ComponentOne

OLAP for ASP.NET AJAX Beta

Corporate Headquarters
ComponentOne LLC
201 South Highland Avenue
3rd Floor
Pittsburgh, PA 15206 • USA

Internet: info@ComponentOne.com
Web site: http://www.componentone.com

Sales
E-mail: sales@componentone.com
Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

Trademarks

The ComponentOne product name is a trademark and ComponentOne is a registered trademark of ComponentOne LLC. All other trademarks used herein are the properties of their respective owners.

Warranty

ComponentOne warrants that the original CD (or diskettes) are free from defects in material and workmanship, assuming normal use, for a period of 90 days from the date of purchase. If a defect occurs during this time, you may return the defective CD (or disk) to ComponentOne, along with a dated proof of purchase, and ComponentOne will replace it at no charge. After 90 days, you can obtain a replacement for a defective CD (or disk) by sending it and a check for $25 (to cover postage and handling) to ComponentOne.

Except for the express warranty of the original CD (or disks) set forth here, ComponentOne makes no other warranties, express or implied. Every attempt has been made to ensure that the information contained in this manual is correct as of the time it was written. We are not responsible for any errors or omissions. ComponentOne's liability is limited to the amount you paid for the product. ComponentOne is not liable for any special, consequential, or other damages for any reason.

Copying and Distribution

While you are welcome to make backup copies of the software for your own use and protection, you are not permitted to make copies for the use of anyone else. We put a lot of time and effort into creating this product, and we appreciate your support in seeing that it is used by licensed users only.

This manual was produced using ComponentOne Doc-To-Help™.

Table of Contents

ComponentOne OLAP for ASP.NET AJAX Overview

Create grids, charts, and ad-hoc reports that can be saved, exported, or printed in no time with **ComponentOne OLAP for ASP.NET AJAX**. Use a single control, **C1OlapPage**, which provides a complete OLAP user interface, or customize your application with the C1OlapPanel, C1OlapGrid, C1OlapChart, and C1OlapPage controls.

Note: This version of **Olap for ASP.NET AJAX** is a beta release.

Namespaces

Namespaces organize the objects defined in an assembly. Assemblies can contain multiple namespaces, which can in turn contain other namespaces. Namespaces prevent ambiguity and simplify references when using large groups of objects such as class libraries.

The namespace for the **OLAP for ASP.NET AJAX** components is **C1.Web.Olap**. The following code fragment shows how to declare a C1OlapPage component using the fully qualified name for this class:

- Visual Basic
```
Dim OlapPage1 As C1.Web.Olap.C1OlapPage
```
- C#
```
C1.Web.Olap .C1OlapPage OlapPage1;
```

Namespaces address a problem sometimes known as *namespace pollution*, in which the developer of a class library is hampered by the use of similar names in another library. These conflicts with existing components are sometimes called *name collisions*.

For example, if you create a new class named C1OlapPage, you can use it inside your project without qualification. However, the **C1.Web.Olap** assembly also implements a class called C1OlapPage. So, if you want to use the C1OlapPage class in the same project, you must use a fully qualified reference to make the reference unique. If the reference is not unique, Visual Studio .NET produces an error stating that the name is ambiguous. The following code snippet demonstrates how to declare these objects:

- Visual Basic
```
' Define a new C1OlapPage object
Dim MyOlapPage as C1OlapPage
' Define a new C1Olap.C1OlapPage object.
Dim C1OlapPage as C1.Web.Olap.C1OlapPage
```
- C#
```
// Define a new C1OlapPage object
C1OlapPage MyOlap;
// Define a new C1Olap.C1OlapPage object.
C1.Web.Olap.C1Olap C1OlapPage;
```

Fully qualified names are object references that are prefixed with the name of the namespace where the object is defined. You can use objects defined in other projects if you create a reference to the class (by choosing **Add Reference** from the **Project** menu) and then use the fully qualified name for the object in your code.

Fully qualified names prevent naming conflicts because the compiler can always determine which object is being used. However, the names themselves can get long and cumbersome. To get around this, you can use the **Imports** statement (**using** in C#) to define an alias — an abbreviated name you can use in place of a fully qualified name. For example, the following code snippet creates aliases for two fully qualified names, and uses these aliases to define two objects:

- Visual Basic

```
Imports C1OlapPage = C1.Web.Olap.C1OlapPage
Imports MyOlapPage = MyProject.C1OlapPage

Dim s1 As C1OlapPage
Dim s2 As MyOlapPage
```

- C#

```
using C1OlapPage = C1.Web.Olap.C1OlapPage;
using MyOlapPage = MyProject.C1OlapPage;

 C1OlapPage s1;
 MyOlapPage s2;
```

If you use the **Imports** statement without an alias, you can use all the names in that namespace without qualification, provided they are unique to the project.

Creating an AJAX-Enabled ASP.NET Project

ComponentOne TabControl for ASP.NET AJAX requires you to create an ASP.NET AJAX-Enabled project so that Microsoft ASP.NET AJAX Extensions and a **ScriptManager** control are included in your project before the controls are placed on the page. This allows you to take advantage of ASP.NET AJAX and certain features such as partial-page rendering and client-script functionality of the Microsoft AJAX Library.

When creating AJAX-Enabled ASP.NET projects, Visual Studios 2008 and 2005 both give you the option of creating a Web site project or a Web application project. MSDN provides detailed information on why you would choose one option over the other.

If you are using Visual Studio 2008 with .NET Framework 2.0 or .NET Framework 3.0 or if you are using Visual Studio 2005, you must install the ASP.NET AJAX Extensions 1.0, which can be found at http://ajax.asp.net/. Additionally for Visual Studio 2005 users, creating a Web application project requires installation of a Visual Studio 2005 update and add-in, which can be found at http://msdn.microsoft.com/; however, if you have Visual Studio 2005 SP1, Web application project support is included and a separate download is not required.

If you are using Visual Studio 2008 and .NET Framework 3.5, you can easily create an AJAX-enabled ASP.NET project without installing separate add-ins because the framework has a built-in AJAX library and controls.

Note: If you are using Visual Studio 2010, see http://www.asp.net/ajax/ for more information on creating an AJAX-Enabled ASP.NET Project.

The following table summarizes the installations needed:

Visual Studio Version	Additional Installation Requirements
Visual Studio 2008, .NET Framework 3.5	None
Visual Studio 2008 and .NET Framework 2.0 or 3.0	ASP.NET AJAX Extensions 1.0

Visual Studio 2005 Service Pack 1	http://www.asp.net/ajax/downloads/archive/
Visual Studio 2005	ASP.NET AJAX Extensions 1.0 Visual Studio update and add-in (2 installs for Web application project support)

The following topics explain how to create both types of projects in Visual Studio 2008 and 2005.

- Creating an AJAX-Enabled Web Site Project in Visual Studio 2008

  To create a Web site project in Visual Studio 2008, complete the following steps:

  1. From the **File** menu, select **New** | Web Site. The New Web Site dialog box opens.
  2. Select **.NET Framework 3.5** or the desired framework in the upper right corner. Note that if you choose .NET Framework 2.0 or 3.0, you must install the extensions first.
  3. In the list of templates, select **AJAX 1.0-Enabled ASP.NET 2.0 Web Site**.
  4. Click **Browse** to specify a location and then click **OK**.

     Note: The Web server must have IIS version 6 or later and the .NET Framework installed on it. If you have IIS on your computer, you can specify http://localhost for the server.

     A new AJAX-Enabled Web Site is created at the root of the Web server you specified. In addition, a new Web Forms page called Default.aspx is displayed and a **ScriptManager** control is placed on the form. The **ScriptManger** is needed to enable certain features of ASP.NET AJAX such as partial-page rendering, client-script functionality of the Microsoft AJAX Library, and Web-service calls.

- Creating an AJAX-Enabled Web Application Project in Visual Studio 2008

  To create a new Web application project in Visual Studio 2008, complete the following steps.

  1. From the **File** menu, select **New** | Project. The New Project dialog box opens.
  2. Select **.NET Framework 3.5** or the desired framework in the upper right corner. Note that if you choose .NET Framework 2.0 or 3.0, you must install the extensions first.
  3. Under **Project Types**, choose either **Visual Basic** or **Visual C#** and then select **Web**. Note that one of these options may be located under **Other Languages**.
  4. Select **AJAX 1.0-Enabled ASP.NET 2.0 Web Application** from the list of Templates in the right pane.
  5. Enter a URL for your application in the **Location** field and click **OK**.

     Note: The Web server must have IIS version 6 or later and the .NET Framework installed on it. If you have IIS on your computer, you can specify http://localhost for the server.

     A new Web Forms project is created at the root of the Web server you specified. In addition, a new Web Forms page called Default.aspx is displayed and a **ScriptManager** control is placed on the form. The **ScriptManger** is needed to enable certain features of ASP.NET AJAX such as partial-page rendering, client-script functionality of the Microsoft AJAX Library, and Web-service calls.

- Creating an AJAX-Enabled Web Site Project in Visual Studio 2005

To create a Web site project in Visual Studio 2005, complete the following steps:

1. From the **File** menu in Microsoft Visual Studio .NET, select **New Web Site**. The **New Web Site** dialog box opens.
2. Select **ASP.NET AJAX-Enabled Web Site** from the list of Templates.
3. Enter a URL for your site in the **Location** field and click **OK**.

> **Note:** The Web server must have IIS version 6 or later and the .NET Framework installed on it. If you have IIS on your computer, you can specify http://localhost for the server.

A new Web Forms project is created at the root of the Web server you specified. In addition, a new Web Forms page called Default.aspx is displayed and a **ScriptManager** control is placed on the form. The **ScriptManger** is needed to enable certain features of ASP.NET AJAX such as partial-page rendering, client-script functionality of the Microsoft AJAX Library, and Web-service calls.

- Creating an AJAX-Enabled Web Application Project in Visual Studio 2005

  To create a new Web application project in Visual Studio 2005, complete the following steps.

  1. From the **File** menu in Microsoft Visual Studio 2005, select **New Project**. The **New Project** dialog box opens.
  2. Under **Project Types**, choose either **Visual Basic Projects** or **Visual C# Projects**. Note that one of these options may be located under **Other Languages**.
  3. Select **ASP.NET AJAX-Enabled Web Application** from the list of Templates in the right pane.
  4. Enter a URL for your application in the **Location** field and click **OK**.

