DATA POINTS

Using Silverlight 2 With ADO.NET Data Services

John Papa

CODE DOWNLOAD AVAILABLE FROM THE MSDN CODE GALLERY

Browse the Code Online

Contents

Cross-Domain Communications
Being Smart About Sensitive Data
Getting Started Quickly with EDM
Referencing ADO.NET Data Services from Silverlight
Retrieving Data Using LINQ
Deferred Loading
Saving Data
Retrieving Data in Advance
To Be Continued

ADO.NET Data Services makes it tremendously easy to expose data and allow updates over HTTP using its RESTful (Representational State Transfer) abilities. Your Silverlight applications can take advantage of ADO.NET Data Services to send and receive data using the unique URIs that map to entities. You can also use LINQ queries in the Silverlight client to interact with the entities on the server by using the ADO.NET Data Services Silverlight client library.

ADO.NET Data Services and Silverlight make a powerful combination, but to make them work well together you need a good understanding of a few aspects that may not be immediately obvious. So, here I'll address some steps you can take to ensure a more successful experience when building applications with ADO.NET Data Services and Silverlight.

Cross-Domain Communications

ADO.NET Data Services does not currently support cross-domain communications. Cross-domain communications are supported with standard REST and SOAP services, but not with ADO.NET Data Services. (As a note: the data services team is exploring the space and will post their progress to the Astoria team blog as they go.) This means that a Silverlight 2 client cannot talk to services exposed through ADO.NET Data Services if those services are hosted on a different domain than the domain that hosts the Silverlight client application. For more information on cross-domain policies, please refer to my September 2008 "Data Points" column. In that column, I discuss the file formats and how the policies work.

Being Smart About Sensitive Data

When you are using HTTP communications, all values sent via URIs are clearly visible through a variety of network sniffing tools. One way to combat this is to use SSL to encrypt all HTTP communications. Also, send no sensitive data, such as social security numbers, or any other private data, in the URI. It is a good idea never to use sensitive data as identifiers. Before choosing any RESTful communication, be sure to use a meaningless identifier such as a GUID, a number, or an IDENTITY value. For example, the following sample URI could retrieve employee data using the employee ID of 11:

```
http://[YourDomainHere]/MyService.svc/Employee(11)
```

The number 11 is transparent in the URI, which is why it is important not to use sensitive information. Fortunately, in this case the number 11 represents a fabricated identifier, so no personal data is exposed.

Getting Started Quickly with EDM

Perhaps the simplest way to get up and running with ADO.NET Data Services is to expose data from a relational database as a logical entity model using the ADO.NET Entity Framework. The Entity Framework's Entity Data Model (EDM) is fully aware of how to allow read and update access to its entities. This built-in ability of the EDM allows it to work in concert with ADO.NET Data Services with very little setup.

The first step in exposing an EDM created with the Entity Framework is to create the ADO.NET Data Services service from the Visual Studio templates dialog. This creates a template that can easily be modified to expose an EDM from the Entity Framework. The class constructor inherits from the DataService<T> base class. The T represents the data source class, which in this case is the Entity Framework data source class named Entities. The data source class, also known as the object context class in the Entity Framework, is what allows access to the EDM to retrieve and save data. Since the data source class is created automatically with the Entity Framework, creating an ADO.NET Data Service that exposes the EDM is quite simple. In **Figure 1**, the code shows the service class NWDataService inheriting from the DataService<Entities>.

Figure 1 Creating the DataService

Copy Code

```
public class NWDataService : DataService<Entities>
//public class NWDataService: DataService< /* TODO: put your data source
//class name here */ >
 {
    // This method is called only once to initialize service-wide
    //policies.
    public static void InitializeService(IDataServiceConfiguration
        config)
    {
        //set rules to indicate which entity sets and service operations
        //are visible, updatable, etc.
        config.SetEntitySetAccessRule("ProductSet", EntitySetRights.All);
        config.SetEntitySetAccessRule("CategorySet",
            EntitySetRights.AllRead);
        config.SetEntitySetAccessRule("SupplierSet",
            EntitySetRights.AllRead);
        config.SetEntitySetAccessRule("OrderSet",
            EntitySetRights.AllRead);
        config.SetEntitySetAccessRule("OrderDetailSet",
            EntitySetRights.All);
        config.SetEntitySetAccessRule("CustomerSet",
            EntitySetRights.AllRead);
        /// The rest of the entity sets are not accessible.
        /// Therefore, no proxy classes are created for them either.
    }
}
```

