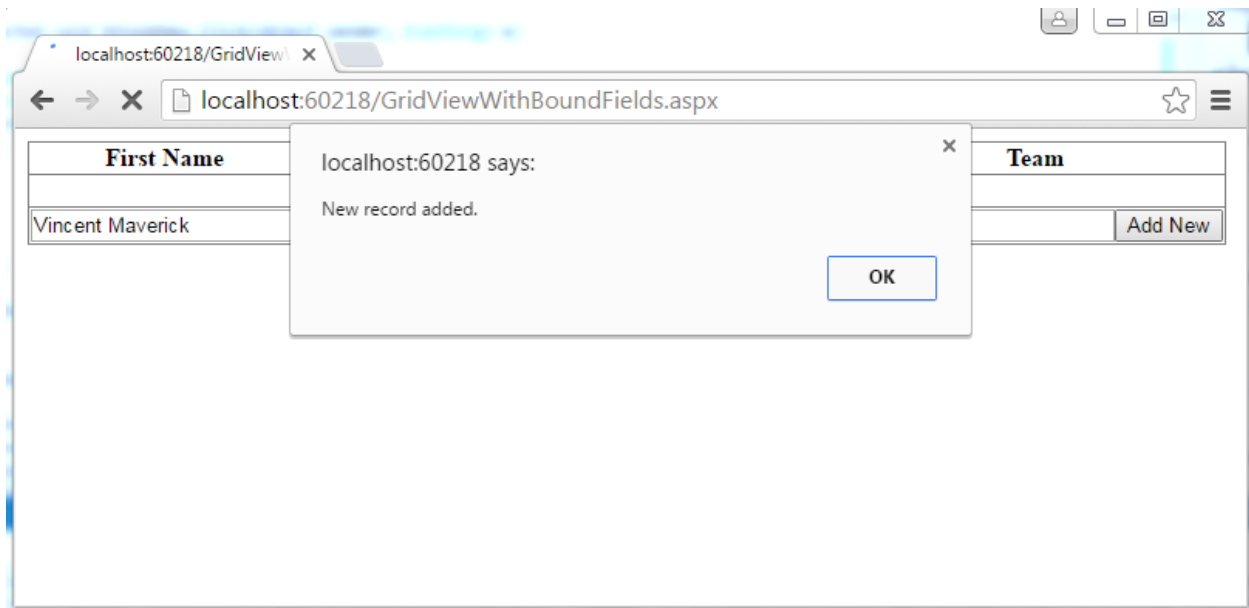


Figure 6: Output

After adding new records to database, the output is given below.

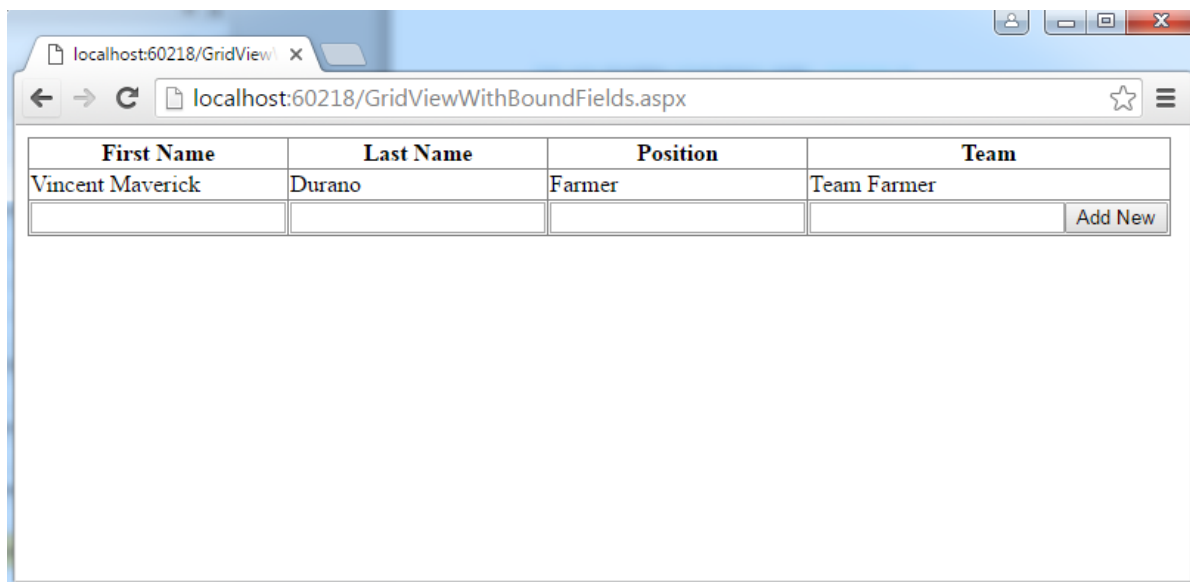


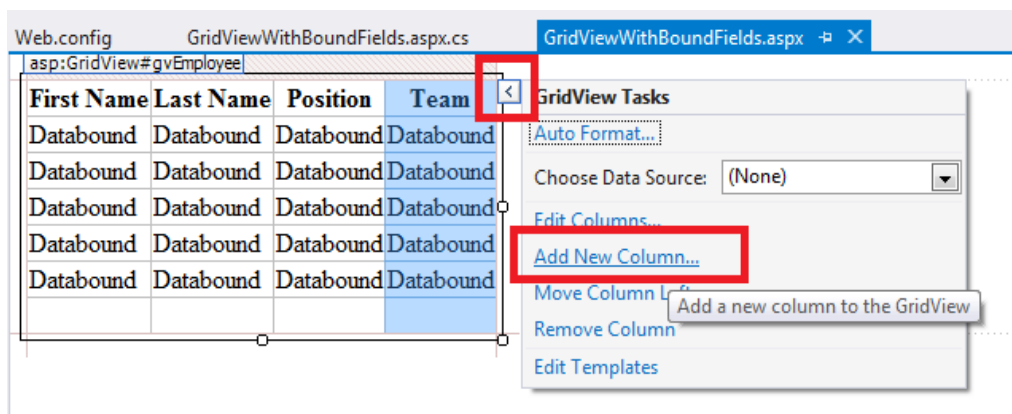
Figure 7: Output

Editing and Updating GridView Rows

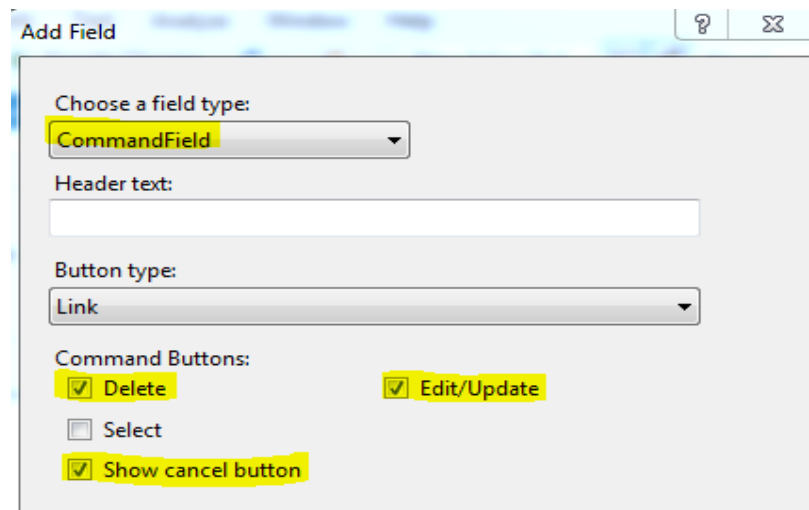
Now, it's time for us to implement edit and update functionality for our GridView. One of the good things about GridView is it provides a built-in CommandField buttons, which allows us to perform certain actions like editing, updating, deleting and selecting of GridView data.

To add the aforementioned command fields, you can follow the steps, given below.

1. Switch to Design View
2. Click on the GridView and select --> Show Smart Tag --> Add New Column. See the figure, given below for the graphical reference.



3. On the List, select CommandField.
4. Check Delete and Edit/Update options, as shown in the figure, given below.



5. Click OK to let Visual Studio generate the markup for us.

If you have noticed, the Edit and Delete CommandFields are automatically added at the last column of GridView. Now, we can start writing the code for editing and updating the information in GridView.

In-order to perform Edit and Update in GridView, we need to use three events, which are `GridView_RowEditing`, `GridView_RowCancelingEdit` and `GridView_RowUpdating`. To generate the events in GridView, you can follow the steps, given below

1. Switch to Design View in Visual Studio Designer.
2. Click on the GridView.
3. Navigate to the GridView Property Pane and then SWITCH to Event Properties.
4. From there, you should be able to find the list of events including the three events, mentioned above.
5. Double click on each event, where we need to generate the Event handler for you.

Now, set the Id field as the `DataKeyNames` in our GridView. Our updated GridView markup should now look something, as given below (see highlighted line).

```
<asp:GridViewID="gvEmployee"runat="server"AutoGenerateColumns="False"
DataKeyNames="Id"
ShowFooter="True"
OnRowCreated="gvEmployee_RowCreated"
OnRowCancelingEdit="gvEmployee_RowCancelingEdit"
OnRowEditing="gvEmployee_RowEditing"
OnRowUpdating="gvEmployee_RowUpdating">
<Columns>
<asp:BoundFieldDataField="FirstName"HeaderText="First Name"/>
<asp:BoundFieldDataField="LastName"HeaderText="Last Name"/>
<asp:BoundFieldDataField="Position"HeaderText="Position"/>
<asp:BoundFieldDataField="Team"HeaderText="Team"/>
<asp:CommandFieldShowDeleteButton="True"ShowEditButton="True"/>
</Columns>
</asp:GridView>
```

We need the Id field as our basis to perform Edit, Update and Delete. Note, we can add the multiple keys within `DataKeyNames` attribute, but for this example, we'll just need only one key. Let's keep rolling and proceed to the code at the backend part of the page. Here, the code for each events is given below.

```
privatebool UpdateEmployee(GridViewRow row)
{
int id = (int)gvEmployee.DataKeys[row.RowIndex].Value;
string firstName = ((TextBox)row.Cells[0].Controls[0]).Text;
string lastName = ((TextBox)row.Cells[1].Controls[0]).Text;
string position = ((TextBox)row.Cells[2].Controls[0]).Text;
string team = ((TextBox)row.Cells[3].Controls[0]).Text;
```

```

if ((firstName ?? lastName ?? position ?? team) != string.Empty)
{
    using (SqlConnection sqlConn = new SqlConnection

(ConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString))
    {
        string sql = @"UPDATE dbo.[Employee]
                        SET FirstName = @FirstName,
                        LastName = @LastName,
                        Position = @Position,
                        Team = @Team
                        WHERE Id = @Id";

        using (SqlCommand sqlCmd = new SqlCommand(sql, sqlConn))
        {
            sqlConn.Open();
            sqlCmd.Parameters.AddWithValue("@Id", id);
            sqlCmd.Parameters.AddWithValue("@FirstName", firstName);
            sqlCmd.Parameters.AddWithValue("@LastName", lastName);
            sqlCmd.Parameters.AddWithValue("@Position", position);
            sqlCmd.Parameters.AddWithValue("@Team", team);

            sqlCmd.ExecuteNonQuery();
        }
    }

    return true;
}

return false;
}

protected void gvEmployee_RowEditing(object sender, GridViewEditEventArgs e)
{
    gvEmployee.EditIndex = e.NewEditIndex; // turn to edit mode
    BindGridView(GetAllEmployee()); // Rebind GridView to show the data in edit mode
}

protected void gvEmployee_RowCancelingEdit(object sender, GridViewCancelEventArgs e)
{
    gvEmployee.EditIndex = -1; // turn to read-only mode
    BindGridView(GetAllEmployee()); // Rebind GridView to show the data in read-only mode
}

protected void gvEmployee_RowUpdating(object sender, GridViewUpdateEventArgs e)
{
    if (UpdateEmployee(gvEmployee.Rows[e.RowIndex]))
        ClientScript.RegisterClientScriptBlock(typeof(Page), "ALERT", "alert('Record
updated.');

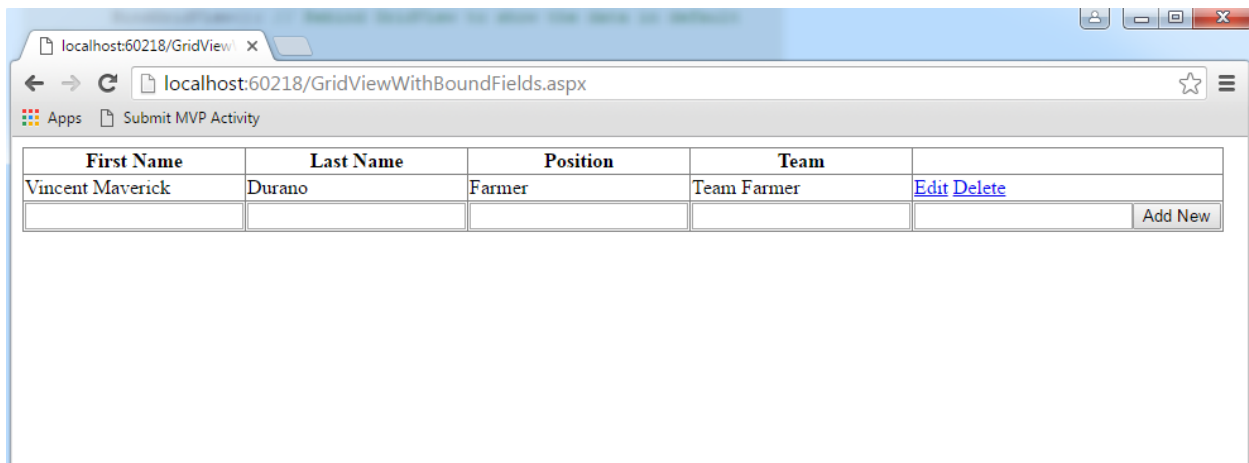
```

There's really not much to say about the code, mentioned above. The implementation of the `UpdateEmployee()` method is pretty much similar to the `InsertNewEmployee()` method, except that we've added a code to get the `Id` field from the `DataKeyNames` property. Adding to it, we also changed our SQL string variable query to do an "Update". The codes for `RowEditing`, `RowCancelingEdit` and `RowUpdating` events were pretty much self-explanatory, as you can see from the code comments.

Testing the app

Let's test the app. running the app should result, as given below.

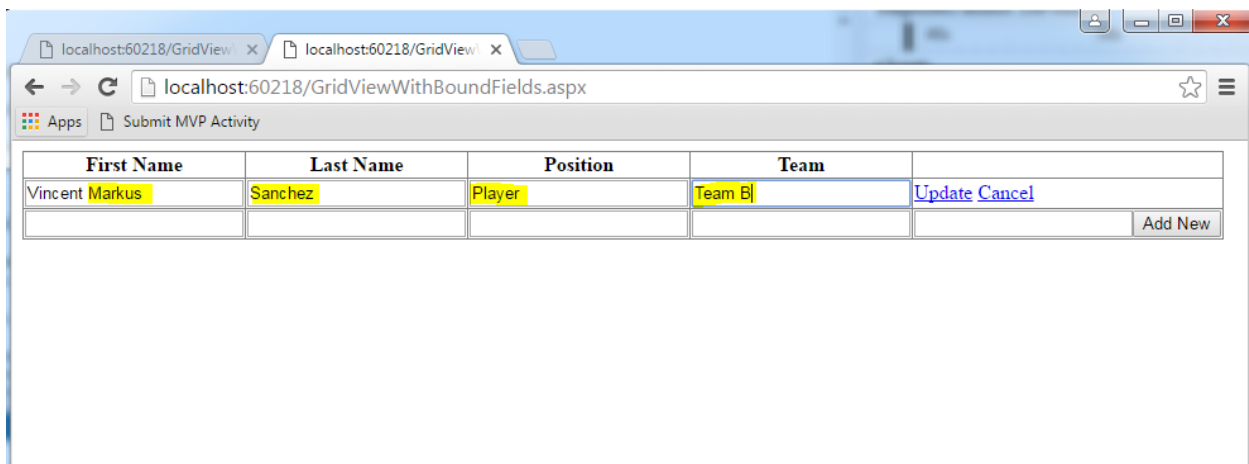
On initial load, the output is given below.



First Name	Last Name	Position	Team	
Vincent	Maverick	Durano	Farmer	Team Farmer

Add New

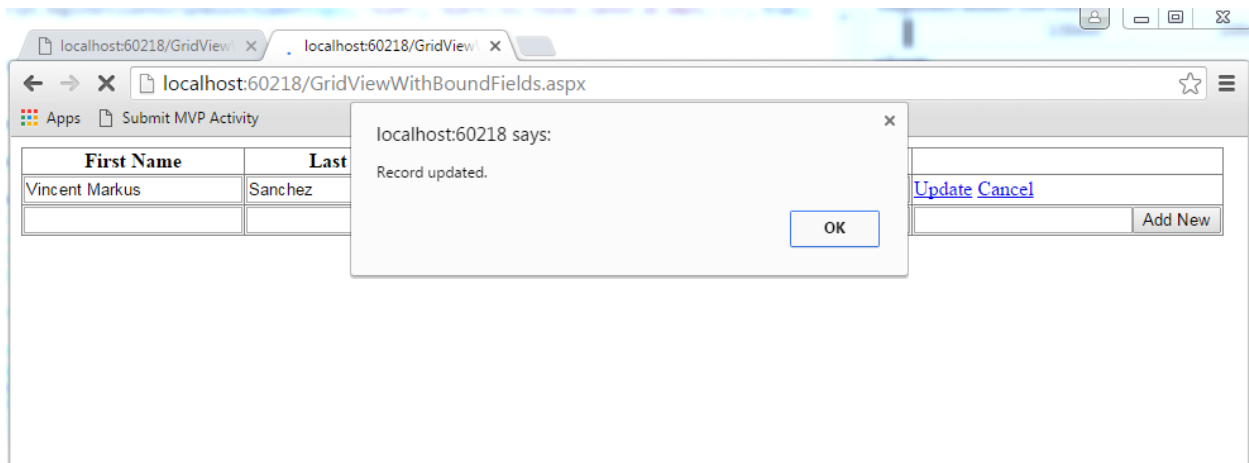
On edit, the output is given below.



First Name	Last Name	Position	Team	
Vincent	Markus	Sanchez	Player	Team B

Add New

After update, the output is given below.



Deleting GridView Rows

Now, let's implement the deletion. Since, we are using the built-in Delete CommandField button in GridView, we can use the GridView_RowDeleting event to delete the specific row in GridView. To add the event, just append the line, given below in your GridView.

```
OnRowDeleting="gvEmployee_RowDeleting"
```

Here, the code block for the Delete functionality is given below.

```
private bool DeleteEmployee(int id)
{
    using (SqlConnection sqlConn = new SqlConnection
        (ConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString))
    {
        string sql = @"DELETE FROM dbo.[Employee] WHERE Id = @Id";
        using (SqlCommand sqlCmd = new SqlCommand(sql, sqlConn))
        {
            sqlConn.Open();
            sqlCmd.Parameters.AddWithValue("@Id", id);
            sqlCmd.ExecuteNonQuery();
        }
    }

    return true;
}

protected void gvEmployee_RowDeleting(object sender, GridViewDeleteEventArgs e)
{
    int id = (int)gvEmployee.DataKeys[e.RowIndex].Value;
    if (DeleteEmployee(id))
        ClientScript.RegisterClientScriptBlock(typeof(Page), "ALERT", "alert('Record
        deleted.');
```

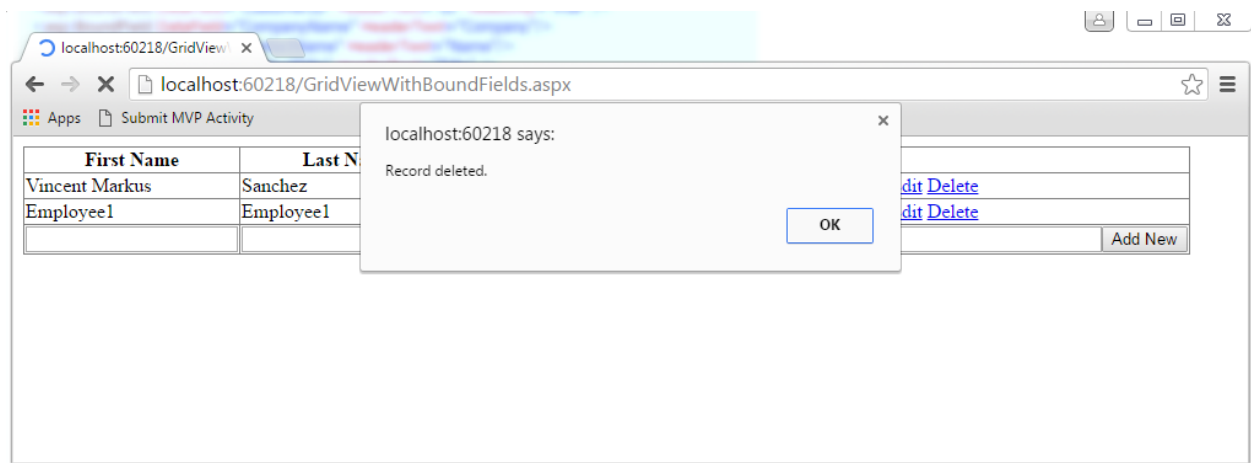


```
BindGridView(GetAllEmployee()); // Rebind GridView to reflect changes
}
```

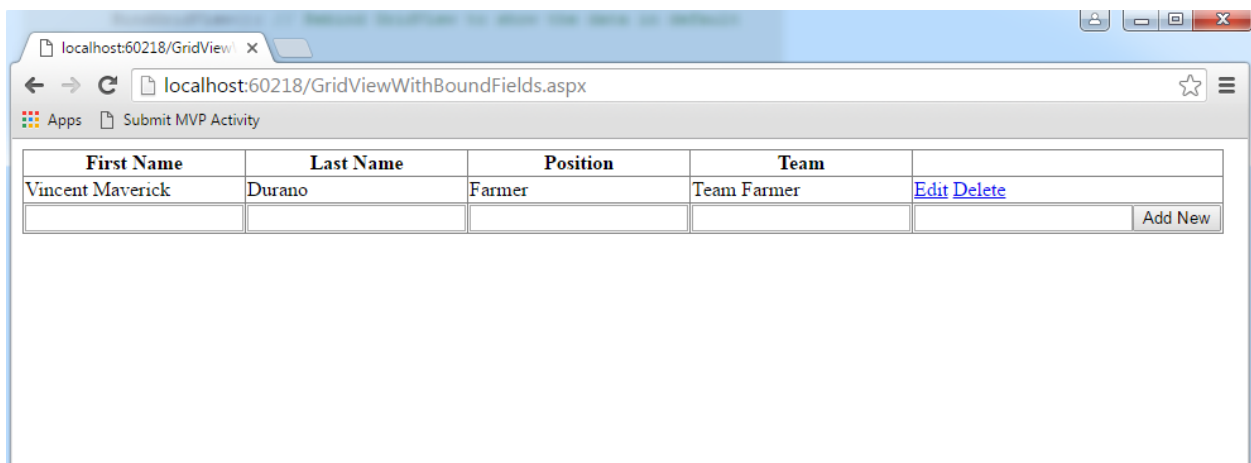
DeleteEmployee() takes an Id as the parameter. The Id will be used as the reference to delete a row from the database. See SQL string variable for SQL statement for the delete. The gvEmployee_RowDeleting event fires right after clicking the delete link from the GridView. We need take the specific Id for a row and pass it to the DeleteEmployee () method. If the deletion is successful, we will display a message to the user in the page to notify them for the changes.

Testing the App

On delete, the output will be given below.



After deletion, the output will be given below.



That's simple. At this point, we now have a fully working CRUD feature in our GridView, using

BouldFields. Now, let's try to explore CRUD operation with TemplateFields.

Performing CRUD Operations in GridView with TemplateFields

In this section, we'll take a look at how to perform CRUD operations in GridView, using TemplateField columns.

We'll be using the same database, which was used in the previous section as our test data. Let's start with binding the GridView with the data.

Binding GridView with Data

Go ahead and create a new WebForm's page and grab a GridView control from Visual Studio ToolBox and place it in the form. Setup the GridView columns to display the data. HTML markup should look, as given below.

```
<asp:GridViewID="gvEmployee"runat="server"
AutoGenerateColumns="false"
ShowFooter="true">
<Columns>
<asp:TemplateFieldHeaderText="First Name">
<ItemTemplate>
<asp:LabelID="lblFirstName"runat="server"Text='<%# Bind("FirstName") %>' />
</ItemTemplate>
<FooterTemplate>
<asp:TextBoxID="txtFirstName"runat="server" />
</FooterTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Last Name">
<ItemTemplate>
<asp:LabelID="lblLastName"runat="server"Text='<%# Bind("LastName") %>' />
</ItemTemplate>
<FooterTemplate>
<asp:TextBoxID="txtLastName"runat="server" />
</FooterTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Position">
<ItemTemplate>
<asp:LabelID="lblPosition"runat="server"Text='<%# Bind("Position") %>' />
</ItemTemplate>
<FooterTemplate>
<asp:TextBoxID="txtPosition"runat="server" />
</FooterTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Team">
<ItemTemplate>
<asp:LabelID="lblTeam"runat="server"Text='<%# Bind("Team") %>' />
</ItemTemplate>
<FooterTemplate>
<asp:TextBoxID="txtTeam"runat="server" />
</FooterTemplate>
```

```
</asp:TemplateField>  
</Columns>  
</asp:GridView>
```

As you can see, we have setup some TemplateField columns to display the data from the database. Using TemplateFields gives us the flexibility or control over the GridView. In other words, it allows us to customize the rendered markup of the GridView, as we can add our own Server controls within it. In regards to this example, we have added some Label and TextBox controls within TemplateFields to display the data. Label controls within <ItemTemplate> will be populated with the data from the database. Notice, the call to the two-way data binding expression in the Label's Text property. In the <FooterTemplate> node, we have setup some TextBox controls to allow the users to add new entry to the database, which is similar to what we did in the previous section. Note, we need to set ShowFooter to true in our GridView to show the footer template at the runtime.

Now, let's switch to the code-backend file of the form and implement the code to bind our GridView with the data from the database. Here, the full code is given below.

```
using System;  
using System.Web.UI;  
using System.Web.UI.WebControls;  
using System.Data;  
using System.Data.SqlClient;  
using System.Configuration;  
  
namespace WebFormsDemo  
{  
    public partial class GridViewWithTemplateFields : System.Web.UI.Page  
    {  
        private string GetDBConnectionString()  
        {  
            return ConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString;  
        }  
  
        private void BindGrid()  
        {  
  
            DataTable dt = new DataTable();  
            using (SqlConnection sqlConn = new SqlConnection(GetDBConnectionString()))  
            {  
                string sql = "SELECT * FROM dbo.[Employee]";  
                using (SqlCommand sqlCmd = new SqlCommand(sql, sqlConn))  
                {  
                    sqlConn.Open();  
                    using (SqlDataAdapter sqlAdapter = new SqlDataAdapter(sqlCmd))  
                    {  
                        sqlAdapter.Fill(dt);  
                    }  
                }  
            }  
        }  
    }  
}
```

```

if(dt.Rows.Count > 0)
{
    gvEmployee.DataSource = dt;
    gvEmployee.DataBind();
}
else
{
    //display no records found here
}

}

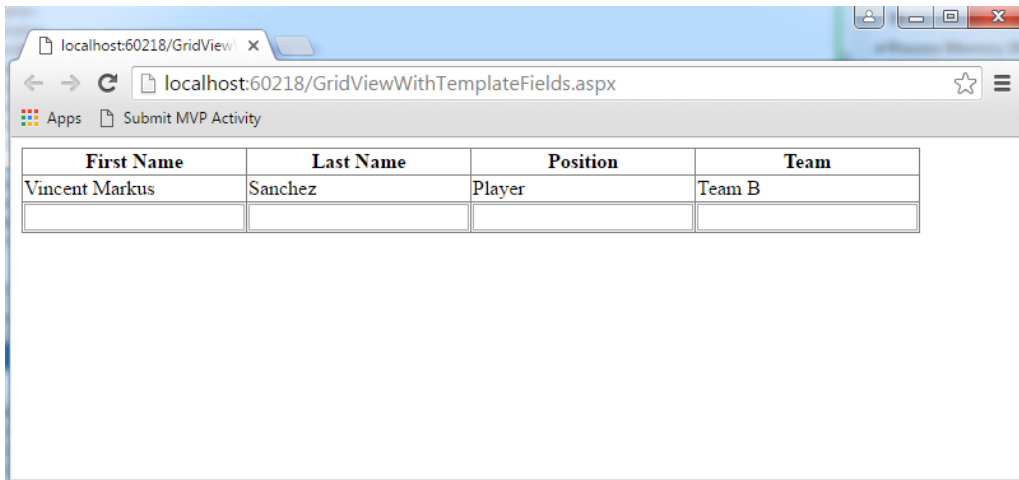
protectedvoid Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
        BindGrid();
}
}

```

There's not much to say about the code, given above, as the implementation is pretty much the same as what we did in the previous section. The only thing, which we have changed there is we place the connection string in a separate method for an easy reference.

