EXTREME ASP.NET

Charting With ASP.NET And LINQ

K. Scott Allen

CODE DOWNLOAD AVAILABLE FROM THE MSDN CODE GALLERY

Browse the Code Online

Contents

Getting Started
Chart Building
Pumping Data
From Charts to Dashboards
Charting the Future

Microsoft recently released a new charting control for ASP.NET 3.5 SP1. The chart control is easy to use and allows you to add attractive data visualizations to your ASP.NET Web applications. The control supports all of the standard chart types, like line charts and pie charts, as well as advanced visualizations like funnel and pyramid charts. In this column, I'll explore the chart control and generate some data using queries written for LINQ to objects. The full source code for this column is available from the MSDN Code Gallery.

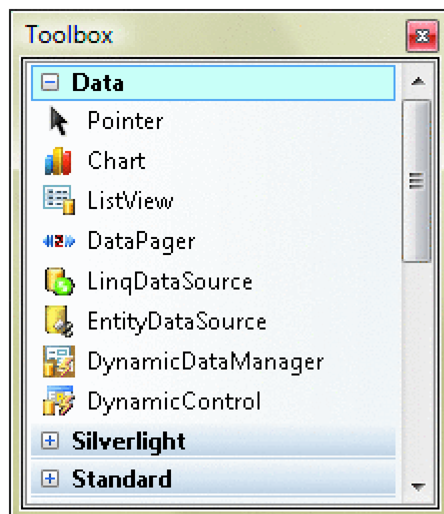

Figure 1 **Visual Studio Tools**

Getting Started

The first step is to download the chart control from the Microsoft Download Center. This download installs the key runtime components you'll need for charting, including the placement of the System.Web.DataVisualization assembly into the global assembly cache.

You'll also want to download the Visual Studio 2008 Tool Support and the charting sample Web site. The tool support will give you toolbox integration and IntelliSense support at design time, while the sample Web site will give you hundreds of examples to examine for inspiration into creating the type of chart you need. Note that you will need Service Pack 1 installed for both the Microsoft .NET Framework 3.5 runtime and Visual Studio 2008.

After the installations are all complete, you should be able to create a new ASP.NET project in Visual Studio and find the Chart control in the Toolbox window (see **Figure 1**). You can drag the chart into an ASPX file using the design view (which will make some necessary configuration changes to your web.config file) or work with the control directly in the Source view of an ASPX file (in which case you'll need to manually make the configuration changes I will soon describe).

When you place a chart into the Web forms designer, Visual Studio will place a new entry into the <controls> section of your web.config file, like so:

Copy Code

```
<add tagPrefix="asp"
```

```
      namespace="System.Web.UI.DataVisualization.Charting"
      assembly="System.Web.DataVisualization,
                Version=3.5.0.0, Culture=neutral,
                PublicKeyToken=31bf3856ad364e35" />
```

This entry allows you to use the chart control with the familiar asp tag prefix that other built-in ASP.NET controls utilize. Visual Studio also adds a new HTTP handler entry into the IIS 7.0 <handlers> section and a similar entry into the <httpHandlers> section (for use with IIS 6.0 and the Visual Studio Web development server):

Copy Code

```
<add name="ChartImageHandler"
     preCondition="integratedMode"
     verb="GET,HEAD"
     path="ChartImg.axd"
     type="System.Web.UI.DataVisualization.Charting.Chart           HttpHandler,
           System.Web.DataVisualization,
           Version=3.5.0.0, Culture=neutral,
           PublicKeyToken=31bf3856ad364e35" />
```

This HTTP handler is responsible for processing requests that arrive for ChartImg.axd, which is the default endpoint the chart control will use for serving charts. I'll go into more detail about the HTTP handler later.

The code in **Figure 2** shows the basic elements of a chart. Every chart includes at least one Series object populated with data. The ChartType property of each series will determine the type of chart used to plot the series (the default type is a column chart). Each chart can also contain one or more ChartArea objects where plotting will occur.

Figure 2 A Basic Chart

Copy Code

```
<asp:Chart ID="Chart1" runat="server" Height="300" Width="400">
    <Series>
        <asp:Series BorderColor="180, 26, 59, 105">
            <Points>
                <asp:DataPoint YValues="45" />
                <asp:DataPoint YValues="34" />
                <asp:DataPoint YValues="67" />
            </Points>
        </asp:Series>
    </Series>
    <ChartAreas>
        <asp:ChartArea />
    </ChartAreas>
</asp:Chart>
```

You can customize nearly every visual element of the ASP.NET chart control, including backgrounds, axes, titles, legends, and labels. The chart control is so highly customizable that you should have a plan in place to keep the charts in your application looking consistent. In this column, I will use a builder strategy to apply consistent fonts and colors to all charts. This builder class will also allow the use of the charting control outside the confines of an ASP.NET page.