The next step is to allow or deny read and write access to each of the entity sets in the EDM as shown in **Figure 1**. This can be done on all entity sets using the * or on an individual entity set level by specifying each entity set name. The SetEntitySetAccessRule method accepts the name of the entity set that the rule will be applied to and one or more EntitySetRights enum values. Notice in **Figure 2** that each of the entity sets for ProductSet, OrderSet, OrderDetailSet, CustomerSet, SupplierSet, and CategorySet allow all access. This means that those six entity sets allow reading, inserting, updating, and deleting. These are service-wide access settings and apply to all requests coming into the system. Any entity set not listed, such as EmployeeSet or RegionSet, is not accessible.

Figure 2 System.Data.Services EntitySetRights Enums

Enum	Description
All	All reads and writes are permitted on the specified entity
AllRead	All reads are permitted
AllWrite	All write operations are permitted
None	No access permitted to the specified entity
ReadMultiple	Reading multiple rows is permitted
ReadSingle	Reading a single row is permitted
WriteAppend	Creating new data is permitted
WriteDelete	Deleting data is permitted
WriteMerge	Merge-based updates are permitted

WriteReplace Replacing is permitted

The EntitySetRights enum values indicate the type of access allowed for the entity set. **Figure 2** shows all of the valid enum values and their descriptions. Permission can be given on a global scale for all entity sets, too. For example, the following line of code would allow all read access to all entity sets, but would not allow write access:

Copy Code

```
config.SetEntitySetAccessRule("*", EntitySetRights.AllRead);
```

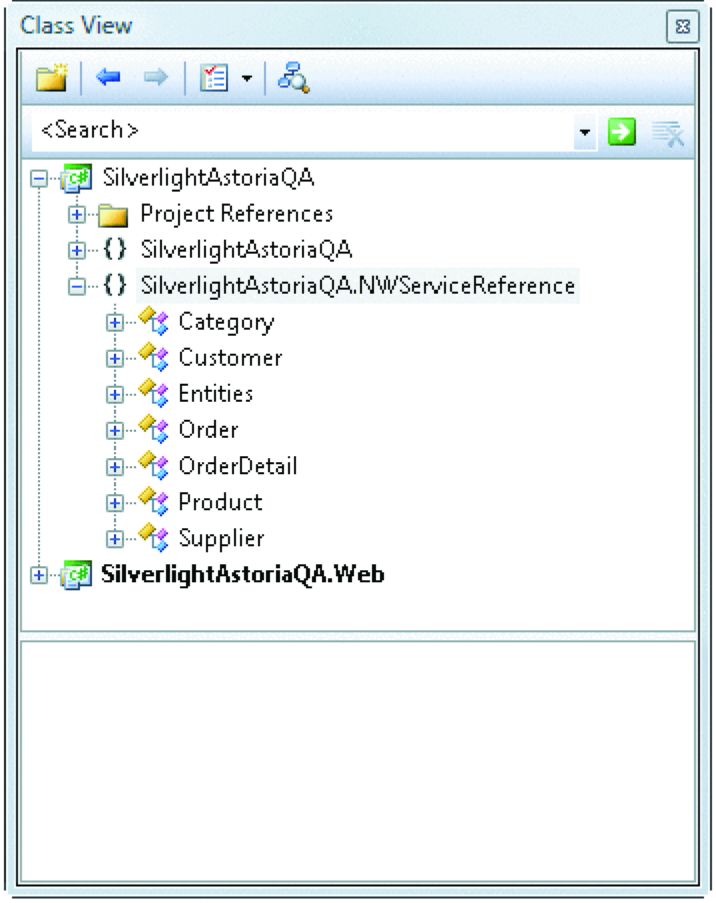
Rights can be combined to allow more than one right for an entity set, as well. The following code makes the ProductSet entity set accessible for reading and updating only:

Copy Code

```
config.SetEntitySetAccessRule("ProductSet",
EntitySetRights.AllRead |
EntitySetRights.WriteMerge |
EntitySetRights.WriteReplace);
```

Following these steps will expose the six entity sets (shown in **Figure 1**) and make them available via an ADO.NET Data Service. Other entity sets and permissions can be customized too.