Testing the app

Running the code should result, as given below.



The screenshot shows a web browser window with the address bar displaying 'localhost:60218/GridViewWithTemplateFields.aspx'. The browser has tabs for 'localhost:60218/GridView' and 'Submit MVP Activity'. The main content area displays a table with the following data:

First Name	Last Name	Position	Team
Vincent Markus	Sanchez	Player	Team B

Pretty simple!

SHARE THIS DOCUMENT AS IT IS. PLEASE DO NOT REPRODUCE, REPUBLISH, CHANGE OR COPY.

```
return false;
}

protected void btnAddNew_Click(object sender, EventArgs e)
{
    Button btn = (Button)sender;
    GridViewRow row = (GridViewRow)btn.NamingContainer;
    if (row != null)
    {
        if (InsertNewEmployee(row))
            ClientScript.RegisterClientScriptBlock(typeof(Page), "ALERT", "alert('New record added.');" , true);
        else
            ClientScript.RegisterClientScriptBlock(typeof(Page), "ALERT", "alert('All fields cannot be empty.');" , true);
    }

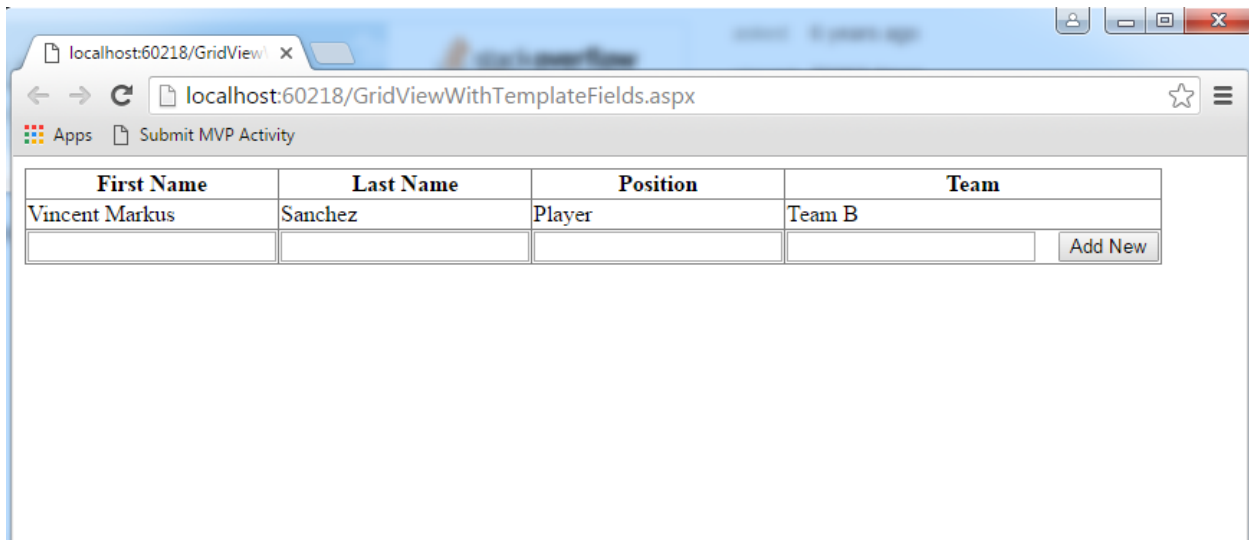
    BindGrid();
}
```

Note, that the code, mentioned above is pretty much the same, as we did in the previous section with BoundFields.

Testing the app

Running the code should result to something, as given below.

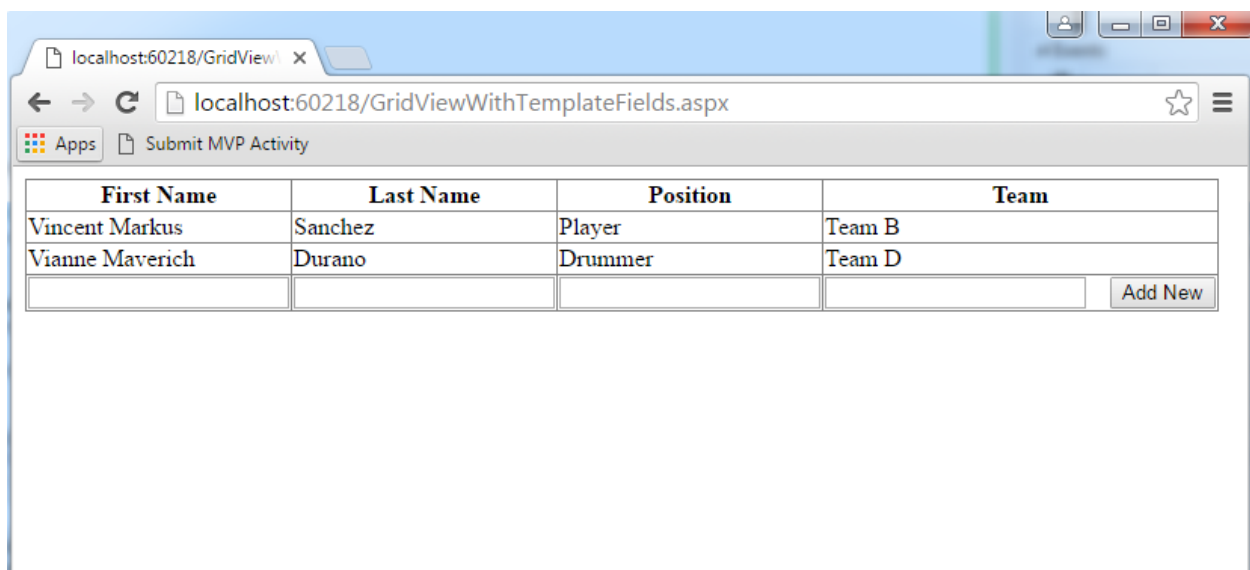
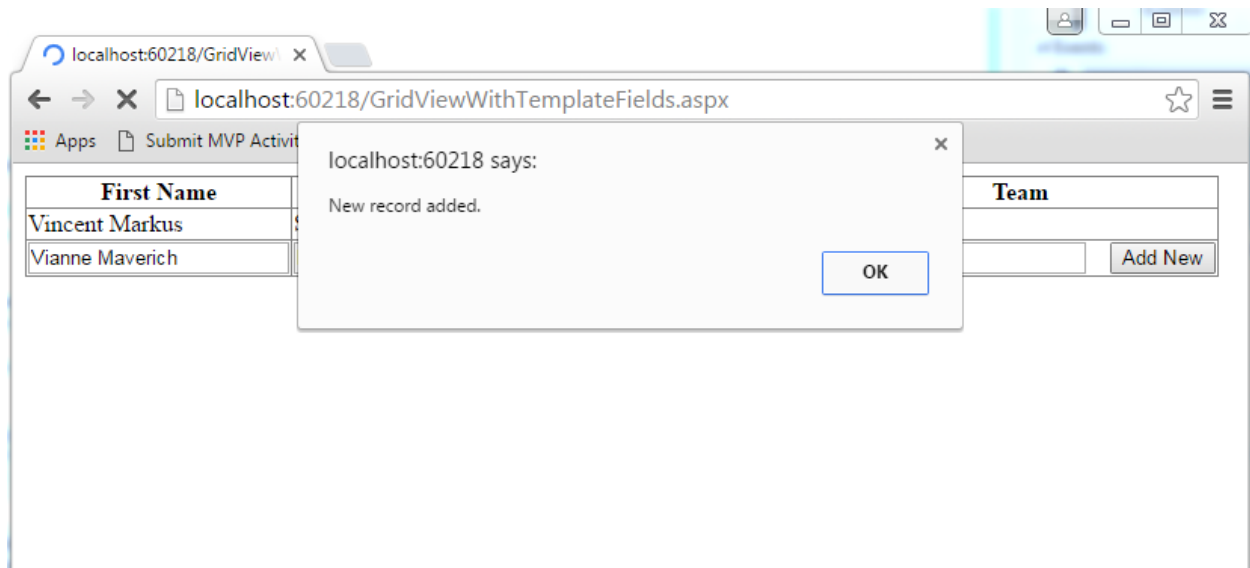
On initial load, the output is given below.



First Name	Last Name	Position	Team
Vincent Markus	Sanchez	Player	Team B

Add New

After adding new rows, the output is given below.



Editing and Updating GridView Rows

Till now, we have learnt the basics of adding new rows to the database. Now, we need to setup some <EditItemTemplate> nodes in order for us to perform an edit and update with TemplateFields. We also need to set the Id field as DataKeyNames. Adding to this, we also need to set the ShowEditButton and generate the required events for editing, updating and cancelling the operations. This could simply mean, where we need to update our existing GridView markup with the following (see highlighted lines, given below).

```
<asp:GridViewID="gvEmployee"runat="server"
AutoGenerateColumns="false"
ShowFooter="true"
DataKeyNames="Id"
OnRowCancelingEdit="gvEmployee_RowCancelingEdit"
OnRowEditing="gvEmployee_RowEditing"
OnRowUpdating="gvEmployee_RowUpdating">
<Columns>
<asp:TemplateFieldHeaderText="First Name">
<ItemTemplate>
<asp:LabelID="lblFirstName"runat="server"Text='<%# Bind("FirstName") %>' />
</ItemTemplate>
<EditItemTemplate>
<asp:TextBoxID="txtEditFirstName"runat="server"Text='<%# Bind("FirstName") %>' />
</EditItemTemplate>
<FooterTemplate>
<asp:TextBoxID="txtFirstName"runat="server" />
</FooterTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Last Name">
<ItemTemplate>
<asp:LabelID="lblLastName"runat="server"Text='<%# Bind("LastName") %>' />
</ItemTemplate>
<EditItemTemplate>
<asp:TextBoxID="txtEditLastName"runat="server"Text='<%# Bind("LastName") %>' />
</EditItemTemplate>
<FooterTemplate>
<asp:TextBoxID="txtLastName"runat="server" />
</FooterTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Position">
<ItemTemplate>
<asp:LabelID="lblPosition"runat="server"Text='<%# Bind("Position") %>' />
</ItemTemplate>
<EditItemTemplate>
<asp:TextBoxID="txtEditPosition"runat="server"Text='<%# Bind("Position") %>' />
</EditItemTemplate>
<FooterTemplate>
<asp:TextBoxID="txtPosition"runat="server" />
</FooterTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Team">
```


[illegible]

Here, the code is given below for edit and update functionality.

```
private bool UpdateEmployee(GridviewRow row)
{
    int id = (int)gvEmployee.DataKeys[row.RowIndex].Value;
    string firstName = ((TextBox)row.Cells[0].FindControl("txtEditFirstName")).Text;
    string lastName = ((TextBox)row.Cells[1].FindControl("txtEditLastName")).Text;
    string position = ((TextBox)row.Cells[2].FindControl("txtEditPosition")).Text;
    string team = ((TextBox)row.Cells[3].FindControl("txtEditTeam")).Text;

    if (!string.IsNullOrEmpty(firstName) && !string.IsNullOrEmpty(lastName)
        && !string.IsNullOrEmpty(position) && !string.IsNullOrEmpty(team))
    {
        using (SqlConnection sqlConn = new SqlConnection(GetDBConnectionString()))
        {
            string sql = @"UPDATE dbo.[Employee]
                           SET FirstName = @FirstName,
                               LastName = @LastName,
                               Position = @Position,
                               Team = @Team
                           WHERE Id = @Id";

            using (SqlCommand sqlCmd = new SqlCommand(sql, sqlConn))
            {
                sqlConn.Open();
                sqlCmd.Parameters.AddWithValue("@Id", id);
                sqlCmd.Parameters.AddWithValue("@FirstName", firstName);
                sqlCmd.Parameters.AddWithValue("@LastName", lastName);
                sqlCmd.Parameters.AddWithValue("@Position", position);
                sqlCmd.Parameters.AddWithValue("@Team", team);

                sqlCmd.ExecuteNonQuery();
            }
        }
    }

    return true;
}

return false;
}
```

```
protectedvoid gvEmployee_RowEditing(object sender, GridViewEditEventArgs e)
{
    gvEmployee.EditIndex = e.NewEditIndex; // turn to edit mode
    BindGrid(); // Rebind GridView to show the data in edit mode
}

protectedvoid gvEmployee_RowCancelingEdit(object sender, GridViewCancelEditEventArgs e)
{
    gvEmployee.EditIndex = -1; // turn to read-only mode
    BindGrid(); // Rebind GridView to show the data in read-only mode
}

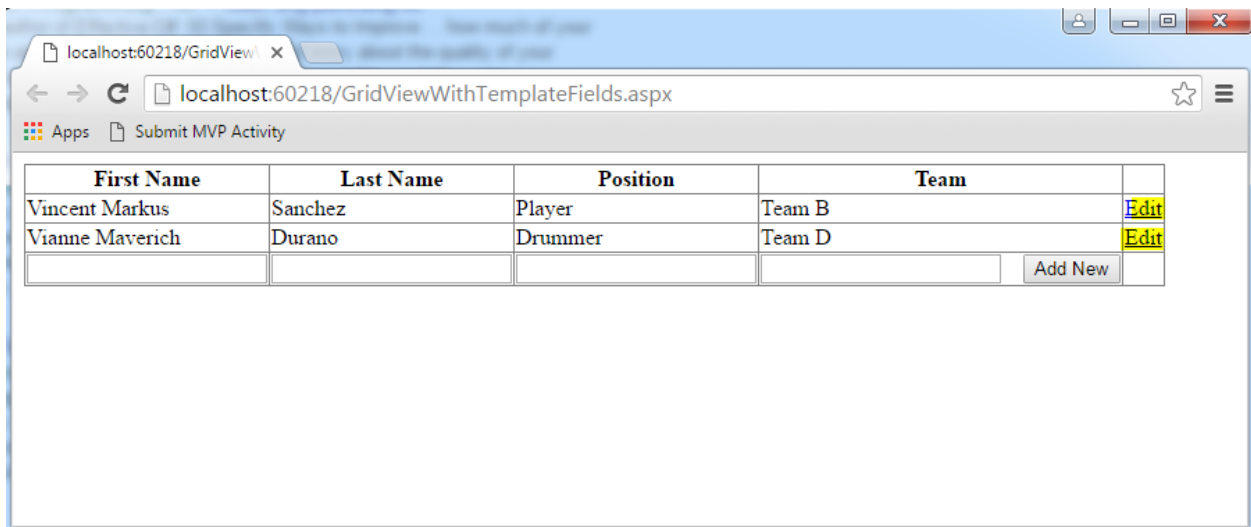
protectedvoid gvEmployee_RowUpdating(object sender, GridViewUpdateEventArgs e)
{
    if (UpdateEmployee(gvEmployee.Rows[e.RowIndex]))
        ClientScript.RegisterClientScriptBlock(typeof(Page), "ALERT", "alert('Record updated.');" , true);
    else
        ClientScript.RegisterClientScriptBlock(typeof(Page), "ALERT", "alert('All fields cannot be empty.');" , true);

    gvEmployee.EditIndex = -1; // turn to read-only mode after update
    BindGrid(); // Rebind GridView to reflect updated data
}
```

Note, the code, given above is also similar to what we did with BoundFields, except that we have used the FindControl() method for referencing the TextBox from <EditItemTemplate> nodes.

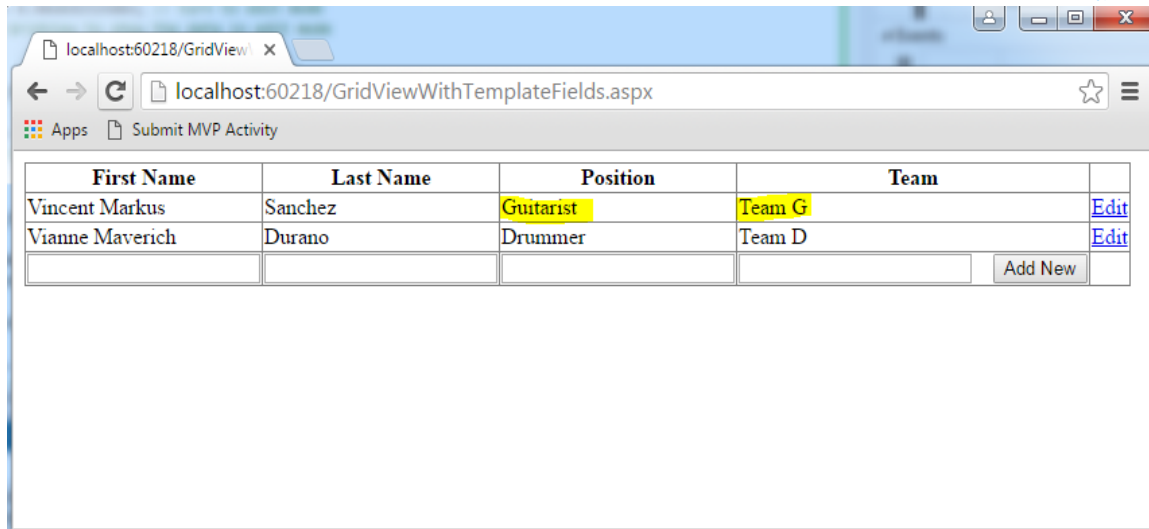
Testing the app

On initial load, the output is given below.

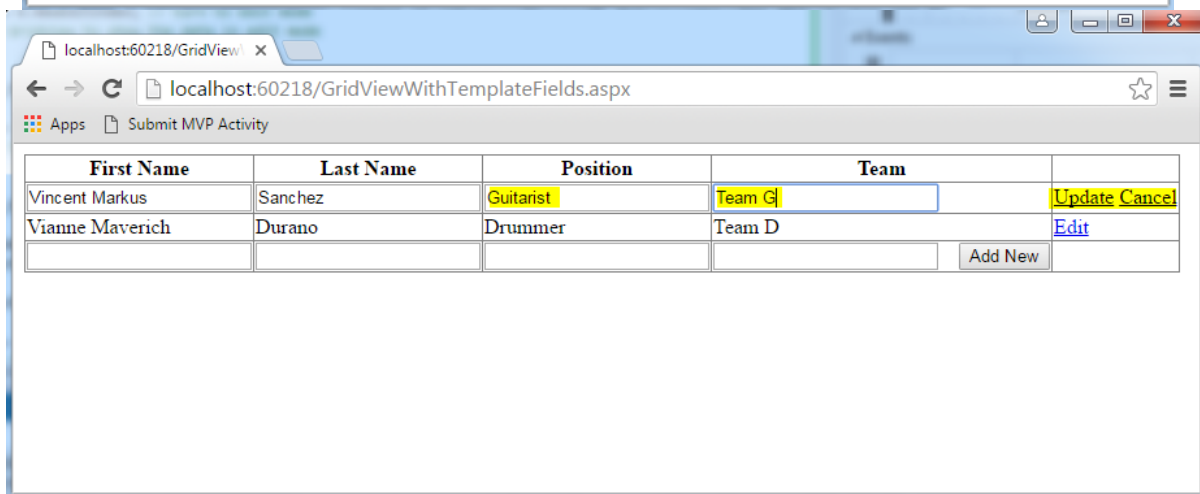


First Name	Last Name	Position	Team	
Vincent Markus	Sanchez	Player	Team B	Edit
Vianne Maverich	Durano	Drummer	Team D	Edit
				Add New

On edit, the output is given below.

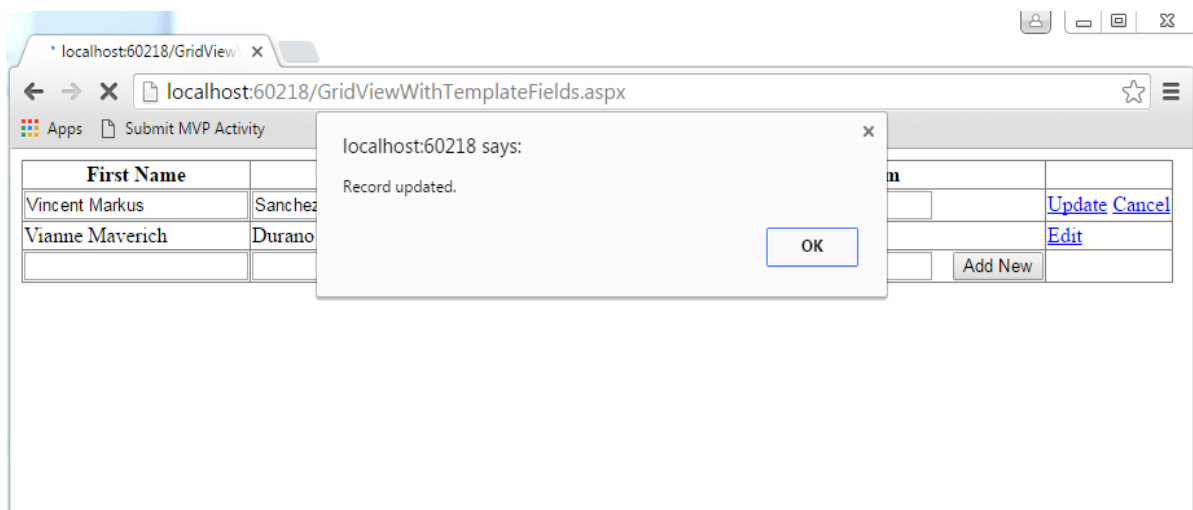


First Name	Last Name	Position	Team	
Vincent Markus	Sanchez	Guitarist	Team G	Edit
Vianne Maverich	Durano	Drummer	Team D	Edit
				Add New



First Name	Last Name	Position	Team	
Vincent Markus	Sanchez	Guitarist	Team G	Update Cancel
Vianne Maverich	Durano	Drummer	Team D	Edit
				Add New

After update, the output is given below.



First Name	Last Name	Position	Team	
Vincent Markus	Sanchez	Guitarist	Team G	Update Cancel
Vianne Maverich	Durano	Drummer	Team D	Edit
				Add New

Pretty easy, right?

Deleting GridView Rows

Now, let's implement the row deletion. First, we need to set the ShowDeleteButton CommandField to true, as given below.

```
<asp:CommandFieldShowEditButton="True"ShowDeleteButton="true"/>
```

Setup the RowDeleting event, as given below.

```
OnRowDeleting="gvEmployee_RowDeleting"
```

Now, switch to the code at the backend file of the form and add the code, given below.

```
privatebool DeleteEmployee(int id)
{
    using (SqlConnection sqlConn = newSqlConnection(GetDBConnectionString())){
        string sql = @"DELETE FROM dbo.[Employee] WHERE Id = @Id";

        using (SqlCommand sqlCmd = newSqlCommand(sql, sqlConn))
        {
            sqlConn.Open();
            sqlCmd.Parameters.AddWithValue("@Id", id);
            sqlCmd.ExecuteNonQuery();
        }
    }

    returntrue;
}

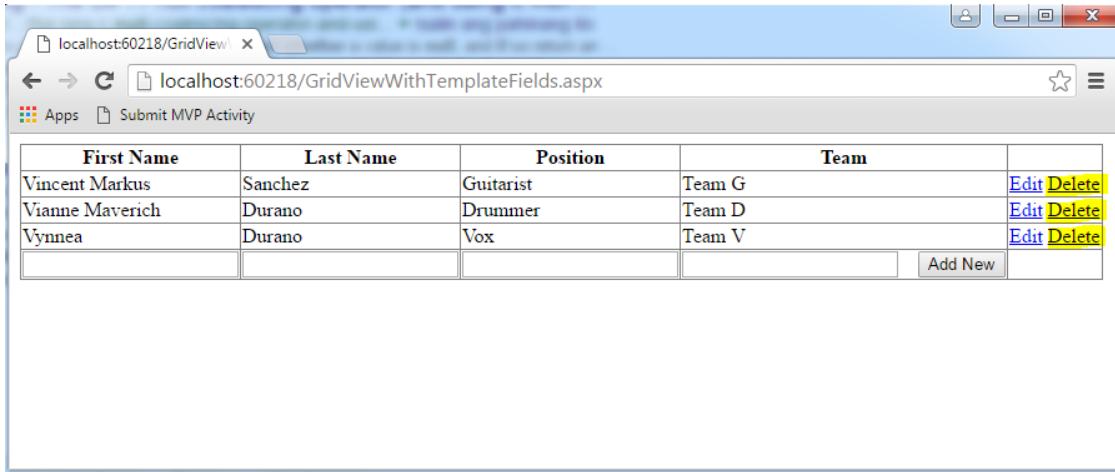
protectedvoid gvEmployee_RowDeleting(object sender, GridViewDeleteEventArgs e)
{
    int id = (int)gvEmployee.DataKeys[e.RowIndex].Value;
    if (DeleteEmployee(id))
        ClientScript.RegisterClientScriptBlock(typeof(Page), "ALERT", "alert('Record
        deleted.');
```

As you may have probably guessed already, the code, shown above is exactly the same, as we did in the previous section with BoundFields.

Testing the app

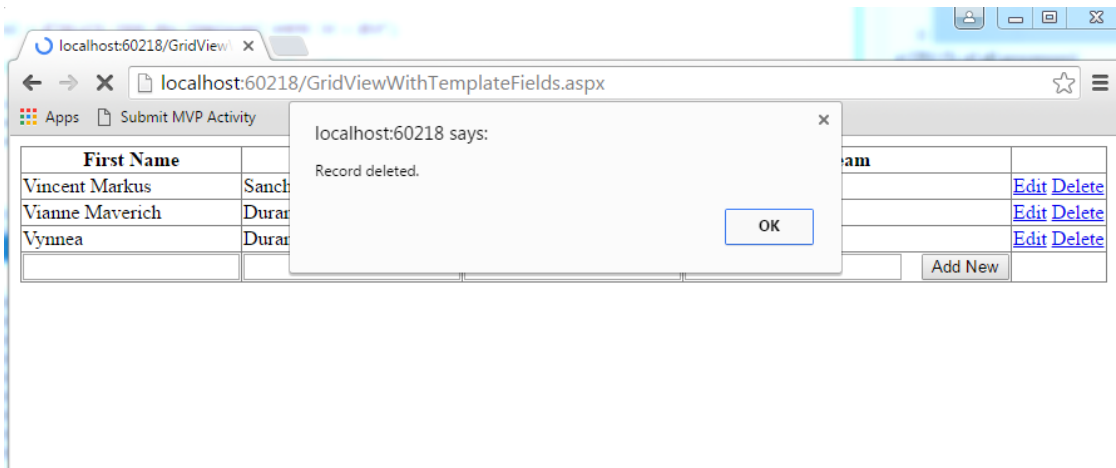
Running the code should result, as given below.

On initial load, the output is given below.

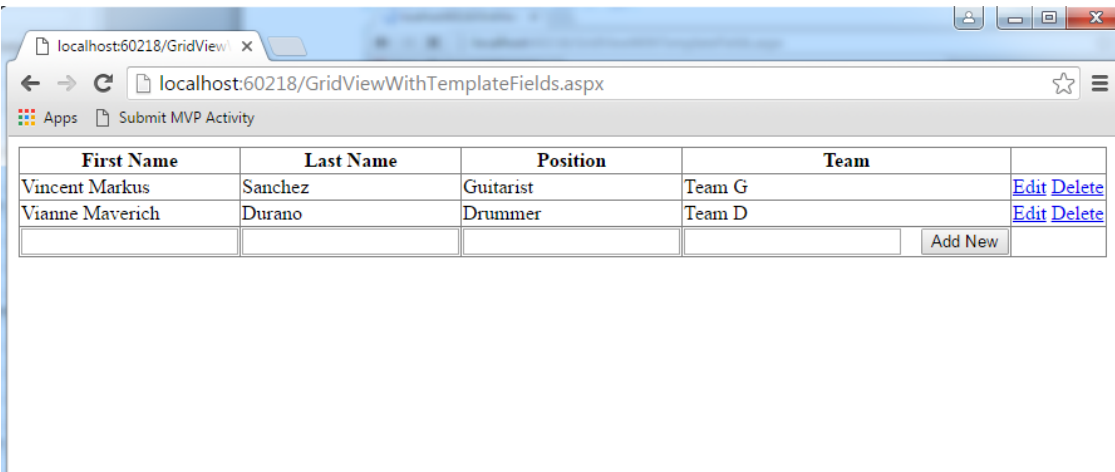


First Name	Last Name	Position	Team	
Vincent Markus	Sanchez	Guitarist	Team G	Edit Delete
Vianne Maverich	Durano	Drummer	Team D	Edit Delete
Vynnea	Durano	Vox	Team V	Edit Delete
				Add New

On delete, the output is given below.



After deletion, the output is given below.



