ComponentOne

Data for Silverlight

GrapeCity US

GrapeCity
201 South Highland Avenue, Suite 301
Pittsburgh, PA 15206
Tel: 1.800.858.2739 | 412.681.4343
Fax: 412.681.4384
Website: https://www.grapecity.com/en/
E-mail: us.sales@grapecity.com

Trademarks

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

Warranty

ComponentOne warrants that the media on which the software is delivered is 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 media 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 the defective media by sending it and a check for $25 (to cover postage and handling) to ComponentOne.

Except for the express warranty of the original media on which the software is delivered is 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. ComponentOne is 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.

Table of Contents

- Data for Silverlight Overview 2
    - Help with Silverlight Edition 2
- Key Features 3
- Introduction to C1.Silverlight.Data 4
- Data-Centric Architecture 5
- The Sample Application 6
    - Create the Application 6
    - Add the References 6-7
    - Create the UI 7-8
    - Implement the Server Side 8
        - Add the Database Access Infrastructure 8-9
        - Implement the Web Service 9-13
    - Implement the Client Side 13
        - Add a Reference to the Web Service 13-14
        - Use the Web Service to Get the Data 14-16
        - Bind the Data to Controls on the Page 16-18
        - Synchronize the Views 18-19
        - Add and Remove Items 19-20
        - Commit Changes to the Server 20-22
    - Optimize the Data Transfers 22
        - Compress the Data on the Server 22
        - Decompress the Data on the Client 22-23
- Using Data with Schedule for Silverlight 24
- Conclusion 25

Data for Silverlight Overview

Data for Silverlight is an implementation of the standard **DataSet**, **DataTable**, and **DataView** classes familiar to Windows Forms and ASP.NET developers. The following topics describe how you can use the **C1.Silverlight.Data** assembly to implement data-centric Silverlight applications.

Help with Silverlight Edition

Getting Started

For information on installing **ComponentOne Studio Silverlight Edition**, licensing, technical support, namespaces and creating a project with the control, please visit Getting Started with Silverlight Edition.

Key Features

Data for Silverlight allows you to create customized, rich applications. Make the most of **Data for Silverlight** by taking advantage of the following key features:

- **Supports All Bindable Controls**
  The Data assembly supports WPF/Silverlight data-binding. Data supports all bindable controls (including the Microsoft DataGrid) and Language Integrated Queries (Linq). Use familiar classes like DataSet, DataTable, and DataView to manipulate your data and communicate with the server, while taking full advantage of Linq and the latest generation of Silverlight controls.
- **Reusable Business Logic**
  Data allows you to work with data in Silverlight applications in the same manner and with the same code as in Windows Forms. You can create a data library encapsulating your data and business logic and use it both in Silverlight applications and Windows Forms.
- **Serialize to XML**
  Empower occasionally connected scenarios with XMLserialization. Save the data in the IsolatedStorage to be consumed later in your Silverlight application or communicate via XML with a Web Service in the server.

Introduction to C1.Silverlight.Data

The classes in the **C1.Silverlight.Data** assembly are a subset of those in the **System.Data** namespace. This means that you can use your existing ADO.NET-based code in Silverlight applications. For example, the code below shows how you can create and populate a DataTable object using **C1.Silverlight.Data**:

C#

```
//Create DataTable
DataTable dt = new DataTable();

// Add some columns
dt.Columns.Add("ID", typeof(int));
dt.Columns.Add("First Name", typeof(string));
dt.Columns.Add("Last Name", typeof(string));
dt.Columns.Add("Active", typeof(bool));
// Set primary key column
DataColumn dc = dt.Columns["ID"];
dc.AutoIncrement = true;
dc.ReadOnly = true;
dt.PrimaryKey = dc; // single column, not array

// Add some rows
for (int i = 0; i < 10; i++)
{
    dt.Rows.Add(i,
    "First " + i.ToString(),
    "Last " + i.ToString(),
    i % 2 == 0);
}
```

If you've used ADO.NET, the code should look very familiar. The only difference is that the PrimaryKey property on ADO.NET **DataTable** objects is a vector of columns, while in **C1.Silverlight.Data** it is a single column instead. Once you have built the DataTable, you can bind it to other controls using the table's DefaultView property:

C#

```
//Bind table to a control
DataGrid dg = new DataGrid();
dg.ItemsSource = dt.DefaultView;
LayoutRoot.Children.Add(dg);
```

Note: You must import the **C1.Silverlight.Data.dll** (contains the **C1Data** classes) and **System.Windows.Controls.Data.dll** (contains Microsoft's **DataGrid** control) namespaces for the above code to work

The **DefaultView** property returns a DataView object associated with the table. Again, this is exactly the same mechanism used in ADO.NET. The DataView object implements an **IEnumerable** interface that allows it to be used as a data source for bindable controls and for LINQ queries.

Because the classes in the **C1.Silverlight.Data** assembly are a subset of those in the **System.Data** namespace, we will not describe them in detail here. If you are not familiar with these classes, please check the descriptions of the System.Data.DataSet and System.Data.DataTable classes on MSDN.

Data-Centric Architecture

Silverlight can be used to build line-of-business and other data-centric applications. This type of application typically involves the following steps:

1. Get the data from the server.
2. Display and edit the data on the client.
3. Submit the changes back to the server.

Steps 1 and 3 typically rely on Web services to transfer the data and traditional data access strategies to query and update a database. Step 2 typically involves Silverlight data-bound controls.

Microsoft offers many tools that can be used to perform the server-side part of the job. The latest such tool is ADO.NET Data Services, which provides Web-accessible endpoints for data models and integrates with Silverlight through the ADO.NET Data Services for Silverlight library (System.Data.Services.Client.dll). A lot has been written lately about this new technology (see for example "Data Services" in MSDN vol. 23, no. 10, September 2008).

ADO.NET Data Services is a powerful new technology that is likely to become a standard for many types of data-centric applications. However, it is not the only option. The traditional data ADO.NET classes (**DataSet**, **DataTable**, **DataAdapters**, and so on) can also be used to retrieve and update data on the server. These classes have been used by developers since .NET 1.0. They are solid, powerful, and easy to use. Furthermore, many developers already have a considerable investment in the form of code that has been used and tested for a long time.