Entity models built with other object-relational mappings (ORM)s, such as LINQ to SQL and NHibernate, can also be used with ADO.NET Data Services. The context objects must implement IQueryable for each entity set that will be queried. For create, update, and delete operations, the context object must implement the IUpdatable interface. ADO.NET Data Services has built-in knowledge of the ADO.NET Entity Framework that allows it to support querying and updating via ADO.NET Data Services. It is likely that future versions of other ORMs will also have support for ADO.NET Data Services or at least have helper classes to help jump-start it. But for now, if you want to use an ORM other than Entity Framework with ADO.NET Data Services, you will need to implement the interfaces described above. In fact, ADO.NET Data Services is not specific to relational data. Any data source can be configured via the techniques noted in this paragraph.



```
SilverlightAstoriaQA
  Project References
  SilverlightAstoriaQA
  SilverlightAstoriaQA.NWServiceReference
    Category
    Customer
    Entities
    Order
    OrderDetail
    Product
    Supplier
SilverlightAstoriaQA.Web
```

Figure 3 **Generated Classes from the Service Reference**

Referencing ADO.NET Data Services from Silverlight

Once the ADO.NET Data Service has been created and built, a Silverlight application can reference the service and interact with it. An ADO.NET Data Service can be referenced from the Silverlight application by right-clicking the Service References node in the Solution Explorer to open the Add Service Reference dialog window. You can either enter the URI of the service in the Address box or you can click the Discover button, and then click the Go button to locate the service. Once the metadata for the service is retrieved, the service and the entity sets that it exposes will appear in a list. Finally, name the service reference and click the OK button.

This creates a proxy class in the Silverlight client that allows you to interact with the ADO.NET Data Service. If you click the Show All Files button in the Solution Explorer window and expand the NWServiceReference node completely, you will see a Reference.cs file. This file contains the generated proxy class that allows you to interact with the ADO.NET Data Service and the generated classes for the entities that are exposed through the service. The list of classes can also be seen by looking in the Class View window, as shown in **Figure 3**.

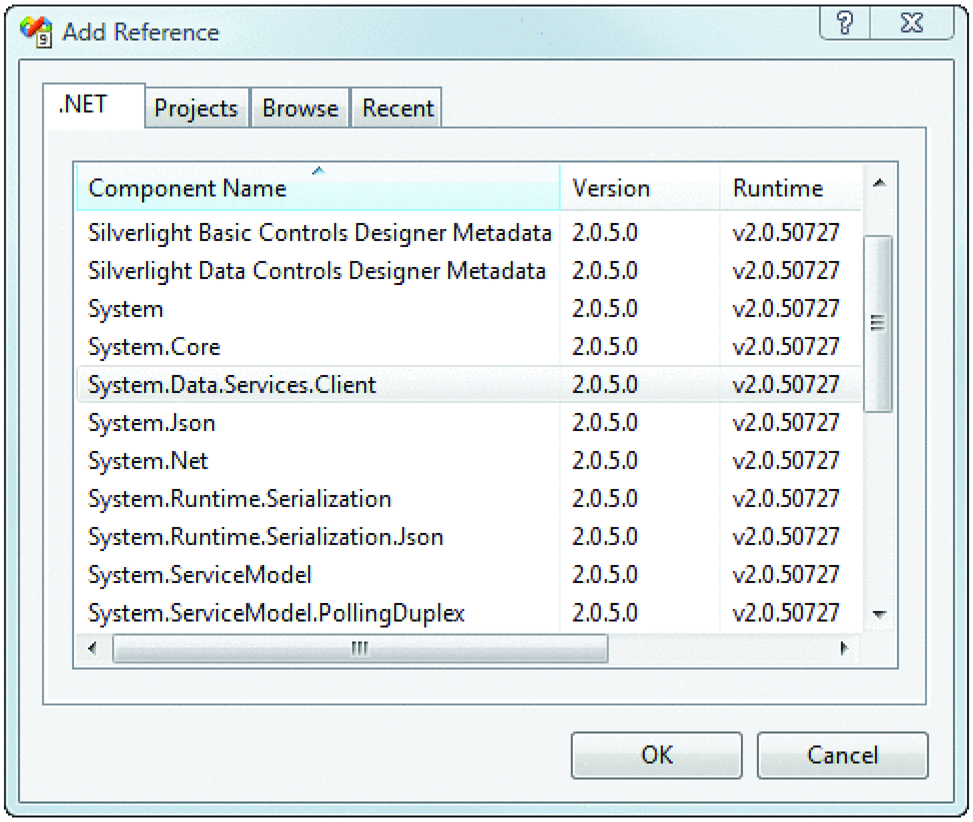
Notice that all of the entities in the Entity Framework's model in the Web project are not listed in the Class View. Only entity sets that are accessible via the data service will have proxy classes generated for them when a service reference is added to the ADO.NET Data Service from a Silverlight client. The six entity sets shown in the Class View were exposed by using the SetEntitySetAccessRule methods in the ADO.NET Data Service, therefore these are the only entity sets available to the Silverlight client. The seventh class in the Class View window, the Entities class, is a proxy class that represents the data service as a whole and facilitates the calls to NWDataService.svc.