First Name	Last Name	Position	Team	
Vincent Markus	Sanchez	Guitarist	Team G	Edit Delete
Vianne Maverich	Durano	Drummer	Team D	Edit Delete
				Add New

Display Confirmation Message on Delete

This section describes the different approaches to display a confirmation message, while deleting a row in GridView and passes the Id of the item to the confirmation message. Confirmation means, a user will be asked first, if he wants to delete a record by choosing an option to perform such an action: (OK and CANCEL).

In this demo, we will be using JavaScript confirm function to display the confirmation message.

To start, let's create our JavaScript confirm function. To do this, switch to .ASPX HTML source and add the code block, given below within the <head> tag, as given below.

After setting up our JavaScript function, we can simply call the confirmOnDelete() function in our code, while deleting a row in GridView. Here, the approaches are given below.

Using the ShowDeleteButton CommandField

If you can recall, ShowDeleteButton CommandField is a built-in command in GridView to perform delete. Here, a way is given below to attach JavaScript, while using ShowDeleteButton.

Assume, we have GridView column mark up, as given below.

```
<Columns>
<asp:BoundFieldDataField="FirstName"HeaderText="First Name"/>
<asp:BoundFieldDataField="LastName"HeaderText="Last Name"/>
<asp:BoundFieldDataField="Position"HeaderText="Position"/>
<asp:BoundFieldDataField="Team"HeaderText="Team"/>
<asp:CommandFieldShowDeleteButton="True"ShowEditButton="True"/>
</Columns>
```

At RowDataBound event, we can add an attribute to the CommandField to invoke JavaScript function, which we have just created earlier. The example is given below.

```
protectedvoid gvEmployee_RowDataBound(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowState != DataControlRowState.Edit) // check for RowState
    {
        if (e.Row.RowType == DataControlRowType.DataRow) //check for RowType
        {
            // Get the id to be deleted
            string id = gvEmployee.DataKeys[e.Row.RowIndex].Value.ToString();

            //cast the ShowDeleteButton link to linkbutton
            LinkButton lb = (LinkButton)e.Row.Cells[4].Controls[2];
            if (lb != null)
            {
                //attach the JavaScript function and pass the ID as the paramter
                lb.Attributes.Add("onclick", "return confirmOnDelete('" + id + "');");
            }
        }
    }
}
```

```

    }
}
}

```

Let see what we just did there.

The code, given above simply attached an onclick event to the LinkButton by adding the attributes to the control. Before it, we need to check the RowState of the GridView first to ensure that the GridView is not on Edit Mode. If you failed to check for its RowState, you will get an unexpected outcome. Subsequently, we checked for the RowType to ensure that we are on the right track to find the control in GridView. Please note, GridView comprises of six RowTypes namely (DataRow, EmptyDataRow, Header, Footer, Separator and Pager) so while accessing a control at RowDataBound event of GridView, you must always check for its RowType.

We have used the method Controls [index] because the CommandField columns doesn't have an ID for us to use the FindControl method. Adding to that, we cannot set an ID to a CommandField column.

Also notice, we have passed the value of 4 in the cells index. This means that the ShowDeleteButton link is located at the 5th column of the GridView (see GridView mark up, shown above), by default, cells index starts at 0.

We passed the value of 2 in the Controls index in order to get the reference to the ShowDeleteButton link. Based on GridView mark up, shown above, we can see the controls sequence of the 5th column, shown below.

e.Row.Cells[4].Controls[0] – index 0 reference to ShowEditButton link

e.Row.Cells[4].Controls[1] – index 1 reference to a Literal control that separates between the ShowEditButton and ShowDeleteButton.

e.Row.Cells[4].Controls[2] – index 2 reference to ShowDeleteButton link

e.Row.Cells[4].Controls[3] – you will get null reference exception, while passing index 3 and up, as there is no control beyond index 3.

Using TemplateField with LinkButton

Consider, we have GridView column mark-up, as given below.

```
<Columns>
<asp:BoundFieldDataField="FirstName"HeaderText="First Name"/>
<asp:BoundFieldDataField="LastName"HeaderText="Last Name"/>
<asp:BoundFieldDataField="Position"HeaderText="Position"/>
<asp:BoundFieldDataField="Team"HeaderText="Team"/>
<asp:TemplateField>
<ItemTemplate>
<asp:LinkButtonID="lnkDelete"runat="server">Delete</asp:LinkButton>
</ItemTemplate>
</asp:TemplateField>
</Columns>
```

The Declarative Approach

While using TemplateField, we can directly use the OnClientClick event of the LinkButton Server control and call JavaScript confirm function, as given below.

```
<ItemTemplate>
<asp:LinkButtonID="lnkDelete"runat="server"
OnClick="lnkDelete_Click"
OnClientClick="return confirmOnDelete('');">Delete</asp:LinkButton>
</ItemTemplate>
```

Using the approach, shown above, it does not allow us to pass the value of the Id as a parameter directly in the confirm function, due to which we passed an empty value instead.

The Code behind Approach

The Code Behind approach is almost the same, as we did using the RowDataBound event to access the ShowDeleteButton. The output is given below.

```
protectedvoid gvEmployee_RowDataBound(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowState != DataControlRowState.Edit) // check for RowState
    {
        if (e.Row.RowType == DataControlRowType.DataRow) //check for RowType
        {
            string id = gvEmployee.DataKeys[e.Row.RowIndex].Value.ToString();

            //cast the ShowDeleteButton link to linkbutton
            LinkButton lb = (LinkButton)e.Row.Cells[4].FindControl("lnkDelete");
            if (lb != null)
            {
                //attach the JavaScript function and pass the ID as the paramter
            }
        }
    }
}
```



```

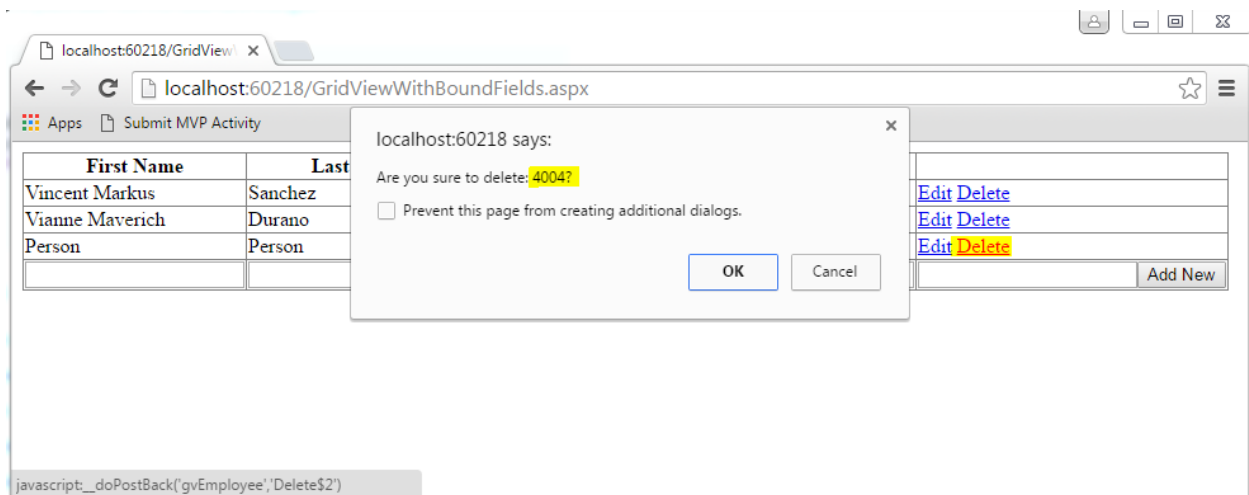
        lb.Attributes.Add("onclick", "return confirmOnDelete('" + id + "');");
    }
}
}

```

The only main difference from the above approach is we are using FindControl method for referencing the control from the TemplateField, based on the LinkButton's ID.

Testing the app

Running the code should result, as shown below.



GridView Multiple Delete with CheckBox and Confirm

This section describes how to implement multiple delete in GridView, using CheckBox control and display a confirmation message upon deletion.

Consider, we have the columns declaration, given below.

```
<Columns>
<asp:TemplateField>
<HeaderTemplate>
<asp:ButtonID="btnDelete"runat="server"Text="Delete"/>
</HeaderTemplate>
<ItemTemplate>
<asp:CheckBoxID="CheckBox1"runat="server"/>
</ItemTemplate>
</asp:TemplateField>
<asp:BoundFieldDataField="CustomerID"HeaderText="ID"ReadOnly="True"/>
<asp:BoundFieldDataField="CompanyName"HeaderText="Company"/>
<asp:BoundFieldDataField="ContactName"HeaderText="Name"/>
<asp:BoundFieldDataField="ContactTitle"HeaderText="Title"/>
<asp:BoundFieldDataField="Address"HeaderText="Address"/>
<asp:BoundFieldDataField="Country"HeaderText="Country"/>
</Columns>
```

Now, let's create the method to delete the multiple rows. First, we need to declare the namespaces, given below, in order for us to use SQLClient libraries, StringBuilder and StringCollection class.

```
using System.Data.SqlClient;
using System.Collections.Specialized;
using System.Text;
```

Here, the code block to delete the multiple records is given below.

```
privatevoid DeleteRecords(StringCollection sc)
{
    StringBuilder sb = newStringBuilder(string.Empty);
    conststring sqlStatement = "DELETE FROM Customers WHERE CustomerID";

    foreach (string item in sc)
    {
        sb.AppendFormat("{0}='{1}'; ", sqlStatement, item);
    }

    using (SqlConnection sqlConn = newSqlConnection(GetDBConnectionString()))
    {
        using (SqlCommand sqlCmd = newSqlCommand(sb.ToString(), sqlConn))
        {
            sqlConn.Open();
            sqlCmd.ExecuteNonQuery();
        }
    }
}
```

```
}
```

DeleteRecords() method takes one parameter, which is basically a string collection of ID to be deleted. The method basically iterates the entire list of IDs, which was stored in a StringCollection object. The values stored in the StringCollection will be formatted together with the value of SQLStatements string, and creates a concatenated delete statements, based on the number of ID's to be deleted. These concatenated values are then stored in the StringBuilder object and pass it in the SQLcommand as a parameter.

We have used this approach, so that we can execute multiple deletes at once and in this way, we only need to hit the database once.

Now, let's call DeleteRecords() method at the Delete Button Click event, which was located at the Header portion of the first column of GridView. In order to generate the click event automatically, you can follow the steps, given below.

1. Switch to Design View in the Visual Studio Designer.
2. Right click on the GridView and select Edit template -- > Column[0].
3. The GridView should switch to edit mode.
4. Find the Delete button in the template and double click on the button to generate the event for you.
5. Write the codes there to call the Delete method and so on.
6. To switch back to default mode in GridView, just right click on the GridView and select End template Editing.
7. You are done.

The code block for the Delete Button Click event is given below.

```
protectedvoid btnDelete_Click(object sender, EventArgs e)
{
    StringCollection sc = newStringCollection();
    string id = string.Empty;
    //loop the GridView Rows
    for (int i = 0; i < GridView1.Rows.Count; i++)
    {
        //find the CheckBox
        CheckBox cb = (CheckBox)GridView1.Rows[i].Cells[0].FindControl("CheckBox1");
        if (cb != null)
        {
            if (cb.Checked)
            {
                // get the id of the field to be deleted
                id = gvEmployee.DataKeys[i].Value.ToString();
                // add the id to be deleted in the StringCollection
                sc.Add(id);
            }
        }
    }
}
```

```

    }

    DeleteRecords(sc); // call the method for delete and pass the StringCollection values
    BindGridView(); // Bind GridView to reflect changes made here
}

```

The code, given above basically loops through the GridView rows and searches for CheckBox, which was checked, using FindControl() method. When a CheckBox is checked, it will then get the corresponding ID for the row and add it in the StringCollection object. Afterwards, it loops through all CheckBoxes, which were checked. We then invoke DeleteRecords() method to perform the deletion.

Note- Don't forget to bind your GridView with the data. Afterwards, you call the delete method to reflect the changes in GridView.

Now, let's create the confirmation function, when a user attempts to delete some records. JavaScript function is given below.

```

<head runat="server">
<title></title>
<script type="text/javascript">
function confirmOnDelete(item)
{
if (confirm("The selected item(s) will be deleted! Continue?")==true)
return true;
else
return false;
}
</script>
</head>

```

The code at the backend to invoke JavaScript function at RowDataBound event of GridView is given below.

```

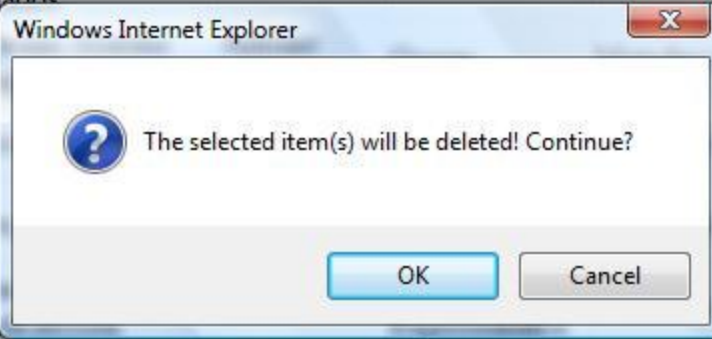
protected void GridView1_RowDataBound(object sender, GridViewRowEventArgs e)
{
if (e.Row.RowType == DataControlRowType.Header) //check for RowType
{
//access the LinkButton from the Header TemplateField using FindControl
Button b = (Button)e.Row.FindControl("btnDelete");
//attach the JavaScript function
b.Attributes.Add("onclick", "return confirmOnDelete();");
}
}

```

Testing the app

Running the app results in the output, given below.

Delete	ID	Company	Name	Title	Address	Country
<input checked="" type="checkbox"/>	ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57	Germany
<input checked="" type="checkbox"/>	ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	Avda. de la Constitución 2222	Mexico
<input type="checkbox"/>	ANTON	Antonio Taitelbaum			2312	Mexico
<input type="checkbox"/>	AROUT	Arnaud Dubois			135 rue Martenot	UK
<input checked="" type="checkbox"/>	BERGS	Bergsten			14, rue des Bouchers	Sweden
<input checked="" type="checkbox"/>	BLAUS	Blauer			57	Germany
<input type="checkbox"/>	BLONP	Blondesddsl père et fils	Frédérique Citeaux	Marketing Manager	24, place Kléber	France
<input type="checkbox"/>	BOLID	Bólido Comidas preparadas	Martin Sommer	Owner	C/ Araquil, 67	Spain
<input checked="" type="checkbox"/>	BONAP	Bon app'	Laurence Lebihan	Owner	12, rue des Bouchers	France
<input checked="" type="checkbox"/>	BOTTM	Bottom-Dollar Markets	Elizabeth Lincoln	Accounting Manager	23 Tsawassen Blvd.	Canada



Sorting GridView Manually With TemplateFields

In this section, we will take a look at how to sort GridView columns manually, using a DataTable.

To get started, let's set the GridView with TemplateField columns. Since we are working with TemplateFields, we need to handle sorting manually by adding a LinkButton control inside the HeaderTemplate of TemplateField column.

Note, for the simplicity of this demo, we will just apply sorting to one column for your reference. You can apply the same way to the other template columns in your GridView, which you want to sort, based on your requirements.

The sample GridView column mark up is given below.

```
<asp:TemplateField>
<HeaderTemplate>
```

```
<asp:LinkButtonID="LinkButtonEmpName"runat="server"Text="Employee Name"
CommandName="Sort"
CommandArgument="Employees">
</asp:LinkButton>
</HeaderTemplate>
<ItemTemplate>
<asp:LabelID="LabelEmployee"runat="server"Text='<%=# Bind("Employees") %>' />
</ItemTemplate>
<FooterTemplate>
<asp:TextBoxID="TextBoxEmployee"runat="server"/>
</FooterTemplate>
</asp:TemplateField>
```

As you can see, we have added a LinkButton under the HeaderTemplate section of the TemplateField column. Set the CommandName and CommandArgument of the LinkButton.

Since we need to perform sorting manually, then we have set the value of CommandName to “Sort” so that we can perform certain actions “like sorting” based on that value at RowCommand event later. Also be sure to set the CommandArgument value, the value of the CommandArgument should be the Field Name from your database table, which corresponds to a particular column. For an example in the above declaration, I set the CommandArgument value to “Employees”.

Now, let’s proceed to the code at the backend of the form and create the method to bind the GridView. Here, the code block is given below.

```
privatestring GetDBConnectionString()
{
returnConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString;
}

privatevoid BindGrid()
{
DataTable dt = newDataTable();
using (SqlConnection sqlConn = newSqlConnection(GetDBConnectionString()))
{
string sql = "SELECT * FROM dbo.[Employee]";
using (SqlCommand sqlCmd = newSqlCommand(sql, sqlConn))
{
sqlConn.Open();
using (SqlDataAdapter sqlAdapter = newSqlDataAdapter(sqlCmd))
{
sqlAdapter.Fill(dt);
}
}
}

if (dt.Rows.Count > 0)
{
DataView dv = dt.DefaultView;
```

```
if (this.ViewState["SortExpression"] != null)
{
    dv.Sort = string.Format("{0} {1}",
this.ViewState["SortExpression"].ToString(),
this.ViewState["SortOrder"].ToString());
}
gvEmployee.DataSource = dt;
gvEmployee.DataBind();
}
else
{
    //display no records found here
}
}

protectedvoid Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
        BindGrid();
}
```

At this point, I presume that you already know the basics of binding GridView with the data. The code, shown above is pretty much the same, as we did in the previous section for binding the GridView. Basically, what is added from the code above is the sorting functionality, using a DataView. Since a LinkButton triggers a PostBack, we have to maintain the sort expression and sort order, using ViewStates. Hence, if the sort expression is available, the DataView's sort property will be set with the ViewState's SortExpression and SortOrder values.

Now, let's implement the sorting functionality at RowCommand event. For those, who do not know how to generate the event in GridView, then you can follow the steps, given below.

1. Switch to Design View in Visual Studio Designer.
2. Click on the GridView.
3. Navigate to the GridView Property pane and then switch to the event properties.
4. From there, you should be able to find the RowCommand event.
5. Double click on it to generate the Event handler for you.
6. Write the codes.

Here, the code block is given below for the RowCommand event.

```
protectedvoid gvEmployee_RowCommand(object sender, GridViewCommandEventArgs e)
{
    if (e.CommandName.Equals("Sort"))
    {
        if (ViewState["SortExpression"] != null)
        {
            if (this.ViewState["SortExpression"].ToString() == e.CommandArgument.ToString())
```

```

    {
    if (ViewState["SortOrder"].ToString() == "ASC")
        ViewState["SortOrder"] = "DESC";
    else
        ViewState["SortOrder"] = "ASC";
    }
    else
    {
        ViewState["SortOrder"] = "ASC";
        ViewState["SortExpression"] = e.CommandArgument.ToString();
    }
    }
    else
    {
        ViewState["SortExpression"] = e.CommandArgument.ToString();
        ViewState["SortOrder"] = "ASC";
    }
    }

    //Bind the Grid to reflect the Sort Order
    BindGrid();
}

```

The code, shown above stores the current order and sort expression in ViewState object. This means, if the column was sorted in an ascending order, the new direction has to be descending.

Testing the app

Running the code, shown above will show something, as shown below.

Employee Name	Position	Team Name	Employee ID	
A	AA	AAA	3	Edit Delete
B	BB	BBB	4	Edit Delete
Then	QAE	SIT	2	Edit Delete
Vinz	SE	ACS	1	Edit Delete
X	XXX	XXX	5	Edit Delete
			<input type="button" value="Add New"/>	

After clicking the Employee Name, Header will re-sort the column and the result is given below.

Employee Name	Position	Team Name	Employee ID	
X	XXX	XXX	5	Edit Delete
Vinz	SE	ACS	1	Edit Delete
Jhen	QAE	STT	2	Edit Delete
B	BB	BBB	4	Edit Delete
A	AA	AAA	3	Edit Delete
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="button" value="Add New"/>	

Implementing Custom Paging in GridView With LINQ

Paging is very helpful to presenting a huge amount of data in the page because this helps speeding up the loading performance of the page and provides a friendlier UI to end users in terms of data presentation. For this example, I will highlight how to implement custom paging in a GridView control, using the power of LINQ and will show you some tips to maximize the performance of a paged grid.

For those, who are not familiar with LINQ, here's a short overview. Language-Integrated Query (LINQ) is a set of features introduced in .NET Framework 3.5, which extends powerful query capabilities to the language syntax of C# and Visual Basic. LINQ introduces standard, easily-learned patterns for querying and updating the data. In addition to it, the technology can be extended to support potentially any kind of data store.

To get started, let's go ahead and fire up Visual Studio. Select new Web Application / Website project. Add a new page, set up your page by adding a GridView and a Repeater control. HTML should look, as shown below.

```
<h2>GridView Custom Paging with LINQ</h2>
<asp:GridViewID="grdCustomer"runat="server"AutoGenerateColumns="false">
<Columns>
<asp:BoundFieldDataField="Company"HeaderText="Company"/>
<asp:BoundFieldDataField="Name"HeaderText="Name"/>
<asp:BoundFieldDataField="Title"HeaderText="Title"/>
<asp:BoundFieldDataField="Address"HeaderText="Address"/>
</Columns>
</asp:GridView>
<asp:RepeaterID="rptPager"runat="server">
<ItemTemplate>
<asp:LinkButtonID="lnkPage"runat="server"
Text='<%#Eval("Text") %>'
CommandArgument='<%#Eval("Value") %>'
```

```

Enabled='<%#Eval("Enabled") %>'
OnClick="Page_Changed"
ForeColor="#267CB2"
Font-Bold="true"/>
</ItemTemplate>
</asp:Repeater>

```

HTML comprises of a GridView and Repeater data representation controls. GridView is where we display the list of customer information from the database. Repeater will serve as our custom pager.

Keep in mind, I used Northwind.mdf as my database, which you can download from(<http://northwinddatabase.codeplex.com/releases/view/71634>). We will be using Entity Framework, so that we can work on the conceptual model. I will not elaborate more about the details on how to pull data from a database, using EF. If you are new to Entity Framework, you can have a look at my article, which outlined the details of EF here: <http://www.c-sharpcorner.com/article/entity-framework-basic-guide-to-perform-fetch-filter-insert-update-and-delete/>

Here, the code at the backend for the entire stuff is given below.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.UI.WebControls;

namespace WebFormsDemo
{
    public class Customer
    {
        public string Company { get; set; }
        public string Name { get; set; }
        public string Title { get; set; }
        public string Address { get; set; }
    }

    public partial class GridViewPagingWithLINQ : System.Web.UI.Page
    {
        private DB.NORTHWNDEntities northWindDB = new DB.NORTHWNDEntities();

        private List<Customer> GetCustomerEntity()
        {
            var customer = from c in northWindDB.Customers
                select new Customer
                {
                    Company = c.CompanyName,
                    Name = c.ContactName,
                    Title = c.ContactTitle,
                    Address = c.Address
                }
        }
    }
}

```

```

        };

return customer.ToList();
    }

privatevoid BindCustomerListGrid(int pageIndex)
{
    int totalRecords = GetCustomerEntity().Count;
    int pageSize = 10;
    int startRow = pageIndex * pageSize;

    grdCustomer.DataSource = GetCustomerEntity().Skip(startRow).Take(pageSize);
    grdCustomer.DataBind();

    BindPager(totalRecords, pageIndex, pageSize);
}

privatevoid BindPager(int totalRecordCount, int currentPageIndex, int pageSize)
{
    double getPageCount = (double)((decimal)totalRecordCount / (decimal)pageSize);
    int pageCount = (int)Math.Ceiling(getPageCount);
    List<ListItem> pages = new List<ListItem>();
    if (pageCount > 1)
    {
        pages.Add(newListItem("FIRST", "1", currentPageIndex > 1));
        for (int i = 1; i <= pageCount; i++)
        {
            pages.Add(newListItem(i.ToString(), i.ToString(), i !=
currentPageIndex + 1));
        }
        pages.Add(newListItem("LAST", pageCount.ToString(), currentPageIndex <
pageCount - 1));
    }

    rptPager.DataSource = pages;
    rptPager.DataBind();
}

protectedvoid Page_Changed(object sender, EventArgs e)
{
    int pageIndex = Convert.ToInt32(((sender asLinkButton).CommandArgument));
    BindCustomerListGrid(pageIndex - 1);
}

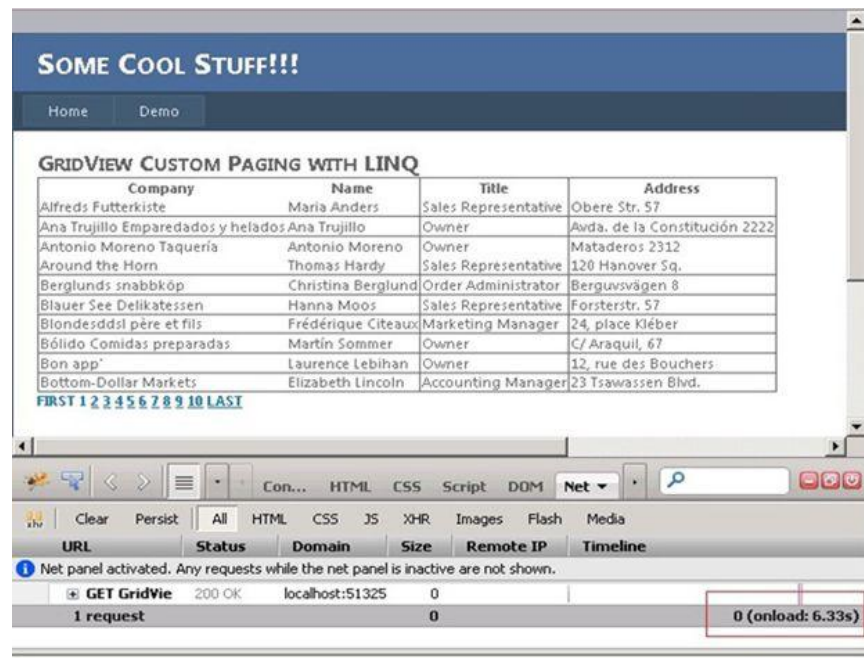
protectedvoid Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        BindCustomerListGrid(0);
    }
}
}

```

GetCustomerEntity() method is where we queried the data from the database, using LINQ syntax. The method returns a list of the customers. BindCustomerListGrid() is the method in which we bind the data to the GridView. You will see in the method that it uses the Take() and Skip() LINQ operators to chunk the data, based on the page size, which we passed in. The good thing about these operators is that it allows you to skip a certain number of rows and only take a limited number of rows from that point. For example, we set the page size to 10, which means that it will only select and display 10 records per page instead of selecting the entire results from the database that SQLDataSource is doing by default. The BindPager() method is where we construct our pager, based on the totalRecords and pageSize. The Page_Changed event handles the paging, which basically sets, what page the grid should display.

Running the code, shown above will show something, as given below.

On initial load, the output will be shown as follows.



SOME COOL STUFF!!!

Home Demo

GRIDVIEW CUSTOM PAGING WITH LINQ

Company	Name	Title	Address
Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57
Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	Avda. de la Constitución 2222
Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312
Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.
Berglunds snabbköp	Christina Berglund	Order Administrator	Berguvsvägen 8
Blauer See Delikatessen	Hanna Moos	Sales Representative	Forsterstr. 57
Blondesddsl père et fils	Frédérique Citeaux	Marketing Manager	24, place Kléber
Bólido Comidas preparadas	Martin Sommer	Owner	C/ Araquil, 67
Bon app'	Laurence Leblan	Owner	12, rue des Bouchers
Bottom-Dollar Markets	Elizabeth Lincoln	Accounting Manager	23 Tsawassen Blvd.

FIRST 1 2 3 4 5 6 7 8 9 10 LAST

GET GridVie 200 OK localhost:51325 0 0 (onload: 6.33s)

After paging, the output will be shown below.

SOME COOL STUFF!!!

Home Demo

GRIDVIEW CUSTOM PAGING WITH LINQ

Company	Name	Title	Address
La maison d'Asie	Annette Roulet	Sales Manager	1 rue Alsace-Lorraine
Laughing Bacchus Wine Cellars	Yoshi Tannamuri	Marketing Assistant	1900 Oak St.
Lazy K Kountry Store	John Steel	Marketing Manager	12 Orchestra Terrace
Lehmanns Marktstand	Renate Messner	Sales Representative	Magazimweg 7
Let's Stop N Shop	Jaime Yorres	Owner	87 Polk St. Suite 5
LILA-Supermercado	Carlos González	Accounting Manager	Carrera 52 con Ave. Bolívar #65-98 Llano Largo
LINO-Delicatesses	Felipe Izquierdo	Owner	Ave. 5 de Mayo Porlamar
Lonesome Pine Restaurant	Fran Wilson	Sales Manager	89 Chiaroscuro Rd.
Magazzini Alimentari Riuniti	Giovanni Rovelli	Marketing Manager	Via Ludovico il Moro 22
Maison Dewey	Catherine Dewey	Sales Agent	Rue Joseph-Bens 532

FIRST 1 2 3 4 5 6 7 8 9 10 LAST

Con... HTML CSS Script DOM Net

Clear Persist All HTML CSS JS XHR Images Flash Media

URL	Status	Domain	Size	Remote IP	Timeline
POST GridV	200 OK	localhost:51325	8.6 KB	127.0.0.1:51325	24ms
1 request			8.6 KB		24ms (onload: 181ms)

Using firebug, we can see the number of milliseconds; the page was rendered on an initial page load and after paging. Now, let's improve the BindCustomerListGrid() method to speed up more our paging functionality, using Application variable and Caching. Here, the modified method is given below.