Data for Silverlight is a set of classes that can be used by Silverlight clients to exchange data with servers using traditional ADO.NET. The typical steps are as follows:

1. **Get the data from the server.**
   - The server populates a DataSet object the traditional way (typically using **DataAdapter** objects to retrieve the data from a SqlServer database).
   - The server calls the DataSet.WriteXml method to serialize the **DataSet** into a stream and sends the stream to the client.
   - The client uses the DataSet.ReadXml method to de-serialize the stream.
2. **Display and edit the data on the client.**
   - The client binds the tables in the **DataSet** to controls (possibly using LINQ queries).
   - The user interacts with the controls to view, edit, add, and delete data items.
3. **Submit the changes back to the server.**
   - The client calls DataSet.GetChanges and **DataSet.WriteXml** to serialize the changes into a stream and sends the stream to the server.
   - The server calls **DataSet.ReadXml** to de-serialize the changes, then uses **DataAdapter** objects to persist the changes into the database.

The role of **C1Data** in this scenario is twofold. First, it provides a symmetric serialization mechanism that makes it easy to transfer data between **DataSet** objects on the client and on the server. Second, it provides a familiar object model to manipulate the data on the client.

Notes:

C1Data does not compete against ADO.NET Data Services. It allows you to leverage your knowledge of ADO.NET and assets you may already have. You can transfer all that to Silverlight with minimal effort, and migrate to ADO.NET Data Services gradually, if such a migration is desired.

C1Data is not a legacy technology. You can use it with LINQ, for example. In fact, **C1Data** enables LINQ features that are available on desktop but not in Silverlight applications (partial anonymous classes).

The next sections describe the implementation of a simple application that performs the steps described above. Despite its simplicity, the application shows how to perform the four CRUD operations (Create, Read, Update, Delete) that are required of most data-centric applications.

The Sample Application

The sample application consists of two grids, arranged to show a master-detail view of the Northwind *Categories* and *Products* tables. The data is retrieved from the server when the application starts, using a Web service. It then populates the tables so the user can browse and edit the data. Finally, it submits any changes back to the server so that the database can be updated.

Create the Application

Let's start by creating a new Silverlight application called "MasterDetail." Complete the following steps:

1. Select **File**, **New**, then **Project** to open the **New Project** dialog box in Visual Studio 2008.
2. In the **Project types** pane, expand either the **Visual Basic** or **Visual C#** node and select **Silverlight**.
3. Choose **Silverlight Application** in the **Templates** pane.
4. Name the project "MasterDetail," specify a location for the project, and click **OK**.
   Next, Visual Studio will prompt you for the type of hosting you want to use for the new project.
5. In the **New Silverlight Application** dialog box, select **OK** to accept the default name ("MasterDetailWeb") and settings, and create the project.

Add the References

Our project uses two additional assemblies:

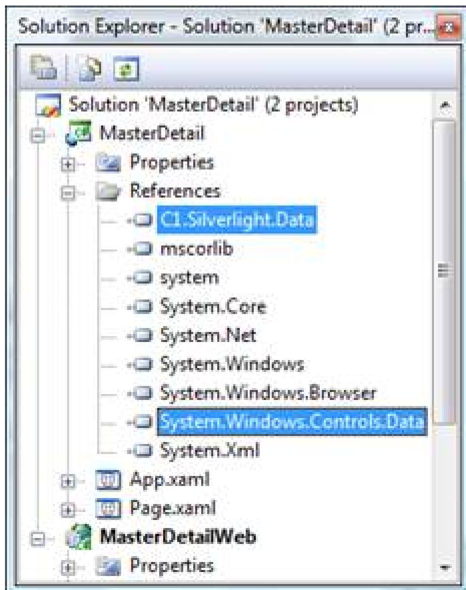
- **C1.Silverlight.Data.dll** contains the **C1Data** classes
- **System.Windows.Controls.Data.dll** contains Microsoft's **DataGrid** control

We use the Microsoft grid instead of ComponentOne's to show that the **C1Data** classes are not tied in any way to other ComponentOne controls; you can use them against any Silverlight control.

To add the assemblies, complete the following steps:

1. Right-click the **MasterDetail** project in the Solution Explorer and select **Add Reference**.
2. In the **Add Reference** dialog box locate and select the following assemblies and click **OK** to add references to your project:
   - C1.Silverlight.Data.dll
   - System.Windows.Controls.Data.dll

After adding references, the solution appears like this:



Create the UI

The user interface consists of two data grids (Categories and Products), and a few buttons used to add and delete items and to commit the changes.

To create the user interface, open the **MainPage.xaml** file and in Source view, copy the following XAML onto the page:

XAML

```
<UserControl x:Class="MasterDetail.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:swc=
"clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Data">
  <Grid x:Name="LayoutRoot" Background="White" >

    <!-- Main grid layout -->
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
      <RowDefinition Height="*" />
      <RowDefinition Height="Auto" />
      <RowDefinition Height="2*" />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <!-- Page title -->
    <TextBlock Text="Master-Detail Sample in Silverlight" Margin="5"
      FontSize="20" FontWeight="Bold" />

    <!-- Master data grid -->
    <TextBlock Text="Categories" Margin="5" Grid.Row="1"
      FontSize="16" FontWeight="Bold" Foreground="Blue" />
    <swc:DataGrid x:Name="_gridCategories" Margin="5" Grid.Row="2" />
```

```
    <!-- Detail data grid -->
    <TextBlock Text="Products" Margin="5" Grid.Row="3"
      FontSize="16" FontWeight="Bold" Foreground="Blue" />
    <swc:DataGrid x:Name="_gridProducts" Margin="5" Grid.Row="4" />

    <!-- Control panel -->
    <StackPanel Margin="5" Orientation="Horizontal" Grid.Row="5" >
      <Button x:Name="_btnAdd" Content=" Add Item "
        Margin="0,0,20,0" FontSize="14" VerticalAlignment="Center" />
      <Button x:Name="_btnRemove" Content=" Remove Item "
        Margin="0,0,20,0" FontSize="14" VerticalAlignment="Center" />
      <Button x:Name="_btnCommit" Content=" Commit Changes "
        Margin="0,0,20,0" FontSize="14" VerticalAlignment="Center" />
      <TextBlock x:Name="_tbStatus" Text="Ready"
        VerticalAlignment="Center" FontSize="12" Foreground="Gray" />
    </StackPanel>
  </Grid>
</UserControl>
```

Try running the application now to make sure everything is OK so far. It consists of two empty grids:

Master-Detail Sample in Silverlight

Categories

Products

Add Item   Remove Item   Commit Changes   Ready

Implement the Server Side

In this step, we implement the server side of the application. It consists of a Web service with two methods:

- **GetData** gets the data from the Northwind database and returns it to the client.
- **UpdateData** saves the changes made by the client back into the database.