Retrieving Data Using LINQ

The next step is to reference the System.Data.Services.Client assembly in the Silverlight project (See **Figure 4**). This assembly makes it easy to interact with the ADO.NET Data Service using LINQ. For example, the following code creates a LINQ query that selects all of the products using ADO.NET Data Services:

Copy Code

```
_products.Clear();
DataServiceQuery<Product> dq = (from p in _ctx.ProductSet select p)
as
DataServiceQuery<Product>;
dq.BeginExecute(new AsyncCallback(FindProduct_Completed), dq);
```



Add Reference

Component Name	Version	Runtime
Silverlight Basic Controls Designer Metadata	2.0.5.0	v2.0.50727
Silverlight Data Controls Designer Metadata	2.0.5.0	v2.0.50727
System	2.0.5.0	v2.0.50727
System.Core	2.0.5.0	v2.0.50727
System.Data.Services.Client	2.0.5.0	v2.0.50727
System.Json	2.0.5.0	v2.0.50727
System.Net	2.0.5.0	v2.0.50727
System.Runtime.Serialization	2.0.5.0	v2.0.50727
System.Runtime.Serialization.Json	2.0.5.0	v2.0.50727
System.ServiceModel	2.0.5.0	v2.0.50727
System.ServiceModel.PollingDuplex	2.0.5.0	v2.0.50727

Figure 4 **Referencing the ADO.NET Data Services Client Library from Silverlight**

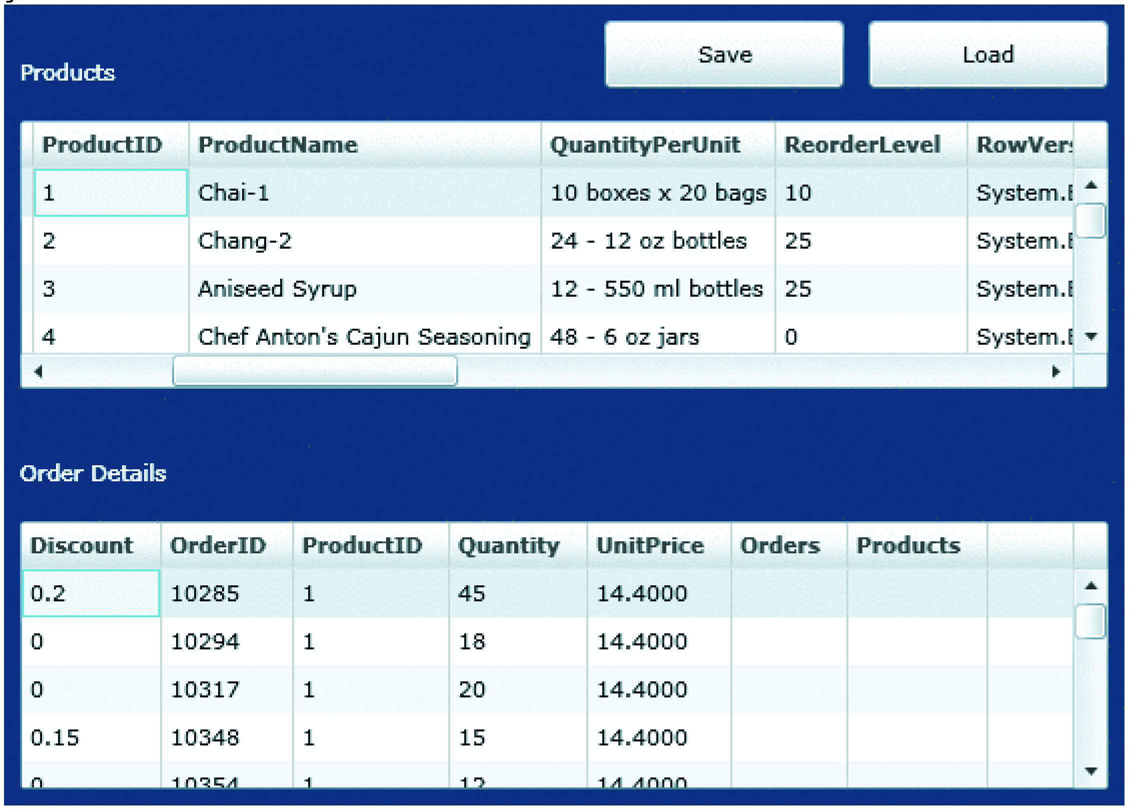
The LINQ query in this code is translated into a URI that ADO.NET Data Services can read. The query is cast to a DataServiceQuery<T>, which is part of the System.Data.Services.Client namespace. The query executes asynchronously, so a valid callback method must be specified to receive the results from the query. When the query in the preceding code sample executes and the data is returned, the FindProduct_Completed method will be invoked.