```
private void BindCustomerListGrid(int pageIndex)
{
    int totalRecords = 0;
    int pageSize = 10;
    int startRow = pageIndex * pageSize;

    if (Convert.ToInt32(Application["RowCount"]) == 0)
    {
        totalRecords = GetCustomerEntity().Count();
        Application["RowCount"] = totalRecords;
    }
    else
    {
        totalRecords = Convert.ToInt32(Application["RowCount"]);
    }

    List<Customer> customerList = new List<Customer>();

    if (Cache["CustomerList"] != null)
    {
        customerList = (List<Customer>)Cache["CustomerList"];
    }
}
```

```

else
{
    customerList = GetCustomerEntity();
    Cache.Insert("CustomerList", customerList, null, DateTime.Now.AddMinutes(3),
    TimeSpan.Zero);
}

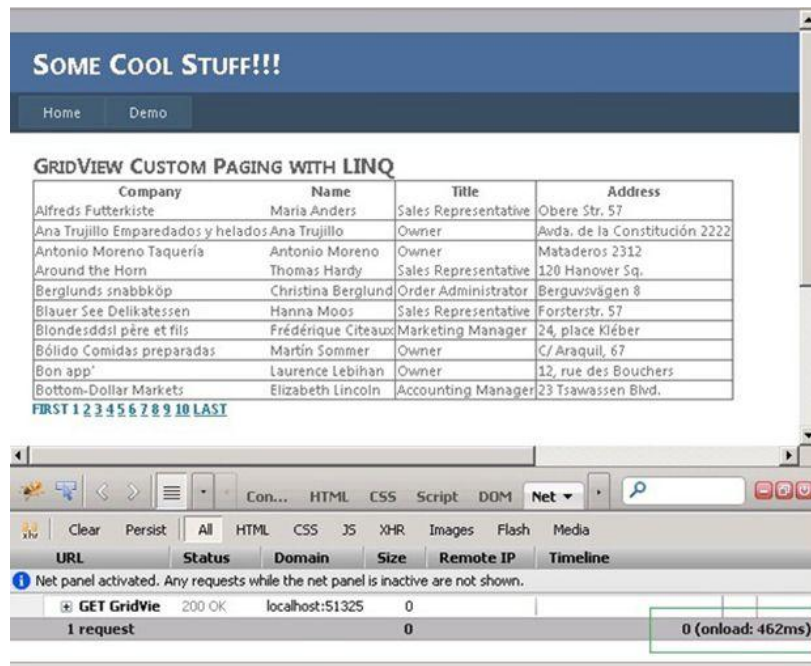
grdCustomer.DataSource = customerList.Skip(startRow).Take(pageSize);
grdCustomer.DataBind();
BindPager(totalRecords, pageIndex, pageSize);
}

```

As you can see, we store the value of totalRecords in an Application variable and use it on the subsequent request. This will minimize database calls because we will not be calling GetCustomerEntity().Count() anymore and instead use the value stored in the Application variable. Another change we made is that we store the result set in a cache, so that again we will not be hitting the database on each subsequent request. You should set the value of the Application variable to 0 and set the Cache to null in the event, where you do an update, insert or delete to refresh the values.

Running the code should now result in the following.

On initial load, the output is given below.



SOME COOL STUFF!!!

Home Demo

GRIDVIEW CUSTOM PAGING WITH LINQ

Company	Name	Title	Address
Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57
Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	Avda. de la Constitución 2222
Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312
Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.
Berglunds snabbköp	Christina Berglund	Order Administrator	Berguvsvägen 8
Blauer See Delikatessen	Hanna Moos	Sales Representative	Forsterstr. 57
Blondesddsl père et fils	Frédérique Citeaux	Marketing Manager	24, place Kléber
Bólido Comidas preparadas	Martin Sommer	Owner	C/ Araquil, 67
Bon app'	Laurence Leblan	Owner	12, rue des Bouchers
Bottom-Dollar Markets	Elizabeth Lincoln	Accounting Manager	23 Tsawassen Blvd.

FIRST 1 2 3 4 5 6 7 8 9 10 LAST

Net panel activated. Any requests while the net panel is inactive are not shown.

URL	Status	Domain	Size	Remote IP	Timeline
GET GridView	200 OK	localhost:51325	0		0 (onload: 462ms)
1 request			0		

After paging, the output is given below.

As you see, there's a big change on the performance of page loading time and on subsequent requests.

Implementing Cascading DropDownList on Edit Mode

This question has been asked many times in the various technical forums (such as in <http://forums.asp.net>) and there are definitely many solutions available. Most of the examples from the Web use DataSource controls (for example SqlDataSource, ObjectDataSource and so on) to implement a cascading DropDownList in a GridView. I can't seem to find a formal example, which shows how to implement it, using DataSource controls.

Note

Before you proceed, make sure you already understand the basics of using GridView control and how to bind it with the data, since I will not include the details of doing edit, update or delete scenarios for this exercise.

Scenario

As a recap, a Cascading DropDownList enables a common scenario in which the contents of one list depends on the selection of another list. In this demo we will show a list of Product orders in a GridView and when the user edits the row, we will present the user with a DropDownList containing the list of available makes of products. Once they have made their selection, we will populate the second DropDownList in the next column with the available Models for the specific make they've chosen. And once they have chosen the model from the second DropDownList we will then update the Price field for that corresponding model.

The HTML Markup

This section will have a look at how to possibly do this. To get started, let's setup our HTML markup (.ASPX). For the simplicity of this demo, I just set it up, as shown below.

```
<asp:GridViewID="gvProducts"runat="server"AutoGenerateColumns="false"DataKeyNames="ProductOrderID"
onrowcancelingedit="gvProducts_RowCancelingEdit"
onrowediting="gvProducts_RowEditing"
onrowupdating="gvProducts_RowUpdating"
onrowdatabound="gvProducts_RowDataBound">
<Columns>
<asp:BoundFieldDataField="ProductOrderID"ReadOnly="true"/>
<asp:TemplateFieldHeaderText="Make">
<ItemTemplate>
<asp:LabelID="lblMake"runat="server"Text="<%# Bind("Make") %>"/>
</ItemTemplate>
<EditItemTemplate>
<asp:DropDownListID="ddlProductMake"runat="server"
```



```

OnSelectedIndexChanged="ddlProductMake_SelectedIndexChanged"
AutoPostBack="true">
</asp:DropDownList>
</EditItemTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Model">
<ItemTemplate>
<asp:LabelID="lblModel"runat="server"Text='<%# Bind("Model") %>' />
</ItemTemplate>
<EditItemTemplate>
<asp:DropDownListID="ddlProductModel"runat="server"
OnSelectedIndexChanged="ddlProductModel_SelectedIndexChanged"
AutoPostBack="true">
</asp:DropDownList>
</EditItemTemplate>
</asp:TemplateField>
<asp:BoundFieldDataField="Price"HeaderText="Price"ReadOnly="True"DataFormatString="{0:c}"
/>
<asp:BoundFieldDataField="Quantity"HeaderText="Quantity"/>
<asp:CommandFieldShowEditButton="True"/>
</Columns>
</asp:GridView>

```

The markup, shown above of 6 columns of various types of control fields. The first column is a BoundField, which holds the ProductOrderID field. You may notice that we set this field as the DataKeyName in the GridView, so we can easily reference the ID of the row later. The 2nd and 3rd columns are TemplateFields, which contains a Label control within an <ItemTemplate> and a DropDownList control within an <EditItemTemplate>. This means that on a read-only state, the Label will be displayed and on an edit-state, the DropDownList will be displayed to allow the users to modify the selection of their choice. Note, you need to set AutoPostBack to TRUE for the DropDownList to trigger the SelectedIndexChanged event. The 4th and 5th columns are BoundFields, which holds the Price and Quantity fields. You may notice that the Price field has an attribute DataFormatString="{0:c}" set to it. This will transform the value to a currency format, when displayed in the Browser. Finally, the last column is a CommandField with ShowEditButton enabled. This will generate the Edit link, when the GridView is bound to the data.

The DataSource

Just for the simplicity of this exercise, I didn't use a database and instead I've just created some classes, which will hold some properties on it. Here, the class definitions are given below.

```
public class ProductMake
{
    public int ProductMakeID { get; set; }
    public string Make { get; set; }
}

public class ProductModel
{
    public int ProductModelID { get; set; }
    public int ProductMakeID { get; set; }
    public string Model { get; set; }
    public decimal Price { get; set; }
}

public class ProductOrder
{
    public int ProductOrderID { get; set; }
    public int ProductMakeID { get; set; }
    public int ProductModelID { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }
    public short Quantity { get; set; }
    public decimal Price { get; set; }
}
```

Note, this is just an example to make this exercise work. You can always replace this with your DataTable, Entity objects, Custom classes and so on, which holds your actual source of data from the database.

The ProductMake class holds two main properties. This is where we store the list of makes of the product. The ProductModel class is where we store all the models for each make. The ProductOrder class is where we store the list of orders the user has chosen. Now, let's go ahead and provide this class with some sample data. The code block is given below.

```
private List<ProductOrder> GetUserProductOrder()
{
    List<ProductOrder> orders = new List<ProductOrder>();
    ProductOrder po = new ProductOrder();

    po.ProductOrderID = 1;
    po.ProductMakeID = 1;
    po.ProductModelID = 1;
    po.Make = "Apple";
}
```

```
po.Model = "iPhone 4";
po.Quantity = 2;
po.Price = 499;
orders.Add(po);

po = new ProductOrder();
po.ProductOrderID = 2;
po.ProductMakeID = 2;
po.ProductModelID = 4;
po.Make = "Samsung";
po.Model = "Galaxy S2";
po.Quantity = 1;
po.Price = 449;
orders.Add(po);

po = new ProductOrder();
po.ProductOrderID = 3;
po.ProductMakeID = 3;
po.ProductModelID = 7;
po.Make = "Nokia";
po.Model = "Lumia";
po.Quantity = 1;
po.Price = 549;
orders.Add(po);

return orders;
}

private List<ProductMake> GetProductMakes()
{
    List<ProductMake> products = new List<ProductMake>();
    ProductMake p = new ProductMake();

    p.ProductMakeID = 1;
    p.Make = "Apple";
    products.Add(p);

    p = new ProductMake();
    p.ProductMakeID = 2;
    p.Make = "Samsung";
    products.Add(p);

    p = new ProductMake();
    p.ProductMakeID = 3;
    p.Make = "Nokia";
    products.Add(p);

    return products;
}

private List<ProductModel> GetProductModels()
{
    List<ProductModel> productModels = new List<ProductModel>();
    ProductModel pm = new ProductModel();
```

```
pm.ProductMakeID = 1;
pm.ProductModelID = 1;
pm.Model = "iPhone 4";
pm.Price = 499;
productModels.Add(pm);

pm = new ProductModel();
pm.ProductMakeID = 1;
pm.ProductModelID = 2;
pm.Model = "iPhone 4s";
pm.Price = 599;
productModels.Add(pm);

pm = new ProductModel();
pm.ProductMakeID = 1;
pm.ProductModelID = 3;
pm.Model = "iPhone 5";
pm.Price = 699;
productModels.Add(pm);

pm = new ProductModel();
pm.ProductMakeID = 2;
pm.ProductModelID = 4;
pm.Model = "Galaxy S2";
pm.Price = 449;
productModels.Add(pm);

pm = new ProductModel();
pm.ProductMakeID = 2;
pm.ProductModelID = 5;
pm.Model = "Galaxy S3";
pm.Price = 549;
productModels.Add(pm);

pm = new ProductModel();
pm.ProductMakeID = 2;
pm.ProductModelID = 6;
pm.Model = "Galaxy Note2";
pm.Price = 619;
productModels.Add(pm);

pm = new ProductModel();
pm.ProductMakeID = 3;
pm.ProductModelID = 7;
pm.Model = "Nokia Lumia";
pm.Price = 659;
productModels.Add(pm);

return productModels;
}

private List<ProductModel> GetProductModelByMake(int productMakeID)
{
var models = (from p in GetProductModels()
```

```

where p.ProductMakeID == productMakeID
select p);

return models.ToList();
}

```

GetUserProductOrder() fetches the list of the orders. We will use this as our DataSource in the GridView later. GetProductMakes() method gets all the available “makes”. In this case, we just added 3 main items to GetProductModel() method, which gets all the available models for each “make”. GetProductModelByMake() method gets the specific model item and its details, based on the ProductMakeID. This method uses LINQ syntax to query the DataSource, based on the parameter, which was passed to it.

The Implementation

Now, it looks, as if we already have some sample source of data to work with. Now, let's go ahead and do the highlight of this exercise (which is the implementation of the cascading dropdownlist). Here, the code block is given below.

```

private void BindGrid()
{
    gvProducts.DataSource = GetUserProductOrder();
    gvProducts.DataBind();
}

protected void gvProducts_RowEditing(object sender, GridViewEditEventArgs e)
{
    gvProducts.EditIndex = e.NewEditIndex;
    BindGrid();
}

protected void gvProducts_RowCancelingEdit(object sender, GridViewCancelEditEventArgs e)
{
    gvProducts.EditIndex = -1;
    BindGrid();
}

protected void gvProducts_RowDataBound(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        if ((e.Row.RowState & DataControlRowState.Edit) > 0)
        {
            DropDownList ddlMake = (DropDownList)e.Row.FindControl("ddlProductMake");
            ddlMake.DataSource = GetProductMakes();
            ddlMake.DataValueField = "ProductMakeID";
            ddlMake.DataTextField = "Make";
            ddlMake.DataBind();

            ddlMake.SelectedValue = gvProducts.DataKeys[e.Row.RowIndex].Value.ToString();
        }
    }
}

```

```

DropDownList ddlModel = (DropDownList)e.Row.FindControl("ddlProductModel");
    ddlModel.DataSource =
GetProductModelByMake(Convert.ToInt32(gvProducts.DataKeys[e.Row.RowIndex].Value));
    ddlModel.DataValueField = "ProductModelID";
    ddlModel.DataTextField = "Model";
    ddlModel.DataBind();

    ddlModel.SelectedValue =
GetProductModelByMake(Convert.ToInt32(gvProducts.DataKeys[e.Row.RowIndex].Value))
    .FirstOrDefault().ProductModelID.ToString();

    }
}
}

protectedvoid ddlProductMake_SelectedIndexChanged(object sender, EventArgs e)
{
    DropDownList ddlMake = (DropDownList)sender;
    GridViewRow row = (GridViewRow)ddlMake.NamingContainer;
    if (row != null)
    {
        if ((row.RowState & DataControlRowState.Edit) > 0)
        {
            DropDownList ddlModel = (DropDownList)row.FindControl("ddlProductModel");
            ddlModel.DataSource =
GetProductModelByMake(Convert.ToInt32(ddlMake.SelectedValue));
            ddlModel.DataValueField = "ProductModelID";
            ddlModel.DataTextField = "Model";
            ddlModel.DataBind();
        }
    }
}

protectedvoid ddlProductModel_SelectedIndexChanged(object sender, EventArgs e)
{
    DropDownList ddlModel = (DropDownList)sender;
    GridViewRow row = (GridViewRow)ddlModel.NamingContainer;
    if (row != null)
    {
        if ((row.RowState & DataControlRowState.Edit) > 0)
        {
            row.Cells[3].Text = string.Format("{0:C}", GetProductModels()
                .Where(o => o.ProductModelID ==
Convert.ToInt32(ddlModel.SelectedValue))
                .FirstOrDefault().Price);
        }
    }
}

```

The gvProducts_RowDataBound event is where we bind the DropDownList with the data from our DataSource. First, we check the RowType to ensure that we are only manipulating the rows of type DataRow. Please note, a GridView comprises of several row types such as Header,

DataRow, EmptyDataRow, Footer, Pager and Separator. The next line in code block, mentioned above is the critical part of the code and it determines the Edit state.

Accessing the controls from within <EditItemTemplate> is a bit tricky, especially if you are not really familiar with how the stuff works within a GridView. Equating the RowState to DataControlState.Edit isn't really accurate and you might get an exception, while doing so. RowState property is a bitwise combination. Therefore, the RowState can indicate that you are in the Edit state and the Alternate state. Hence, you cannot do a simple equality check, while you are in Edit mode. Instead, you must do something, given below.

```
if ((e.Row.RowState & DataControlRowState.Edit) > 0)
{
    //do your stuff here
}
```

We use the bitwise “&” operator to determine, if the GridView is in Edit mode and check the result, if it is greater than zero. For details about the Bitwise operator, see Bitwise operators in C#.

Once we have managed to determine the Edit-state, we can now begin accessing the controls, using the FindControl() method and bind it with the corresponding DataSources. If you notice, I've set the SelectedValue for the ddlMake and ddlModel DropDownLists, so that by the time the user clicks edit, the DropDownList will be pre-selected with the previous item; the user has chosen.

The ddlProductMake_SelectedIndexChanged event is where we actually do the cascading feature by populating the second DropDownList, based on the value selected from the first DropDownList but before it, we need to cast the sender of the object triggering the event to determine which DropDownList is triggered within the GridView row. We then cast the NamingContainer of the sender to a type of GridViewRow to determine the actual row; the user is editing.

The ddlProductModel_SelectedIndexChanged event is where we update the Price value, based on the selected Model from the second DropDownList. It basically uses LINQ syntax to query the DataSource and to get the Price, based on the selected value of the ddlModel.

Binding the GridView

Finally, let's call the BindGrid() method to populate the GridView. Here, the code block is given below.

```
protectedvoid Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
        BindGrid();
}
```

Testing the App

Running the code will produce the output, given below.

SOME COOL STUFF!!!					
Home Demo					
	Make	Model	Price	Quantity	
1	Apple	iPhone 4	\$499.00	2	Edit
2	Samsung	Galaxy S2	\$449.00	1	Edit
3	Nokia	Lumia	\$549.00	1	Edit

SOME COOL STUFF!!!					
Home Demo					
	Make	Model	Price	Quantity	
1	Apple ▼	iPhone 4 ▼	\$499.00	2	Update Cancel
2	Samsung	Galaxy S2	\$449.00	1	Edit
3	Nokia	Lumia	\$549.00	1	Edit

SOME COOL STUFF!!!

[Home](#)
[Demo](#)

	Make	Model	Price	Quantity	
1	Apple	iPhone 4	\$499.00	2	Update Cancel
2	Samsung	Galaxy S2	\$449.00	1	Edit
3	Nokia	Lumia	\$549.00	1	Edit

SOME COOL STUFF!!!

[Home](#)
[Demo](#)

	Make	Model	Price	Quantity	
1	Samsung	Galaxy S2	\$499.00	2	Update Cancel
2	Samsung	Galaxy S2	\$449.00	1	Edit
3	Nokia	Galaxy S3	\$549.00	1	Edit
		Galaxy Note2			

SOME COOL STUFF!!!

[Home](#)
[Demo](#)

	Make	Model	Price	Quantity	
1	Samsung	Galaxy S3	\$549.00	2	Update Cancel
2	Samsung	Galaxy S2	\$449.00	1	Edit
3	Nokia	Lumia	\$549.00	1	Edit

Move Multiple Rows Between GridViews

In this section, we'll take a look at how to move the multiple rows between GridViews. The main idea here is to use a CheckBox control to select the rows to be removed from one GridView to another and vice versa.

To start, let's build the form, as shown below.

```
<table>
<tr>
<tdvalign="top">
<asp:GridViewID="GridView3"runat="server"AutoGenerateColumns="false">
<Columns>
<asp:TemplateField>
<ItemTemplate>
<asp:CheckBoxID="CheckBox1"runat="server"/>
</ItemTemplate>
</asp:TemplateField>
<asp:BoundFieldDataField="FirstName"HeaderText="First Name"/>
<asp:BoundFieldDataField="LastName"HeaderText="Last Name"/>
<asp:BoundFieldDataField="Position"HeaderText="Position"/>
<asp:BoundFieldDataField="Team"HeaderText="Team"/>
</Columns>
</asp:GridView>
</td>
<td>
<asp:ButtonID="Button3"runat="server"Text="">>>>"onclick="Button1_Click"/>
<br/>
<asp:ButtonID="Button4"runat="server"Text=""><<<"onclick="Button2_Click"/>
</td>
<tdvalign="top">
<asp:GridViewID="GridView4"runat="server"AutoGenerateColumns="false">
<Columns>
<asp:TemplateField>
<ItemTemplate>
<asp:CheckBoxID="CheckBox2"runat="server"/>
</ItemTemplate>
</asp:TemplateField>
<asp:BoundFieldDataField="FirstName"HeaderText="First Name"/>
<asp:BoundFieldDataField="LastName"HeaderText="Last Name"/>
<asp:BoundFieldDataField="Position"HeaderText="Position"/>
<asp:BoundFieldDataField="Team"HeaderText="Team"/>
</Columns>
</asp:GridView>
</td>
</tr>
</table>
```

The markup above comprises of two identical GridViews and couple of buttons for the move operations. Now, the full code for the functionality is given below.

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Configuration;
using System.Web.UI.WebControls;

namespace WebFormsDemo
{
    public partial class MoveGridViewRows : System.Web.UI.Page
    {
        private DataTable _clonedTable;

        private string GetDBConnectionString()
        {
            return ConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString;
        }

        private DataTable GetEmployee()
        {
            DataTable dt = new DataTable();
            using (SqlConnection sqlConn = new SqlConnection(GetDBConnectionString()))
            {
                string sql = "SELECT * FROM dbo.[Employee]";
                using (SqlCommand sqlCmd = new SqlCommand(sql, sqlConn))
                {
                    sqlConn.Open();
                    using (SqlDataAdapter sqlAdapter = new SqlDataAdapter(sqlCmd))
                    {
                        sqlAdapter.Fill(dt);
                    }
                }
            }

            return dt;
        }

        private void BindGrid(DataTable dt, GridView gv)
        {
            if (dt.Rows.Count > 0)
            {
                dt = TrimEmptyRow(dt);
                gv.DataSource = SortData(dt);
                gv.DataBind();
            }
            else
            {
                ShowNoResultFound(dt, gv);
            }

            if (gv.ID == "GridView1")
                ViewState["OrigData"] = dt;
            else
                ViewState["MovedData"] = dt;
        }
    }
}

```

```

    }

    private DataTable SortData(DataTable dt)
    {
        DataView dv = dt.DefaultView;
        dv.Sort = "FirstName ASC";
        dt = dv.ToTable();
        return dt;
    }

    private void MoveRows(DataTable dt, GridView gv)
    {
        for (int i = gv.Rows.Count - 1; i >= 0; i--)
        {
            CheckBox cb = (CheckBox)gv.Rows[i].Cells[0].FindControl("CheckBox1");
            if (cb != null)
            {
                if (cb.Checked)
                {
                    AddRow(dt.Rows[i], gv);
                    dt.Rows.Remove(dt.Rows[i]);
                }
            }
        }

        BindGrid(dt, gv);
    }

    private void AddRow(DataRow row, GridView gv)
    {
        DataTable dt = null;
        if (ViewState["MovedData"] == null)
        {
            dt = (DataTable)ViewState["ClonedTable"];
            dt.ImportRow(row);
            dt.Rows.Remove(dt.Rows[0]);
            ViewState["MovedData"] = dt;
            BindGrid(dt, GridView2);
        }
        else
        {
            if (gv.ID == "GridView1")
            {
                dt = (DataTable)ViewState["MovedData"];
                dt.ImportRow(row);
                ViewState["MovedData"] = dt;
                BindGrid(dt, GridView2);
            }
            else
            {
                dt = (DataTable)ViewState["OrigData"];
                dt.ImportRow(row);
                ViewState["OrigData"] = dt;
                BindGrid(dt, GridView1);
            }
        }
    }

```

```

    }
}

private DataTable TrimEmptyRow(DataTable dt)
{
    if (dt.Rows[0][0].ToString() == string.Empty)
        dt.Rows.Remove(dt.Rows[0]);

    return dt;
}

private void ShowNoResultFound(DataTable source, GridView gv)
{
    source.Rows.Add(source.NewRow()); // create a new blank row to the DataTable

    // Bind the DataTable which contain a blank row to the GridView

    gv.DataSource = source;

    gv.DataBind();

    // Get the total number of columns in the GridView to know what the Column Span should be
    int columnsCount = gv.Columns.Count;

    gv.Rows[0].Cells.Clear(); // clear all the cells in the row

    gv.Rows[0].Cells.Add(new TableCell()); //add a new blank cell

    gv.Rows[0].Cells[0].ColumnSpan = columnsCount; //set the column span to the
    new added cell

    //You can set the styles here

    gv.Rows[0].Cells[0].HorizontalAlign = HorizontalAlign.Center;

    gv.Rows[0].Cells[0].ForeColor = System.Drawing.Color.Red;

    gv.Rows[0].Cells[0].Font.Bold = true;

    //set No Results found to the new added cell

    gv.Rows[0].Cells[0].Text = "NO ITEMS FOUND!";

}

protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        DataTable source = GetEmployee();
        BindGrid(source, GridView1);

        _clonedTable = source.Clone();
        ViewState["ClonedTable"] = _clonedTable;
    }
}

```

```

        ShowNoResultFound(_clonedTable, GridView2);
    }
}

protectedvoid Button1_Click(object sender, EventArgs e)
{
    DataTable dtOrigData = (DataTable)ViewState["OrigData"];
    MoveRows(SortData(dtOrigData), GridView1);
}

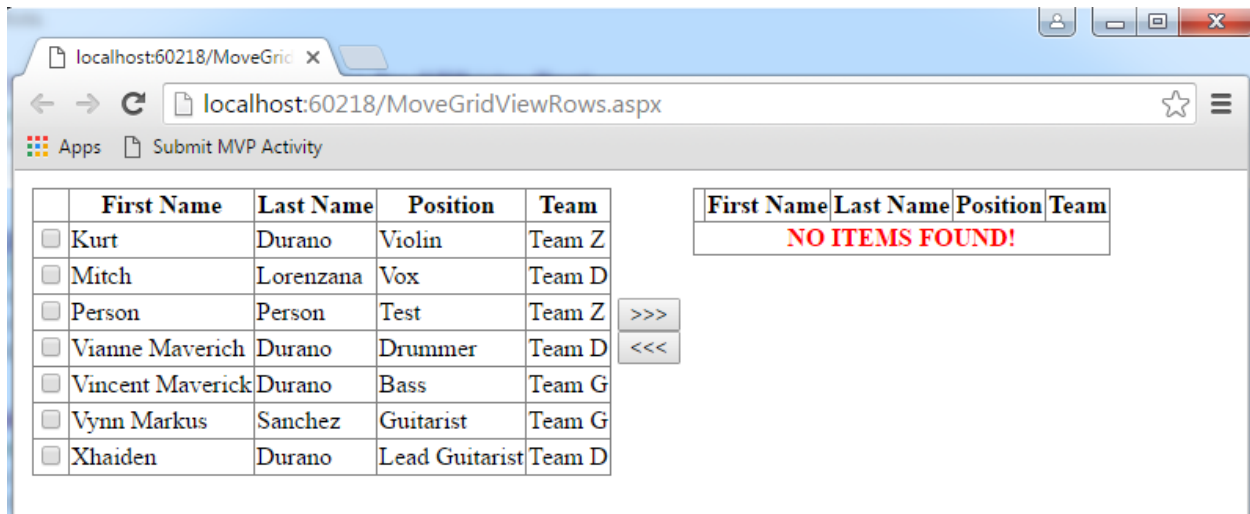
protectedvoid Button2_Click(object sender, EventArgs e)
{
    DataTable dtMovedData = (DataTable)ViewState["MovedData"];
    MoveRows(SortData(dtMovedData), GridView2);
}
}
}

```

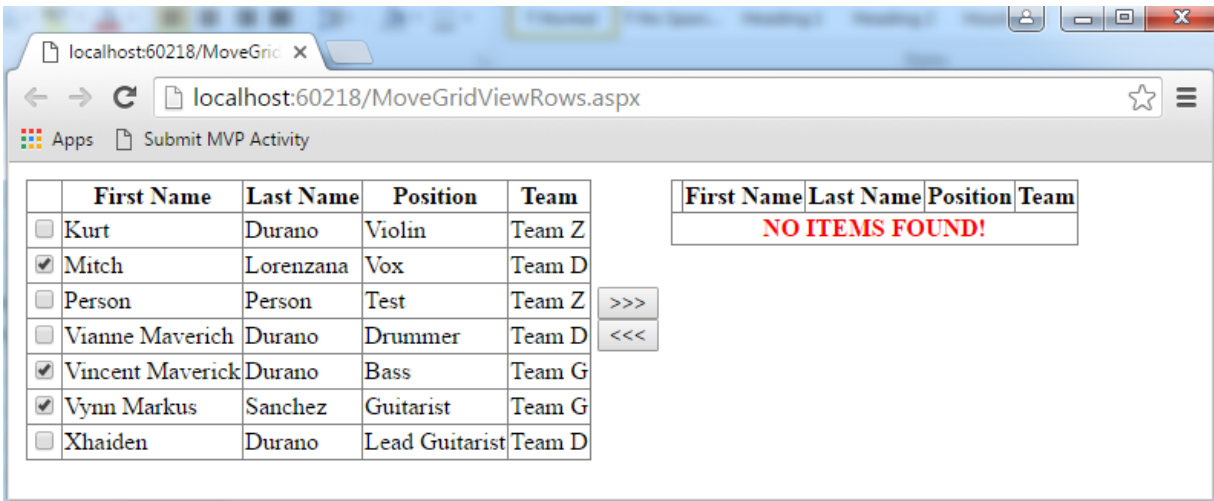
Testing the app

Running the code should produce the results, given below.