Add the Database Access Infrastructure

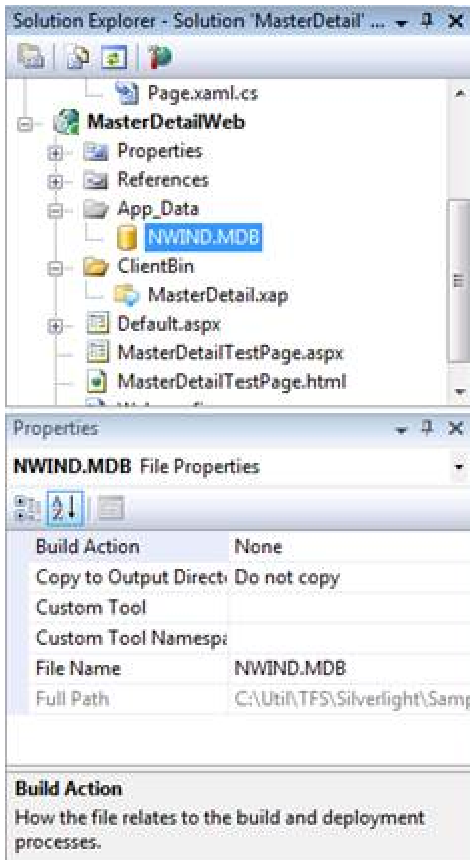
Before implementing the Web services, add the database itself to the project. This step is optional. We use a connection string to specify where the database is located, so we could use any existing copy of the MDB file already in the system. We will create a local copy here to facilitate deployment and to avoid making changes to the original

database.

To add the database to the project, complete the following steps:

1. Right-click the **App_Data** node in the **MasterDetailWeb** project and select **Add | Existing Item**.
2. In the dialog box, locate the **NWIND.MDB** file and add it to the project.
3. In the Properties window, set the **Build Action** property to **None**.

The image below shows the Solution Explorer window after this step is completed:



In addition to the database file, we need a mechanism to transfer data from and to the database. In this sample, we accomplish this using a utility class called **SmartDataSet**. **SmartDataSet** extends the regular ADO.NET **DataSet** class and adds the following:

- A **ConnectionString** property that specifies the database type and location.
- A **Load** method that loads data from selected tables.
- An **Update** method that saves changes back into the database.

The **SmartDataSet** is convenient but not necessary. It knows how to create and configure **DataAdapter** objects used to load and save data, but you could also write standard ADO.NET code to accomplish the same thing. Because it only uses standard ADO.NET techniques, we do not list it here.

To add the **SmartDataSet.cs** file to the project, complete the following:

1. Right-click the **MasterDetailWeb** node in the Solution Explorer, select **Add | Existing Item**.
2. In the **Add Existing Item** dialog box, locate the **SmartDataSet.cs** file in the **C1.Silverlight** distribution package and click **Add** to add it to the project.

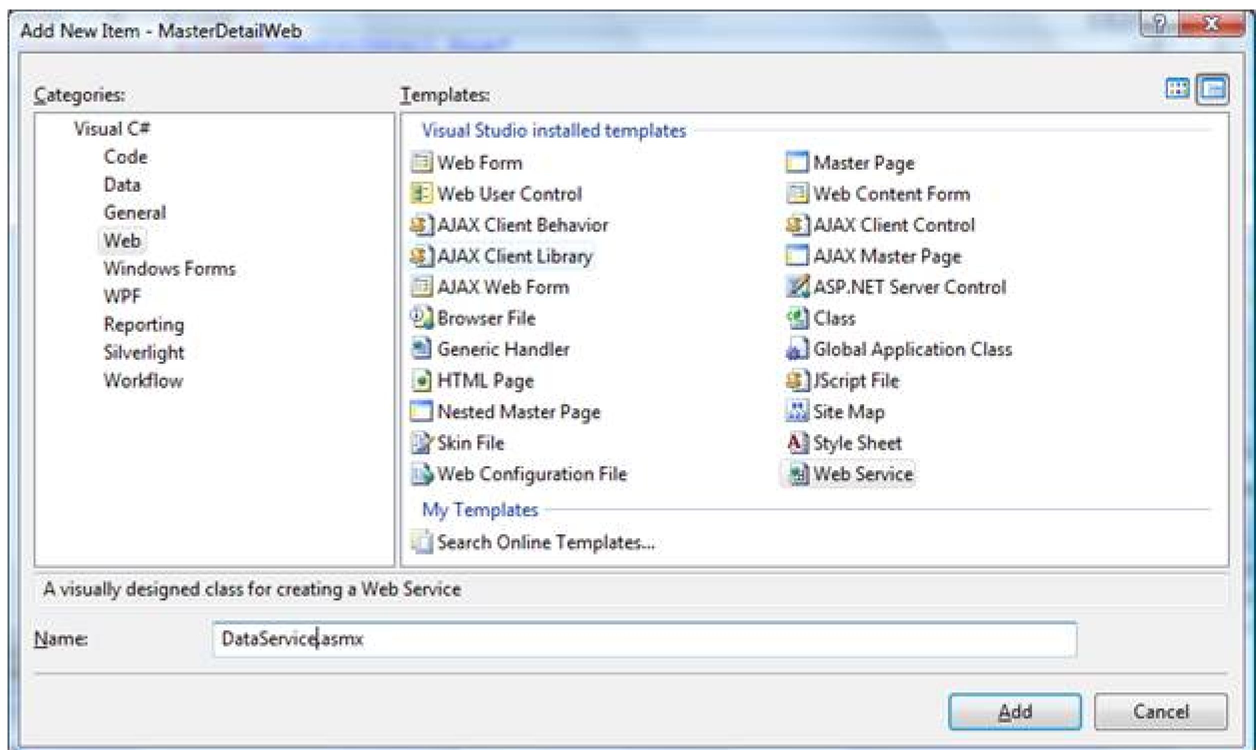
Implement the Web Service

The server side of the application consists of Web services that load the data from the database and returns it to the server, or receive a list of changes from the client and applies it to the database.