The FindProduct_Completed method (shown in **Figure 5**) accepts an IAsyncResult parameter containing the results from the query. The results are read by calling the EndExecute method, which yields a set of Product objects. Each product is then examined and an event handler is assigned to its PropertyChanged event. (In the sample code, I extended the Product class using a partial class to add the INotifyPropertyChanged implementation.) This step ensures that when changes are made to any Product instance in Silverlight, that the Product will notify the DataServiceContext (Entities, in **Figure 5**) by calling the UpdateObject method. Without this code, DatServiceContext would be unaware of any changes made by the user to a Product instance. Assuming that the _products variable is of type ObservableCollection<Product> and that it is bound to the DataContext of a DataGrid control, the products will appear in the DataGrid control (see **Figure 6**).

Figure 5 FindProduct_Completed Method

Copy Code

```
private void FindProduct_Completed(IAsyncResult result)
{
    DataServiceQuery<Product> query = (DataServiceQuery<Product>)result.
         AsyncState;
    try
    {
        var entities = query.EndExecute(result);
        foreach (Product item in entities)
        {
            item.PropertyChanged += ((sender, e) =>
                                     {
                                        Product entity = (Product)
                                                         sender;
                                        _ctx.UpdateObject(entity);
                                    });
            _products.Add(item);
        }
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Failed to retrieve data: " + ex.ToString());
    }
}
```



ProductID	ProductName	QuantityPerUnit	ReorderLevel	RowVer
1	Chai-1	10 boxes x 20 bags	10	System.I
2	Chang-2	24 - 12 oz bottles	25	System.I
3	Aniseed Syrup	12 - 550 ml bottles	25	System.I
4	Chef Anton's Cajun Seasoning	48 - 6 oz jars	0	System.I

Discount	OrderID	ProductID	Quantity	UnitPrice	Orders	Products
0.2	10285	1	45	14.4000		
0	10294	1	18	14.4000		
0	10317	1	20	14.4000		
0.15	10348	1	15	14.4000		
0	10354	1	12	14.4000		

Figure 6 **Loading Products and Order Details**

Deferred Loading

The ADO.NET Data Services client library in Silverlight offers the ability to load objects associated with an object already in the DataServiceContext. For example, when a product is selected in the upper DataGrid in the Silverlight control shown in **Figure 6**, before the order details for that product can be bound to the lower DataGrid, they must be retrieved. The first step in this process is to assign an event handler to the SelectionChanged event of the productDataGrid. Then when a product is selected, the event handler (shown in **Figure 7**) uses the BeginLoadProperty method to ask ADO.NET Data Services to get the OrderDetails objects for the selected product.