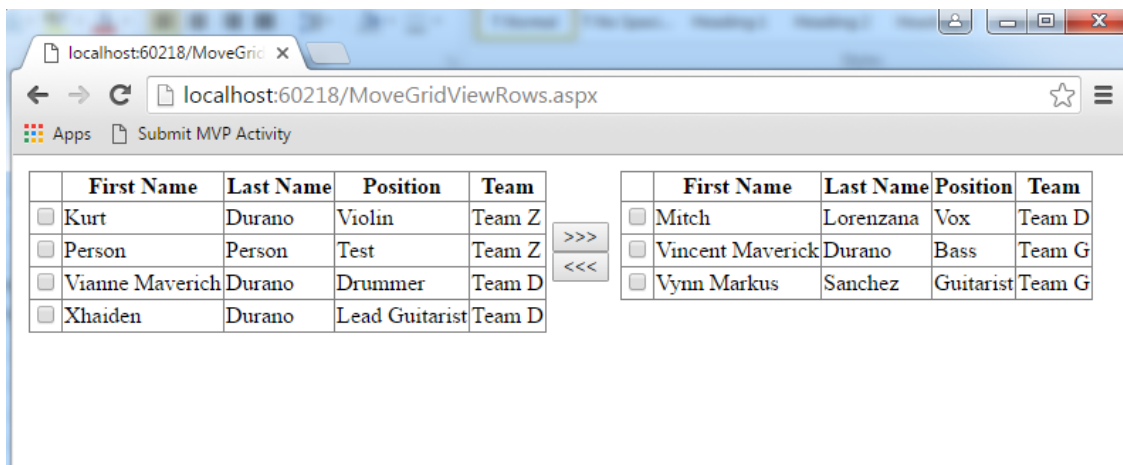
On initial load, the output is given below.



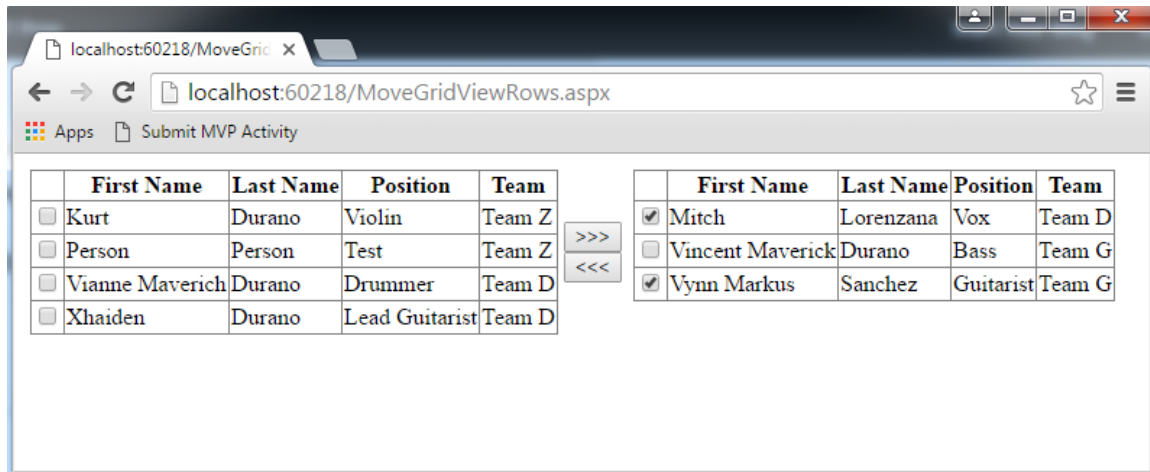
Selecting multiple rows from the left GridView



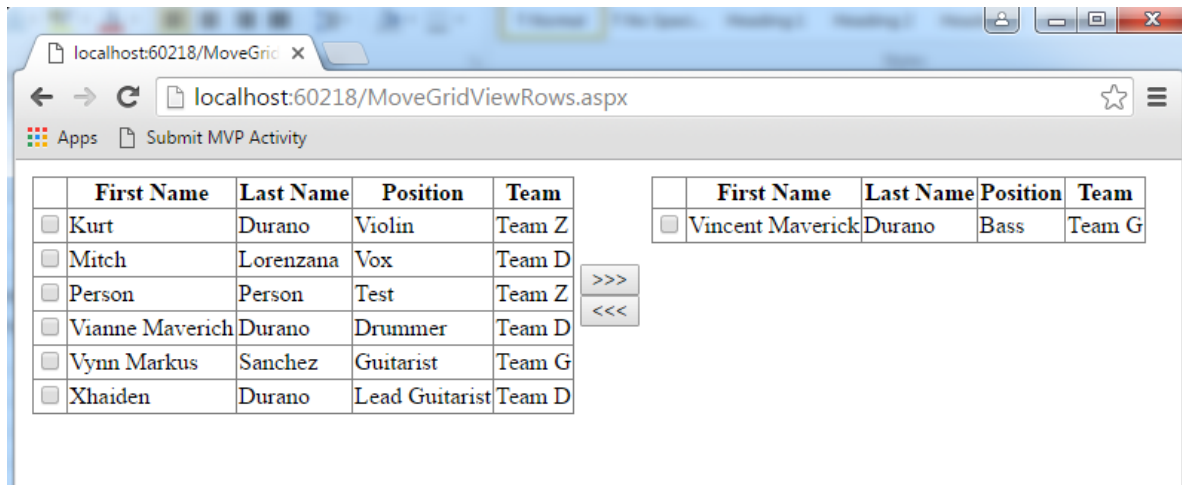
After moving the selected rows to the right GridView, the output is shown below.



Selecting rows from right GridView reveals the output as follows.



After moving the selected rows to the left GridView, the output is given below.



Pivot Data in GridView - A Generic Pivot Method with DataTable

In this section, we'll take a look at how to "PIVOT" the original data, which is displayed in the GridView.

To start, let's grab two GridViews from the Visual Studio Toolbox and place them in the form. ASPX source would look, as mentioned below.

```
<div>
    ORIGINAL Table:
    <asp:GridViewID="GridView1"runat="server">
    </asp:GridView>
    <br/><br/>
    PIVOTED Table:
    <asp:GridViewID="GridView2"runat="server"ShowHeader="false">
    </asp:GridView>
</div>
```

Now, let's create the GenericPivot method. Here, the code block is given below.

```
privateDataTable PivotTable(DataTable origTable)
{
    int counter = 1;
    DataTable newTable = newDataTable();
    DataRow dr = null;
    //Add Columns to new Table
    for (int i = 0; i <= origTable.Rows.Count; i++)
    {
        newTable.Columns.Add(newDataColumn(counter.ToString(), typeof(String)));
        counter++;
    }

    //Loop through the columns
    for (int cols = 0; cols < origTable.Columns.Count; cols++)
    {
        dr = newTable.NewRow();
        for (int rows = 0; rows < origTable.Rows.Count; rows++)
        {
            if (rows < newTable.Columns.Count)
            {
                // Add the Column Name in the first Column
                dr[0] = origTable.Columns[cols].ColumnName;
                dr[rows + 1] = origTable.Rows[rows][cols];
            }
        }
    }

    //add the DataRow to the new Table rows collection
    newTable.Rows.Add(dr);
}
```

```
//remove the ID column
    newTable.Rows[0].Delete();
return newTable;
}
```

As you have seen, the method PivotTable() returns a DataTable and basically accepts a DataTable as the parameter.

Let's bind GridViews with the original data from the database and with the Pivot data. Here, the code block is given below.

```
privatestring GetDBConnectionString()
{
returnConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString;
}

privatevoid BindGrid()
{

DataTable dt = newDataTable();
using (SqlConnection sqlConn = newSqlConnection(GetDBConnectionString()))
{
string sql = "SELECT * FROM dbo.[Employee]";
using (SqlCommand sqlCmd = newSqlCommand(sql, sqlConn))
{
sqlConn.Open();
using (SqlDataAdapter sqlAdapter = newSqlDataAdapter(sqlCmd))
{
sqlAdapter.Fill(dt);
}
}
}

if (dt.Rows.Count > 0)
{
//Bind the First GridView with the original data from the DataTable
GridView1.DataSource = dt;
GridView1.DataBind();

//Pivot the Original data from the DataTable by calling the
//method PivotTable and pass the dt as the parameter

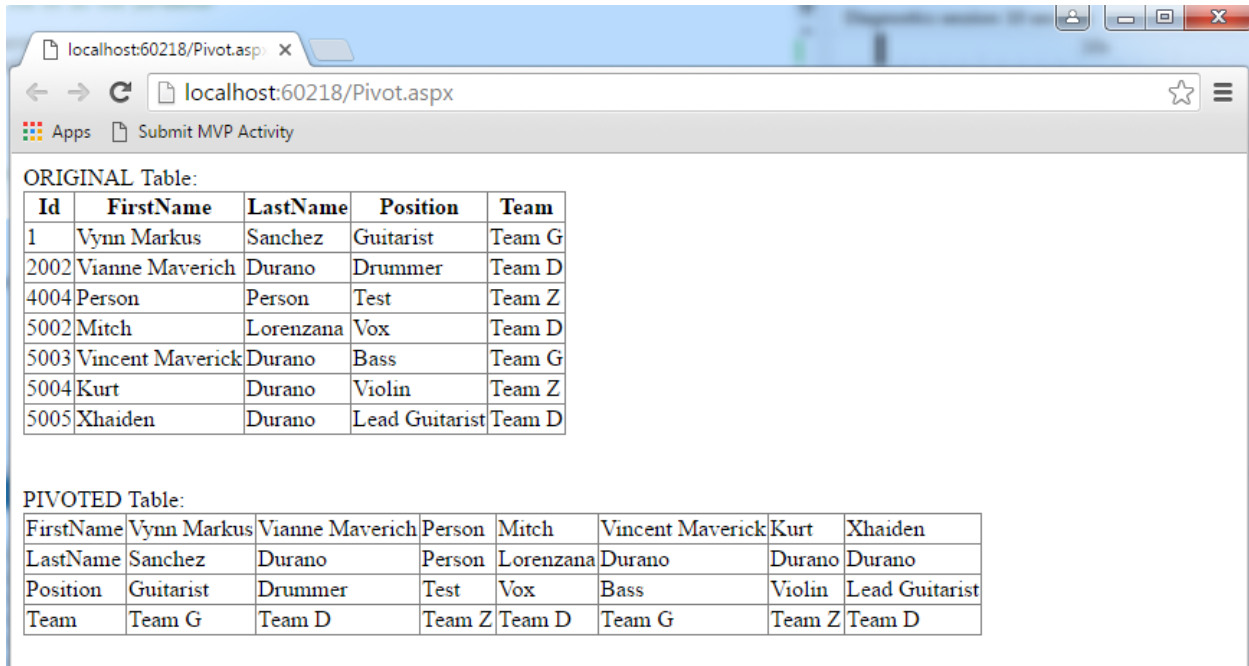
DataTable pivotedTable = PivotTable(dt);
GridView2.DataSource = pivotedTable;
GridView2.DataBind();
}
else
{
//display no records found here
}
}

protectedvoid Page_Load(object sender, EventArgs e)
```

```
{
if (!IsPostBack)
    BindGrid();
}
```

Here, the code blocks are given below.

Running the code should result to something, as given below.



localhost:60218/Pivot.aspx

Submit MVP Activity

ORIGINAL Table:

ID	FirstName	LastName	Position	Team
1	Vynn Markus	Sanchez	Guitarist	Team G
2002	Vianne Maverich	Durano	Drummer	Team D
4004	Person	Person	Test	Team Z
5002	Mitch	Lorenzana	Vox	Team D
5003	Vincent Maverick	Durano	Bass	Team G
5004	Kurt	Durano	Violin	Team Z
5005	Khaiden	Durano	Lead Guitarist	Team D

PIVOTED Table:

FirstName	Vynn Markus	Vianne Maverich	Person	Mitch	Vincent Maverick	Kurt	Khaiden
LastName	Sanchez	Durano	Person	Lorenzana	Durano	Durano	Durano
Position	Guitarist	Drummer	Test	Vox	Bass	Violin	Lead Guitarist
Team	Team G	Team D	Team Z	Team D	Team G	Team Z	Team D

Highlight Row in GridView with Colored Columns

In this section, we'll take a look at how to highlight the rows in GridView when the existing columns have color style applied to it. To make it clear, let's build a sample Application.

HTML Markup

```
<asp:GridView runat="server" id="GridView1"
    onrowcreated="GridView1_RowCreated"
    onrowdatabound="GridView1_RowDataBound">
</asp:GridView>
```

The Code Behind

```
private DataTable FillData()
{
    DataTable dt = new DataTable();
    DataRow dr = null;

    //Create DataTable columns
    dt.Columns.Add(new DataColumn("RowNumber", typeof(string)));
    dt.Columns.Add(new DataColumn("Col1", typeof(string)));
    dt.Columns.Add(new DataColumn("Col2", typeof(string)));
    dt.Columns.Add(new DataColumn("Col3", typeof(string)));

    //Create Row for each columns
    dr = dt.NewRow();
    dr["RowNumber"] = 1;
    dr["Col1"] = "A";
    dr["Col2"] = "B";
    dr["Col3"] = "C";
    dt.Rows.Add(dr);

    dr = dt.NewRow();
    dr["RowNumber"] = 2;
    dr["Col1"] = "AA";
    dr["Col2"] = "BB";
    dr["Col3"] = "CC";
    dt.Rows.Add(dr);

    dr = dt.NewRow();
    dr["RowNumber"] = 3;
    dr["Col1"] = "A";
    dr["Col2"] = "B";
    dr["Col3"] = "CC";
    dt.Rows.Add(dr);

    dr = dt.NewRow();
    dr["RowNumber"] = 4;
    dr["Col1"] = "A";
```

```

dr["Col2"] = "B";
dr["Col3"] = "CC";
dt.Rows.Add(dr);

dr = dt.NewRow();
dr["RowNumber"] = 5;
dr["Col1"] = "A";
dr["Col2"] = "B";
dr["Col3"] = "CC";
dt.Rows.Add(dr);

return dt;
}

protectedvoid Page_Load(object sender, EventArgs e)
{
if (!IsPostBack)
{
    GridView1.DataSource = FillData();
    GridView1.DataBind();
}
}

```

There's nothing really fancy from the code, mentioned above. It just contains a method, which fills a DataTable with a dummy data in it. Now, the code for row highlighting is given below.

```

protectedvoid GridView1_RowCreated(object sender, GridViewRowEventArgs e)
{
//Set Background Color for Columns 1 and 3
e.Row.Cells[1].BackColor = System.Drawing.Color.Beige;
e.Row.Cells[3].BackColor = System.Drawing.Color.Red;

//Attach onmouseover and onmouseout for row highlighting
e.Row.Attributes.Add("onmouseover", "this.style.backgroundColor='Blue'");
e.Row.Attributes.Add("onmouseout", "this.style.backgroundColor=''");
}

```

Running the code, mentioned above will reveal the output, as given below in the Browser.

On initial load

PROUDMONKEY

[Home](#)
[Demo](#)

GRIDVIEW SAMPLE DEMO

RowNumber	Col1	Col2	Col3
1	A	B	C
2	AA	BB	CC
3	A	B	CC
4	A	B	CC
5	A	B	CC

On mouseover of GridView row

PROUDMONKEY

[Home](#)
[Demo](#)

GRIDVIEW SAMPLE DEMO

RowNumber	Col1	Col2	Col3
1	A	B	C
2	AA	BB	CC
3	A	B	CC
4	A	B	CC
5	A	B	CC

Notice, Col1 and Col3 were not highlighted. Why? The reason is that Col1 and Col3 cells have a background color set on it. We only highlight the rows (TR) and not the columns (TD) due to which on mouseover, only the rows will be highlighted. To fix the issue, we will create a JavaScript method, which would remove the background color of the columns, while highlighting a row and on mouseout, set back the original color, which is set on Col1 and Col3. Here, the codes are given below.

The JavaScript Code

```
<script type="text/javascript">
function HighlightRow(rowIndex, colIndex, colIndex2, flag) {
var gv = document.getElementById("<%= GridView1.ClientID %>");
var selRow = gv.rows[rowIndex];
if (rowIndex > 0) {
if (flag == "sel") {
    gv.rows[rowIndex].style.backgroundColor = 'Blue';
    gv.rows[rowIndex].style.color = "White";
    gv.rows[rowIndex].cells[colIndex].style.backgroundColor = '';
    gv.rows[rowIndex].cells[colIndex2].style.backgroundColor = '';
}
else {
    gv.rows[rowIndex].style.backgroundColor = '';
    gv.rows[rowIndex].style.color = "Black";
    gv.rows[rowIndex].cells[colIndex].style.backgroundColor = 'Beige';
    gv.rows[rowIndex].cells[colIndex2].style.backgroundColor = 'Red';
}
}
}
</script>
```

The HighlightRow() method is a JavaScript function, which accepts four (4) parameters- the rowIndex, colIndex, colIndex2 and the flag. The rowIndex is the current row index of the selected row in GridView. The colIndex is the index of Col1 and colIndex2 is the index of col3. We are passing these indexes because these columns have background color on it and we need to toggle its backgroundColor, while highlighting the row in GridView. Finally, the flag is something, which would determine, if it's selected or not. Here, the code for calling JavaScript function, given above.

```
protected void GridView1_RowCreated(object sender, GridViewRowEventArgs e)
{
    //Set Background Color for Columns 1 and 3
    e.Row.Cells[1].BackColor = System.Drawing.Color.Beige;
    e.Row.Cells[3].BackColor = System.Drawing.Color.Red;

    //Attach onmouseover and onmouseout for row highlighting
    //and call the HighlightRow method with the required parameters
    int index = e.Row.RowIndex + 1;
    e.Row.Attributes.Add("onmouseover", "HighlightRow(" + index + "," + 1 + "," + 3 +
    "','sel')");
    e.Row.Attributes.Add("onmouseout", "HighlightRow(" + index + "," + 1 + "," + 3 +
    "','dsel')");
}
```

Running the code, given above will result to something, as given below.

PROUDMONKEY

Home
Demo

GRIDVIEW SAMPLE DEMO

RowNumber	Col1	Col2	Col3
1	A	B	C
2	AA	BB	CC
3	A	B	CC
4	A	B	CC
5	A	B	CC

Highlight GridView Row On Click And Retain Selected Row On Postback

In this section, we'll take a look at how to highlight a row in GridView and retain the selected row across Postbacks.

In this demo, we're going to use a combination of plain JavaScript and jQuery to do the client-side manipulation. I presume that you already know how to bind the grid with the data because I will not include the codes for populating the GridView. For binding a GridView, please refer the previous section of this book.

For simplicity, we'll just set up the page, as shown below.

```
<h2>You have selected Row: (<asp:LabelID="Label1"runat="server"/>)</h2>
<asp:HiddenFieldID="hfCurrentRowIndex"runat="server"></asp:HiddenField>
<asp:HiddenFieldID="hfParentContainer"runat="server"></asp:HiddenField>
<asp:ButtonID="Button1"runat="server"onclick="Button1_Click"Text="Trigger Postback"/>
<asp:GridViewID="grdCustomer"runat="server"AutoGenerateColumns="false"onrowdatabound="grd
Customer_RowDataBound">
<Columns>
<asp:BoundFieldDataField="Company"HeaderText="Company"/>
<asp:BoundFieldDataField="Name"HeaderText="Name"/>
<asp:BoundFieldDataField="Title"HeaderText="Title"/>
<asp:BoundFieldDataField="Address"HeaderText="Address"/>
</Columns>
</asp:GridView>
```

Note

Since the action is done at the client-side, when we do a Postback (clicking on a button), the page will be re-created and you will lose the highlighted row. This is normal because the Server doesn't know anything about the Client/Browser, unless you do something to notify the Server that something has changed. To retain the settings, we will use some HiddenFields control to store the data, so that when it does Postback, we can reference the value from there.

Here are the JavaScript functions, given below.

```
<scriptsrc="http://ajax.googleapis.com/ajax/libs/jquery/1.4/jquery.min.js" type="text/java
script">
</script>
<scripttype="text/javascript">
var prevRowIndex;

function ChangeRowColor(row, rowIndex)
{
var parent = document.getElementById(row);
var currentRowIndex = parseInt(rowIndex) + 1;

if (prevRowIndex == currentRowIndex)
{
return;
} elseif (prevRowIndex != null)
{
parent.rows[prevRowIndex].style.backgroundColor = "#FFFFFF";
}

parent.rows[currentRowIndex].style.backgroundColor = "#FFFD66";
prevRowIndex = currentRowIndex;

$('#<%= Label1.ClientID %>').text(currentRowIndex);

$('#<%= hfParentContainer.ClientID %>').val(row);
$('#<%= hfCurrentRowIndex.ClientID %>').val(rowIndex);
}

$(function()
{
RetainSelectedRow();
});

function RetainSelectedRow()
{
var parent = $('#<%= hfParentContainer.ClientID %>').val();
var currentIndex = $('#<%= hfCurrentRowIndex.ClientID %>').val();
if (parent != null)
{
ChangeRowColor(parent, currentIndex);
}
}
</script>
```

The `ChangeRowColor()` is a function that sets the background color of the selected row. It is also, where we set the previous row and `RowIndex` values in `HiddenFields`. The `$(function(){});` is a short-hand for the jQuery `document.ready` event. This event will be fired, once the page is posted back to the Server due to which we called the function `RetainSelectedRow()`. The `RetainSelectedRow()` function is where we referenced the current selected values stored from the `HiddenFields` and passed these values to the `ChangeRowColor()` function to retain the highlighted row.

Finally, the code at the backend part is given below.

```
protectedvoid grdCustomer_RowDataBound(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        e.Row.Attributes.Add("onclick", string.Format("ChangeRowColor('{0}','{1}');",
            e.Row.ClientID,
            e.Row.RowIndex));
    }
}
```

The code given above is responsible for attaching JavaScript on click event for each row, calling the `ChangeRowColor()` function, passing the `e.Row.ClientID` and `e.Row.RowIndex` to the function.

Here, the sample output is given below.

SOME COOL STUFF!!!

[Home](#)
[Demo](#)

YOU HAVE SELECTED ROW: (5)

Trigger Postback

Company	Name	Title	Address
Gourmet Lanchonetes	André Fonseca	Sales Associate	Av. Brasil, 442
Great Lakes Food Market	Howard Snyder	Marketing Manager	2732 Baker Blvd.
GROSELLA-Restaurante	Manuel Pereira	Owner	5ª Ave. Los Palos Grandes
Hanari Carnes	Mario Pontes	Accounting Manager	Rua do Paço, 67
HILARION-Abastos	Carlos Hernández	Sales Representative	Carrera 22 con Ave. Carlos Soublette #8-35
Hungry Coyote Import Store	Yoshi Latimer	Sales Representative	City Center Plaza 516 Main St.
Hungry Owl All-Night Grocers	Patricia McKenna	Sales Associate	8 Johnstown Road
Island Trading	Helen Bennett	Marketing Manager	Garden House Crowther Way
Königlich Essen	Philip Cramer	Sales Associate	Maubelstr. 90
La corne d'abondance	Daniel Tonini	Sales Representative	67, avenue de l'Europe

FIRST 1 2 3 4 5 6 7 8 9 10 LAST

Using Radio Button in GridView With JavaScript Validation

As you may know, setting the group name attribute of a radio button will not work, if the radio button is located within a data representation control like a GridView. This is because the radio button inside a GridView behaves differently. Since a GridView is rendered as a table element, at run time it will assign a different "name" to each radio button. Hence, you are able to select multiple rows.

In this Section, I'm going to show how to select one radio button at a time in GridView and add a simple validation to it, using plain JavaScript. To get started, let's go ahead and fire up Visual Studio and then create a new Web Application/ Website project. Add a WebForm and then grab GridView. The mark up would look something, as given below.

```
<asp:GridViewID="GridView1"runat="server"AutoGenerateColumns="false">
<Columns>
<asp:TemplateField>
<ItemTemplate>
<asp:RadioButtonID="rb"runat="server"/>
</ItemTemplate>
</asp:TemplateField>
<asp:BoundFieldDataField="RowNumber"HeaderText="Row Number"/>
<asp:BoundFieldDataField="Col1"HeaderText="First Column"/>
<asp:BoundFieldDataField="Col2"HeaderText="Second Column"/>
</Columns>
</asp:GridView>
```

If you notice, we have added a templatefield column, so that we can add the radio button there. Also, I have set up some BoundField columns and set the DataFields as a RowNumber, Col1 and Col2. These columns are just dummy columns and are used for the simplicity of this example. Now, where do these columns come from? These columns are created by hand at the code backend file of ASPX. Here, the code is given below.

```
privateDataTable FillData()
{
    DataTable dt = newDataTable();
    DataRow dr = null;

    //Create DataTable columns
    dt.Columns.Add(newDataColumn("RowNumber", typeof(string)));
    dt.Columns.Add(newDataColumn("Col1", typeof(string)));
    dt.Columns.Add(newDataColumn("Col2", typeof(string)));

    //Create Row for each columns
    dr = dt.NewRow();
    dr["RowNumber"] = 1;
    dr["Col1"] = "A";
    dr["Col2"] = "B";
    dt.Rows.Add(dr);
}
```

```

dr = dt.NewRow();
dr["RowNumber"] = 2;
dr["Col1"] = "AA";
dr["Col2"] = "BB";
dt.Rows.Add(dr);

dr = dt.NewRow();
dr["RowNumber"] = 3;
dr["Col1"] = "A";
dr["Col2"] = "B";
dt.Rows.Add(dr);

dr = dt.NewRow();
dr["RowNumber"] = 4;
dr["Col1"] = "A";
dr["Col2"] = "B";
dt.Rows.Add(dr);

dr = dt.NewRow();
dr["RowNumber"] = 5;
dr["Col1"] = "A";
dr["Col2"] = "B";
dt.Rows.Add(dr);

return dt;
}

```

The code for binding the GridView with the dummy data is mentioned.

```

protectedvoid Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        GridView1.DataSource = FillData();
        GridView1.DataBind();
    }
}

```

We now have a GridView data with a radio button on each row. Now, let's go ahead and switch back to ASPX mark up. In this example, I'm going to use a JavaScript for validating the radio button to select one radio button at a time. (You could also use jQuery to take advantage of the jQuery selectors for easy DOM manipulation.) Here, JavaScript code is given below.

```

<script>
function CheckOtherIsCheckedByGVID(rb) {
    var isChecked = rb.checked;
    var row = rb.parentNode.parentNode;
    if (isChecked) {
        row.style.backgroundColor = '#B6C4DE';
        row.style.color = 'black';
    }
}
var currentRdbID = rb.id;
parent = document.getElementById("<%= GridView1.ClientID %>");
var items = parent.getElementsByTagName('input');

```

```
for (i = 0; i < items.length; i++) {
if (items[i].id != currentRdbID && items[i].type == "radio") {
if (items[i].checked) {
items[i].checked = false;
items[i].parentNode.parentNode.style.backgroundColor = 'white';
items[i].parentNode.parentNode.style.color = '#696969';
}
}
}
}
</script>
```

The function, mentioned above sets the row of the current selected radio button's style to determine that the row is selected and then loops through the radio buttons in the GridView, de-select the previous selected radio button and set the row style back to its default. The next thing is to call JavaScript function, mentioned above in the onclick event of the radio button, mentioned below.

```
<asp:RadioButtonID="rb"runat="server"
onclick="javascript:CheckOtherIsCheckedByGVID(this);"/>
```

Here, the output is given below.

On Load

PROUDMONKEY			
Home Demo			
	Row Number	First Column	Second Column
<input type="radio"/>	1	A	B
<input type="radio"/>	2	AA	BB
<input type="radio"/>	3	A	B
<input type="radio"/>	4	A	B
<input type="radio"/>	5	A	B

After selecting a Radio Button, the output is given below.