  > **Note:** The Web server must have IIS version 6 or later and the .NET Framework installed on it. If you have IIS on your computer, you can specify http://localhost for the server.

  A new Web Forms project is created at the root of the Web server you specified. In addition, a new Web Forms page called Default.aspx is displayed and a **ScriptManager** control is placed on the form. The **ScriptManger** is needed to enable certain features of ASP.NET AJAX such as partial-page rendering, client-script functionality of the Microsoft AJAX Library, and Web-service calls.

Adding the OLAP for ASP.NET AJAX Components to a Project

When you install **ComponentOne OLAP for ASP.NET AJAX**, the **Create a ComponentOne Visual Studio Toolbox Tab** checkbox is checked, by default, in the installation wizard. When you open Visual Studio, you will notice a **ComponentOne OLAP for ASP.NET AJAX** tab containing the ComponentOne controls has automatically been added to the Toolbox.

If you decide to uncheck the **Create a ComponentOne Visual Studio Toolbox Tab** checkbox during installation, you can manually add ComponentOne controls to the Toolbox at a later time.

ComponentOne OLAP for ASP.NET AJAX provides the following controls:

- **C1OlapPanel**
- **C1OlapPage**
- **C1OlapPrintDocument**
- **C1OlapChart**
- **C1OlapGrid**

- **C1PdfDocument**
- **C1FlexGrid**
- **C1Chart**

To use these controls, add them to the form or add a reference to the C1.Web.Olap assembly to your project.

Adding OLAP for ASP.NET AJAX Controls to the Toolbox

To add **ComponentOne OLAP for ASP.NET AJAX** controls to the Visual Studio Toolbox:

1. Open the Visual Studio IDE (Microsoft Development Environment). Make sure the Toolbox is visible (select **Toolbox** in the **View** menu if necessary) and right-click it to open the context menu.
2. To make the **OLAP for ASP.NET AJAX** components appear on their own tab in the Toolbox, select **Add Tab** from the context menu and type in the tab name, **C1Olap**, for example.
3. Right-click the tab where the components are to appear and select **Choose Items** from the context menu. The **Choose Toolbox Items** dialog box opens.
4. In the dialog box, select the **.NET Framework Components** tab. Sort the list by Namespace (click the Namespace column header) and check the check boxes for the components belonging to namespace C1.Web.Olap. Note that there may be more than one component for each namespace.

Adding OLAP for ASP.NET AJAX Controls to the Form

To add **OLAP for ASP.NET AJAX** to a form:

1. Add the **OLAP for ASP.NET AJAX** controls to the Visual Studio Toolbox.
2. Double-click a control or drag it onto your form.

Note that when you add a **C1OlapPanel**, **C1OlapPage**, **C1OlapPrintDocument**, **C1OlapChart**, or **C1OlapGrid** control to your form, references to all four assemblies (C1.C1Pdf.2.dll, C1.Win.C1Chart.2.dll, C1.Win.C1FlexGrid.2.dll, and C1.Web.Olap) are added to your project.

Adding a Reference to the C1.Web.Olap Assembly

To add a reference to the assembly:

1. Select the **Add Reference** option from the **Project** menu of your project.
2. Select the **ComponentOne C1Olap** assembly from the list on the **.NET** tab or browse to find the C1.Web.Olap.dll file and click **OK**.
3. Double-click the form caption area to open the code window. At the top of the file, add the following **Imports** statements (**using** in C#):
4. Imports C1.Web.Olap

Note: This makes the objects defined in the C1.Web.Olap assembly visible to the project. See Namespaces (page 1) for more information.

What is OLAP for ASP.NET AJAX

OLAP for ASP.NET AJAX is a suite of .NET controls that provide analytical processing features similar to those found in Microsoft Excel's Pivot Tables and Pivot Charts.

OLAP for ASP.NET AJAX takes raw data in any format and provides an easy-to-use interface so users can quickly and intuitively create summaries that display the data in different ways, uncovering trends and providing valuable insights interactively. As the user modifies the way in which he wants to see the data, **OLAP for ASP.NET AJAX** instantly provides grids, charts, and reports that can be saved, exported, or printed.

Introduction to OLAP for ASP.NET AJAX

OLAP means "online analytical processing". It refers to technologies that enable the dynamic visualization and analysis of data.

Typical OLAP tools include "OLAP cubes", and pivot tables such as the ones provided by Microsoft Excel. These tools take large sets of data and summarize it by grouping records based on a set of criteria. For example, an OLAP cube might summarize sales data grouping it by product, region, and period. In this case, each grid cell would display the total sales for a particular product, in a particular region, and for a specific period. This cell would normally represent data from several records in the original data source.

OLAP tools allow users to redefine these grouping criteria dynamically (on-line), making it easy to perform ad-hoc analysis on the data and discover hidden patterns.

For example, consider the following table:

Date	Product	Region	Sales
Oct 2007	Product A	North	12
Oct 2007	Product B	North	15
Oct 2007	Product C	South	4
Oct 2007	Product A	South	3
Nov 2007	Product A	South	6
Nov 2007	Product C	North	8
Nov 2007	Product A	North	10
Nov 2007	Product B	North	3

Now suppose you were asked to analyze this data and answer questions such as:

- Are sales going up or down?
- Which products are most important to the company?
- Which products are most popular in each region?

In order to answer these simple questions, you would have to summarize the data to obtain tables such as these:

Sales by Date and by Product

Date	Product A	Product B	Product C	Total
Oct 2007	15	15	4	**34**
Nov 2007	16	3	8	**27**
Total	**31**	**18**	**12**	**61**

Sales by Product and by Region

Product	North	South	Total
Product A	22	9	**31**
Product B	18		**18**
Product C	8	4	**12**
Total	**48**	**13**	**61**

Each cell in the summary tables represents several records in the original data source, where one or more values fields are summarized (sum of sales in this case) and categorized based on the values of other fields (date, product, or region in this case).

This can be done easily in a spreadsheet, but the work is tedious, repetitive, and error-prone. Even if you wrote a custom application to summarize the data, you would probably have to spend a lot of time maintaining it to add new views, and users would be constrained in their analyses to the views that you implemented.

OLAP tools allow users to define the views they want interactively, in ad-hoc fashion. They can use pre-defined views or create and save new ones. Any changes to the underlying data are reflected automatically in the views, and users can create and share reports showing these views. In short, OLAP is a tool that provides flexible and efficient data analysis.

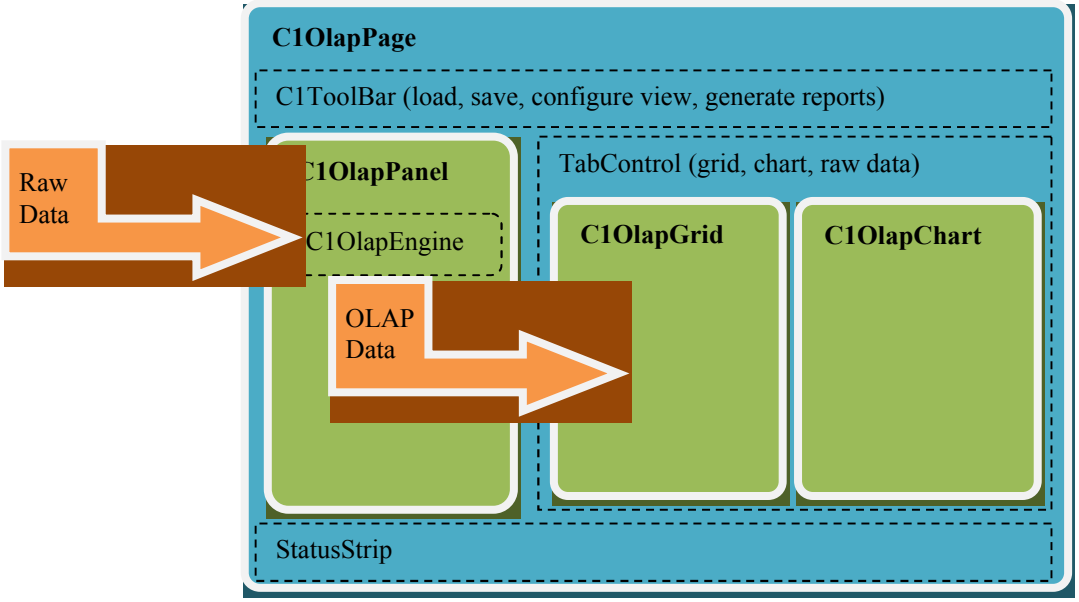
OLAP for ASP.NET AJAX Architecture

OLAP for ASP.NET AJAX includes the following controls:

C1OlapPage

The **C1OlapPage** control is the easiest way to develop OLAP applications quickly and easily. It provides a complete OLAP user interface built using the other controls in **OLAP for ASP.NET AJAX**. The **C1OlapPage** object model exposes the inner controls, so you can easily customize it by adding or remove interface elements. If you want more extensive customization, the source code is included and you can use it as a basis for your own implementation.

The diagram below shows how the C1OlapPage is organized:



C1OlapPanel

The **C1OlapPanel** control is the core of the **OLAP for ASP.NET AJAX** product. It has a **DataSource/DataSourceID** property that takes raw data as input, and an **OlapTable** property that provides custom views summarizing the data according to criteria provided by the user. The **OlapTable** is a regular **DataTable** object that can be used as a data source for any regular control.

The C1OlapPanel also provides the familiar, Excel-like drag and drop interface that allows users to define custom views of the data. The control displays a list containing all the fields in the data source, and users can drag the fields to lists that represent the row and column dimensions of the output table, the values summarized in the output data cells, and the fields used for filtering the data.

At the core of the C1OlapPanel control, there is a C1OlapEngine object that that is responsible for summarizing the raw data according to criteria selected by the user. These criteria are represented by C1OlapField objects, which contain a connection a specific column in the source data, filter criteria, formatting and summary options. The user creates custom views by dragging C1OlapField objects from the source **Fields** list to one of four auxiliary lists: the **RowFields**, **ColumnFields**, **ValueFields**, and **FilterFields** lists. Fields can be customized using a context menu.