Chart Building

In looking for sample data to use in this column, I briefly considered using historical data from the stock market, but the data from the past year has been depressing, so instead I decided to use data from the United States Bureau of Transportation Statistics (bts.gov). The sample application for this column includes a file with information about every domestic flight originating from my home airport (Baltimore Washington International, or BWI) during January of 2008. The data includes the destination city, distance, taxiing times, and delays. This data is represented in C# using the class in **Figure 3**.

Figure 3 Flight Data

Copy Code

```
public class Flight {
    public DateTime Date { get; set; }
```

```
    public string Airline { get; set; }
    public string Origin { get; set; }
    public string Destination { get; set; }
    public int TaxiOut { get; set; }
    public int TaxiIn { get; set; }
    public bool Cancelled { get; set; }
    public int ArrivalDelay { get; set; }
    public int AirTime { get; set; }
    public int Distance { get; set; }
    public int CarrierDelay { get; set; }
    public int WeatherDelay { get; set; }
}
```

The first chart I built from this data was a chart to show the most popular destinations for flights from the Baltimore airport (see **Figure 4**). This chart is built using very little code in the ASPX file or its associated codebehind file. The ASPX file includes a chart control on the page, but only sets the Width and Height properties, as you can see:

Copy Code

```
  <form id="form1" runat="server">
  <div>
    <asp:Chart runat="server" Width="800" Height="600" ID="_chart">
    </asp:Chart>
  </div>
</form>
```

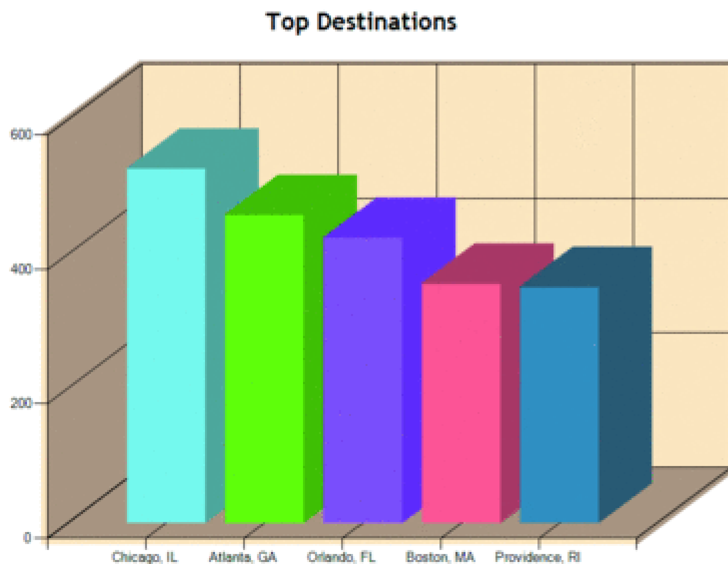

Figure 4 **Top Destinations** (Click the image for a larger view)

The codebehind delegates all of its Page_Load event handling work to a TopDestinationsChartBuilder class, as you see here:

Copy Code

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        var builder = new TopDestinationsChartBuilder(_chart);
        builder.BuildChart();
    }
}
```

The TopDestinationsChartBuilder inherits from a ChartBuilder class. This ChartBuilder base class uses a template method design pattern to assemble the pieces of a chart. The template method specifies the basic

algorithm required to produce an aesthetically consistent and functional chart, but provides hooks for a subclass to customize the assembled pieces. The template method is named BuildChart, and it is shown in **Figure 5**.

Figure 5 BuildChart

Copy Code

```
public void BuildChart() {
    _chart.ChartAreas.Add(BuildChartArea());
    _chart.Titles.Add(BuildChartTitle());
    if (_numberOfSeries > 1) {
        _chart.Legends.Add(BuildChartLegend());
    }
    foreach (var series in BuildChartSeries()) {
        _chart.Series.Add(series);
    }
}
```

Each Build method in the ChartBuilder class has an associated Customize method. Once the Build method has constructed its piece of the chart, it invokes the Customize method. A derived class can override the customization method to apply chart-specific settings. One such example is the BuildChartTitle method, shown in **Figure 6**. The TopDestinationsChartBuilder class overrides the CustomizeChartTitle to apply the specific text for a title:

Copy Code