PROUDMONKEY

Home

Demo

	Row Number	First Column	Second Column
<input type="radio"/>	1	A	B
<input checked="" type="radio"/>	2	AA	BB
<input type="radio"/>	3	A	B
<input type="radio"/>	4	A	B
<input type="radio"/>	5	A	B

As you can see, on initial load, there's no default selected radio button in the GridView. Now, let's add a simple validation for it. We will basically display an error message, if a user clicks a button, which triggers a Postback without selecting a radio button in the GridView. Here, JavaScript for the validation is given below.

```
function ValidateRadioButton(sender, args) {
var gv = document.getElementById("<%= GridView1.ClientID %>");
var items = gv.getElementsByTagName('input');
for (var i = 0; i < items.length; i++) {
if (items[i].type == "radio") {
if (items[i].checked) {
args.IsValid = true;
return;
}
else {
args.IsValid = false;
}
}
}
}
```

The function, mentioned above loops through the rows in GridView and find all the radio buttons within it. Subsequently, it will check each radio button's checked property. If a radio is checked, set IsValid to true else set it to false. The reason why I'm using IsValid is because I'm using ASP validator control for validation. Now, add the markup, given below under the GridView declaration.

```
<br/>
<asp:LabelID="lblMessage"runat="server"/>
<br/>
<asp:ButtonID="btn"runat="server"Text="POST"onclick="btn_Click"
```

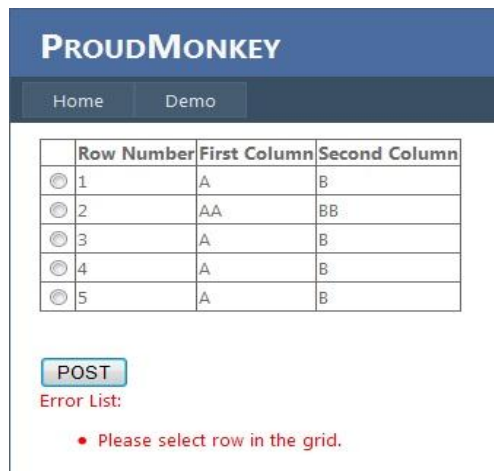
```
ValidationGroup="GroupA"/>
<asp:CustomValidatorID="CustomValidator1"runat="server"
ErrorMessage="Please select row in the grid."
ClientValidationFunction="ValidateRadioButton"
ValidationGroup="GroupA"style="display:none"/>
<asp:ValidationSummaryID="ValidationSummary1"runat="server"
ValidationGroup="GroupA"
HeaderText="Error List:"
DisplayMode="BulletList"
ForeColor="Red"/>
```

At the button's click event, add simple code, given below to test if the validation works.

```
protectedvoid btn_Click(object sender, EventArgs e)
{
    lblMessage.Text = "Postback at: " + DateTime.Now.ToString("hh:mm:ss tt");
}
```

Running the page will show something, given below in the Browser.

When the validation triggers, the output is given below.



PROUDMONKEY

Home Demo

	Row Number	First Column	Second Column
<input type="radio"/>	1	A	B
<input type="radio"/>	2	AA	BB
<input type="radio"/>	3	A	B
<input type="radio"/>	4	A	B
<input type="radio"/>	5	A	B

Error List:

- Please select row in the grid.

When, it is successful, the output is ven below.

PROUDMONKEY

Home
Demo

	Row Number	First Column	Second Column
<input type="radio"/>	1	A	B
<input type="radio"/>	2	AA	BB
<input checked="" type="radio"/>	3	A	B
<input type="radio"/>	4	A	B
<input type="radio"/>	5	A	B

Postback at: 05:10:03 PM

How To: Do Calculations in GridView

I've seen many fresh developers in various forums struggling at implementing a simple calculation in GridView. Usually, they wanted to calculate the total amount, while typing the value into a TextBox control and display the grand total value at the Footer. As you may already know, this can be easily done using a Server-side approach

Let's take a quick review on how to implement a Server-side approach calculation. Consider, we have GridView markup, given below.

Server-Side Approach

Suppose, we have GridView markup, given below.

```
<asp:GridViewID="GridView1"runat="server"
AutoGenerateColumns="false"
ShowFooter="true">
<Columns>
<asp:BoundFieldDataField="ItemDescription"HeaderText="Item"/>
<asp:TemplateFieldHeaderText="Amount">
<ItemTemplate>
<asp:TextBoxID="TextBox1"runat="server"
AutoPostBack="True"
onTextChanged="TextBox1_TextChanged">
</asp:TextBox>
</ItemTemplate>
</asp:TemplateField>
</Columns>
</asp:GridView>
```


The markup, mentioned above is pretty much simple. A GridView contains a BoundField column to display the item and a TemplateField for the amount column. Now, let's populate the grid with data from the database, using ADO.NET. The code block is given below.

```
private string GetConnectionString()
{
    //Where MYDBConnection is the connection string that was set up from the web config file
    return System.Configuration.ConfigurationManager.ConnectionStrings
        ["MyDBConnection"].ConnectionString;
}

// Method for Binding the GridView Control
private void BindGridView()
{
    using (SqlConnection connection = new SqlConnection(GetConnectionString()))
    {
        string sql = "SELECT * FROM YourTable";
        using (SqlCommand cmd = new SqlCommand(sql, connection))
        {
            connection.Open();
            using (var adapter = new SqlDataAdapter(cmd))
            {
                adapter.Fill(dt)
            }
            if (dt.Rows.Count > 0)
            {
                GridView1.DataSource = dt;
                GridView1.DataBind();
            }
        }
    }
}

//Bind GridView on initial postback
protected void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
        BindGridView();
}
```

The code block to calculate the total is given below.

```
//Calculate the Totals in the TextBox rows
protected void TextBox1_TextChanged(object sender, EventArgs e)
{
    double total = 0;
    foreach (GridViewRow gvr in GridView1.Rows)
    {
        TextBox tb = (TextBox)gvr.Cells[1].FindControl("TextBox1");
        double sum;
        if (double.TryParse(tb.Text.Trim(), out sum))
        {
            total += sum;
        }
    }
}
```

```

    }
}
//Display the Totals in the Footer row
GridView1.FooterRow.Cells[1].Text = total.ToString();
}

```

Running the code will now provide you a grid with the enabled total amount calculation. Now, there are certain cases, where you may be required to implement a client-side calculation. We will take a look at how we will implement it. Keep in mind that the Server-side approach isn't really ideal to do such operation as the TextChanged event will trigger a Server Postback everytime to change/type a value in the text box.

The Client-Side Approach with JavaScript

To get started, let's setup the form. For simplicity, let's just setup the form, as given below.

```

<asp:gridviewID="GridView1"runat="server"
ShowFooter="true"
AutoGenerateColumns="false">
<Columns>
<asp:BoundFieldDataField="RowNumber"HeaderText="Row Number"/>
<asp:BoundFieldDataField="Description"HeaderText="Item Description"/>
<asp:TemplateFieldHeaderText="Item Price">
<ItemTemplate>
<asp:LabelID="LBLPrice"runat="server"
Text='<%# Eval("Price", "{0:C}") %>'></asp:Label>
</ItemTemplate>
<FooterTemplate>
<b>Total Qty:</b>
</FooterTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Quantity">
<ItemTemplate>
<asp:TextBoxID="TXTQty"runat="server"
onkeyup="CalculateTotals();"></asp:TextBox>
</ItemTemplate>
<FooterTemplate>
<asp:LabelID="LBLQtyTotal"runat="server"
Font-Bold="true"ForeColor="Blue"Text="0"></asp:Label>
<b>Total Amount:</b>
</FooterTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Sub-Total">
<ItemTemplate>
<asp:LabelID="LBLSubTotal"runat="server"
ForeColor="Green"Text="0.00"></asp:Label>
</ItemTemplate>
<FooterTemplate>
<asp:LabelID="LBLTotal"runat="server"
ForeColor="Green"Font-Bold="true"Text="0.00"></asp:Label>

```

```
</FooterTemplate>  
</asp:TemplateField>  
</Columns>  
</asp:gridview>
```

As you can see, there is really nothing fancy about the markup, mentioned above. It just contains a standard GridView with BoundFields and TemplateFields in it. For the purpose of the demo, I will just use a dummy data to populating the GridView. The code block is given below.

```
publicpartialclassGridCalculation : System.Web.UI.Page  
{  
    privatevoid BindDummyDataToGrid()  
    {  
  
        DataTable dt = newDataTable();  
        DataRow dr = null;  
  
        dt.Columns.Add(newDataColumn("RowNumber", typeof(int)));  
        dt.Columns.Add(newDataColumn("Description", typeof(string)));  
        dt.Columns.Add(newDataColumn("Price", typeof(string)));  
  
        dr = dt.NewRow();  
        dr["RowNumber"] = 1;  
        dr["Description"] = "Nike";  
        dr["Price"] = "1000";  
        dt.Rows.Add(dr);  
  
        dr = dt.NewRow();  
        dr["RowNumber"] = 2;  
        dr["Description"] = "Converse";  
        dr["Price"] = "800";  
        dt.Rows.Add(dr);  
  
        dr = dt.NewRow();  
        dr["RowNumber"] = 3;  
        dr["Description"] = "Adidas";  
        dr["Price"] = "500";  
        dt.Rows.Add(dr);  
  
        dr = dt.NewRow();  
        dr["RowNumber"] = 4;  
        dr["Description"] = "Reebok";  
        dr["Price"] = "750";  
        dt.Rows.Add(dr);  
        dr = dt.NewRow();  
        dr["RowNumber"] = 5;  
        dr["Description"] = "Vans";  
        dr["Price"] = "1100";  
        dt.Rows.Add(dr);  
        dr = dt.NewRow();  
        dr["RowNumber"] = 6;  
        dr["Description"] = "Fila";  
        dr["Price"] = "200";  
        dt.Rows.Add(dr);  
    }  
}
```

```
//Bind the GridView
    GridView1.DataSource = dt;
    GridView1.DataBind();
}

protectedvoid Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
        BindDummyDataToGrid();
}
}
```

Running the page should result in something, as mentioned below.

Row Number	Item Description	Item Price	Quantity	Sub-Total
1	Nike	1000	<input type="text"/>	
2	Converse	800	<input type="text"/>	
3	Adidas	500	<input type="text"/>	
4	Reebok	750	<input type="text"/>	
5	Vans	1100	<input type="text"/>	
6	Fila	200	<input type="text"/>	
			Total Amount:	

Note, the client-side implementation is quite different, since you need to deal with the DOM elements to extract the controls and you'll need to understand JavaScript syntax, which is complex compared to C# with Server-side implementation. Now, let's go into the “meat” of this demo and this is the implementation of the client-side calculation. The main functionality comprises of the following.

- Number validation
- Formatting values into readable money format with separators
- Calculation for sub-totals and total amount

Here is the JavaScript code (this should be placed inside the <head> section of your WebForm page):

```
<scripttype="text/javascript">

function CalculateTotals() {
var gv = document.getElementById("<%= GridView1.ClientID %>");
var tb = gv.getElementsByTagName("input");
var lb = gv.getElementsByTagName("span");
```

```

var sub = 0;
var total = 0;
var indexQ = 1;
var indexP = 0;
var price = 0;

for (var i = 0; i < tb.length; i++) {
if (tb[i].type == "text") {
    ValidateNumber(tb[i]);

    price = lb[indexP].innerHTML.replace
("$", "").replace(",", "");
    sub = parseFloat(price) * parseFloat(tb[i].value);

if (isNaN(sub)) {
    lb[i + indexQ].innerHTML = "0.00";
    sub = 0;
}

else {
    lb[i + indexQ].innerHTML =
FormatToMoney(sub, "$", ",", "."); ;
}

    indexQ++;
    indexP = indexP + 2;

    total += parseFloat(sub);
}
}

lb[lb.length - 1].innerHTML =
FormatToMoney(total, "$", ",", ".");
}

function ValidateNumber(o) {
if (o.value.length > 0) {
    o.value = o.value.replace(/^[^d]+/g, ''); //Allow only whole numbers
}
}

function isThousands(position) {
if (Math.floor(position / 3) * 3 == position) returntrue;
returnfalse;
};

function FormatToMoney(theNumber, theCurrency, theThousands, theDecimal) {
var theDecimalDigits = Math.round((theNumber * 100) - (Math.floor(theNumber) * 100));
    theDecimalDigits = "" + (theDecimalDigits + "0").substring(0, 2);
    theNumber = "" + Math.floor(theNumber);
var theOutput = theCurrency;
for (x = 0; x < theNumber.length; x++) {
    theOutput += theNumber.substring(x, x + 1);
if (isThousands(theNumber.length - x - 1) && (theNumber.length - x - 1 != 0)) {
    theOutput += theThousands;
};
};
theOutput += theDecimal + theDecimalDigits;

```

```
return theOutput;
    }
</script>
```

Let's try to evaluate each JavaScript function, mentioned above. The FormatToMoney() is a function, which would format the numeric values to money by passing the numeric value, the currency, thousands and decimal separators where isThousand() function evaluates the value and returns Boolean. This function is used within the FormatToMoney() function to determine, if the value is on thousand. ValidateNumber() is a function, which validates, if the value supplied is a valid number. Finally, CalculateTotals() is the main function, which extracts each element from

the GridView, calculates the values and sets the calculated values back to the GridView element, which in this case is the sub-total and total amount.

Now, call the JavaScript CalculateTotals() function on “onkeyup” or “onkeypress” event, as mentioned below.

```
<ItemTemplate>
<asp:TextBoxID="TXTQty"runat="server"onkeyup="CalculateTotals();"></asp:TextBox>
</ItemTemplate>
```

Running the page will provide you with the output, mentioned below.

On initial load, the output will be as follows.

Row Number	Item Description	Item Price	Quantity	Sub-Total
1	Nike	\$1,000.00		0.00
2	Converse	\$800.00		0.00
3	Adidas	\$500.00		0.00
4	Reebok	\$750.00		0.00
5	Vans	\$1,100.00		0.00
6	Fila	\$200.00		0.00
			Total Amount:	0.00

After entering values into the TextBox

Row Number	Item Description	Item Price	Quantity	Sub-Total
1	Nike	\$1,000.00	1	\$1,000.00
2	Converse	\$800.00	2	\$1,600.00
3	Adidas	\$500.00	1	\$500.00
4	Reebok	\$750.00		0.00
5	Vans	\$1,100.00		0.00
6	Fila	\$200.00		0.00
			Total Amount:	\$3,100.00

Inserting and Deleting Sub Rows in GridView

In this section, we will take a look at how to insert sub rows in GridView and add a delete functionality for the inserted sub rows. The basic idea is to insert a row data in the DataSource since the GridView rows will be generated, based on the DataSource data.

To make it more clear, let's build up a sample Application. Here, the sample markup for the demo is given below.

```
<asp:gridviewID="GridView1"runat="server"AutoGenerateColumns="false"
onrowdatabound="GridView1_RowDataBound">
<Columns>
<asp:BoundFieldDataField="RowNumber"HeaderText="Row Number"/>
<asp:TemplateFieldHeaderText="Header 1">
<ItemTemplate>
<asp:TextBoxID="TextBox1"runat="server"></asp:TextBox>
</ItemTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Header 2">
<ItemTemplate>
<asp:TextBoxID="TextBox2"runat="server"></asp:TextBox>
</ItemTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Header 3">
<ItemTemplate>
<asp:TextBoxID="TextBox3"runat="server"></asp:TextBox>
</ItemTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Action">
<ItemTemplate>
<asp:LinkButtonID="LinkButton1"runat="server">
```

```
onclick="LinkButton1_Click"Text="Insert"/>
</ItemTemplate>
</asp:TemplateField>
</Columns>
</asp:gridview>
```

Here, the full code of the implementation is given below.

```
privateDataTable FillData()
{
    DataTable dt = newDataTable();
    DataRow dr = null;

    //Create DataTable columns
    dt.Columns.Add(newDataColumn("RowNumber", typeof(string)));

    //Create Row for each columns
    dr = dt.NewRow();
    dr["RowNumber"] = 1;
    dt.Rows.Add(dr);

    dr = dt.NewRow();
    dr["RowNumber"] = 2;
    dt.Rows.Add(dr);

    dr = dt.NewRow();
    dr["RowNumber"] = 3;
    dt.Rows.Add(dr);

    dr = dt.NewRow();
    dr["RowNumber"] = 4;
    dt.Rows.Add(dr);

    dr = dt.NewRow();
    dr["RowNumber"] = 5;
    dt.Rows.Add(dr);

    //Store the DataTable in ViewState for future reference
    ViewState["CurrentTable"] = dt;

    return dt;
}

privatevoid BindGridView(DataTable dtSource)
{
    GridView1.DataSource = dtSource;
    GridView1.DataBind();
}

privateDataRow InsertRow(DataTable dtSource, string value)
{
    DataRow dr = dtSource.NewRow();
```



```

        dr["RowNumber"] = value;
    return dr;
}
//private DataRow DeleteRow(DataTable dtSource,

protectedvoid Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        BindGridView(FillData());
    }
}

protectedvoid LinkButton1_Click(object sender, EventArgs e)
{
    LinkButton lb = (LinkButton)sender;
    GridViewRow row = (GridViewRow)lb.NamingContainer;
    DataTable dtCurrentData = (DataTable)ViewState["CurrentTable"];
    if (lb.Text == "Insert")
    {
        //Insert new row below the selected row

        dtCurrentData.Rows.InsertAt(InsertRow(dtCurrentData, row.Cells[0].Text + "-sub"),
row.RowIndex + 1);

    }
    else
    {
        //Delete selected sub row
        dtCurrentData.Rows.RemoveAt(row.RowIndex);
    }

    BindGridView(dtCurrentData);
    ViewState["CurrentTable"] = dtCurrentData;
}

protectedvoid GridView1_RowDataBound(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        if (e.Row.Cells[0].Text.Contains("-sub"))
        {
            ((LinkButton)e.Row.FindControl("LinkButton1")).Text = "Delete";
        }
    }
}

```

As you can see the code, mentioned above is pretty much straight forward and self explanatory. To give you a short explanation, the code, mentioned above comprises of three (3) private methods: the FillData(), BindGridView() and InsertRow(). FillData() method returns a DataTable and basically creates a dummy data for us to test something out. You can replace the code in the method, if you want to use actual data from the database, but for the purpose of this example, we just filled the DataTable with some sample data. BindGridView() is a method,

which handles the actual binding of GridView. The InsertRow() is a method, which returns a DataRow. This method handles the insertion of the sub row.

At the LinkButton OnClick event, we casted the sender to a LinkButton to determine the specific object, which fires up the event and get the row values. We then reference the data from ViewState to get the current data, which is being used in the GridView. If the LinkButton text is "Insert", we will insert a new row to the DataSource (in this case the DataTable) based on the rowIndex. If it is not there, delete the sub row, which was added.

Here, some screen shots of the output are mentioned below.

On initial load, the output is given below.

PROUDMONKEY

Home

Demo

GRIDVIEW SAMPLE DEMO

Row Number	Header 1	Header 2	Header 3	Action
1	<input type="text"/>	<input type="text"/>	<input type="text"/>	Insert
2	<input type="text"/>	<input type="text"/>	<input type="text"/>	Insert
3	<input type="text"/>	<input type="text"/>	<input type="text"/>	Insert
4	<input type="text"/>	<input type="text"/>	<input type="text"/>	Insert
5	<input type="text"/>	<input type="text"/>	<input type="text"/>	Insert

After inserting a sub row, the output is given below.

PROUDMONKEY				
Home Demo				
GRIDVIEW SAMPLE DEMO				
Row Number	Header 1	Header 2	Header 3	Action
1	<input type="text"/>	<input type="text"/>	<input type="text"/>	Insert
2	<input type="text"/>	<input type="text"/>	<input type="text"/>	Insert
2-sub	<input type="text"/>	<input type="text"/>	<input type="text"/>	Delete
3	<input type="text"/>	<input type="text"/>	<input type="text"/>	Insert
4	<input type="text"/>	<input type="text"/>	<input type="text"/>	Insert
5	<input type="text"/>	<input type="text"/>	<input type="text"/>	Insert
	<input type="text"/>	<input type="text"/>	<input type="text"/>	

Dynamically Adding and Deleting Rows in GridView and Saving All Rows at Once

Many years ago, I wrote a series of articles, which had shown how to add the dynamic textboxes, dynamic DropDownLists and a combination of both the controls in a GridView control. I have posted another couple of posts about how to delete the rows for the dynamically created rows and how to save them all at once.

Questions like “how do I generate dynamic controls in GridView on a click of a button” have been frequently asked up until now. In this section, I'm going to wrap up everything into one place for an easy reference. The main features are given below.

- Adding rows of TextBox and DropDownList.
- Retain TextBox values and DropDownList selected values across Postbacks.
- Ability to remove the rows.
- Save all values at once.

To get started, fire up Visual Studio and add a new WebForm page. Add a GridView control to the page. Here, the GridView HTML markup is mentioned below.

The HTML Markup

```
<asp:LabelID="lblMessage"runat="server"ForeColor="Green"/>
<asp:gridviewID="Gridview1"runat="server"ShowFooter="true"
AutoGenerateColumns="false"
OnRowCreated="Gridview1_RowCreated">
<Columns>
<asp:BoundFieldDataField="RowNumber"HeaderText="Row Number"/>
<asp:TemplateFieldHeaderText="Header 1">
<ItemTemplate>
<asp:TextBoxID="TextBox1"runat="server"></asp:TextBox>
</ItemTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Header 2">
<ItemTemplate>
<asp:TextBoxID="TextBox2"runat="server"></asp:TextBox>
</ItemTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Header 3">
<ItemTemplate>
<asp:DropDownListID="DropDownList1"runat="server"
AppendDataBoundItems="true">
<asp:ListItemValue="-1">Select</asp:ListItem>
</asp:DropDownList>
</ItemTemplate>
</asp:TemplateField>
<asp:TemplateFieldHeaderText="Header 4">
<ItemTemplate>
<asp:DropDownListID="DropDownList2"runat="server"
AppendDataBoundItems="true">
<asp:ListItemValue="-1">Select</asp:ListItem>
</asp:DropDownList>
</ItemTemplate>
<FooterStyleHorizontalAlign="Right"/>
<FooterTemplate>
<asp:ButtonID="ButtonAdd"runat="server"
Text="Add New Row"
onclick="ButtonAdd_Click"
OnClientClick="return ValidateEmptyValue();" />
</FooterTemplate>
</asp:TemplateField>
<asp:TemplateField>
<ItemTemplate>
<asp:LinkButtonID="LinkDelete"runat="server"
onclick="LinkDelete_Click">Remove</asp:LinkButton>
</ItemTemplate>
</asp:TemplateField>
</Columns>
</asp:gridview>
```

As you can see from the markup, mentioned above, I have setup a BoundField to display the RowNumber and some TemplateField columns, so that the GridView will automatically generate a row of TextBoxes and DropDownLists, while adding a new row. You will also see that I have

©2016 C# CORNER.

SHARE THIS DOCUMENT AS IT IS. PLEASE DO NOT REPRODUCE, REPUBLISH, CHANGE OR COPY.

added a button control under the FooterTemplate at the last DropDownList column and a LinkButton at the last column in the GridView to remove the rows.

Note- Since we added a control at the GridView footer, be sure to set ShowFooter to true in the GridView.

The Code at the backend

For the simplicity of the demo, I am creating a dummy data, using ArrayList as the data source for our DropDownList. In a real scenario, you may query your database and bind it to your DropDownList. The full code is given below.

```
using System;
using System.Collections;
using System.Data;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Collections.Specialized;
using System.Text;
using System.Data.SqlClient;

namespace WebFormsDemo
{
    public partial class DynamicGrid : System.Web.UI.Page
    {
        private ArrayList GetDummyData()
        {
            ArrayList arr = new ArrayList();

            arr.Add(new ListItem("Item1", "1"));
            arr.Add(new ListItem("Item2", "2"));
            arr.Add(new ListItem("Item3", "3"));
            arr.Add(new ListItem("Item4", "4"));
            arr.Add(new ListItem("Item5", "5"));

            return arr;
        }

        private void FillDropDownList(DropDownList ddl)
        {
            ArrayList arr = GetDummyData();

            foreach (ListItem item in arr)
            {
                ddl.Items.Add(item);
            }
        }

        private void SetInitialRow()
        {

```

```

DataTable dt = new DataTable();
DataRow dr = null;

        dt.Columns.Add(new DataColumn("RowNumber", typeof(string)));
        dt.Columns.Add(new DataColumn("Column1", typeof(string))); //for TextBox value
        dt.Columns.Add(new DataColumn("Column2", typeof(string))); //for TextBox value
        dt.Columns.Add(new DataColumn("Column3", typeof(string))); //for DropDownList
selected item
        dt.Columns.Add(new DataColumn("Column4", typeof(string))); //for DropDownList
selected item

        dr = dt.NewRow();
        dr["RowNumber"] = 1;
        dr["Column1"] = string.Empty;
        dr["Column2"] = string.Empty;
        dt.Rows.Add(dr);

//Store the DataTable in ViewState for future reference
ViewState["CurrentTable"] = dt;

//Bind the Gridview
Gridview1.DataSource = dt;
Gridview1.DataBind();

//After binding the gridview, we can then extract and fill the DropDownList with Data
DropDownList ddl1 =
(DropDownList)Gridview1.Rows[0].Cells[3].FindControl("DropDownList1");
DropDownList ddl2 =
(DropDownList)Gridview1.Rows[0].Cells[4].FindControl("DropDownList2");
        FillDropDownList(ddl1);
        FillDropDownList(ddl2);
    }

private void AddNewRowToGrid()
{
    if (ViewState["CurrentTable"] != null)
    {
        DataTable dtCurrentTable = (DataTable)ViewState["CurrentTable"];
        DataRow drCurrentRow = null;

        if (dtCurrentTable.Rows.Count > 0)
        {
            drCurrentRow = dtCurrentTable.NewRow();
            drCurrentRow["RowNumber"] = dtCurrentTable.Rows.Count + 1;

//add new row to DataTable
            dtCurrentTable.Rows.Add(drCurrentRow);

            for (int i = 0; i < dtCurrentTable.Rows.Count - 1; i++)
            {

//extract the TextBox values

```

```

TextBox box1 = (TextBox)GridView1.Rows[i].Cells[1].FindControl("TextBox1");
TextBox box2 = (TextBox)GridView1.Rows[i].Cells[2].FindControl("TextBox2");

        dtCurrentTable.Rows[i]["Column1"] = box1.Text;
        dtCurrentTable.Rows[i]["Column2"] = box2.Text;

//extract the DropDownList Selected Items

DropDownList ddl1 =
(DropDownList)GridView1.Rows[i].Cells[3].FindControl("DropDownList1");
DropDownList ddl2 =
(DropDownList)GridView1.Rows[i].Cells[4].FindControl("DropDownList2");

// Update the DataRow with the DDL Selected Items

        dtCurrentTable.Rows[i]["Column3"] = ddl1.SelectedItem.Text;
        dtCurrentTable.Rows[i]["Column4"] = ddl2.SelectedItem.Text;

    }

//Store the current data to ViewState for future reference
ViewState["CurrentTable"] = dtCurrentTable;

//Rebind the Grid with the current data to reflect changes
GridView1.DataSource = dtCurrentTable;
GridView1.DataBind();
    }
}
else
{
    Response.Write("ViewState is null");
}

//Set Previous Data on Postbacks
SetPreviousData();
}

privatevoid SetPreviousData()
{
    int rowIndex = 0;
    if (ViewState["CurrentTable"] != null)
    {
        DataTable dt = (DataTable)ViewState["CurrentTable"];
        if (dt.Rows.Count > 0)
        {
            for (int i = 0; i < dt.Rows.Count; i++)
            {

                TextBox box1 = (TextBox)GridView1.Rows[i].Cells[1].FindControl("TextBox1");
                TextBox box2 = (TextBox)GridView1.Rows[i].Cells[2].FindControl("TextBox2");
            }
        }
    }
}

```

```

DropDownList ddl1 =
(DropDownList)GridView1.Rows[rowIndex].Cells[3].FindControl("DropDownList1");
DropDownList ddl2 =
(DropDownList)GridView1.Rows[rowIndex].Cells[4].FindControl("DropDownList2");

//Fill the DropDownList with Data
FillDropDownList(ddl1);
FillDropDownList(ddl2);

if (i < dt.Rows.Count - 1)
{

//Assign the value from DataTable to the TextBox
box1.Text = dt.Rows[i]["Column1"].ToString();
box2.Text = dt.Rows[i]["Column2"].ToString();

//Set the Previous Selected Items on Each DropDownList on Postbacks
ddl1.ClearSelection();

ddl1.Items.FindByText(dt.Rows[i]["Column3"].ToString()).Selected = true;

ddl2.ClearSelection();

ddl2.Items.FindByText(dt.Rows[i]["Column4"].ToString()).Selected = true;

}

rowIndex++;
}
}
}

protectedvoid Page_Load(object sender, EventArgs e)
{
if (!Page.IsPostBack)
{
SetInitialRow();
}
}

protectedvoid ButtonAdd_Click(object sender, EventArgs e)
{
AddNewRowToGrid();
}

protectedvoid GridView1_RowCreated(object sender, GridViewRowEventArgs e)
{
if (e.Row.RowType == DataControlRowType.DataRow)
{
DataTable dt = (DataTable)ViewState["CurrentTable"];
LinkButton lb = (LinkButton)e.Row.FindControl("LinkButton1");
if (lb != null)
{
if (dt.Rows.Count > 1)
{

```



```

if (e.Row.RowIndex == dt.Rows.Count - 1)
    {
        lb.Visible = false;
    }
else
    {
        lb.Visible = false;
    }
}
}

protectedvoid LinkDelete_Click(object sender, EventArgs e)
{
    LinkButton lb = (LinkButton)sender;
    GridViewRow gvRow = (GridViewRow)lb.NamingContainer;
    int rowID = gvRow.RowIndex;
    if (ViewState["CurrentTable"] != null)
    {
        DataTable dt = (DataTable)ViewState["CurrentTable"];
        if (dt.Rows.Count > 1)
        {
            if (gvRow.RowIndex < dt.Rows.Count - 1)
            {
                //Remove the Selected Row data and reset row number
                dt.Rows.Remove(dt.Rows[rowID]);
                ResetRowID(dt);
            }

            //Store the current data in ViewState for future reference
            ViewState["CurrentTable"] = dt;

            //Re bind the GridView for the updated data
            Gridview1.DataSource = dt;
            Gridview1.DataBind();
        }

        //Set Previous Data on Postbacks
        SetPreviousData();
    }

    privatevoid ResetRowID(DataTable dt)
    {
        int rowNumber = 1;
        if (dt.Rows.Count > 0)
        {
            foreach (DataRow row in dt.Rows)
            {
                row[0] = rowNumber;
                rowNumber++;
            }
        }
    }
}

```

```
}  
}
```

Method Definitions

- **GetDummyData()**- A method that returns an ArrayList. Basically this method contains a static dummy data for populating the DropDownList. You may want to use a database when dealing with real world scenarios.
- **FillDropDownList(DropDownList DDL)**: A method that fills the DropDownList with the dummy data.
- **SetInitialRow()**- A method, which binds the GridView on the initial load with a single row of data. The DataTable defined in this method is stored in ViewState, so that it can be referenced anywhere in the code across Postbacks. Basically this table will serve as the original DataSource for the GridView. Keep in mind that this is just for demo, so be careful when using ViewState to avoid page performance issue. Also ViewState has a limit when it comes to size so make sure that you don't store a huge amount of data in it.
- **AddNewRowToGrid()**- A method, which adds a new row to the GridView when a button is clicked and store the newly added row values in the Original Table that was defined in the SetInitialRow() method.
- **SetPreviousData()**- A method, which retains all the items, which were selected from the DropDownList and TextBox, when it Postbacks.
- **ResetRowID()**- A method, which refreshes the grid's row number, while a row is deleted.