Notice that the **OLAP for ASP.NET AJAX** architecture is open. The **C1OlapPanel** takes any regular collection as a **DataSource**, including data tables, generic lists, and LINQ enumerations; it then summarizes the data and produces a regular **DataTable** as output. **OLAP for ASP.NET AJAX** includes two custom controls that are optimized for displaying the OLAP data, the C1OlapGrid and C1OlapChart, but you could use any other control as well.

C1OlapGrid

The C1OlapGrid control is used to display OLAP tables. It extends the C1GridView control and provides automatic data binding to C1OlapPanel objects, grouped row and column headers, as well as custom behaviors for resizing columns, copying data to the clipboard, and showing details for any given cell.

C1OlapChart

The C1OlapChart control is used to display OLAP charts. It extends the C1WebChart control and provides automatic data binding to **C1OlapPanel** objects, automatic tooltips, chart type and palette selection.

The C1OlapChart control extends the C1WebChart control, our general-purpose charting control. This means the whole C1WebChart object model is also available to **OLAP for ASP.NET** users. For example, you can export the chart to different file formats including PNG and JPG or customize the chart styles and interactivity.

C1OlapReport

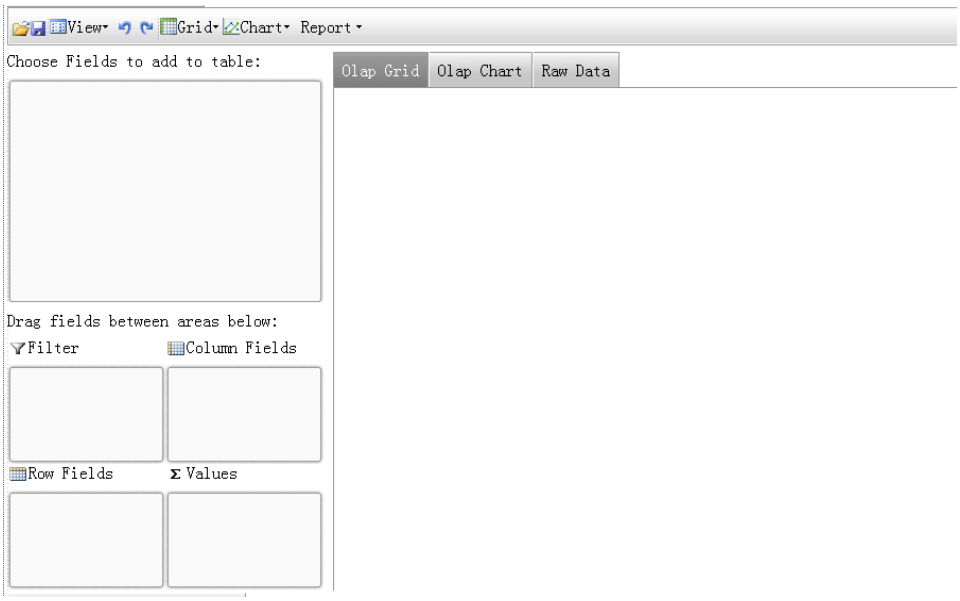
The C1OlapReport is used to create reports based on OLAP views. It extends the C1Report class and provides properties that allow you to specify content and formatting for showing OLAP grids, charts, and the raw data used to create the report.

OLAP for ASP.NET AJAX Quick Start

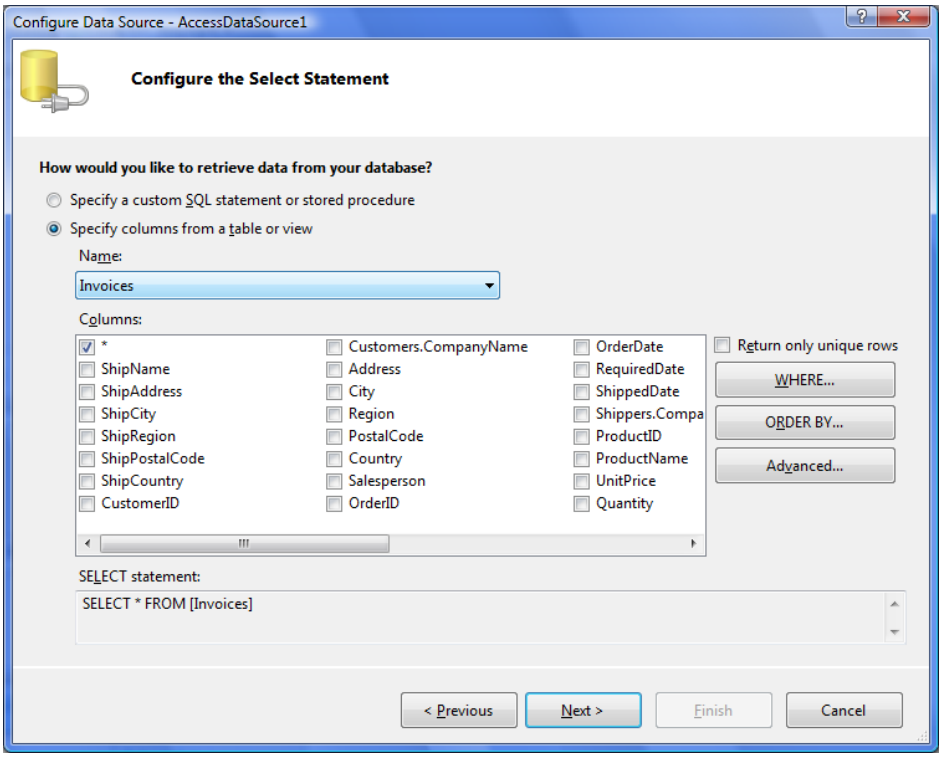
This section presents code walkthroughs that start with the simplest **OLAP for ASP.NET AJAX** application and progress to introduce commonly used features.

An Olap application with no code

To create the simplest **OLAP for ASP.NET AJAX** application, start by creating a new Web site and dragging a **ScriptManager** onto the Page and then dragging a C1OlapPage control onto the Page. Notice that the C1OlapPage control should look like this:



Now, let us -create a data source for the application. Dragging an AccessDataSource onto the page and activate the smart designer by clicking the small rectangle that appears at the top right corner of the control. Click "Configure Data Source…" to configure the data source. For this sample, find the C1NWinddatabase and select the "Invoices" view as shown below:

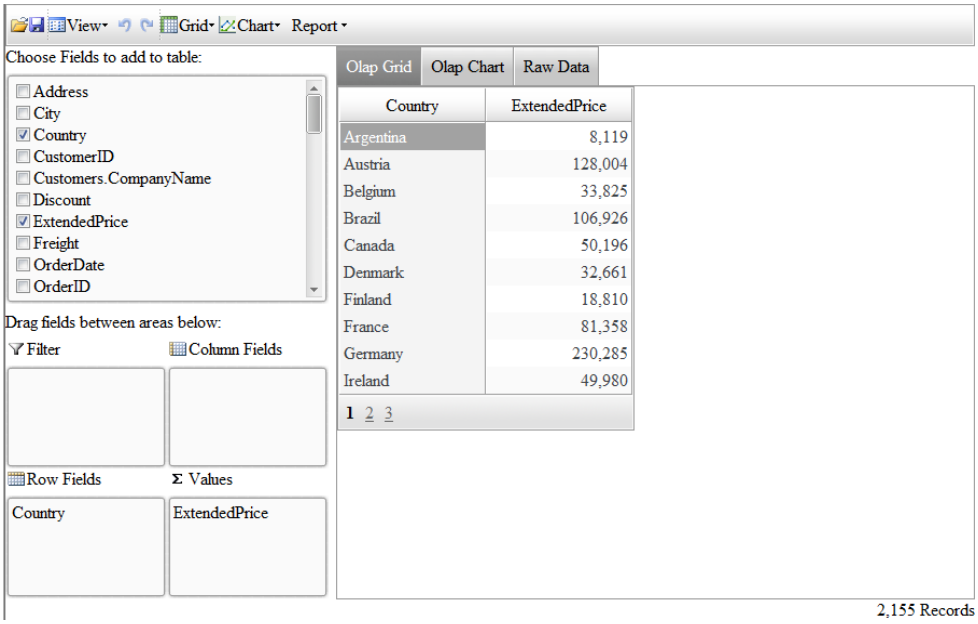


Now, let us select the created data source for the C1OlapPage. Activate the smart designer of by clicking the small rectangle that appears at the top right corner of the C1OlagePage. Use the combobox next to "Choose Data Source" and assign AccessDataSource1 to the control.

The application is now ready. The following sections describe the functionality provided by default, without writing a single line of code.

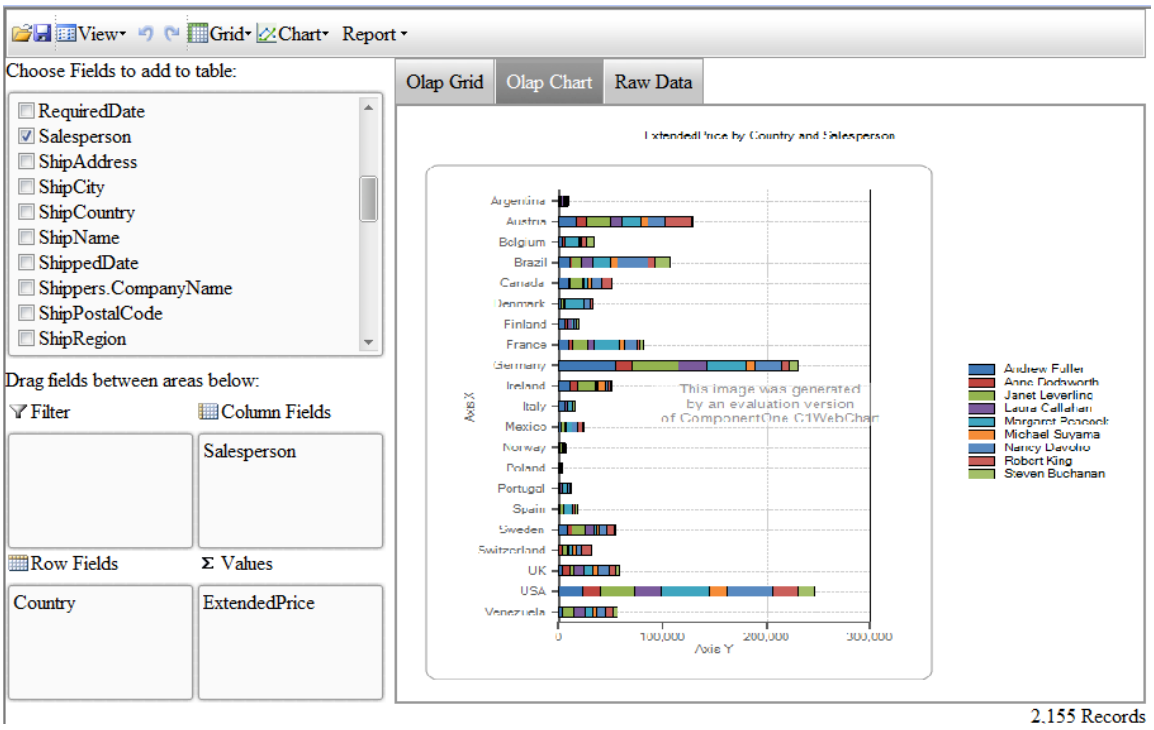
Creating OLAP Views

Run the Web site and you will see an interface similar to the one in Microsoft Excel. Drag the "Country" field to the "Row Fields" list and "ExtendedPrice" to the "Value Fields" list, and you will see a summary of prices charged by country as shown below:

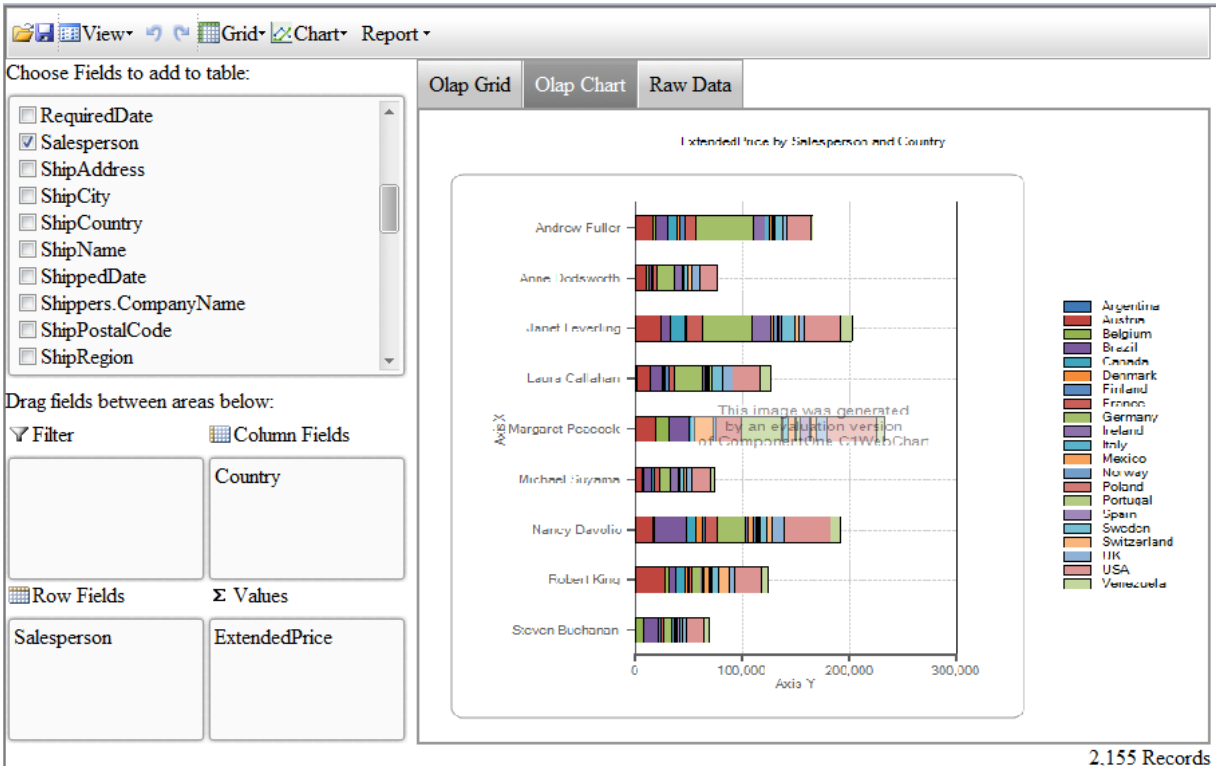


Click the "Olap Chart" tab and you will see the same data in chart format, showing that the main customers are the US, Germany, and Austria.

Now drag the "SalesPerson" field into the "Column Fields" list to see a new summary, this time of sales per country and per sales person. If you still have the chart tab selected, you should be looking at a chart similar to the previous one, except this time the bars are split to show how much was sold by each salesperson:



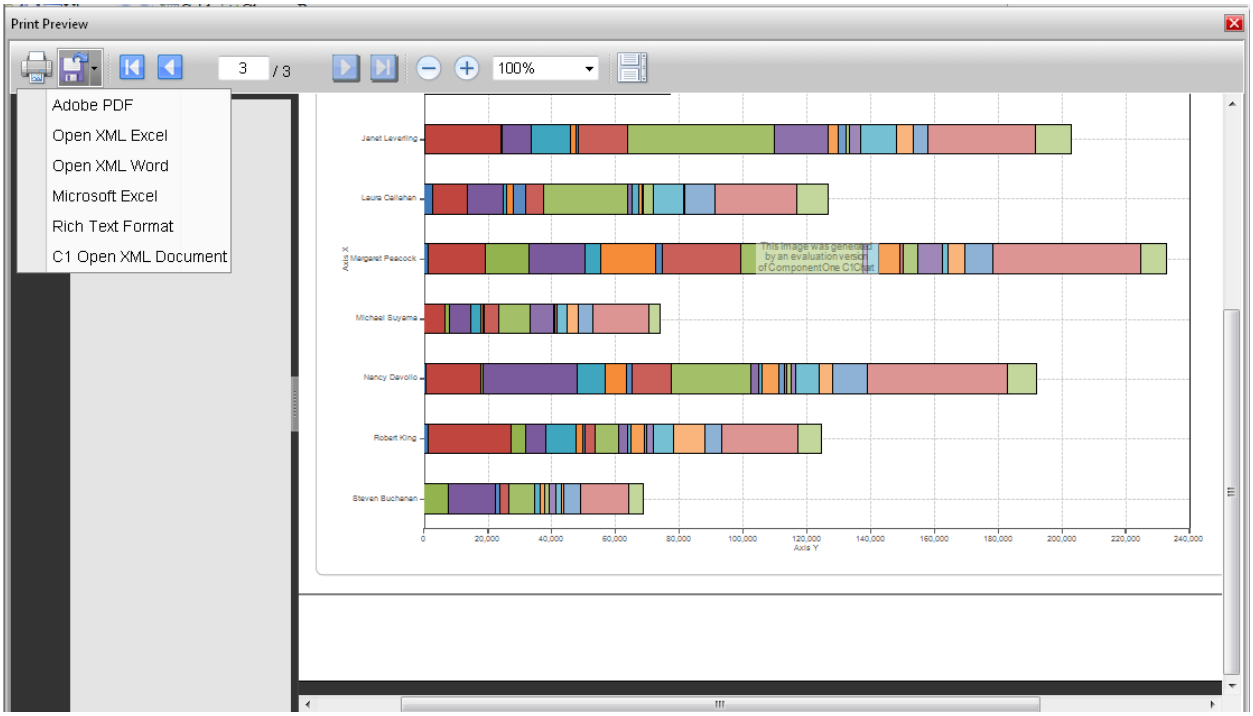
Now create a new view by swapping the “SalesPerson” and “Country” fields by dragging them to the opposite lists. This will create a new chart that emphasizes salesperson instead of country:



The chart shows that Margaret Peacock was the top salesperson in the period being analyzed, followed closely by Janet Leverling and Nancy Davolio.

Creating OLAP Reports

This is an interesting chart, so let's create a report that we can e-mail to other people in the company. Click the "Report" button at the top of the page and you will see a preview showing the data on the previous pages and the chart on the last page. The report should look like this:



Now you can print the report or click the split button with the export icons to generate a various types of file that you can send to others or post on the web.

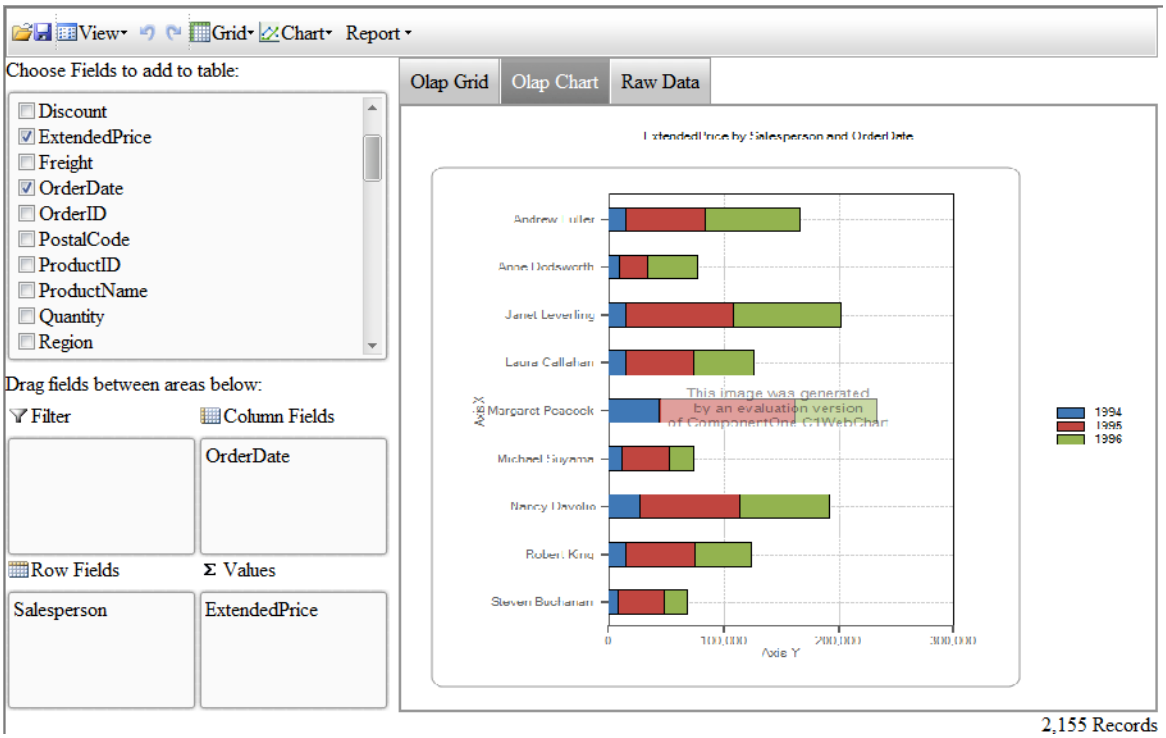
Summarizing Data

Before we move on to the next example, let's create a new view to illustrate how you can easily summarize data in different ways.