```
protected override void  CustomizeChartTitle(Title title)
{
    title.Text = "Top Destinations";
}
```

Figure 6 BuildChartTitle

Copy Code

```
private Title BuildChartTitle() {
    Title title = new Title() {
        Docking = Docking.Top,
        Font = new Font("Trebuchet MS", 18.0f, FontStyle.Bold),
    };
    CustomizeChartTitle(title);
    return title;
}

protected virtual void CustomizeChartTitle(Title title) { }
```

The majority of the work for the TopDestinationsChartBuilder is dedicated to customizing the chart series, which includes adding all of the data points for display. Fortunately, finding the top five destination cities from a collection of Flight objects is ridiculously easy using LINQ to Objects, as you can see in **Figure 7**. The code first applies LINQ's GroupBy operator to group the flights by their destination city. The OrderByDescending operator then sorts the sequence of groupings by the number of flights in each group. Finally, the code uses the Take operator to identify the top five destinations from the sequence.

Figure 7 Get Top Five Cities

Copy Code

```
protected override void CustomizeChartSeries(IList<Series> seriesList) {
    var repository = FlightRepositoryFactory.CreateRepository();
    var query = repository.Flights
                          .GroupBy(flight => flight.Destination)
                          .OrderByDescending(group => group.Count())
                          .Take(5);
    Series cities = seriesList.Single();
    cities.Name = "Cities";
    foreach (var record in query) {
        cities.Points.AddXY(record.Key, record.Count());
    }
```

```
}
```

The LINQ query produces a sequence of groupings. You can loop through these groupings to add information to the chart. The Key property of each grouping represents the value of the key selected in the GroupBy operator (in this case, the value of the Destination city). The code uses the destination as the X value for each data point. Each grouping also contains an enumerable sequence of the Flight objects it grouped under this destination. You only need to use the Count operator to get the total number of flights to each destination and add the count as the Y value for each data point.

Pumping Data

Instead of adding data to a chart series one point at a time, you can use a DataBindXY method on the chart's DataPointCollection to insert a sequence of data points without using a loop. You can see this happen in the DelaysByDayChartBuilder, which calculates the total of all delays (in minutes) for each day in the month. This builder also produces a second series of data showing the total weather-related delays. The builder starts with a LINQ query that groups the flights by the day of the month. The Key property for each grouping now represents the day of the month (from 1 to 31 for January):

Copy Code

```
var query = repository.Flights
                              .GroupBy(flight => flight.Date.Day)
                              .OrderBy(group => group.Key)
                              .ToList();
```

The code in **Figure 8** uses the DataBindXY method. First, all the X values are collected into a list using the Select operator to grab the Key value of each group. Next, additional processing is applied to the initial grouping query to sum the ArrivalDelay and WeatherDelay values inside of each group.

Figure 8 Insert Values without Looping

Copy Code

```
var xValues = query.Select(group => group.Key).ToList();

totalDelaySeries.Points.DataBindXY(
        xValues,
        query.Select(
            group => group.Sum(
                flight => flight.ArrivalDelay)).ToList());

weatherDelaySeries.Points.DataBindXY(
        xValues,
        query.Select(
            group => group.Sum(
                flight => flight.WeatherDelay)).ToList());
```

As you can see, the standard LINQ operators make it easy to slice and manipulate the data for reporting and charting. Another great example is in the TaxiTimeChartBuilder. This class assembles a radar chart to display the total "taxi out" time for each day of the week. The "taxi out" time is the amount of time an airplane spends between leaving the gate and lifting off of the airport runway. At congested airports the "taxi out" time can soar as planes queue up and wait for the runway to clear. The TaxiTimeChartBuilder highlights the data points where the "taxi out" time exceeds some threshold value. This job is made trivially easy using a LINQ query against the data points in a series:

Copy Code