Figure 7 Asking for Order Details

Copy Code

```
void productDataGrid_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    Product product = productDataGrid.SelectedItem as Product;
    if (product == null) return;

    _orderDetails.Clear();
    if (product.OrderDetails == null || product.OrderDetails.Count == 0)
    {
        _ctx.BeginLoadProperty(product, "OrderDetails", FindOrderDetail_Completed,
null);
    }
    else
    {
        LoadOrderDetails();
    }
}
```
The BeginLoadProperty method makes a network request to go get the OrderDetails records, and when it returns it will invoke the callback method, FindOrderDetail_Completed. If any additional state needs to be passed from this code to the callback method, it can be passed in the fourth parameter of the BeginLoadProperty method. For example, the same event handler may be used to receive the results from several asynchronous queries. A value could be passed in the state parameter to help the callback method determine whose asynchronous call query results it is receiving.

When the callback is invoked, the results are read into the DataServiceContext object using its EndLoadProperty method, as shown here:

Copy Code
```
_ctx.EndLoadProperty(result);
Deployment.Current.Dispatcher.BeginInvoke(() => LoadOrderDetails());
```
The order details are then loaded by the method LoadOrderDetails. Since the code runs as the result of an asynchronous operation completing, there is no guarantee that this code will be running on the UI thread. If the code is not running on the UI thread, then the new set of order details will not appear in element DataGrid. Basically, any UI operation must run on the UI thread. One way to make sure an operation runs on the UI thread is to use the Dispatcher object's BeginInvoke method. The preceding code uses the Dispatcher to make sure the order details are loaded on the UI thread by calling LoadOrderDetails (shown in **Figure 8**).

Figure 8 Loading Order Details

Copy Code
```
private void LoadOrderDetails()
{
    Product product = productDataGrid.SelectedItem as Product;
    if (product == null) return;

    var query = (from od in product.OrderDetails
                 orderby od.OrderID ascending
                 select od);
    foreach (OrderDetail item in query)
    {
        item.PropertyChanged += ((sender, e) =>
        {
            OrderDetail entity = (OrderDetail)sender;
            _ctx.UpdateObject(entity);
        });
        _orderDetails.Add(item);
    }
}
```
LoadOrderDetails grabs the currently selected Product instance and creates a LINQ query that will select all of the OrderDetails objects from the selected Product. This is possible since the order details were loaded into DataServiceContext using EndLoadProperty. Once the order details are retrieved using the LINQ query in **Figure 8**, each OrderDetail object instance has its PropertyChanged event handler assigned a lambda expression that tells DataServiceContext if any property values are changed. Assuming the _orderDetails object is an ObservableCollection<OrderDetail> element that is bound to orderDetailsDataGrid, the order details will appear (as you saw in **Figure 6**).

Saving Data

The sample shown also allows the user to save changes to both the products and the order details. There are two basic steps to allow for saving changes: notifying DataServiceContext when changes occur and issuing the save operation asynchronously.

These hierarchical sets of data each have a partial class in Silverlight that extends the Product and OrderDetail entities. The partial class for Product (shown in **Figure 9**) implements the INotifyPropertyChanged interface, which requires that the PropertyChanged event be implemented. The partial Product class that is generated by creating the service reference to the ADO.NET Data Service creates partial methods for each of the public properties on the class. Each property gets a method that fires when a property is about the change and one when the property has already changed. For example, the Product class's ProductName property has an OnProductNameChanging partial method and an OnProductNameChanged partial method. **Figure 9** shows that the OnProductNameChanged partial method in the custom code (not the generated class) fires the PropertyChanged event. This is the key for tracking all changes to all entities' property values. DataServiceContext needs to know when the property values change and what they change to; otherwise it cannot save the changes.

Figure 9 Implementing Change Notification

Copy Code

```
public partial class Product :INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    private void FirePropertyChanged(string propertyName)
    {
        var handler = PropertyChanged;
        if (handler != null)
            handler(this, new PropertyChangedEventArgs(propertyName));
    }

    partial void OnProductIDChanged() { FirePropertyChanged("ProductID"); }
    partial void OnProductNameChanged() { FirePropertyChanged("ProductName"); }
    partial void OnDiscontinuedChanged() { FirePropertyChanged("Discontinued"); }
    partial void OnDiscontinuedDateChanged() {
FirePropertyChanged("DiscontinuedDate"); }
    partial void OnQuantityPerUnitChanged() {
FirePropertyChanged("QuantityPerUnit"); }
    partial void OnReorderLevelChanged() { FirePropertyChanged("ReorderLevel"); }
    partial void OnRowVersionStampChanged() {
FirePropertyChanged("RowVersionStamp"); }
    partial void OnUnitPriceChanged() { FirePropertyChanged("UnitPrice"); }
    partial void OnUnitsInStockChanged() { FirePropertyChanged("UnitsInStock"); }
    partial void OnUnitsOnOrderChanged() { FirePropertyChanged("UnitsOnOrder"); }
}
```

Once the property changed event handlers are set for each property, DataServiceContext will be made aware of any changes and the rest of the work to save the data is relatively simple. When a user clicks the Save button the BeginSaveChanges method is invoked on DataServiceContext, as shown here:

Copy Code

```
private void SaveButton_Clicked(object sender, RoutedEventArgs e)
{
    _ctx.BeginSaveChanges(SaveChangesOptions.Batch,
        new AsyncCallback(Save_Complete), null);
}
```

This method makes a POST operation through ADO.NET Data Services, sending all changes that DataServiceContext is aware of. The POST is issued asynchronously, so a callback method is required to process the results. The SaveChangesOption.Batch parameter indicates that all changes should be saved within a transaction in batch mode. If any save operations fail, they will all fail and the transaction will be rolled back. This technique works whether you are saving a single record from one entity or multiple records from multiple associated entities.

Retrieving Data in Advance

Earlier I discussed how to perform deferred loading using the BeginLoadProperty method of DataServiceContext to grab records for an existing entity and attach them to DataServiceContext. This technique is great for getting additional data on demand, especially when the data is not always needed. Deferred loading saves the cost of retrieving data that the user may not want to see, unless the user asks for the data (which happens when the user selects the Product in the productDataGrid control).
Another technique for retrieving hierarchical data is to ask for it all in advance. For example, when retrieving Product records it might be beneficial to also get each Product's Category and Supplier. Otherwise, the Category and Supplier properties of a Product instance will be null. The BeginLoadProperty technique could be used, but that would require making many network requests. A better technique to get all of the data at once is to use the Expand method in the LINQ query, since it would require only a single HTTP request. The following query shows the Expand method asking for all categories and suppliers for the selected products in the LINQ query.

Copy Code
```
DataServiceQuery<Product> dq = (
 from p in _ctx.ProductSet.Expand("Categories").Expand("Suppliers")
 select p)
 as DataServiceQuery<Product>;
```
The Expand method comes at a cost as it retrieves additional data. This should be used only when it's necessary to grab all data up front. The Expand method shown in the preceding code grabs child records for each Product entity. In other words, the Category and Supplier are both direct children of a Product. Notice that the name of the property, not the name of the entity set, is passed to the Expand method.
If multiple levels of a hierarchy are required, the syntax of the Expand method can be adapted to get those records too. For example, if you want to grab the OrderDetail element for each Product as well as each of those OrderDetail's Orders, the syntax could look like this:

Copy Code
```
DataServiceQuery<Product> dq = (
    from p in _ctx.ProductSet.Expand("OrderDetails/Orders")
    select p)
    as DataServiceQuery<Product>;
```
This code indicates that the query should get the entities for the OrderDetails property of each Product as well as each entity for the Orders. (The Product class has an OrderDetails property and the OrderDetail class has an Orders property.) The OrderDetail records are implied since they are required when getting the Orders property for each Product. This query will return over 8MB of XML data to the Silverlight client application. This is a lot of data and could negatively affect performance on slower connections. I recommend using the Expand method only when the data is needed and even in those cases, use it with the most restrictive filter possible to avoid retrieving data you don't need.

To Be Continued

There is quite a bit of functionality exposed by the combination of ADO.NET Data Services and Silverlight that make for a robust data-driven application. In a future installment of Data Points, I hope to revisit the topic and offer more tips. For now, check out my blog: johnpapa.net, the Silverlight.net Web site, and previous installments of Data Points for more on Silverlight data-centric applications.

Send your questions and comments for John to mmdata@microsoft.com.

John Papa (johnpapa.net) is a Senior Consultant with ASPSOFT and a baseball fan who spends summer nights rooting for the Yankees with his family. John, a C# MVP, Silverlight Insider, and INETA speaker, has authored several books, including his latest titled *Data-Driven Services with Silverlight 2* (O'Reilly, 2009). He often speaks at conferences such as Mix, DevConnections, and VSLive