This time, drag the "SalesPerson" field to the "Row Fields" list and the "OrderDate" field to the "Column Fields" list. The resulting view contains one column for each day when an order was placed. This is not very useful information, because there are too many columns to show any trends clearly. We would like to summarize the data by month or year instead.

One way to do this would be to modify the source data, either by creating a new query in SQL or by using LINQ. Both of these techniques will be described in later sections. Another way is simply to modify the parameters of the "OrderDate" field. To do this, right-click the "OrderDate" field and select the "Field Settings" menu. Then select the "Format" tab in the dialog, choose the "Custom" format, enter "yyyy", and click OK.

The dates are now formatted and summarized by year, and the OLAP chart looks like this:

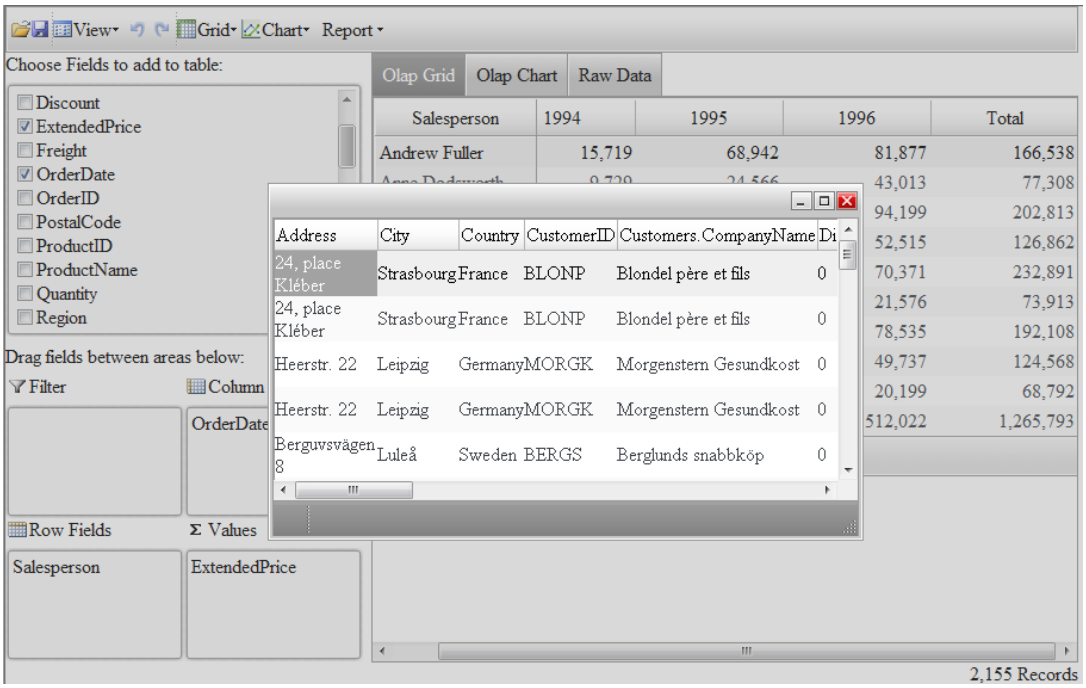


If you wanted to check how sales are placed by month or weekday, you could simply change the format to "MM" or "dddd".

Drilling Down on the Data

As we mentioned before, each cell in the OLAP grid represents a summary of several records in the data source. You can see the underlying records behind each cell in the OLAP grid by right clicking it with the mouse.

To see this, click the "Olap Grid" tab and right-click the first cell on the grid, the one that represents Andrew Fullers's sales in 1994. You will see another grid showing the 31 records that were used to compute the total displayed in the Olap grid:



Customizing the C1OlapPage

The previous example showed how you can create a complete OLAP application using only a C1OlapPage control and no code at all. This is convenient, but in most cases you will want to customize the application and the user interface to some degree.

Creating Predefined Views

In addition to the ViewDefinition property, which gets or sets the current view as an XML string, the C1OlapPage control also exposes **ReadXml** and **WriteXml** methods that allow you to persist views to files and streams. These methods are automatically invoked by the C1OlapPage when you click the "Load" and "Save" buttons in the built-in toolbar.

These methods allow you to implement predefined views very easily. To do this, start by creating some views and saving each one by pressing the "Save" button. For this sample, we will create five views showing sales by:

1. Product and Country
2. Salesperson and Country
3. Salesperson and Year
4. Salesperson and Month
5. Salesperson and Weekday

Once you have created and saved all the views, create a new XML file called "DefaultViews.xml" with a single "OlapViews" node, then copy and paste all your default views into this document. Next, add an "id" tag to each view and assign each one a unique name. This name will be shown in the user interface (it is not required by **C1Olap**). Your XML file should look like this:

```
<OlapViews>

  <C1OlapPage id="Product vs Country">
    <!-- view definition omitted... -->
  <C1OlapPage id="SalesPerson vs Country">
```

```
    <!-- view definition omitted... -->
  <C1OlapPage id="SalesPerson vs Year">
    <!-- view definition omitted... -->
  <C1OlapPage id="SalesPerson vs Month">>
    <!-- view definition omitted... -->
  <C1OlapPage id="SalesPerson vs Weekday">
    <!-- view definition omitted... -->

</OlapViews>
```

Now add this file to the Web site as a resource.

Now that the view definitions are ready, we need to expose them in a menu so the user can select them. To do this, set the property "ViewPath" as ViewPath="~/View/OlapViews.xml" :

If you need further customization, you can also choose not to use the C1OlapPage at all, and build your interface using the lower-level C1OlapPanel, C1OlapGrid, and C1OlapChart controls. The source code for the C1OlapPage control is included with the package and can be used as a starting point. The example in the "Building a custom User Interface" section shows how this is done.

Large data sources

All the examples discussed so far loaded all the data into memory. This is a simple and convenient way to do things, and it works in many cases.

In some cases, however, there may be too much data to load into memory at once. Consider for example a table with a million rows or more. Even if you could load all this data into memory, the process would take a long time.

There are many ways to deal with these scenarios. You could create queries that summarize and cache the data on the server, or use specialized OLAP data providers. In either case, you would end up with tables that can be used with **OLAP for ASP.NET AJAX**.

But there are also simpler options. Suppose the database contains information about thousands of companies, and users only want to see a few at a time. Instead of relying only on the filtering capabilities of **C1WebOlap**, which happen on the client, you could delegate some of the work to the server, and load only the companies the user wants to see. This is easy to accomplish and does not require any special software or configurations on the server.

For example, consider the following **CachedDataTable** class (this class is used in the "SqlFilter" sample installed with **OLAP for ASP.NET AJAX**):

```
/// <summary>
/// Extends the <see cref="DataTable"/> class to load and cache
/// data on demand using a <see cref="Fill"/> method that takes
/// a set of keys as a parameter.
/// </summary>
class CachedDataTable : DataTable
{
  public string ConnectionString { get; set; }
  public string SqlTemplate { get; set; }
  public string WhereClauseTemplate { get; set; }
  Dictionary<object, bool> _values =
     new Dictionary<object, bool>();

  // constructor
  public CachedDataTable(string sqlTemplate,
    string whereClauseTemplate, string connString)
  {
```

```
    ConnectionString = connString;
    SqlTemplate = sqlTemplate;
    WhereClauseTemplate = whereClauseTemplate;
  }

  // populate the table by adding any missing values
  public void Fill(IEnumerable filterValues, bool reset)
  {
    // reset table if requested
    if (reset)
    {
      _values.Clear();
      Rows.Clear();
    }

    // get a list with the new values
    List<object> newValues = GetNewValues(filterValues);
    if (newValues.Count > 0)
    {
      // get sql statement and data adapter
      var sql = GetSqlStatement(newValues);
      using (var da = new OleDbDataAdapter(sql, ConnectionString))
      {
        // add new values to the table
        int rows = da.Fill(this);
      }
    }
  }
  public void Fill(IEnumerable filterValues)
  {
    Fill(filterValues, false);
  }
```

This class extends the regular **DataTable** class and provides a **Fill** method that can either repopulate the table completely or add additional records based on a list of values provided. For example, you could start by filling the table with two customers (out of several thousand) and then add more only when the user requested them.

Note that the code uses an **OleDbDataAdapter**. This is because the sample uses an MDB file as a data source and an OleDb-style connection string. To use this class with Sql Server data sources, you would replace the **OleDbDataAdapter** with a **SqlDataAdapter.**

The code above is missing the implementation of two simple methods given below:

```
  // gets a list with the filter values that are not already in the
  // current values collection;
  // and add them all to the current values collection.
  List<object> GetNewValues(IEnumerable filterValues)
  {
    var list = new List<object>();
    foreach (object value in filterValues)
    {
      if (!_values.ContainsKey(value))
      {
        list.Add(value);
        _values[value] = true;
      }
```

```
    }
    return list;
  }

  // gets a sql statement to add new values to the table
  string GetSqlStatement(List<object> newValues)
  {
    return string.Format(SqlTemplate, GetWhereClause(newValues));
  }
  string GetWhereClause(List<object> newValues)
  {
    if (newValues.Count == 0 || string.IsNullOrEmpty(WhereClauseTemplate))
    {
      return string.Empty;
    }

    // build list of values
    StringBuilder sb = new StringBuilder();
    foreach (object value in newValues)
    {
      if (sb.Length > 0) sb.Append(", ");
      if (value is string)
      {
        sb.AppendFormat("'{0}'", ((string)value).Replace("'", "''"));
      }
      else
      {
        sb.Append(value);
      }
    }

    // build where clause
    return string.Format(WhereClauseTemplate, sb);
  }
}
```

The **GetNewValues** method returns a list of values that were requested by the user but are still not present in the **DataTable**. These are the values that need to be added.The **GetSqlStatement** method builds a new SQL statement with a WHERE clause that loads the records requested by the user that haven't been loaded yet. It uses string templates provided by the caller in the constructor, which makes the class general.