```
var overThresholdPoints =
    taxiOutSeries.Points
                 .Where(p => p.YValues.First() > _taxiThreshold);
foreach (var point in overThresholdPoints)
{
    point.Color = Color.Red;
}
```

Here the color of every data point that exceeds the specified threshold is changed to red. This behavior might make you think about an airport dashboard report, so let's look at dashboards next.

From Charts to Dashboards

In the business intelligence circles, dashboards are used to display key performance information about a business in graphical fashion. This performance information can come from various sources and can be visualized using a series of charts and widgets. A user should be able to glance at a dashboard display and quickly identify any anomalies that might affect the business.

One of the challenges in producing a dashboard display is performance. Putting together all of the information for a dashboard can result in a multitude of queries to both relational and multi-dimensional databases. Even when the raw data for a dashboard is cached, the sheer number of charts and widgets on a dashboard can burden a server.

Figure 9 shows a simple dashboard for an airport displaying the top destinations, taxi time by day of week, total flights for each day of the week, and the total delay time for each day of the month. One approach to building this dashboard would be to place all four chart controls on a single Web page and use the chart builder classes we've been building to assemble the charts. However, imagine if each chart required two seconds to build. Normally, two seconds is not a long time to wait for a complicated query, but since I have a total of four charts on the page (which is a small number for a dashboard), the user will be waiting at least eight seconds for the first graphic to appear.

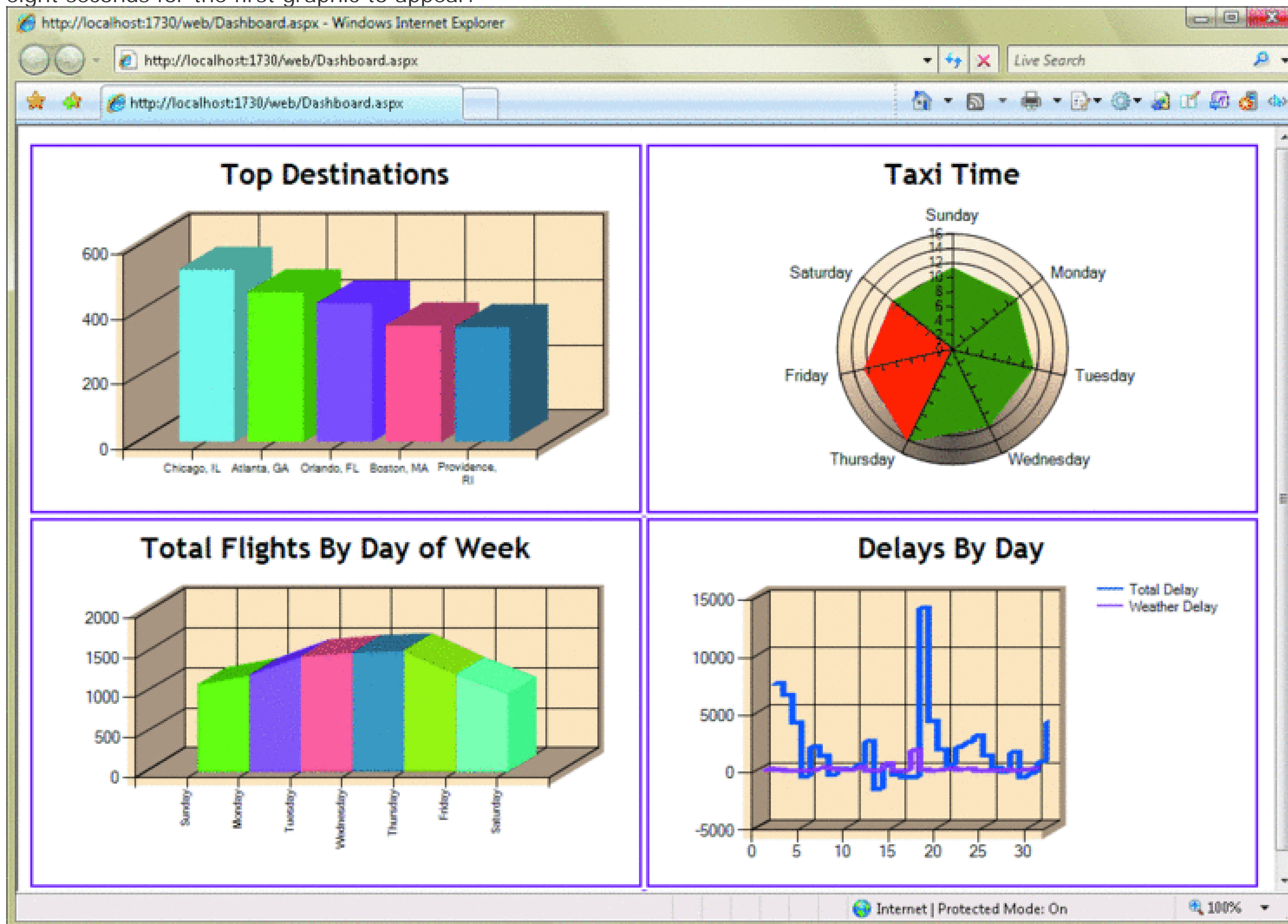

Figure 9 **Airport Activity Dashboard**

A different approach to building this dashboard page is to define placeholders for each chart in the page and build the chart images asynchronously. An asynchronous approach would allow the user to begin seeing the

first result as soon as it is available. This solution could leverage Windows Communication Foundation (WCF) proxies in JavaScript to display charts as they become available.
Fortunately, the builder classes I've defined for the charts make it easier to move the chart generation logic behind WCF Web services. The code in **Figure 10** shows a WCF service method that first constructs an empty chart, and then constructs one of the ChartBuilder-derived classes implemented in the project. The builder is specified as a parameter to the method, and the code checks this parameter against a map of known builders to find the actual type of builder to instantiate (using Activator.CreateInstance).

Figure 10 Instantiating a Builder

Copy Code
```
[OperationContract]
public string GenerateChart(string builderName) {
    Chart chart = new Chart() {
        Width = 500, Height = 300
    };

    ChartBuilder builder =
        Activator.CreateInstance(_typeMap[builderName], chart)
            as ChartBuilder;
    builder.BuildChart();

    return SaveChart(builderName, chart);
}
```
Using a map of known builder types means we don't have to pass the method an actual type name, and it also provides a layer of insulation against malicious input from over the Internet. We will only instantiate types the service knows about.
Producing the chart image in the Web service is tricky. As we mentioned earlier, the chart works with an HTTP Handler to render charts on the client. Specifically, the Chart control gives the ChartHttpHandler class the bytes representing the chart's finished image. The ChartHttpHandler responds with a unique identifier. When the Chart control renders, it produces a regular HTML <img> tag with the src attribute referencing ChartImg.axd and including the unique identifier in the query string. When this image request reaches the HTTP Handler, the handler can look up the proper chart for display. You can read more details about this process, including all the configuration options, on Delian Tchoparino's blog.
Unfortunately, the APIs for the HTTP Handler are not public, and therefore are not available in a Web service. Instead, the SaveChart method, which the service code in **Figure 10** invokes, uses the SaveImage method of the Chart control to write the chart image to the file system in PNG format. The service then returns the name of the file to the client. By generating physical files we can also introduce a caching strategy and avoid the queries and image generation during periods of heavy load.
The code in **Figure 11** uses the WCF service from JavaScript to set the src attribute of each chart's image placeholder. (See Fritz Onion's column "Client-Side Web Service Calls with AJAX Extensions" to understand how to generate JavaScript proxies and invoke Web services with JavaScript.) The document object model (DOM) identifier and builder class for each chart is defined in a JavaScript array that is part of a "context" object. This context is tunneled through successive Web service invocations in the userState parameter of the Web service proxy methods. The JavaScript uses the context object to track its progress in updating the dashboard charts. For dynamic dashboard pages, the server could dynamically generate the array.

Figure 11 Updating the Chart

Copy Code
```
/// <reference name="MicrosoftAjax.js" />
function pageLoad() {
    var context = {
        index: 0,
        client: new ChartingService(),
        charts:
        [
            { id: "topDestinations", builder: "TopDestinations" },
            { id: "taxiTime", builder: "TaxiTime" },
```

```
            { id: "dayOfWeek", builder: "DayOfWeek" },
            { id: "delaysByDay", builder: "DelaysByDay" }
        ]
    };

    context.client.GenerateChart(
        context.charts[context.index].builder,
        updateChart,
        displayError,
        context);
}

function updateChart(result, context) {
    var img = $get(context.charts[context.index].id);
    img.src = result;
    context.index++;
    if (context.index < context.charts.length) {
        context.client.GenerateChart(
            context.charts[context.index].builder,
            updateChart, displayError, context);
    }
}

function displayError() {
    alert("There was an error creating the dashboard charts");
}
```

Charting the Future

I've covered quite a bit of technology, including the template method design pattern, LINQ operators, and JavaScript-compatible WCF Web services. Since this column demonstrated only a small fraction of the features available in the chart control, you should be sure to check the sample Web site to see the amazing breadth of features available. Combining this great visualization tool with the flexibility and expressiveness of LINQ means that your future charting applications will be more flexible, effective, and useful.

K. Scott Allen is a member of the Pluralsight technical staff and the founder of OdeToCode. You can reach Scott at scott@OdeToCode.com or read his blog at odetocode.com/blogs/scott.