We could implement the server side as a generic handler class (ashx), as a Web Service (asmx), or as a Silverlight-enabled WCF service (SVC). For this sample, any of the choices would do. We chose a classic Web Service.

Follow these steps to create the service:

1. Right-click the **MasterDetailWeb** project.
2. Select **Add | New Item**.
3. In the **Add New Item** dialog box, select **Web** from the list of categories in the left pane.
4. In the right pane, select the **Web Service** template from the list of templates.
5. Name the new service "DataService.asmx."
   The dialog box looks like the image below.



6. Click the **Add** button to create the new Web Service.

After you click the **Add** button, Visual Studio opens the newly created **DataService.asmx.cs** (or **DataService.asmx.vb**) file. The file contains a single **HelloWorld** method.

Edit the file as follows:

1. Add the following import statements to the statements block at the start of the file:

   C#

   ```
   using System.IO;
   using System.Data;
   ```

2. Uncomment the line below so the service can be called from Silverlight:

   C#

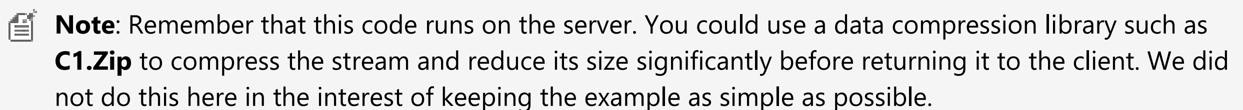
   ```
   [System.Web.Script.Services.ScriptService]
   ```

3. Delete the **HelloWorld** method that Visual Studio created and replace it with the following code:

   C#

```
[WebMethod]
public byte[] GetData(string tables)
{
  // Create DataSet with connection string
  var ds = GetDataSet();

  // Load data into DataSet
  ds.Fill(tables.Split(','));

  // Persist to stream
  var ms = new System.IO.MemoryStream();
  ds.WriteXml(ms, XmlWriteMode.WriteSchema);

  // Return stream data
  return ms.ToArray();
}
```

The method starts by creating a **SmartDataSet**, then fills it with the tables specified by the **tables** parameter. Finally, it uses the WriteXml method to persist the DataSet into a stream, converts the stream into a byte array, and returns the result.



Note: Remember that this code runs on the server. You could use a data compression library such as **C1.Zip** to compress the stream and reduce its size significantly before returning it to the client. We did not do this here in the interest of keeping the example as simple as possible.

4. Add the method that saves the changes back into the database with the following code:

C#

```
[WebMethod]
public string UpdateData(byte[] dtAdded, byte[] dtModified, byte[] dtDeleted)
{
  try
  {
    UpdateData(dtAdded, DataRowState.Added);
    UpdateData(dtModified, DataRowState.Modified);
    UpdateData(dtDeleted, DataRowState.Deleted);
    return null;
  }
  catch (Exception x)
  {
    return x.Message;
  }
}
```

The method takes three parameters, each corresponding to a different type of change to apply to the database: records added, modified, and deleted. It calls the **UpdateData** helper method to apply each set of changes, and returns null if all changes apply successfully. If there are any errors, the method returns a message that describes the exception.

5. Add the following code for the **UpdateData** helper method:

C#

```
void UpdateData(byte[] data, DataRowState state)
{
  // No changes, no work
  if (data == null)
```

```
    return;

  // Load data into dataset
  var ds = GetDataSet();
  var ms = new MemoryStream(data);
  ds.ReadXml(ms);
  ds.AcceptChanges();

  // Update row states with changes
  foreach (DataTable dt in ds.Tables)
  {
    foreach (DataRow dr in dt.Rows)
    {
      switch (state)
      {
        case DataRowState.Added:
          dr.SetAdded();
          break;
        case DataRowState.Modified:
          dr.SetModified();
          break;
        case DataRowState.Deleted:
          dr.Delete();
          break;
      }
    }
  }

  // Update the database
  ds.Update();
}
```

The method starts by creating a **SmartDataSet** and loading all the changes into it. It then changes the **RowState** property on each row to identify the type of change to the row (addition, modification, or deletion). Finally, it calls the **SmartDataSet.Update** method to write the changes to the database.

6. The server-side code is almost ready. The only part still missing is the method that creates and configures the **SmartDataSet**. Add the following code for the implementation:

C#

```
SmartDataSet GetDataSet()
{
  // Get physical location of the mdb file
  string mdb = Path.Combine(
    Context.Request.PhysicalApplicationPath, @"App_Data\nwind.mdb");

  // Check that the file exists
  if (!File.Exists(mdb))
  {
    string msg = string.Format("Cannot find database file {0}.", mdb);
    throw new FileNotFoundException(msg);
  }

  // Make sure file is not read-only (source control often does this...)
```

```
  FileAttributes att = File.GetAttributes(mdb);
  if ((att & FileAttributes.ReadOnly) != 0)
  {
    att &= ~FileAttributes.ReadOnly;
    File.SetAttributes(mdb, att);
  }

  // Create and initialize the SmartDataSet
  var dataSet = new SmartDataSet();
  dataSet.ConnectionString =
    "provider=microsoft.jet.oledb.4.0;data source=" + mdb;
  return dataSet;
}
```

The method starts by locating the database file, making sure it exists, and checking that it is not read-only (or the updates would fail). Once that is done, it creates a new **SmartDataSet**, initializes its **ConnectionString** property, and returns the newly created **SmartDataSet** to the caller.