Now that the **CachedDataTable** is ready, the next step is to connect it with C1WebOlap and enable users to analyze the data transparently, as if it were all loaded in memorTo do this, open the default page and add a **ScriptManager** and a C1OlapPage control to it.

Next, we need to get a complete list of all the customers in the database so the user can select the ones he wants to look at. Note that this is a long list but compact list. It contains only the customer name, not any of the associated details such as orders, order details, etc. Here is the code that loads the full customer list:

```
//get complete list of customers.
var customerList = new List<string>();
var sql = @"SELECT DISTINCT Customers.CompanyName AS [Customer] FROM Customers";
var da = new OleDbDataAdapter(sql, GetConnectionString());
var dt = new DataTable();
da.Fill(dt);
foreach (DataRow dr in dt.Rows)
```

```
{
      customerList.Add((string)dr["Customer"]);
}
ViewState["CustomerList"] = customerList;
```

Next, we need a list that contains the customers that the customer wants to look at. We should add a hidden input onto the page to store the ShowValues. <input type="hidden" id="ShowValuesInput__jsonserverstate" runat="server" value = "Hanari Carnes,Hungry Coyote Import Store,Island Trading,Laughing Bacchus Wine Cellars,Rancho grande,La maison d'Asie,La corne d'abondance" />.

And here is the code that loads the "active" customer list from the new setting:

```
var showValues = ShowValuesInput__jsonserverstate.Value;
List<string> activeCustomerList = new List<string>();
string[] values = showValues.Split(new char[]{','},
StringSplitOptions.RemoveEmptyEntries);
for (int i = 0; i < values.Length; i++)
{
      activeCustomerList.Add(values[i]);
}
```

Now we are ready to create a **CachedDataTable** and assign it to the DataSource property:

```
var dtSales = new CachedDataTable(GetConnectionString());
dtSales.Fill(activeCustomerList);
C1OlapPage1.DataSource = dtSales;
C1OlapPage1.DataBind();
if (activeCustomerList != null && activeCustomerList.Count > 0)
{
var field = C1OlapPage1.OlapPanel.Fields["Customer"];
var filter = field.Filter;
filter.ShowValues = activeCustomerList.ToArray();
}
```

The **CachedDataTable** constructor uses three parameters:

SqlTemplate
This is a standard SQL SELECT statement where the "WHERE" clause is replaced by a placeholder. The statement is fairly long, and is defined as an application resource. To see the actual content please refer to the "SqlFilter" sample.

WhereTemplate
This is a standard SQL WHERE statement that contains a template that will be replaced with the list of values to include in the query. It is also defined as an application resource which contains this string: "WHERE Customers.CompanyName in ({0})".

ConnectionString
This parameter contains the connection string that is used to connect to the database. Our sample uses the same GetConnectionString method introduced earlier, that returns a reference to the NorthWind database installed with **OLAP for ASP.NET AJAX**.

Now that the data source is ready, we need to connect it to C1WebOlap to ensure that:

1. The user can see all the customers in the C1WebOlap filter (not just the ones that are currently loaded) and
2. When the user modifies the filter, new data is loaded to show any new customers requested.

To accomplish item 1, we need to assign the complete list of customers to the Values property. This property contains a list of the values that are displayed in the filter. By default, **OLAP for ASP.NET AJAX** populates this list with values found in the raw data. In this case, the raw data contains only a partial list, so we need to provide the complete version instead.

To accomplish item 2, we need to listen to the Propertychanged event at client side, which fires when the user modifies any field properties including the filter and C1OlapPanel.Updating event at the client side. When this happens, we retrieve the list of customers selected by the user and post them back to the server and then pass that list to the data source.

This is the code that accomplishes this:

```
C1OlapPage1.OlapPanel.OnClientUpdating = "C1OlapPage1_OlapPanelUpdating";
filter.OnClientPropertyChanged =
"C1OlapPage1_CustomerFilterOnClientPropertyChanged";
```

And here is the event handler that updates the data source when the filter changes at client side:

```
var MAX_CUSTOMERS = 12;
function C1OlapPage1_CustomerFilterOnClientPropertyChanged(sender) {
      var showValues = sender.get_showValues();
      if (showValues.length == 0) {
            alert('You have selected no customers.');
      }
      else if (showValues.length > MAX_CUSTOMERS) {
            alert('You have selected too many customers.');
      }
};

function C1OlapPage1_OlapPanelUpdating(sender) {
      var filter = sender.get_fields().getFieldByName("Customer").get_filter();
      var pState = filter.get_propertiesState();
      var showValues = [];
      var values = pState["Values"];
      var notShowValuesIndexes = pState["NotShowValuesIndexes"];
      var showValuesIndexes = pState["ShowValuesIndexes"];

      if (values != null) {
            if (showValuesIndexes != null) {
                  for (var i = 0; i < showValuesIndexes.length; i++) {
                      var idx = showValuesIndexes[i];
                      if (idx >= 0 && idx <= values.length - 1) {
                           showValues.push(values[idx]);
                      }
                  }
            }
            else if (notShowValuesIndexes != null) {
                  if (notShowValuesIndexes.length == 0) {
                         showValues = values;
                  }
                  else {
                        for (var idx = 0; idx < values.length; idx++) {
                              var i = 0;
                              for (; i < notShowValuesIndexes.length; i++) {
                                    if (idx == i) {
                                          break;
                                    }
```

```
                                }
if (i < notShowValuesIndexes.length) {
                                    continue;
                                }
showValues.push(values[idx]);
                            }
                        }
                    }
var str = showValues.join(",");
                $("#ShowValuesInput").attr("value", str);
C1.Web.UI.Utils.CallbackHelper.updateWebFormsInternals("ShowValuesInput", str);
        }
}
```

The code starts by setting the MAX_CUSTOMERS, which means the maximum number of customers that can be selected at any time. Set the maximum number of customers to a relatively small value to prevent users from loading too much data into the application at once.

The **C1OlapPage1_CustomerFilterOnClientPropertyChanged** function is raised when the filter's changed which is used to check the ShowValues of the specified field.

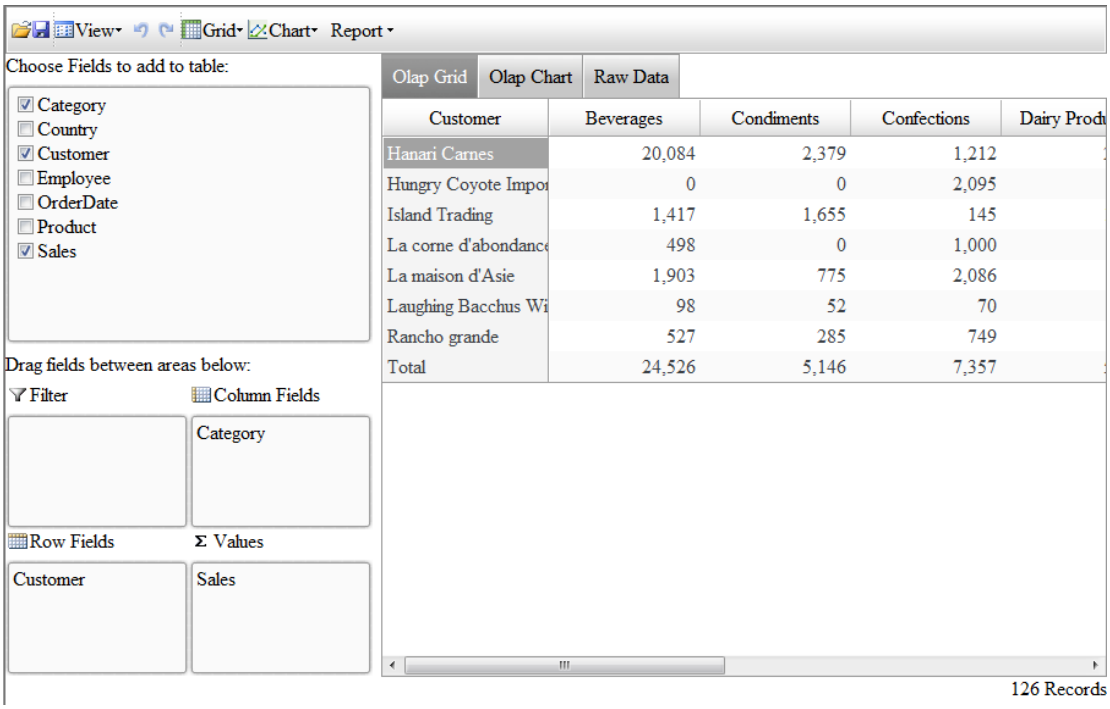
The **C1OlapPage1_OlapPanelUpdating** function is raised before updating the settings to the server which is used to prepare the activeCustomerList for server side and store it to the hidden input.

Before running the application, there is one last item to consider. The field's **Filter** property is only taken into account by the C1OlapEngine if the field in "active" in the view. "Active" means the field is a member of the **RowFields**, **ColumnFields**, **ValueFields**, or **FilterFields** collections. In this case, the "Customers" field has a special filter and should always be active. To ensure this, we must handle the engine's **Updating** event and make sure the "Customers" field is always active.