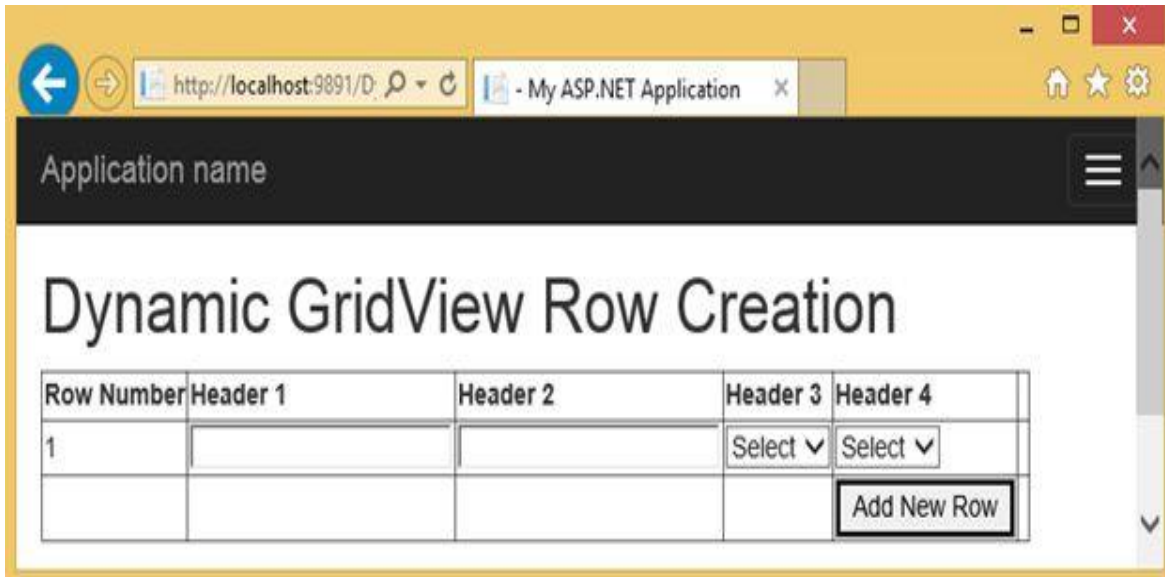
The Events

- **ButtonAdd_Click**- Calls AddNewRowToGrid() method.
- **LinkDelete_Click**- This method will be invoked once the “remove” link is clicked from the grid. This is where the data from the data source will be remove, based on the row index, reset the row number afterwards and finally store the updated data source in ViewState again and bind it to the grid to reflect the changes.
- **Gridview1_RowCreated**- This is where we put the basic validation in GridView for not allowing the users to see the “remove” button in the last row.

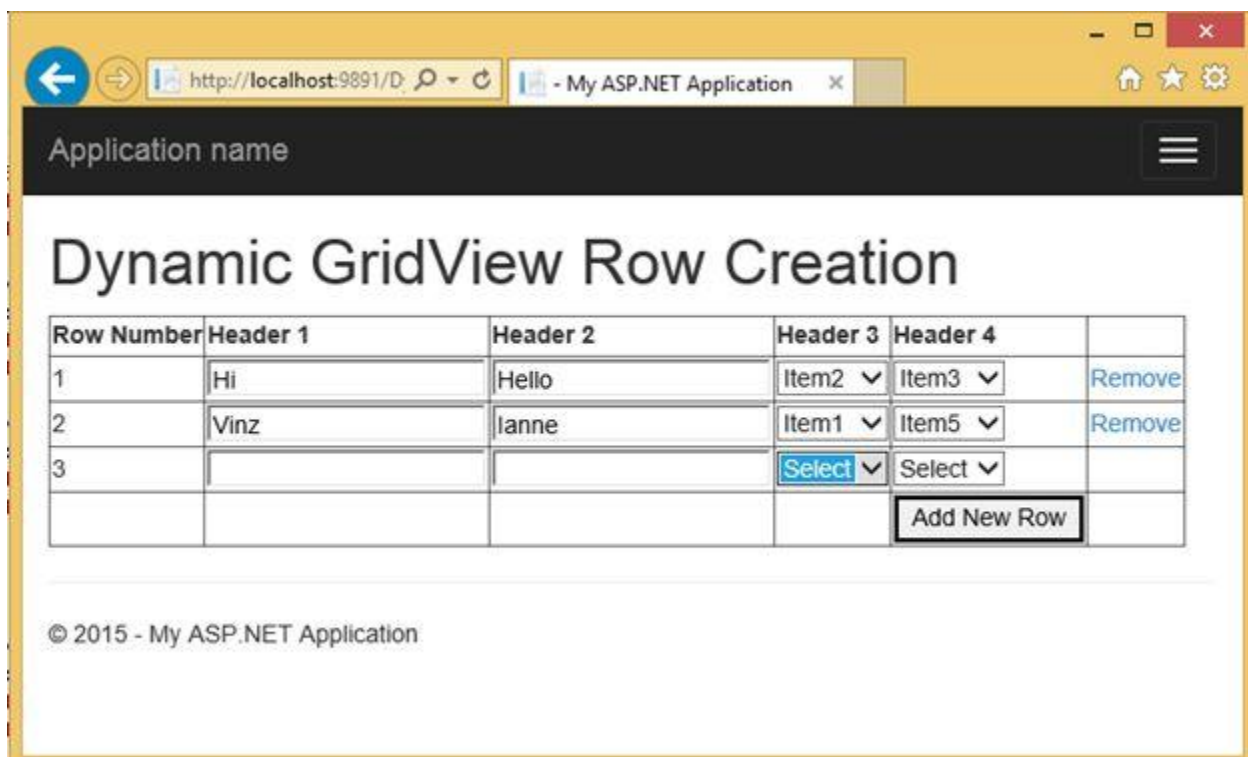
The Output

Running the page will display something, as shown below in the Browser.

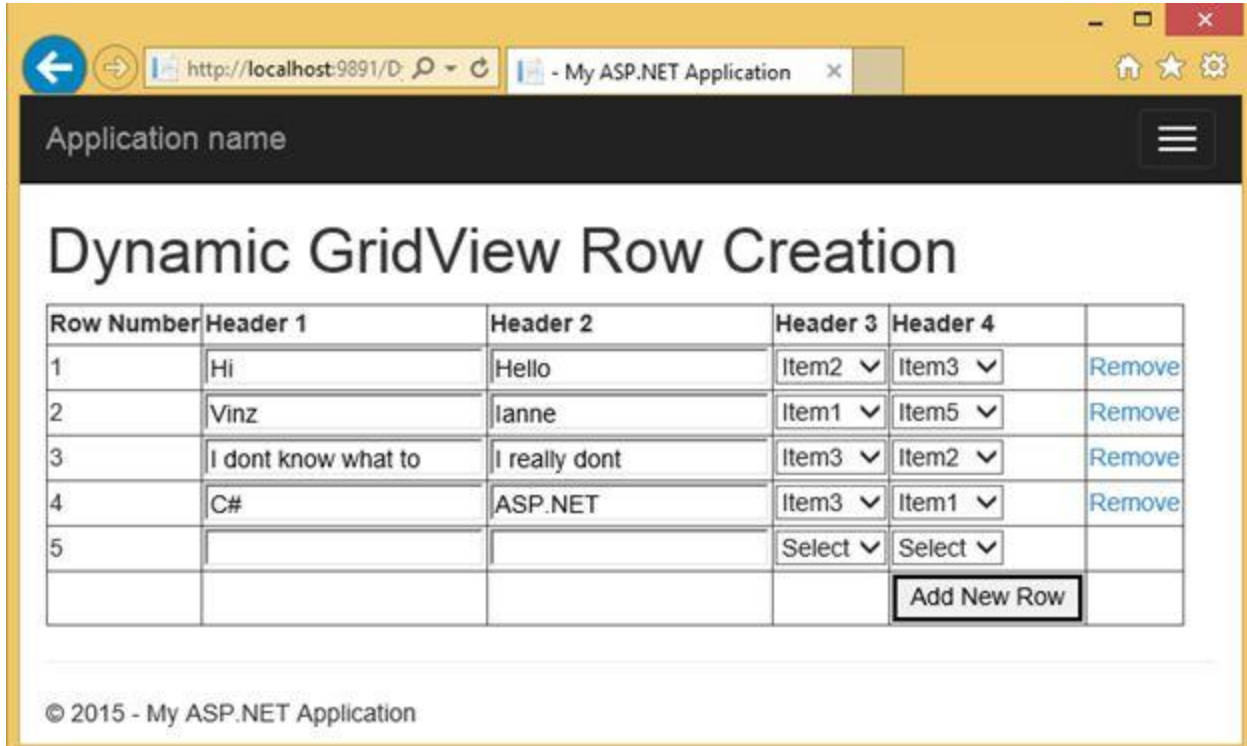
On Initial load, the output is mentioned below.



Adding some new rows yields the output, mentioned below.



Removing a row helps to get the output as follows.



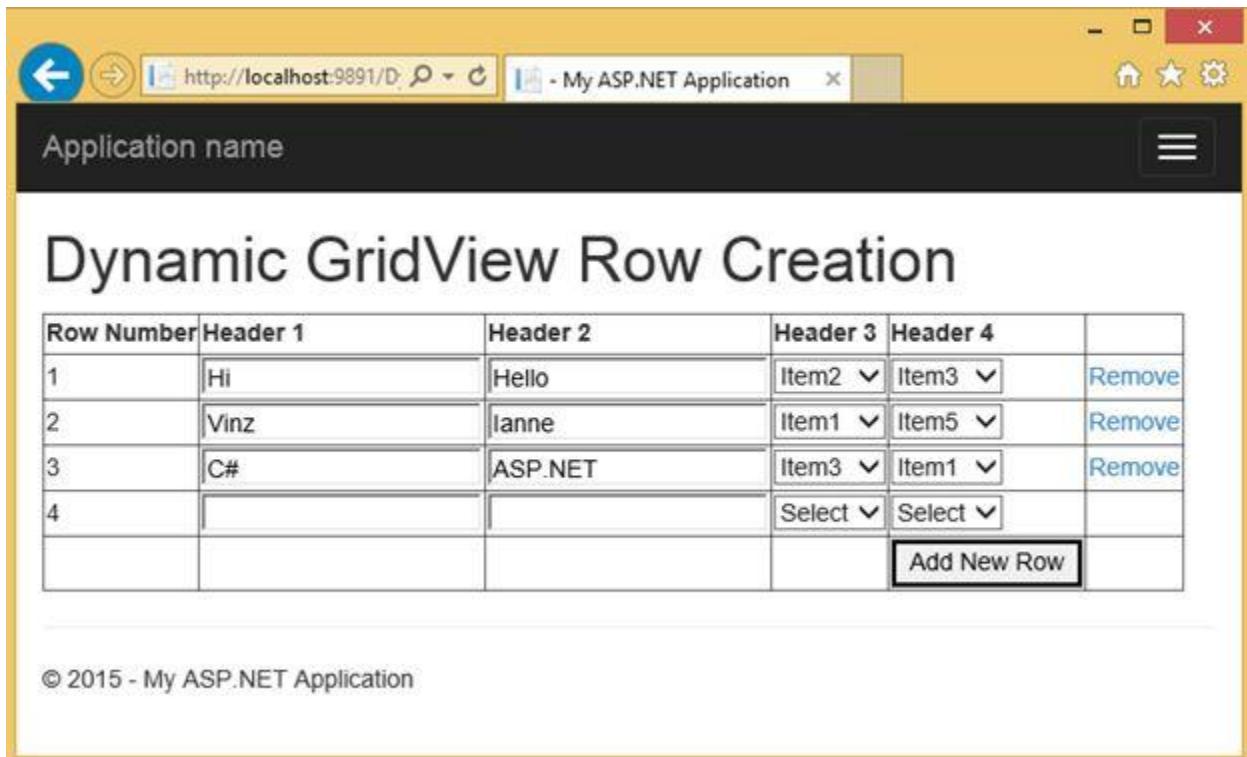
Application name

Dynamic GridView Row Creation

Row Number	Header 1	Header 2	Header 3	Header 4	
1	Hi	Hello	Item2 ▼	Item3 ▼	Remove
2	Vinz	Ianne	Item1 ▼	Item5 ▼	Remove
3	I dont know what to	I really dont	Item3 ▼	Item2 ▼	Remove
4	C#	ASP.NET	Item3 ▼	Item1 ▼	Remove
5			Select ▼	Select ▼	
					Add New Row

© 2015 - My ASP.NET Application

After removing a row, the output retrieved is mentioned below.



Application name

Dynamic GridView Row Creation

Row Number	Header 1	Header 2	Header 3	Header 4	
1	Hi	Hello	Item2 ▼	Item3 ▼	Remove
2	Vinz	Ianne	Item1 ▼	Item5 ▼	Remove
3	C#	ASP.NET	Item3 ▼	Item1 ▼	Remove
4			Select ▼	Select ▼	
					Add New Row

© 2015 - My ASP.NET Application

Now, the next thing, which you guys might be asking is how to save the data in the database. Don't worry, because in the next step, I am going to show you how.

Saving All Data at Once

The first thing to do is you need to create a database and a table to store the data so fire up SQL Management Studio or the Express version of SQL Server and create the table with the fields, mentioned below.

```
CREATETABLE[dbo].[GridViewDynamicData](  
    [RowID][tinyint] IDENTITY(1, 1) NOTNULL, [Field1][varchar](50) NULL,  
    [Field2][varchar](50) NULL, [Field3][varchar](50) NULL, [Field4][varchar](50) NULL  
    ) ON[PRIMARY]  
  
GO
```

Save the table to whatever you like but for this demo, I named the table as “GridViewDynamicData”.

Note

I set the RowID to auto increment, so that the Id will be automatically generated for the every new added row in the table. To do this, select the Column name “RowID” and in the column properties sets the “Identity Specification” to yes.

Once you've created the table, switch back to Visual Studio and add a button control to the form.

For example

```
<asp:ButtonID="BtnSave"runat="server"Text="Save All"OnClick="BtnSave_Click"/>
```

Now, let's create the method to save the data to the database. The first thing, we need here is to set up the connection string, so that we can communicate with our database from our code. For this example, we will use the web.config file to set up the connection string. See the markup, mentioned below.

```
<connectionStrings>  
<addname="DBConnection"connectionString="Data Source=win-ehm93ap21cf\SQLEXPRESS;Initial  
Catalog=DemoDB;Integrated Security=SSPI;"  
providerName="System.Data.SqlClient"/>  
</connectionStrings>
```

Note

You may need to change the value of DataSource and Initial Catalog.

We can now proceed to create the method to save the data to the database. First, add the namespaces, mentioned below.

```
using System.Data;
using System.Data.SqlClient;
```

We need to declare the namespaces, mentioned above, so that we can use SQLClient, ADO.NET objects to work with the data.

Second, create the method to call the connection string, which was setup from the web.config file.

```
private string GetConnectionString()
{
    return ConfigurationManager.ConnectionStrings["DBConnection"].ConnectionString;
}
```

The code block is mentioned below to insert all the rows into our database.

```
private void InsertRecords(DataTable source)
{
    using (SqlConnection connection = new SqlConnection(GetConnectionString()))
    {
        using (SqlBulkCopy sqlBulkCopy = new SqlBulkCopy(connection))
        {
            //Set the Database Table to use
            sqlBulkCopy.DestinationTableName = "dbo.GridViewDynamicData";

            // [OPTIONAL]: Mappings of columns from Datatable against Database table
            sqlBulkCopy.ColumnMappings.Add("Field1", "Field1");
            sqlBulkCopy.ColumnMappings.Add("Field2", "Field2");
            sqlBulkCopy.ColumnMappings.Add("Field3", "Field3");
            sqlBulkCopy.ColumnMappings.Add("Field4", "Field4");
            connection.Open();
            sqlBulkCopy.WriteToServer(source);
        }
    }

    lblMessage.Text = "Records successfully saved!";
}
```

The InsertRecords() method takes a DataTable object as the parameter. The DataTable object contains all the values from the dynamic grid. We use the SQLBulkCopy.WriteToServer method to load all the rows from the DataTable to a specified table name from a database at once.

Finally, the code block for the button click event is mentioned below.

```
protectedvoid BtnSave_Click(object sender, EventArgs e)
{
    if (GridView1.Rows.Count > 0)
    {
        DataTable dt = new DataTable();
        dt.Columns.AddRange(new DataColumn[4] {
            new DataColumn("Field1", typeof(string)),
            new DataColumn("Field2", typeof(string)),
            new DataColumn("Field3", typeof(string)),
            new DataColumn("Field4", typeof(string))});

        foreach (GridViewRow row in GridView1.Rows)
        {
            string field1 = ((TextBox)row.Cells[1].FindControl("TextBox1")).Text;
            string field2 = ((TextBox)row.Cells[2].FindControl("TextBox2")).Text;
            string field3 =
            ((DropDownList)row.Cells[3].FindControl("DropDownList1")).SelectedItem.Text;
            string field4 =
            ((DropDownList)row.Cells[4].FindControl("DropDownList2")).SelectedItem.Text;

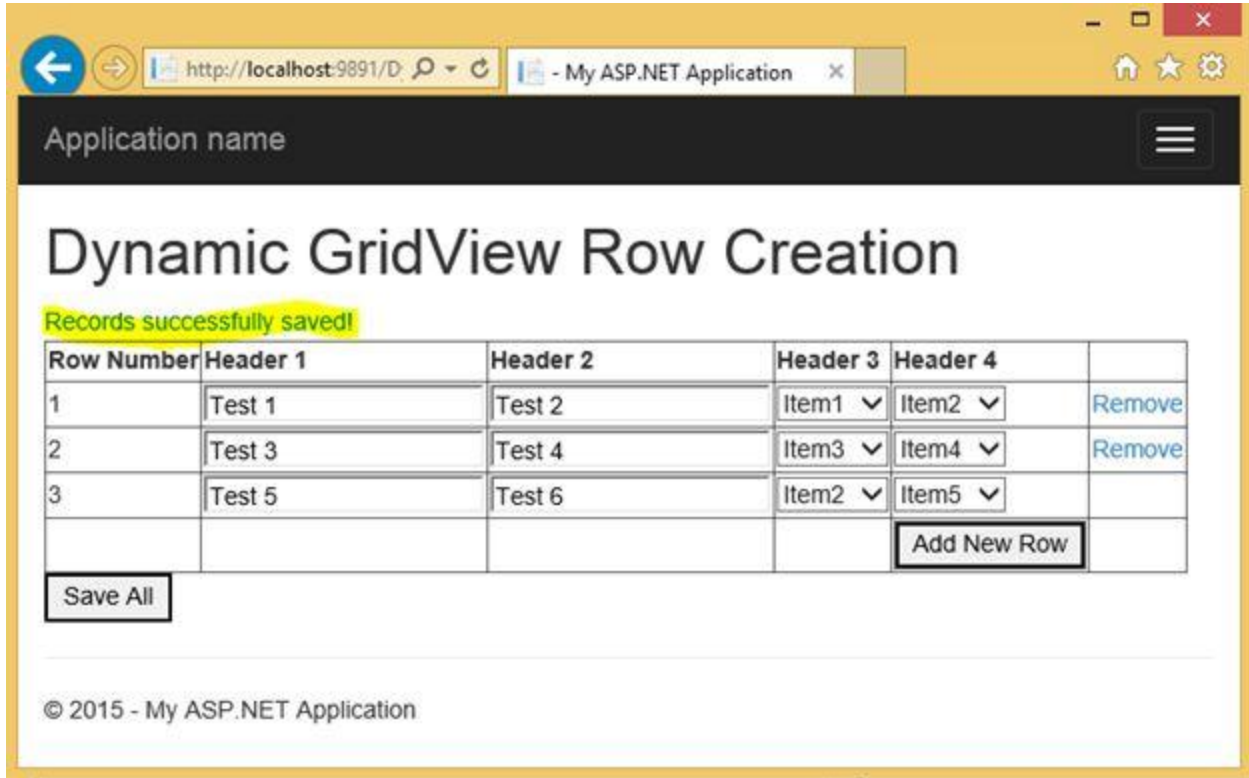
            dt.Rows.Add(field1, field2, field3, field4);
        }
        InsertRecords(dt);
    }
}
```

The code, mentioned above is pretty much straight forward. First, we create a new DataTable and define some column definitions for us to store some values to it. We then loop through the rows from the GridView to extract the control values and then add each control values in the

DataTable. After all the values are added, we call the method InsertRecords() to actually execute the inserts to the database.

Output

The output is mentioned below after clicking on the “Save All” button.



Application name

Dynamic GridView Row Creation

Records successfully saved!

Row Number	Header 1	Header 2	Header 3	Header 4	
1	Test 1	Test 2	Item1 ▾	Item2 ▾	Remove
2	Test 3	Test 4	Item3 ▾	Item4 ▾	Remove
3	Test 5	Test 6	Item2 ▾	Item5 ▾	
				Add New Row	

[Save All](#)

© 2015 - My ASP.NET Application

Here, the captured data is stored in the database.

Results		Messages			
	RowID	Field1	Field2	Field3	Field4
1	1	Test 1	Test 2	Item1	Item2
2	2	Test 3	Test 4	Item3	Item4
3	3	Test 5	Test 6	Item2	Item5

If you want validate for the empty values in each text box, you can use JavaScript, mentioned below.

```
<script>
function ValidateEmptyValue() {
var gv = document.getElementById("<%= Gridview1.ClientID %>");
var tb = gv.getElementsByTagName("input");

for (var i = 0; i < tb.length; i++) {
if (tb[i].type == "text") {
if (tb[i].value < 1) {
alert("Field cannot be blank!");
returnfalse;
}
}
}
}
```



```
    }  
    return true;  
  }  
</script>
```

You can call JavaScript function in the ButtonAdd control, as mentioned below.

```
<asp:ButtonID="ButtonAdd"runat="server"  
Text="Add New Row"  
onclick="ButtonAdd_Click"  
OnClientClick="return ValidateEmptyValue();"/>
```

Tips and Tricks

Accessing AutoGenerated Columns in GridView

I have seen many developers in the forums were asking the stuff like “how do we set read-only for the autogenerated columns in gridview?” or “how to access autogenerated bound columns in GridView?”. Well as you may know, the autogenerated columns are created dynamically, so we need to manually access each column in the code before we can set their properties.

A quick code snippet is mentioned below on setting the boundfield column for the autogenerated columns in GridView to ReadOnly.

```
protectedvoid GridView1_RowCreated(object sender, GridViewRowEventArgs e)
{
    foreach (TableCell cell in e.Row.Cells)
    {
        if (!string.IsNullOrEmpty(cell.Text) && cell.Text != " ")
        {
            BoundField field = (BoundField)((DataControlFieldCell)cell).ContainingField;
            if (field.DataField == "ID")
                field.ReadOnly = true;
        }
    }
}
```

GridView cells comprises of different DataControlFields and basically AutoGenerated Columns uses a BoundField to display the data. If you have noticed from the code, mentioned above, we loop through the cells and cast them to a DataControlFieldCell type to get the ContainingField. We then cast the ContainingField to a BoundField type, so that we can check the DataField used in a particular column.

You can also use the code, mentioned above, if you want to (for example) hide a specific column in your auto generated grid.

Why Not Just Use This?

```
GridView1.Columns[index].Visible = false;
```

Using the code, mentioned above will give you "index was out of range error". Why? This is because auto generated columns are not added in the GridView columns collection.

The quick snippet for hiding a specific column is mentioned below.

```
protectedvoid GridView1_RowCreated(object sender, GridViewRowEventArgs e)
{
    foreach (TableCell cell in e.Row.Cells)
```

```
{
    BoundField field = (BoundField)((DataControlFieldCell)cell).ContainingField;
    if (field.DataField == "ColumnName")
    {
        field.Visible = false;
    }
}
```

Dynamic Cascading DropDownList using JavaScript

A user in the forums was asking how to populate one dropdownlist, based on a value selected from another dropdownlist, using JavaScript in ASP.NET. The question was quite interesting and I thought, I'd share the solution. I posted on the thread as a reference to others, who might face the same problem.

The scenario is that the dropdownlists are inside a GridView TemplateFields and the user wanted to implement a cascading dropdown by dynamically creating the items on the fly without using jQuery, AJAX and database, but purely JavaScript.

To implement it, let's add a GridView in the markup and setup some TemplateField columns with DropDownList on it. Our ASPX markup should look, as mentioned below.

```
<formid="form1"runat="server">
<asp:GridViewID="GridView3"runat="server">
<Columns>
<asp:TemplateField>
<ItemTemplate>
<asp:DropDownListID="ddlStatus"
runat="server"
onchange="buildDropDown(this);">
<asp:ListItem></asp:ListItem>
<asp:ListItemValue="Ma">Ma</asp:ListItem>
<asp:ListItemValue="Mi">Mi</asp:ListItem>
<asp:ListItemValue="Others">Others</asp:ListItem>
</asp:DropDownList>
</ItemTemplate>
</asp:TemplateField>
<asp:TemplateField>
<ItemTemplate>
<asp:DropDownListID="ddlReason"runat="server">
</asp:DropDownList>
</ItemTemplate>
</asp:TemplateField>
</Columns>
</asp:GridView>
</form>
```

There's nothing fancy in the markup, mentioned above, except that we wired-up the onchange event to DDLStatus DropDownList.

Now, let's create JavaScript function. Here, the code block is mentioned below.

```
<script type="text/javascript">
function buildDropDown(obj) {
var status = obj.options[obj.selectedIndex].value;
var row = obj.parentNode.parentNode;
var rowIndex = row.rowIndex - 1;
//you may need to change the index of cells value based on the location
//of your ddlReason DropDownList
var ddlReason = row.cells[1].getElementsByTagName('SELECT')[0];

switch (status) {
case "Ma":
            ddlReason.options[0] = new Option("OK", "OK");
            ddlReason.options[1] = new Option("Good", "Good");

break;
case "Mi":
            ddlReason.options[0] = new Option("Data", "Data");
            ddlReason.options[1] = new Option("Resoure", "Resoure");
            ddlReason.options[2] = new Option("Time", "Time");

break;
case "Others":
            ddlReason.options[0] = new Option("Some Item", "Some Item");

break;
        }
}
</script>
```

The function buildDropDown() takes a parameter obj. This parameter is a reference to a DropDownList control, which is inside a GridView. Now, let's take a look at what we've done there.

The status variable holds the selected value of the first DropDownList called DDLStatus.

The row variable serves as the naming container to which the DropDownList DDLStatus is located. Remember, a grid contains rows, so we need to determine which DropDownList to populate.

The DDLReason variable represents the DDLReason DropDownList. Once we get the row, we can easily find an element in the cells, using row.cells[1].getElementsByTagName('SELECT')[0]. This line means get the first SELECT element in the array, i.e. within the second cColumn of the GridView. cells[1] represents the 2nd column. Note, index always starts at 0, so you may need to change the index, based on your requirements.

The switch statement simply determines which items to be populated in the DDLReason DropDownList, based on the status value.

MasterPage, WebUserControl, ModalPopup and UpdatePanel – Passing GridView Values to Parent Page

A developer in the forums is asking how to pass the data, which is coming from a WebUserControl control to the main page. Basically, the scenario is that he has a WebUserControl, which contains a GridView, which is wrapped around inside an ASPNET AJAX UpdatePanel control. The WebUserControl will then be used in the page (ContentPage), which is hosted within a MasterPage. The page contains some text box in which it will be populated, once the user selects a row from the GridView and ModalPopup is defined within that page.

The question is quite interesting and I have given it a shot and created a simple demo. Here, the solution , which I came up with and provide it to the developer.

The Page Markup

```
<%@PageTitle=""Language="C#"MasterPageFile=~\Site.Master"AutoEventWireup="true"
CodeBehind="Modal.aspx.cs"Inherits="WebAppDemo.Modal"%>

<%@RegisterAssembly="AjaxControlToolkit"Namespace="AjaxControlToolkit"TagPrefix="asp"%>
<%@Registersrc="UserControl/WebUserControl1.ascx"tagName="WebUserControl1"tagprefix="uc1"
%>

<asp:ContentID="Content1"ContentPlaceHolderID="HeadContent"runat="server">
<styletype="text/css">
.modal-bg{
background-color:Gray;
filter:alpha(opacity=50);
opacity:0.6;
z-index:999;
}
.modal{
position:absolute;
}
</style>

</asp:Content>
<asp:ContentID="Content2"ContentPlaceHolderID="MainContent"runat="server">
<asp:ToolkitScriptManagerID="ToolkitScriptManager1"runat="server">
</asp:ToolkitScriptManager>

<asp:TextBoxID="txtProductName"runat="server"/>
<asp:TextBoxID="txtPrice"runat="server"/>

<asp:LinkButtonID="LinkButton1"runat="server"Text="Show"/>
<asp:PanelID="pnlPopUp"runat="server"
style="display:none"CssClass="modal">
<uc1:WebUserControl1ID="WebUserControl11"runat="server"/>
```

```

</asp:Panel>
<asp:ModalPopupExtenderID="ModalPopupExtender1"runat="server"
TargetControlID="LinkButton1"
PopupControlID="pnlPopUp"
BackgroundCssClass="modal-bg">
</asp:ModalPopupExtender>

</asp:Content>

```

There is nothing really fancy about the markup, mentioned above. It just contains two text boxes, a ModalPopup, a LinkButton as the target control for the Modal and a Panel control, which contains the WebUserControl.

You may also have noticed that I have created a simple CSS for the modal.

The WebUserControl Markup

```

<asp:UpdatePanelID="UpdatePanel1"runat="server">
<ContentTemplate>
<asp:GridViewID="GridView1"runat="server"AutoGenerateColumns="False">
<Columns>
<asp:BoundFieldDataField="ID"/>
<asp:BoundFieldDataField="ProductName"HeaderText="Product Name"/>
<asp:TemplateField>
<ItemTemplate>
<asp:TextBoxID="txtPrice"runat="server"
Text="<%# Eval("Price") %>" />
</ItemTemplate>
</asp:TemplateField>
<asp:TemplateField>
<ItemTemplate>
<asp:TextBoxID="txtAmount"runat="server"
Text="<%# Eval("Amount") %>" />
</ItemTemplate>
</asp:TemplateField>
<asp:TemplateField>
<ItemTemplate>
<asp:ButtonID="Button1"runat="server"
Text="Select"onclick="Button1_Click"/>
</ItemTemplate>
</asp:TemplateField>
</Columns>
</asp:GridView>
</ContentTemplate>
<Triggers>
<asp:PostBackTriggerControlID="GridView1"/>
</Triggers>
</asp:UpdatePanel>

```

WebUserControl is where we defined the GridView. Notice, the GridView is wrapped within an UpdatePanel control. We need to set the ID of GridView as the PostbackTrgigger for the

UpdatePanel to trigger the control events within GridView – in this case, the Click event of the button control.

The Code at the backend

Now, the code at the backend is mentioned below for the WebUserControl.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Data;
using AjaxControlToolkit;

namespace WebAppDemo.UserControl
{
    public partial class WebUserControl1 : System.Web.UI.UserControl
    {

        private DataTable GetSampleData()
        {
            //NOTE: THIS IS JUST FOR DEMO
            //If you are working with database
            //You can query your actual data and fill it to the DataTable
            DataTable dt = new DataTable();
            DataRow dr = null;

            //Create DataTable columns
            dt.Columns.Add(new DataColumn("ID", typeof(string)));
            dt.Columns.Add(new DataColumn("ProductName", typeof(string)));
            dt.Columns.Add(new DataColumn("Price", typeof(string)));
            dt.Columns.Add(new DataColumn("Amount", typeof(string)));

            //Create Row for each columns
            dr = dt.NewRow();
            dr["ID"] = 1;
            dr["ProductName"] = "Galaxy S";
            dr["Price"] = "100";
            dr["Amount"] = "10";

            dt.Rows.Add(dr);

            dr = dt.NewRow();
            dr["ID"] = 2;
            dr["ProductName"] = "iPhone 4";
            dr["Price"] = "200";
            dr["Amount"] = "2";

            dt.Rows.Add(dr);
        }
    }
}
```

```

        dr = dt.NewRow();
        dr["ID"] = 3;
        dr["ProductName"] = "HTC Mobile";
        dr["Price"] = "50";
        dr["Amount"] = "10";

        dt.Rows.Add(dr);

return dt;
    }

    private void BindGrid(DataTable source)
    {
        GridView1.DataSource = source;
        GridView1.DataBind();
    }

    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
            BindGrid(GetSampleData());
    }

    protected void Button1_Click(object sender, EventArgs e)
    {
        Button b = (Button)sender;
        GridViewRow gvRow = (GridViewRow)b.NamingContainer;

        string productName = gvRow.Cells[1].Text;
        string price = ((TextBox)gvRow.FindControl("txtPrice")).Text;

        Type thePage = Page.GetType();
        System.Reflection.PropertyInfo[] pi = thePage.GetProperties();
        foreach (System.Reflection.PropertyInfo pinfo in pi)
        {
            if (pinfo.Name == "ProductName")
            {
                pinfo.SetValue(Page, productName, null);
            }
            elseif (pinfo.Name == "Price")
            {
                pinfo.SetValue(Page, price, null);
            }
        }

        ((ModalPopupExtender)this.Parent.Parent.FindControl(
            "ModalPopupExtender1")).Hide();
    }
}

```

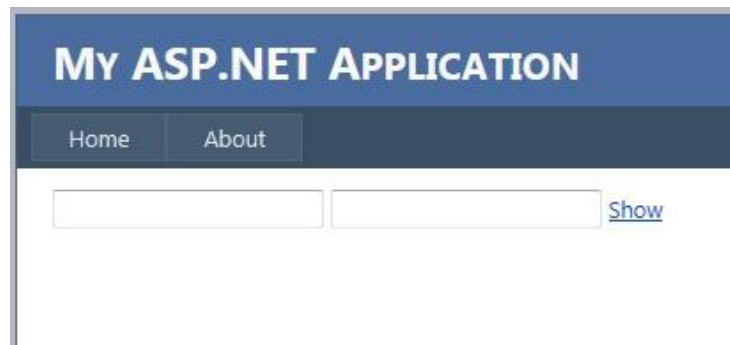
You may have noticed, I just used a fake data as the DataSource for GridView. At the button's Click event, I have used reflection to access the property metadata. Subsequently, I used the SetValue to populate the controls in the main page, based on the selected row values from the GridView. Note, you can also use the FindControl() method to reference a control and assign a

value, but I opt to use the reflection in this particular demo because I just want to pass the data and I don't want to deal with the naming containers to find the specific controls in the page. FindControl method is useful when you want to manipulate the control, like for example changing the Width or perhaps toggling the Enabled or Visible property of the control.

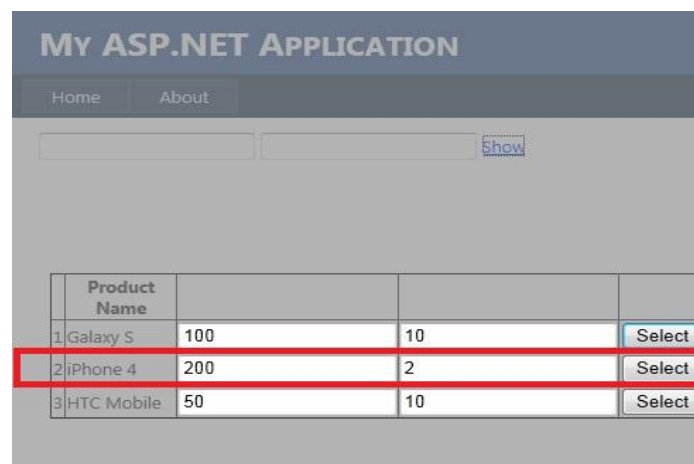
You can also use properties, if you want to get the value from a UserControl in the main page. For example, you expose a public property within your WebUserControl, then in your main page, you simply reference the property defined in your WebUserControl. For example at button's click event. Though using properties isn't applicable for this particular scenario, since we are passing the data to the main page right after selecting a row and not getting the data.

Output

Running the page will display the output, as shown below.



After clicking, the "Show" link's snapshot is displayed, which is as follows.



After selecting a row from GridView, the output is given below.



Highlighting GridView Rows on Mouseover

Here, an example on how to highlight GridView rows on mouseover at RowCreated or RowDataBound event of GridView by simply adding the attributes to the row.

Here, the code block is mentioned below.

```
protectedvoid GridView1_RowCreated(object sender, GridViewRowEventArgs e)
{
    string onmouseoverStyle = "this.style.backgroundColor='blue'";
    string onmouseoutStyle = "this.style.backgroundColor='white'";
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        e.Row.Attributes.Add("onmouseover", onmouseoverStyle);
        e.Row.Attributes.Add("onmouseout", onmouseoutStyle);
    }
}
```

Remove GridView Row Highlighting on Edit Mode

Typically, we attach the mouseover and mouseout client-side events on the gridview rows to highlight the rows on mouseover, but there are cases, which we don't want to make the row highlighted when we are on edit mode. To do this, we can check the GridView EditIndex to determine if the row is on edit mode and then do the validation there. Here, a sample code block is mentioned below.

```
protectedvoid GridView1_RowDataBound(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        string onmouseoverStyle = "this.style.backgroundColor='blue';this.style.color''";
        string onmouseoutStyle = "this.style.backgroundColor='white';this.style.color''";

        if (GridView1.EditIndex != -1)
        {
            e.Row.Attributes.Remove("onmouseover");
        }
    }
}
```

©2016 C# CORNER.

SHARE THIS DOCUMENT AS IT IS. PLEASE DO NOT REPRODUCE, REPUBLISH, CHANGE OR COPY.

```

    }
else
    {
        e.Row.Attributes.Add("onmouseover", onmouseoverStyle);
        e.Row.Attributes.Add("onmouseout", onmouseoutStyle);
    }
}

```

As you can see, the code, mentioned above is very straight forward. It checks if the GridView is on edit mode. When it's on edit mode, we removed the onmouseover attribute, else we attach the onmouseover attribute again and apply the desired styles.

Ways On Hiding AutoGenerateColumns in GridView

As you may know, we cannot "directly" hide AutoGenerateColumns in our codes using the code below:

```
GridView1.Columns[index].Visible = false;
```

Why?

This is because auto generated columns are not added in the GridView columns collection. Using the code, mentioned above will give you "index was out of range error".

In this example, we'll take a look at how to show the different ways on hiding a specific column in GridView with AutoGenerateColumns set to true.

Using the Cells Index

```

protectedvoid GridView1_RowCreated(object sender, GridViewRowEventArgs e)
{
    //hides the first column.
    //by passing 0 as the index of cells
    e.Row.Cells[0].Visible = false;
}

```

Looping through GridView Row Controls Collections

```

protectedvoid GridView1_RowCreated(object sender, GridViewRowEventArgs e)
{
    foreach (TableRow row in GridView1.Controls[0].Controls)
    {
        row.Cells[0].Visible = false;
    }
}

```

You can use the options, mentioned above to hide the columns, if you are sure with the order of the columns in the table. Please note that autogenerated columns will display all the columns from the DataSource, so you must be careful when using an index to hide the columns.

Looping through GridView Cells

As you may know, GridView cells comprises of different DataControlFields and basically AutoGenerated fields use a BoundField to display the data. In this case, we can loop through the cells generated by the GridView and cast the cell to a DataControlFieldCell type to get the ContainingField. We can then cast this ContainingField to a BoundField, so that we can check the DataField used in a particular AutoGenerated BoundField and Hide them, using its visible property.

To make it more clear, you can check the code block, mentioned below.

```
protectedvoid GridView1_RowCreated(object sender, GridViewRowEventArgs e)
{
    foreach (TableCell cell in e.Row.Cells)
    {
        BoundField field = (BoundField)((DataControlFieldCell)cell).ContainingField;
        if (field.DataField == "ColumnName")
        {
            field.Visible = false;
        }
    }
}
```

As you can see, we will check ColumnName first before hiding the column instead of using column indexing.

The ColumnName, mentioned above indicates the field from your DataSource, which you want to hide. You can use this option to hide the columns, if you do not know the sequence of the columns from the DataSource.

Wrap Particular Column Data in GridView

Here, the code block is mentioned below.

```
protectedvoid GridView1_RowDataBound(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        e.Row.Cells[0].Attributes.Add("style", "word-break:break-all;word-wrap:break-word");
    }
}
```

```

}

protectedvoid GridView1_RowCreated(object sender, GridViewRowEventArgs e)
{
    this.GridView1.Columns[0].ItemStyle.Width = newUnit(100);
}

```

Accessing Controls in GridView TemplateFields on GridView Edit Mode

This example demonstrates how to access ASP.NET Server controls, which resides within the TemplateField column of GridView on Edit mode. In this example, we are using the PreRender event of GridView instead of RowEditing event since we cannot directly access the controls at RowEditing event of GridView.

Here, a sample code snippet is mentioned below.

```

protectedvoid GridView1_PreRender(object sender, EventArgs e)
{
    if (this.GridView1.EditIndex != -1)
    {
        Button b = GridView1.Rows[GridView1.EditIndex].FindControl("Button1")
        asButton;
        if (b != null)
        {
            //do something
        }
    }
}

```

Limiting the Data Being displayed in GridView and Display as Tooltip

Here's a sample code snippet:

```

protectedvoid GridView1_RowDataBound(object sender, GridViewRowEventArgs e)
{
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        ViewState["OrigData"] = e.Row.Cells[0].Text;
        //Just change the value of 30 based on your requirements
        if (e.Row.Cells[0].Text.Length >= 30)
        {
            e.Row.Cells[0].Text = e.Row.Cells[0].Text.Substring(0, 30) + "...";
            e.Row.Cells[0].ToolTip = ViewState["OrigData"].ToString();
        }
    }
}

```

Move AutoGenerate Columns at the LeftMost Part of the GridView Columns

There are certain scenarios, where we need to combine AutoGenerated columns with TemplateField columns or even BoundField columns in the GridView. As we all know, by default, the GridView will automatically display all the AutoGenerated columns to the rightmost column of the GridView. Let's see this in action.

Consider, we have GridView markup, as shown below.

```
<asp:GridViewID="GridView1"runat="server">
<Columns>
<asp:TemplateField>
<ItemTemplate>
<asp:ButtonID="Button1"runat="server"Text="Button A"/>
</ItemTemplate>
</asp:TemplateField>
<asp:TemplateField>
<ItemTemplate>
<asp:ButtonID="Button2"runat="server"Text="Button B"/>
</ItemTemplate>
</asp:TemplateField>
</Columns>
</asp:GridView>
```

Note

By default, the AutoGenerateColumns property of GridView is set to true, so we don't need to set it manually in the GridView.

Now, let's bind the GridView with the data from the database. The code will look as follows.

```
privatestring GetDBConnectionString()
{
returnConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString;
}

privatevoid BindGrid()
{
DataTable dt = newDataTable();
using (SqlConnection sqlConn = newSqlConnection(GetDBConnectionString()))
{
string sql = "SELECT * FROM dbo.[Employee]";
using (SqlCommand sqlCmd = newSqlCommand(sql, sqlConn))
{
sqlConn.Open();
using (SqlDataAdapter sqlAdapter = newSqlDataAdapter(sqlCmd))
{
sqlAdapter.Fill(dt);
}
}
}
}
```

```

    }

    if(dt.Rows.Count > 0)
    {
        gvEmployee.DataSource = dt;
        gvEmployee.DataBind();
    }

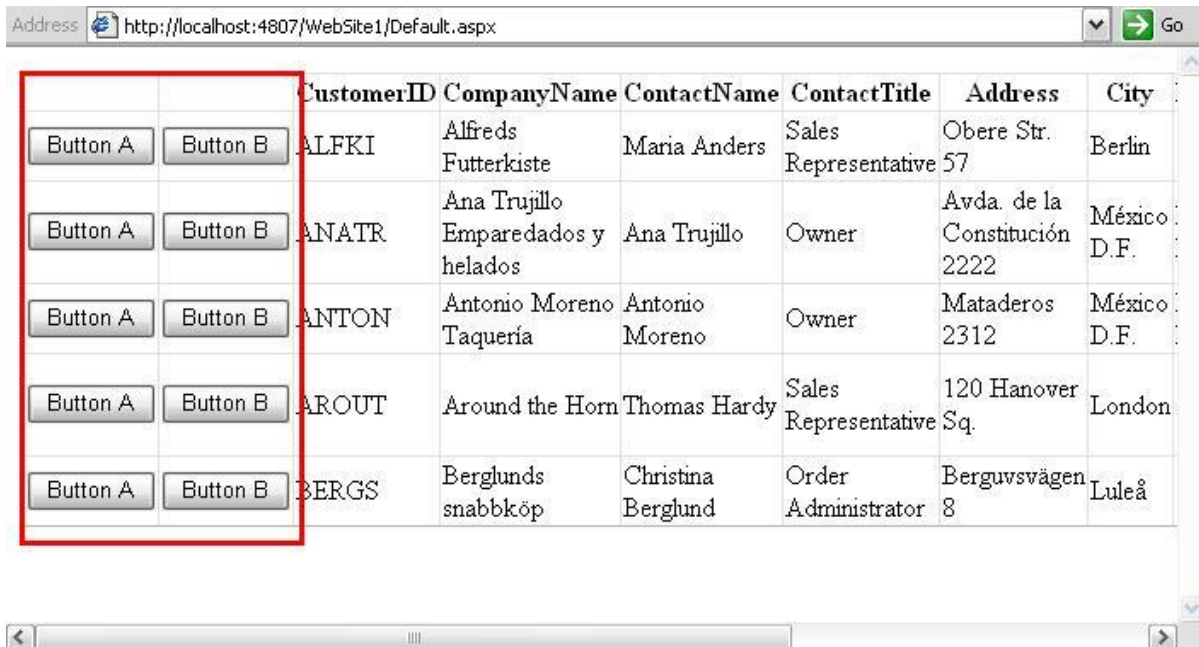
    else
    {
        //display no records found here
    }

}

protectedvoid Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
        BindGrid();
}

```

Running the code, mentioned above will give us this output in the page, mentioned below.



CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Button A	Button B
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57	Berlin	Button A	Button B
ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	Avda. de la Constitución 2222	México D.F.	Button A	Button B
ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.	Button A	Button B
AROUT	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.	London	Button A	Button B
BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Berguvsvägen 8	Luleå	Button A	Button B

As expected, the columns for TemplateFields (with red box) were generated at the leftmost columns in the GridView and the AutoGenerate columns will be rendered at the rightmost columns after the TemplateFields.

The Solution

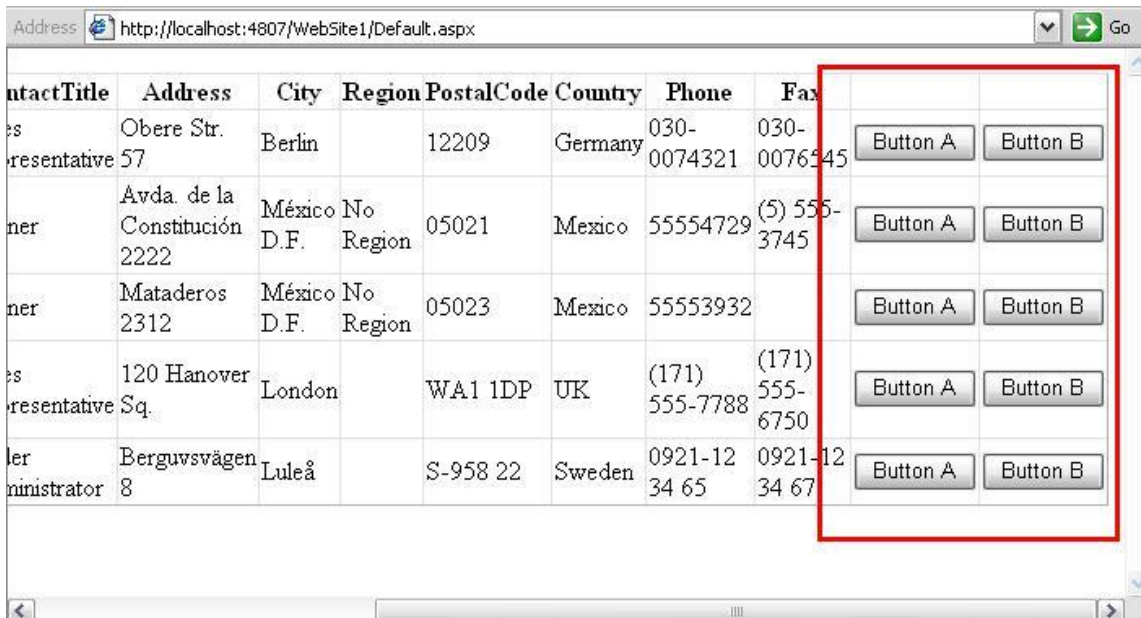
In order to move those AutoGenerate columns to the leftmost columns of the GridView, we can do some Server side manipulations at the RowCreated event of GridView.

Here, the trick is mentioned below.

```
protectedvoid GridView1_RowCreated(object sender, GridViewRowEventArgs e)
{
    GridViewRow row = e.Row;
    // Initialize TableCell list
    List<TableCell> columns = new List<TableCell>();
    foreach (DataControlField column in GridView1.Columns)
    {
        //Get the first Cell /Column
        TableCell cell = row.Cells[0];
        // Then Remove it after
        row.Cells.Remove(cell);
        //And Add it to the List Collections
        columns.Add(cell);
    }

    // Add cells
    row.Cells.AddRange(columns.ToArray());
}
```

Running the code will now result in the output, mentioned below.



ContactTitle	Address	City	Region	PostalCode	Country	Phone	Fax		
Representative	Obere Str. 57	Berlin		12209	Germany	030-0074321	030-0076545	Button A	Button B
Manager	Avda. de la Constitución 2222	México D.F.	No Region	05021	Mexico	55554729	(5) 555-3745	Button A	Button B
Manager	Mataderos 2312	México D.F.	No Region	05023	Mexico	55553932		Button A	Button B
Representative	120 Hanover Sq.	London		WA1 1DP	UK	(171) 555-7788	(171) 555-6750	Button A	Button B
Administrator	Berguvsvägen 8	Luleå		S-958 22	Sweden	0921-12 34 65	0921-12 34 67	Button A	Button B

How to Get Hidden Column Values in GridView

Hidden columns are the fields in GridView, which you don't want to expose or show in the page, usually this field is the primary key of the data. Since this is a confidential data, you might want to hide it to the users. Most people usually use BoundField columns to display the data and just hide the field, which contains the primary key.

In this example, I will demonstrate two ways on how to access the hidden columns when a specific row is selected in GridView.

Suppose, we have this GridView markup, as shown below.

```
<asp:GridViewID="GridView3"runat="server"AutoGenerateColumns="False"
onselectedindexchanged="GridView1_SelectedIndexChanged">
<Columns>
<asp:CommandFieldShowSelectButton="True"/>
<asp:BoundFieldHeaderText="Header 1"DataField="ID"Visible="false"/>
<asp:BoundFieldHeaderText="Header 2"DataField="SomeField"/>
</Columns>
</asp:GridView>
```

Notice, the Visibility of the BoundField column, which holds the ID data, which was set to false means that the field will be hidden/invisible.

Let's try to access the value of the hidden column, while selecting a specific row in the GridView. To do this, we can use the SelectedIndexChanged event of GridView.

Here, the code block is mentioned below.

```
protectedvoid GridView1_SelectedIndexChanged(object sender, EventArgs e)
{
    string strValue = GridView1.SelectedRow.Cells[1].Text;
    //display the selected value in a popup
    Page.ClientScript.RegisterClientScriptBlock(typeof(Page), "SCRIPT",
    string.Format("alert('{0}');", strValue), true);
}
```

Running the codes, mentioned above will give you an empty value when you select a row in the GridView.

Why?

This is because, for the security reasons, BoundField columns with Visibility are set to false will not be rendered in the page and thus we cannot directly get the value in our code.

The workarounds

Using the DataKeyNames of GridView (recommended)

The easiest way to store the hidden fields (primary keys) is by using DataKeyNames property of the GridView control, because this provides a convenient way to access the primary keys of each row.

Here, the updated markup is stated below.

```
<asp:GridView ID = "GridView1" runat="server" AutoGenerateColumns="False"
DataKeyNames="ID"
onselectedindexchanged="GridView1_SelectedIndexChanged">
<Columns>
<asp:CommandField ShowSelectButton = "True" />
<asp:BoundField HeaderText = "Header 2" DataField="SomeField" />
</Columns>
</asp:GridView>
```

Notice, we removed the hidden BoundField column, which contains the ID data in the GridView and set up the DataKeyNames in the GridView.

Here, the code to access the primary key of the row uses the DataKeys.

```
protectedvoid GridView1_SelectedIndexChanged(object sender, EventArgs e){
//get the selected DataKey
int rowIndex = GridView1.SelectedIndex;
string id = GridView1.DataKeys[rowIndex].Value.ToString();

//display the selected value in a pop up
Page.ClientScript.RegisterClientScriptBlock(typeof(Page), "SCRIPT",
string.Format("alert('{0}');", strValue), true);
}
```

Using a TemplateField Column with a HiddenField control

Another way would be storing the primary key column in a HiddenField control.

An example is shown below.

```
<asp:GridViewID="GridView1"runat="server"AutoGenerateColumns="False"
```

```

onselectedindexchanged="GridView1_SelectedIndexChanged">
<Columns>
<asp:CommandFieldShowSelectButton="True"/>
<asp:TemplateField>
<ItemTemplate>
<asp:HiddenFieldID="HiddenField1"runat="server"Value='<%# Bind("ID") %>' />
</ItemTemplate>
</asp:TemplateField>
<asp:BoundFieldHeaderText="Header 2"DataField="SomeField"/>
</Columns>
</asp:GridView>

```

The relevant code to access the primary key data, which was stored in a HiddenField control is mentioned below.

```

protectedvoid GridView1_SelectedIndexChanged(object sender, EventArgs e){

string id =
((HiddenField)GridView1.SelectedRow.Cells[1].FindControl("HiddenField1")).Value;

    Page.ClientScript.RegisterClientScriptBlock(typeof(Page), "SCRIPT",
string.Format("alert('{0}');", strValue), true);
}

```

Copy All Selected GridView Rows to an Array

Here, a sample snippet is mentioned below.

```

protectedvoid GridView1_SelectedIndexChanged(object sender, EventArgs e)
{
DataControlFieldCell[] arrCells = newDataControlFieldCell[GridView1.Columns.Count];
    GridView1.SelectedRow.Cells.CopyTo(arrCells, 0);

foreach (DataControlFieldCell cell in arrCells)
    {
        Response.Write(cell.Text + "<BR/>");
    }
}

```

Summary

In this book, we learnt some of the most basic operations in GridView, including CRUD. We also learnt some of the most frequently asked questions about GridView control.