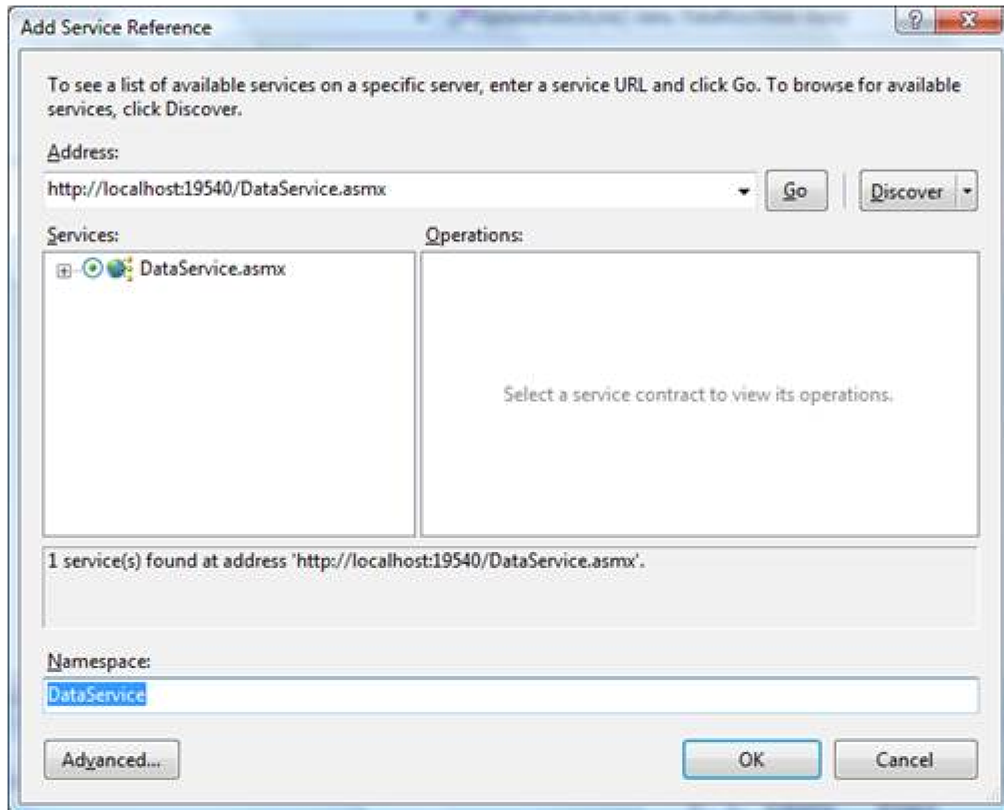
Implement the Client Side

Now let's return to the client side of the application. In the following sections you will implement the client side.

Add a Reference to the Web Service

The first thing to do is add a reference to the Web service we created, so we can invoke it from the client. Complete the following steps:

1. Right-click the **References** node in the **MasterDetail** project and select **Add Service Reference**.
2. In the dialog box that follows, click the **Discover** button. Visual Studio locates the service we added in the previous step.
3. In the **Namespace** field, type "DataService." The dialog box looks like this:



4. Click **OK** to add the reference.

Use the Web Service to Get the Data

We are finally ready to get the data and show it to the users. Complete the following:

1. Open the **MainPage.xaml.cs** file and add the following using statements to the block at the top of the file:

C#

```
using System.IO;
using C1.Silverlight.Data;
using MasterDetail.DataService;
```

2. Next, edit the page constructor as follows:

C#

```
public Page()
{
    InitializeComponent();
    LoadData();
}
```

3. Add the following code to implement the **LoadData** method to invoke the Web service to retrieve the data into a **DataSet** object, which we bind to the controls on the page:

C#

```
DataSet _ds = null;
void LoadData()
{
  // Invoke Web service
  var svc = GetDataService();
```

```
  svc.GetDataCompleted += svc_GetDataCompleted;
  svc.GetDataAsync("Categories,Products");
}
void svc_GetDataCompleted(object sender, GetDataCompletedEventArgs e)
{
  // Handle errors
  if (e.Error != null)
  {
    _tbStatus.Text = "Error downloading data...";
    return;
  }

  // Parse data stream from server (DataSet as XML)
  _tbStatus.Text = string.Format(
    "Got data, {0:n0} kBytes", e.Result.Length / 1024);
  var ms = new MemoryStream(e.Result);
  _ds = new DataSet();
  _ds.ReadXml(ms);

  // Got the data, bind controls to it
  BindData();
}
```

4. Add the following **GetDataService** method implementation:

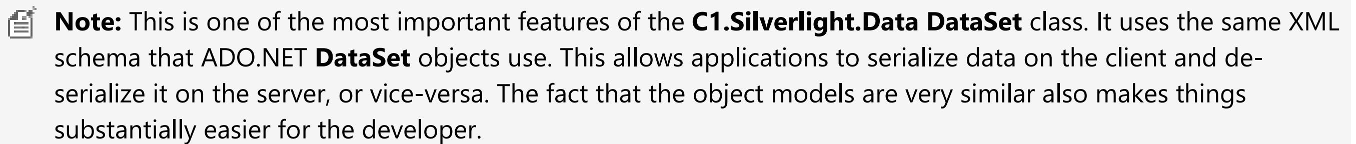
C#

```
// Get data service relative to current host/domain
DataServiceSoapClient GetDataService()
{
  // Increase buffer size
  var binding = new System.ServiceModel.BasicHttpBinding();
  binding.MaxReceivedMessageSize = 2147483647; // int.MaxValue
  binding.MaxBufferSize = 2147483647; // int.MaxValue

  // Get absolute service addres
  Uri uri = GetAbsoluteUri("DataService.asmx");
  var address = new System.ServiceModel.EndpointAddress(uri);

  // Return new service client
  return new DataServiceSoapClient(binding, address);
}
public static Uri GetAbsoluteUri(string relativeUri)
{
  Uri uri = System.Windows.Browser.HtmlPage.Document.DocumentUri;
  string uriString = uri.AbsoluteUri;
  int ls = uriString.LastIndexOf('/');
  return new Uri(uriString.Substring(0, ls + 1) + relativeUri);
}
```

The **GetDataService** method instantiates and returns a new **DataServiceSoapClient** object. We don't use the default constructor because that would refer to the development environment (http://localhost and so on). It would work correctly on the development machine, but would break when the application is deployed. Also, the default constructor uses a 65k buffer that might be too small for our data transfers. The above **GetDataService** method

implementation takes care of both issues.

The **LoadData** method above instantiates the service and invokes the **GetDataAsync** method. When the method finishes executing, it invokes the **svc_DataCompleted** delegate. The delegate instantiates a DataSet object, uses the ReadXml method to de-serialize the data provided by the server, then calls **BindData** to bind the data to the controls on the page.



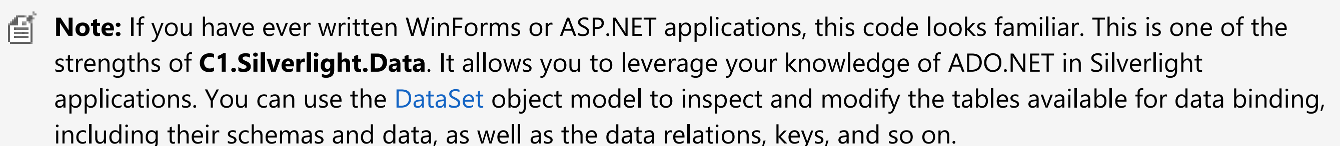
Note: This is one of the most important features of the **C1.Silverlight.Data DataSet** class. It uses the same XML schema that ADO.NET **DataSet** objects use. This allows applications to serialize data on the client and de-serialize it on the server, or vice-versa. The fact that the object models are very similar also makes things substantially easier for the developer.

Bind the Data to Controls on the Page

After the data is retrieved, we bind it to the controls on the page. Here is the implementation of the **BindData** method that handles that task:

C#

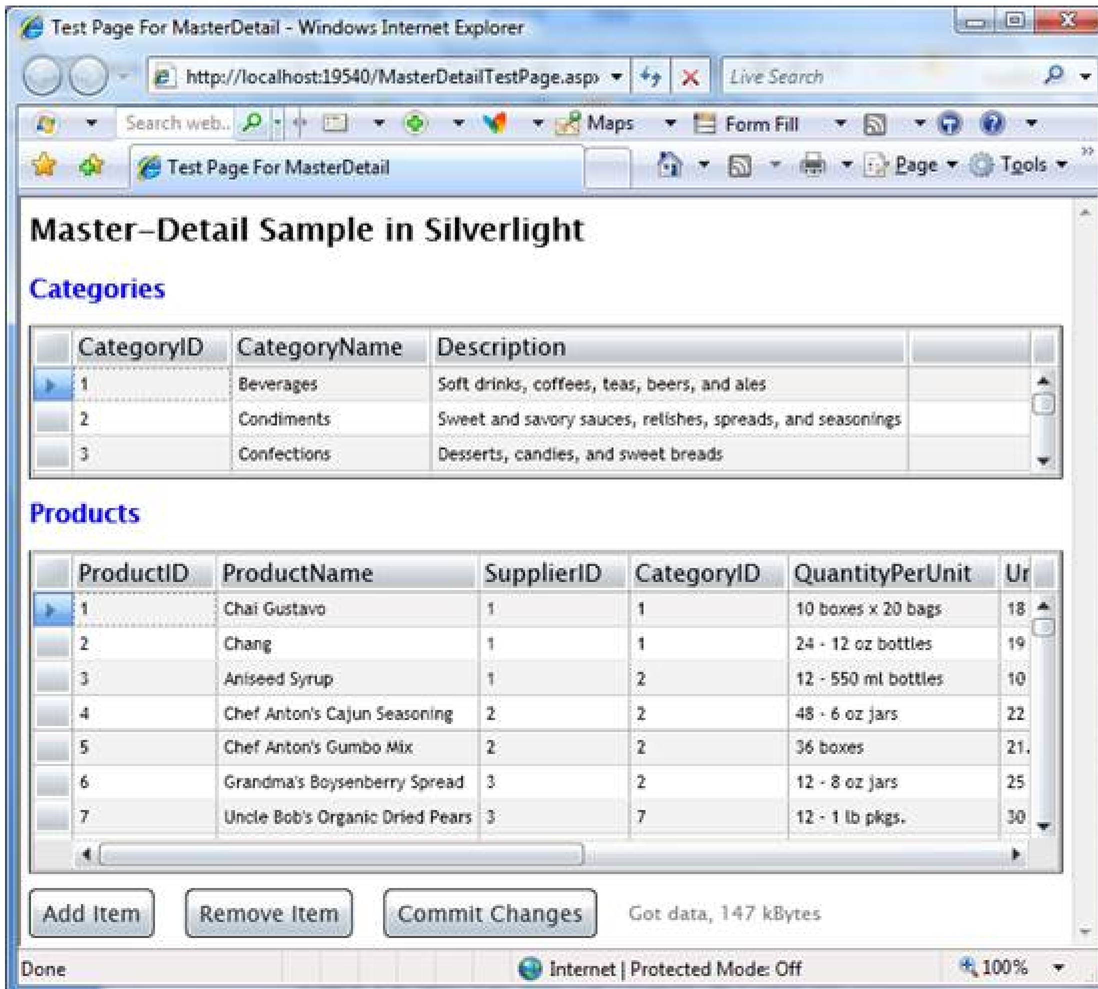
```
void BindData()
{
  // Get the tables we want
  DataTable dtCategories = _ds.Tables["Categories"];
  DataTable dtProducts = _ds.Tables["Products"];

  // Populate categories grid
  _gridCategories.ItemsSource =
      new DataView(dtCategories,
          "CategoryID", "CategoryName", "Description");

  // Populate products categories grid
  _gridProducts.ItemsSource = dtProducts.DefaultView;
}
```



Note: If you have ever written WinForms or ASP.NET applications, this code looks familiar. This is one of the strengths of **C1.Silverlight.Data**. It allows you to leverage your knowledge of ADO.NET in Silverlight applications. You can use the DataSet object model to inspect and modify the tables available for data binding, including their schemas and data, as well as the data relations, keys, and so on.

The only slightly unusual statement is the one that creates a DataView object and specifies which columns to include in the view. This is an extension provided by the C1.Silverlight.Data implementation that is not present in the original ADO.NET **DataView** class.

If you run the project now, you see the data retrieved from the server:



Before we continue with the sample, a few comments are in order.

Although the code that binds the data to the controls looks familiar, what happens under the covers is quite different from the traditional WinForms and ASP.NET scenarios. All data binding in WPF and Silverlight is done through reflection. But the DataRowView objects exposed by **DataView** collections do not have properties that match the columns in the underlying DataTable ("CategoryID", "CategoryName", "Description", and so on).

> **Note:** The binding is possible because the **DataView** class dynamically creates anonymous types that derive from **DataRowView** and expose additional properties that correspond to the table columns. Controls can then use reflection to find these properties and bind to them.

The DataRow class also leverages this mechanism through its GetRowView method. This method builds a wrapper object that exposes selected properties of the **DataRow** and can be used for data binding. You can use this method in your LINQ queries.

For example, the code below builds a LINQ query that selects all categories that start with "C" and exposes their name and description:

C#

```
_gridCategories.ItemsSource =
    from dr
    in dtCategories.Rows
    where ((string)dr["CategoryName"]).StartsWith("C")
    select dr.GetRowView("CategoryName", "Description");
```

If you call **GetRowView** with no parameters, all columns are exposed.

Synchronize the Views

Our master-detail view is not synchronized yet. All products are displayed in the bottom grid, regardless of the category selected on the top grid.

To synchronize the views, we need to handle the **SelectionChanged** event on the categories grid and filter the products to display only those that belong to the selected category.

To do this, we start by adding an event handler to the page constructor:

C#

```
public Page()
{
    InitializeComponent();
    LoadData();
    _gridCategories.SelectionChanged += _gridCategories_SelectionChanged;
}
```

Implement the event handler as follows:

C#

```
void _gridCategories_SelectionChanged(object sender, EventArgs e)
{
  var drv = _gridCategories.SelectedItem as DataRowView;
  if (drv != null)
  {
    var dv = _ds.Tables["Products"].DefaultView;
    dv.RowFilter = string.Format("CategoryID = {0}", drv.GetData("CategoryID"));
  }
}
```

The code gets the selected category as a DataRowView object, gets the default view for the "Products" table (used as a data source for the second grid), and applies the filter. Again, the code looks familiar to WinForms and ASP.NET developers.

If you run the application again, notice that the views are now synchronized.

Note that instead of applying a filter to the DataView, we could have used a LINQ query to update the products table. For example:

C#

```
void _gridCategories_SelectionChanged(object sender, EventArgs e)
{
  var drv = _gridCategories.SelectedItem as DataRowView;
  if (drv != null)
  {
    int id = (int)drv.GetData("CategoryID");
    _gridProducts.ItemsSource =
      from dr
```

```
        in _ds.Tables["Products"].Rows
        where (int)dr["CategoryID"] == id
        select dr.GetRowView();
    }
}
```

This also achieves the goal of synchronizing the master and detail views. The main difference is that the previous method would affect any other controls bound to the same view, while this method affects only the products grid.

Add and Remove Items

Unlike most WinForms grids, the Silverlight **DataGrid** control does not allow users to add or remove items from the grid. We use the buttons already on the page for that. The first thing to do is connect the event handlers on the page constructor:

C#

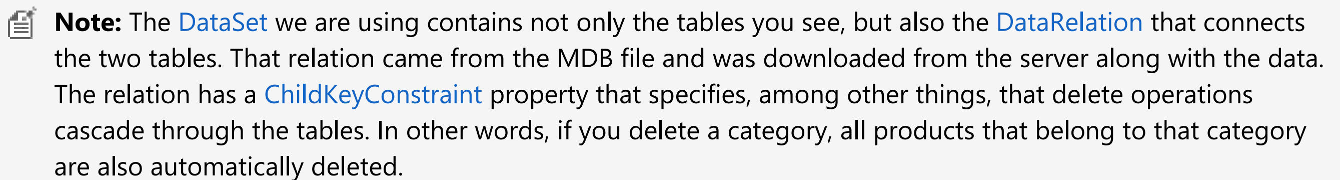
```
// Load data, hook up event handlers
public Page()
{
  InitializeComponent();
  LoadData();

  _gridCategories.SelectionChanged += _gridCategories_SelectionChanged;
  _btnAdd.Click += _btnAdd_Click;
  _btnRemove.Click += _btnRemove_Click;
}
```

The event handlers are simple, they look like regular ADO.NET code you would write on a WinForms application:

C#

```
// Add a new row
private void _btnAdd_Click(object sender, RoutedEventArgs e)
{
  DataTable dt = _ds.Tables["Categories"];
  DataRow newRow = dt.NewRow();
  newRow["CategoryName"] = "New category";
  newRow["Description"] = "This is a new category...";
  dt.Rows.Add(newRow);
}

// Delete a row
private void _btnRemove_Click(object sender, RoutedEventArgs e)
{
  DataRowView drv = _gridCategories.SelectedItem as DataRowView;
  if (drv != null)
  {
    DataRow dr = drv.GetRow();
    dr.Delete();
  }
}
```

If you run the application now, you can add, remove, and modify the items displayed in the grids.



Note: The DataSet we are using contains not only the tables you see, but also the DataRelation that connects the two tables. That relation came from the MDB file and was downloaded from the server along with the data. The relation has a ChildKeyConstraint property that specifies, among other things, that delete operations cascade through the tables. In other words, if you delete a category, all products that belong to that category are also automatically deleted.

Commit Changes to the Server

We are almost done now. The only piece still missing is the code that submits the changes made by the user back to the server, so they can be applied to the database.

The first step is to modify the page constructor once again and connect the event handler for the "Commit Changes" button:

C#

```
// Load data, hook up event handlers
public Page()
{
  InitializeComponent();
  LoadData();

  _gridCategories.SelectionChanged += _gridCategories_SelectionChanged;
  _btnAdd.Click += _btnAdd_Click;
  _btnRemove.Click += _btnRemove_Click;
  _btnCommit.Click += _btnCommit_Click;
}
```

And here is the implementation for the event handler:

C#

```
// Commit changes to server
private void _btnCommit_Click(object sender, RoutedEventArgs e)
{
  SaveData();
}
```

And here is the implementation for the **SaveData** method, which does the real work:

C#

```
// Save data back into the database
void SaveData()
{
  if (_ds != null)
  {
    // get changes of each type
    byte[] dtAdded = GetChanges(DataRowState.Added);
```

```
    byte[] dtModified = GetChanges(DataRowState.Modified);
    byte[] dtDeleted = GetChanges(DataRowState.Deleted);

    // Invoke service
    var svc = GetDataService();
    svc.UpdateDataCompleted += svc_UpdateDataCompleted;
    svc.UpdateDataAsync(dtAdded, dtModified, dtDeleted);
  }
}
void svc_UpdateDataCompleted(object sender, UpdateDataCompletedEventArgs e)
{
  if (!string.IsNullOrEmpty(e.Result))
  {
    throw new Exception("Error updating data on the server: " + e.Result);
  }
  _tbStatus.Text = "Changes accepted by server.";
  _ds.AcceptChanges();
}
```

The method starts by calling the **GetChanges** method to build three byte arrays. Each one represents a DataSet with the rows that have been added, modified, or deleted since the data was downloaded from the server. Then the method invokes the Web service we implemented earlier and listens for the result. If any errors were detected while updating the server, we throw an exception (real applications would deal with the error in a more elegant way).

The only piece still missing is the **GetChanges** method. Here it is:

C#

```
byte[] GetChanges(DataRowState state)
{
    DataSet ds = _ds.GetChanges(state);
    if (ds != null)
    {
        MemoryStream ms = new MemoryStream();
        ds.WriteXml(ms);
        return ms.ToArray();
    }
    return null;
}
```

The method uses the DataSet.GetChanges method to obtain a new **DataSet** object containing only the rows that have the DataRowState specified by the caller. This is the same method available on the ADO.NET **DataSet** class.

The method then serializes the **DataSet** containing the changes into a **MemoryStream**, and returns the stream contents as a byte array.

Try running the application now. Make some changes, then click the "Commit Changes" button to send the changes to the server. If you stop the application and start it again, you should see that the changes were indeed persisted to the database.

Note: The update may fail depending on the changes you make. For example, if you delete one of the existing categories, all products that belong to that category will also be removed from the **DataSet** on the client. When you try to apply these changes to the server, the transaction will likely fail because the database may contain tables that still refer to the products you are trying to delete (the **Orders** table in this case). To test the delete/commit action, try creating a category, committing the changes, then deleting the new category and

> committing again. This will succeed because the new category won't have any products associated with it in the database.

A real application would have to deal with this type of problem more intelligently. For example, we could have loaded the **Orders** table as well, inspect the **DataSet** to detect whether the user is trying to delete an item that he should not, and issue a warning. That is left as an exercise for the reader.

Optimize the Data Transfers

Now that our sample application is ready, we could easily go a step further and optimize its performance by compressing the data streams between client and server. This would require two small modifications to our project:

Compress the Data on the Server

Before sending the data to the client, we can compress the stream using the **C1.Zip** library (the desktop version included with ComponentOne Studio, not the Silverlight version). To do this, we modify the **MasterDetailWeb** project by adding a reference to the C1.Zip library, then changing the **GetData** method in the **DataService.asmx.cs** Web service as follows:

C#

```
[WebMethod]
public byte[] GetData(string tables)
{
  // Create DataSet with connection string
  var ds = GetDataSet();

  // Load data into DataSet
  ds.Fill(tables.Split(','));

  // Persist to stream
  var ms = new MemoryStream();
  using (var sw = new C1.C1Zip.C1ZStreamWriter(ms))
    ds.WriteXml(sw, XmlWriteMode.WriteSchema);
  //ds.WriteXml(ms, XmlWriteMode.WriteSchema);

  // Return stream data
  return ms.ToArray();
}
```

This small change is all that's required to compress the data on the server before sending it to the client. Instead of persisting the DataSet directly to the stream, we are now persisting it through a **C1ZStreamWriter** that compresses the data before writing it to the stream. Because the data is encoded as XML, compression rates will be very good.

Decompress the Data on the Client

To decompress the data on the client, we can modify the **MasterDetail** project by adding a reference to the **C1.Zip** library (Silverlight version), then changing the **svc_GetDataCompleted** method in the **MainPage.xaml.cs** file as follows:

C#

```
void svc_GetDataCompleted(object sender, GetDataCompletedEventArgs e)
{
  // Handle errors
  if (e.Error != null)
  {
    _tbStatus.Text = "Error downloading data...";
    return;
  }

  // Parse data stream from server (DataSet as XML)
  _tbStatus.Text = string.Format(
    "Got data, {0:n0} kBytes", e.Result.Length / 1024);
  var ms = new MemoryStream(e.Result);
  _ds = new DataSet();
  using (var zr = new C1.C1Zip.C1ZStreamReader(ms))
    _ds.ReadXml(zr);
  //_ds.ReadXml(ms);

  // Got the data, bind controls to it
  BindData();
}
```

Instead of reading the DataSet directly from the compressed stream, we now read it through a **C1ZStreamReader** that decompresses it automatically.

Note: Without compression, the application transfers about 150k bytes of data on startup. After making these small changes, the initial data transfer is reduced to less than 50k. These two small changes reduce the amount of data transferred by about two-thirds. This reduces network traffic and the time it takes for the application to initialize and show the data.

Using Data with Schedule for Silverlight

It is possible to bind **Data for Silverlight** to **Schedule for Silverlight**. For an in-depth sample, see the **C1Scheduler_DataBinding** sample installed with **Silverlight Edition**.

The **C1Scheduler_DataBinding** sample is similar to the **MasterDetail** sample except for the following differences:

- It uses the **Schedule.mdb** database, installed by default with **Studio for WPF**.
- It loads the **Appointments** table from the server.
- The UI is primarily represented by the **C1Scheduler** control with additional navigation controls.
- Data is bound to the **C1Scheduler** control in the same way as it is done for all platforms:

C#

```
// set mappings for AppointmentStorage
// set mapping names to the names of Appointment table columns
AppointmentMappingCollection mappings =
sched1.DataStorage.AppointmentStorage.Mappings;
mappings.IdMapping.MappingName = "Id";
mappings.Subject.MappingName = "Subject";
mappings.Body.MappingName = "Body";
mappings.End.MappingName = "End";
mappings.Start.MappingName = "Start";
mappings.Location.MappingName = "Location";
mappings.AppointmentProperties.MappingName = "Properties";
...
// set C1Scheduler data source to DataTable loaded from server
sched1.DataStorage.AppointmentStorage.DataSource = _dtAppointments;
```

- When users add, remove or edit appointments, all changes are propagated to the underlying DataTable automatically by **C1Scheduler** with no code required.

Conclusion

It is possible to build data-centric Silverlight applications today, using familiar tools and techniques. The introduction of new data technologies does not mean ADO.NET developers have to throw out all their knowledge, tools, and existing code. In most cases, these new technologies extend rather than replace the existing tools and patterns.

C1.Silverlight.Data provides a Silverlight implementation of a substantial subset of the ADO.NET classes. This allows developers to:

- Leverage their knowledge of ADO.NET and the rich object model it provides, while at the same time enjoying the benefits of new technologies such as LINQ, WCF, and so on.
- Exchange relational data between client and server in a simple and efficient way;
- Write code that uses similar classes and object models on client and server. Server and client use the same language, similar classes, and similar object models.