Here is the code that ensures the "Customers" field is always active:

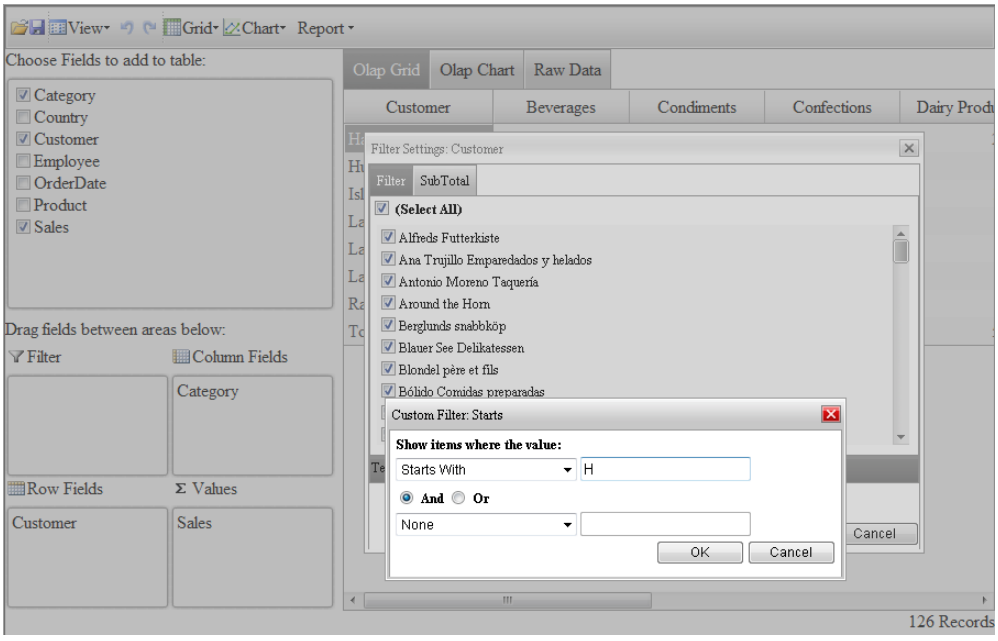
```
protected void C1OlapPage1_Updating(object sender, EventArgs e)
{
      var field = C1OlapPage1.OlapPanel.Fields["Customer"];
      if (!field.IsActive)
      {
            C1OlapPage1.OlapPanel.FilterFields.Add("Customer");
      }
}
```

If you run the application now, you will notice that only the customers included in the "Customers" setting are included in the view:

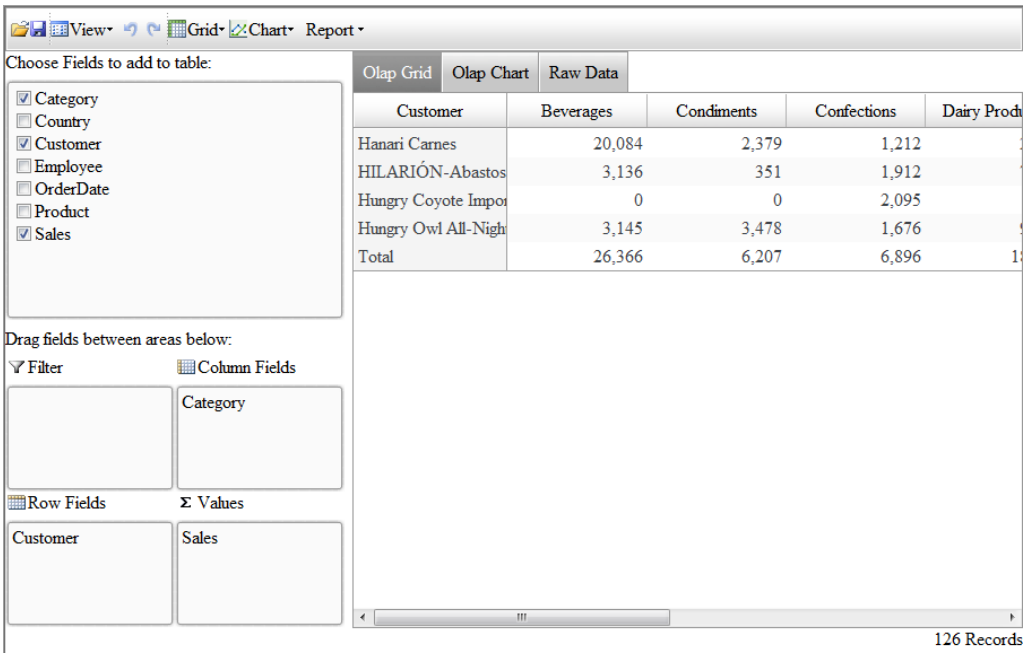


This looks like the screens shown before. The difference is that this time the filtering is done on the server. Data for most customers has not even been loaded into the application.

To see other customers, right-click the "Customer" field and select "Field Settings",uncheck the (Select All) checkbox and then edit the filter by selecting specific customers or by defining a condition as shown below:



When you click OK, the application will detect the change and will request the additional data from the **CachingDataTable** object. Once the new data has been loaded, **OLAP for ASP.NET AJAX** will detect the change and update the OLAP table automatically:

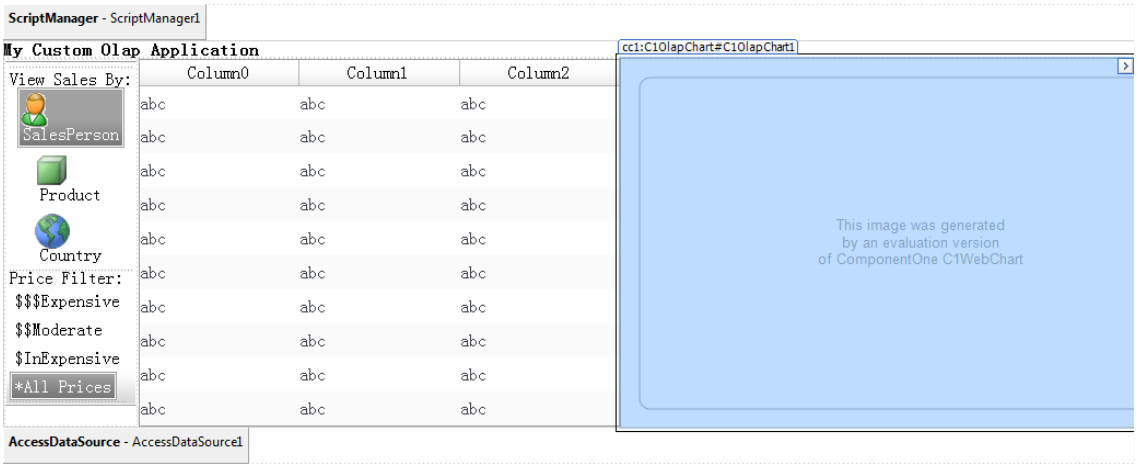


Building a custom User Interface

The examples in previous sections all used the **C1OlapPage** control, which contains a complete UI and requires little or no code. In this section, we will walk through the creation of an OLAP application that does not use the **C1OlapPage**. Instead, it creates a complete custom UI using the **C1OlapGrid**, **C1OlapChart**, and somestandard .NET controls and C1 webcontrols.

The complete source code for this Web site is included in the "CustomUI" sample installed with **OLAP for ASP.NET AJAX**.

The image below shows the application in design view:



There is a **ScriptManager** on the top of the page. and a label docked to the top of the page showing the application title. There is a vertical c1toolbar control at the left of the form with two groups of buttons. The top group allows users to pick one of three pre-defined views: sales by salesperson, by product, or by country. The next group allows users to apply a filter to the data based on product price (expensive, moderate, or inexpensive).

The remaining area of the form if filled with a **C1OlapGrid** on the left and a **C1OlapChart** on the right. These are the controls that will display the view currently selected.

Finally, there is a **C1OlapPanel** control on the form. Its **DisplayVisible** property is set to false, so users won't ever see it. This invisible control is used as a data source for the grid and the chart, and is responsible for filtering and summarizing the data. Both the grid and the chart have their **DataSource** property set to the **C1OlapPanel** And their **DataSource** property should be assigned as the code below:

```
The html tag:
<cc1:C1OlapGrid Width="440px" Height="335px" ID="C1OlapGrid1" runat="server"
onpreparedatasource="C1OlapGrid1_PrepareDataSource" />

<cc1:C1OlapChart ID="C1OlapChart1" runat="server" Height="335px" Width="550px"
onpreparedatasource="C1OlapChart1_PrepareDataSource"></cc1:C1OlapChart>

The code behind:

protected void C1OlapGrid1_PrepareDataSource(object sender, EventArgs e)
{
      this.C1OlapGrid1.DataSource = this.C1OlapPanel1;
}
protected void C1OlapChart1_PrepareDataSource(object sender, EventArgs e)
{
      this.C1OlapChart1.DataSource = this.C1OlapPanel1;
}
```

Once all the controls are in place, let's add the code that connects them all and makes the page work.

First, let's get the data and assign it to the **C1OlapPanel**:

Dragging an AccessDataSource onto the page, and choose the "C1NWind.mdb" file as the datasource file, choose the Invoices as the data table and apply. Set the ID of AccessDataSource to the **C1OlapPanel**.

When first loads the page, the **C1OlapPanel** control uses an **InitFields** helper method below :

```
private void InitFields()
{
      C1OlapPanel1.ColumnFields.Clear();
      C1OlapPanel1.ValueFields.Clear();
      C1OlapPanel1.ColumnFields.Add("OrderDate");
      C1OlapPanel1.Fields["OrderDate"].Format = "yyyy";
      C1OlapPanel1.ValueFields.Add("ExtendedPrice");
}
```

The code sets the format of the "OrderDate" field to "yyyy" so sales are grouped by year and rebuilds view by clearing the engine's **RowFields**, **ColumnFields**, and **ValueFields** collections, then adding the "ExtendedPrice" field to **ValueFields**.

When all are done, the code calls **BindData("Salesperson")** so the **C1OlapPanel** will update the output table.The **BindData** method clears and adds the specified field to the RowFields. The "rowFieldName" parameter passed by the caller contains the name of the only field that changes between views in this example.

```
private void BindData(string rowFieldName)
      {
            C1OlapPanel1.RowFields.Clear();
```

```
            C1OlapPanel1.RowFields.Add(rowFieldName);
            C1OlapPanel1.Update();
      }
```

The event handlers for the **C1ToolBar** that select the current view look like this:

```
protected void C1ToolBar1_ItemClick(object sender, C1ToolBarEventArgs e)
{
      if (e.CommandName.StartsWith("ViewBy"))
      {
            BindData(e.CommandName.Replace("ViewBy", ""));
      }
      else if (e.CommandName.EndsWith("Price"))
      {
            if (e.CommandName.Equals("ExpensivePrice"))
            {
                  SetFilter(50, int.MaxValue);
            }
            else if (e.CommandName.Equals("ModeratePrice"))
            {
                  SetFilter(20, 50);
            }
            else if (e.CommandName.Equals("InExpensivePrice"))
            {
                  SetFilter(0, 20);
            }
            else if (e.CommandName.Equals("AllPrice"))
            {
                  C1OlapPanel1.FilterFields.Clear();
                  C1OlapPanel1.Update();
            }
      }
}
```

The **BindData** helper method will be invoked when the first group's item of the **C1ToolBar** is clicked and **SetFilter** helper method will be invoked when the second group's item is clicked.

```
private void SetFilter(double min, double max)
{
      C1OlapPanel1.FilterFields.Clear();
      C1OlapPanel1.FilterFields.Add("UnitPrice");
      C1WebOlapFilter olapFilter = C1OlapPanel1.Fields["UnitPrice"].Filter;

      olapFilter.Condition1.Operator = ConditionOperator.GreaterThanOrEqualTo;
      olapFilter.Condition1.Parameter = min;
      olapFilter.Condition2.Operator = ConditionOperator.LessThanOrEqualTo;
      olapFilter.Condition2.Parameter = max;
      olapFilter.AndConditions = true;

      C1OlapPanel1.Update();
}
```

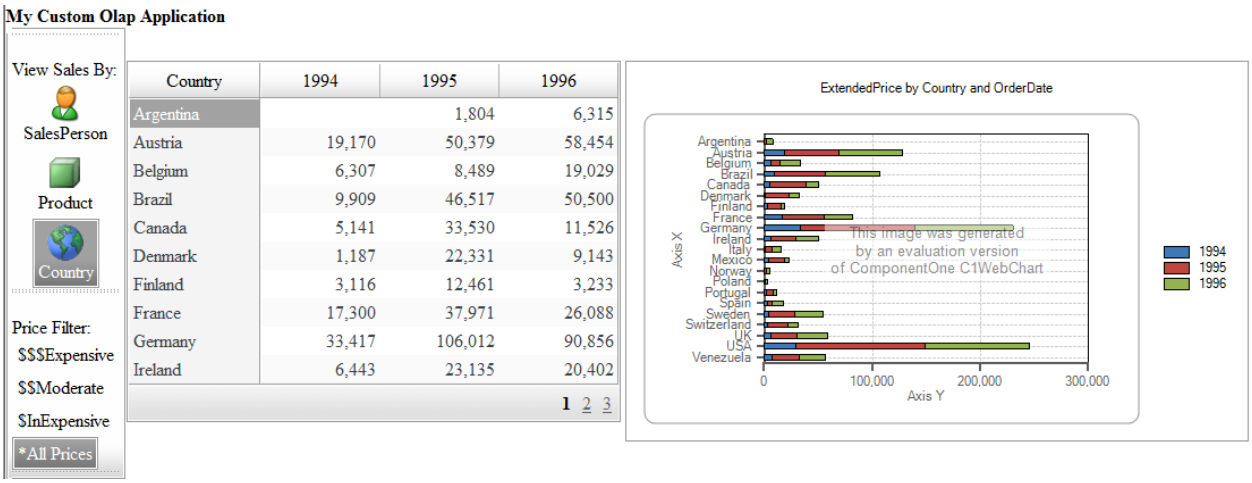
It gets a reference to the "UnitPrice" field that will be used for filtering the data. The "UnitPrice" field is added to the C1OlapPanel's **FilterFields** collection so the filter will be applied to the current view.

This is an important detail. If a field is not included in any of the view collections (**RowFields**, **ColumnFields**, **ValueFields**, **FilterFields**), then it is not included in the view at all, and its **Filter** property does not affect the view in any way.

The code proceeds to configure the **Filter** property of the "UnitPrice" field by setting two conditions that specify the range of values that should be included in the view. The range is defined by the "min" and "max" parameters. Instead of using conditions, you could provide a list of values that should be included. Conditions are usually more convenient when dealing with numeric values, and lists are better for string values and enumerations.

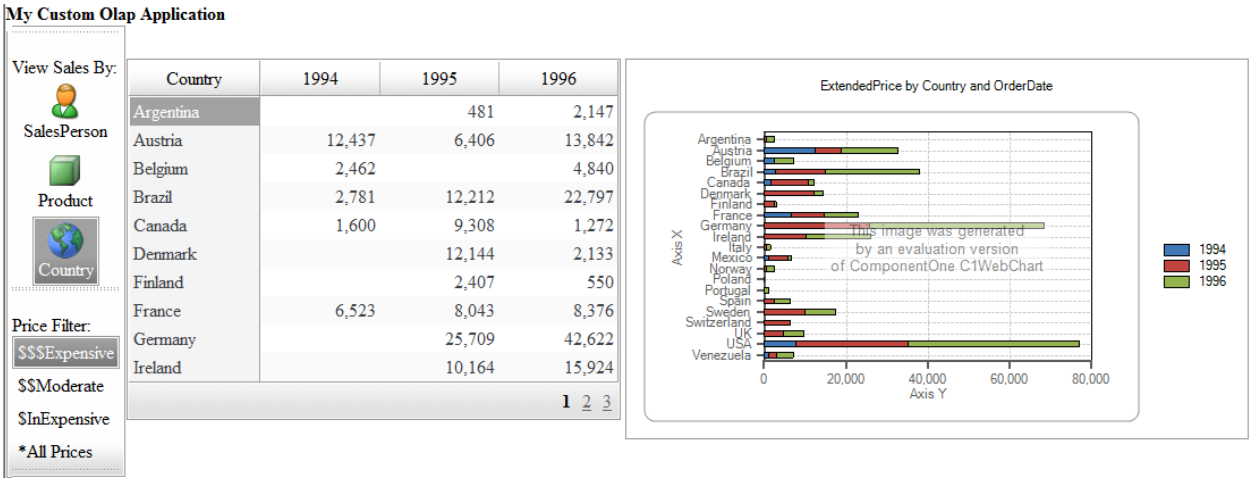
Finally, the code calls **Update**.

The application is almost ready. You can run it now and test the different views and filtering capabilities of the application, as illustrated below:



View showing sales for all products, grouped by year and country. Notice how the chart shows values approaching $300,000.

If you click the "$$$ Expensive" button, the filter is applied and the view changes immediately. Notice how now the chart shows values approaching $80,000 instead. Expensive values are responsible for about one third of the sales:



Configuring Fields in Code

One of the main strengths in Olap applications is interactivity. Users must be able to create and modify views easily and quickly see the results. **OLAP for ASP.NET AJAX** enables this with its Excel-like user interface and user friendly, simple dialogs.

But in some cases you may want to configure views using code. **C1WebOlap** enables this with its simple yet powerful object model, especially the **Field** and **Filter** classes.

The example that follows shows how you can create and configure views with **C1WebOlap**.

Start by creating a new **Web site** adding a **C1OlapPage** control to the page.

Dragging an AccessDataSource onto the page, and choose the "C1NWind.mdb" file as the datasource file, choose the Invoices as the data table and apply. Set the ID of AccessDataSource to the **C1OlapPanel**.

Switch to code view and add the following code to load some data and assign it to the **C1OlapPage** control:

```
protected void Page_Load(object sender, EventArgs e)
{
      if (!this.Page.IsPostBack && !this.Page.IsCallback)
      {
            this.C1OlapPage1.OlapPanel.ValueFields.Add("ExtendedPrice");
            this.C1OlapPage1.OlapPanel.RowFields.Add("ProductName");
            this.C1OlapPage1.OlapPanel.RowFields.Add("OrderDate");
            this.C1OlapPage1.OlapPanel.Update();
}
}
```

The code builds an initial view that shows the sum of the "ExtendedPrice" values by product and by order date. This is similar to the examples given above.

If you run the sample now, you will see an Olap view including all the products and dates.

Next, let's use the **C1WebOlap** object model to change the format used to display the order dates and extended prices:

```
      var field = C1OlapPage1.OlapPanel.Fields["OrderDate"];
      field.Format = "yyyy";

      field = C1OlapPage1.OlapPanel.Fields["ExtendedPrice"];
      field.Format = "c";
      field.Subtotal = C1.Olap.Subtotal.Average;
```

The code retrieves the individual fields from the **Fields** collection which contains all the fields specified in the data source. Then it assigns the desired values to the **Format** and **Subtotal** properties. **Format** takes a regular .NET format string, and **Subtotal** determines how values are aggregated for display in the Olap view. By default, values are added, but many other aggregate statistics are available including average, maximum, minimum, standard deviation, and variance.

Now suppose you are interested only in a subset of the data, say a few products and one year. A user would right-click the fields and apply filters to them. You can do the exact same thing in code as shown below:

```
var filter = C1OlapPage1.OlapPanel.Fields["ProductName"].Filter;
filter.Clear();
filter.ShowValues = "Chai,Chang,Geitost,Ikura".Split(',');

// apply condition filter to show only some dates
```

```
filter = C1OlapPage1.OlapPanel.Fields["OrderDate"].Filter;
filter.Clear();
filter.Condition1.Operator = C1.Olap.ConditionOperator.GreaterThanOrEqualTo;
filter.Condition1.Parameter = new DateTime(1996, 1, 1);
filter.Condition2.Operator = C1.Olap.ConditionOperator.LessThanOrEqualTo;
filter.Condition2.Parameter = new DateTime(1996, 12, 31);
filter.AndConditions = true;
```

The code starts by retrieving the **C1WebOlapFilter** object that is associated with the "ProductName" field. Then it clears the filter and sets its **ShowValues** property. This property takes an array of values that should be shown by the filter. In **C1Olap** we call this a "value filter".

Next, the code retrieves the filter associated with the "OrderDate" field. This time, we want to show values for a specific year. But we don't want to enumerate all days in the target year. Instead, we use a "condition filter" which is defined by two conditions.

The first condition specifies that the "OrderDate" should be greater than or equal to January 1st, 1996. The second condition specifies that the "OrderDate" should be less than or equal to December 31st, 1996. The **AndConditions** property specifies how the first and second conditions should be applied (AND or OR operators). In this case, we want dates where both conditions are true, so **AndConditions** is set to true.

If you run the project again, you should see the following: