SOA DATA ACCESS

Flexible Data Access With LINQ To SQL And The Entity Framework

Anthony Sneed – December 2008

CODE DOWNLOAD AVAILABLE FROM THE MSDN CODE GALLERY

Browse the Code Online

THIS ARTICLE DISCUSSES:

- LINQ to SQL
- ADO.NET Entity Framework, service-oriented architecture
- Data transfer objects
- Persistence, updates, and change tracking

THIS ARTICLE USES THE FOLLOWING TECHNOLOGIES:

LINQ to SQL, ADO.NET Entity Framework, and SOA

Contents

With the releases of LINQ to SQL and the ADO.NET Entity Framework, developers now have two products from Microsoft designed to reduce the impedance mismatch that exists between the worlds of relational data and object-oriented programming. Each of these eliminates much of the plumbing code you would otherwise have to write to perform object persistence. However, incorporating these object relational mapping (ORM) technologies into a service-oriented application architecture presents a whole new set of challenges for the application developer.

For instance, how do you create a data access layer (DAL) that abstracts object persistence from other parts of the application, so that you can swap out one ORM provider for another should the need arise? How do you track changes to entities without requiring LINQ to SQL or the Entity Framework on the client? How do you perform inserts, updates, and deletes for multiple entities in a single transaction with just one trip to the service?

In this article, I will provide you with a roadmap and make some suggestions for how you can address these issues. I'll start by creating a DAL for processing orders from the Northwind sample database. The DAL will rely on an interface with two implementations, one using LINQ to SQL, and the other using the Entity Framework (see **Figure 1**). Both LINQ to SQL and the Entity Framework have tools that generate entities based on a database schema, but rather than using these entities, the DAL will only expose data transfer objects (DTOs) that are basically ignorant when it comes to their own persistence.

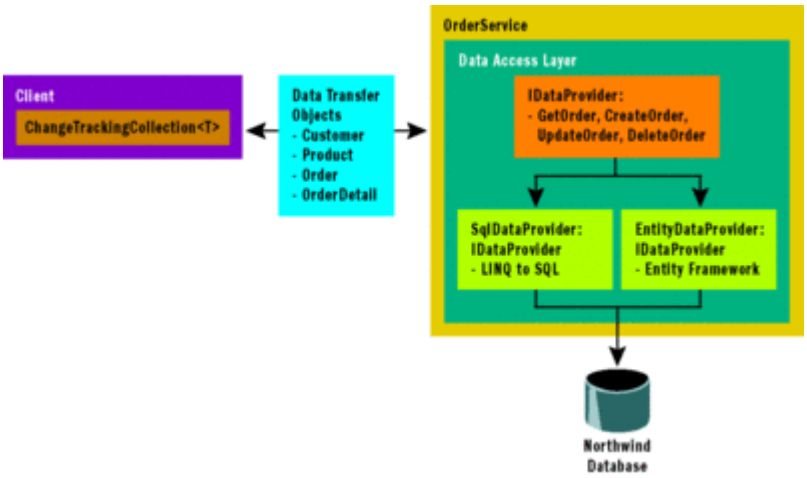

Figure 1 **Service-Oriented Architecture for an Order Processing App** (Click the image for a larger view)

LINQ to SQL and the Entity Framework each have different methods for persisting disconnected entities, depending on whether you are creating a new entity or updating an existing one. Therefore, in order to invoke the appropriate API, you need to know the state of the entity beforehand. If, for example, you have an UpdateOrder method that accepts an order with order details, you could add three parameters to the UpdateOrder method accepting created, updated, and deleted order details. However, this tends to pollute the method signature with plumbing details.

Alternatively, you could pass change state information out-of-band, for instance, in a SOAP or HTTP header, but this could couple change tracking to the communications protocol. Another approach, and the one I am advocating here, is to make change state part of the data contract for each entity:

Copy Code

```
[DataContract]
public enum TrackingInfo
{
    [EnumMember]
    Unchanged,
    [EnumMember]
    Created,
    [EnumMember]
    Updated,
    [EnumMember]
    Deleted
}
```

In addition to other properties, each DTO would have a TrackingState property using the TrackingInfo enum. All that's required is for both sides to agree on this data contract.

Even though I've added a TrackingState property to each DTO, I must still rely on the client, which has no knowledge of LINQ to SQL or the Entity Framework, to track which objects have been created, updated or deleted and to set the TrackingState property to the corresponding value (see **Figure 1**). I've crafted a generic change-tracking collection to perform this task. There are two types of the collection, one that extends ObservableCollection<T> for Windows Presentation Foundation (WPF) applications, and another that extends BindingList<T> for use with Windows Forms applications. Each caches deleted items before removing them from the collection and has a GetChanges method which returns only items that have been inserted, updated, or deleted.

Creating the Data Access Layer

The DAL helps insulate other parts of the application from the details of object persistence. For this reason, it is important that objects exposed through the DAL exclude remnants of any specific data access technology. Both LINQ to SQL and the Entity Framework allow you to manufacture entities using both command-line tools and Visual Studio designers; however, these code-generated entities possess some artifacts that betray their origins.

LINQ to SQL, for example, represents data relations using the EntityRef<T> and EntitySet<T> collection types in order to support deferred loading of associated objects, while the Entity Framework represents navigation properties using EntityReference<T> and EntityCollection<T>. Likewise, these entities incorporate code elements intended to support client-server scenarios, such as validation (using partial methods) and data binding (with INotifyPropertyChanged), which are unnecessary when you only use them for communication with a Windows Communication Foundation (WCF) service.

Because both LINQ to SQL and the Entity Framework support serialization of entities using the DataContractSerializer (by marking them with DataContract and DataMember attributes), you may be tempted to use them in your service operations anyway, despite the extra baggage they carry. After all, collection types such as those used by LINQ to SQL and the Entity Framework show up as simple arrays on the client when you set a WCF service reference.

Consider, however, that when you set the SerializationMode of the LINQ to SQL DataContext from None to Unidirectional, only the "many" side of one-to-many relations will be marked with the DataMember attribute. This means that if you have an Order entity with Customer and Order_Details properties, Order_Details will be marked with DataMember and included in the data contract, but Customer will not. The Entity Framework, on the other hand, will include more fields than you may want in the data contract, generating classes on the client for the Entity Framework-specific features, such as EntityKey, EntityKeyMember, EntityObject, and EntityReference.

For these reasons, you should avoid exposing either LINQ to SQL or Entity Framework tool-generated entities through the DAL and instead return simple DTOs whose sole purpose in life is to carry data across service boundaries. For example, the Order DTO (shown in **Figure 2**) contains only properties and no methods. All the DTOs are defined in a DataTransferObjects namespace, to distinguish them from tool-generated classes with the same name.

Figure 2 DTO Representing an Order from Northwind

Copy Code

```
namespace DataTransferObjects
{
    [DataContract]
    public class Order
    {
        [DataMember]
        public int OrderID { get; set; }

        [DataMember]
        public string CustomerID { get; set; }

        [DataMember]
        public string CustomerName { get; set; }

        [DataMember]
        public DateTime? OrderDate { get; set; }

        [DataMember]
        public List<OrderDetail> OrderDetails { get; set; }

        [DataMember]
        public byte[] Updated { get; set; }

        [DataMember]
        public TrackingInfo TrackingState { get; set; }
    }
}
```

A flexible DAL should buffer the Order Service from changes to the underlying persistence technology. Suppose, for example, you decide to use LINQ to SQL to persist objects because you've selected SQL Server for your data store and your object model bears a close resemblance to the database schema. But later on,

after you've written the LINQ to SQL persistence logic, you decide to use another database system, such as Oracle, or you want to take advantage of the Entity Framework's mapping capabilities to decouple your conceptual and logical schemas. Perhaps a new data access technology comes along that you'd like to use. If you've designed a flexible DAL that's based on a plug-in architecture, you should be able to switch providers on the fly, based on an entry in app.config.

To achieve this flexibility, I've created an interface called IDataProvider, which has methods for retrieving and updating customer orders, as well as retrieving supporting customer and product information (see **Figure 3**). The sample application includes two classes that both implement IDataProvider: SqlDataProvider, which uses LINQ to SQL, and EntityDataProvider, which uses the Entity Framework and LINQ to Entities (see **Figure 1**). You can simply enter the fully qualified class name for the "DataProvider" setting in the DAL's app.config file, and use the CreateInstance method of the Assembly class to create an instance of the class, casting it to IDataProvider. The Order Service doesn't care at all about which class implements IDataProvider, giving you the option to use whatever data provider you choose:

Copy Code

```
IDataProvider provider =
    Assembly.GetExecutingAssembly()
    .CreateInstance(Settings.Default.DataProvider, true)
    as IDataProvider;
```

Figure 3 Decoupling the DAL with an Interface

Copy Code

```
namespace DataAccessLayer
{
    // Alias the namespace containing Data Transfer Objects
    using DTO = DataTransferObjects;

    public interface IDataProvider
    {
        // Retrieve Customers, Products, Orders
        DTO.Customer[] GetCustomers();
        DTO.Product[] GetProducts();
        DTO.Order[] GetCustomerOrders(string customerID);

        // CRUD operations for Order
        DTO.Order GetOrder(int orderID);
        DTO.Order CreateOrder(DTO.Order order);
        DTO.Order UpdateOrder(DTO.Order order);
        void DeleteOrder(DTO.Order order);
    }
}
```

By default, both LINQ to SQL and the Entity Framework will generate entities within the same namespace as the project. But by placing a new "LINQ to SQL Classes" item in a project folder called L2S, the tool will use the folder name as a nested namespace, which helps you differentiate LINQ to SQL entities from your DTO entities. Similarly, placing a new "ADO.NET Entity Data Model" in an L2E project folder causes the Entity Framework entities to be placed in a nested L2E namespace. Alternatively, you could specify the namespace in the property window of the LINQ to SQL and the Entity Framework visual component designers.

Here I am making a choice to generate entities both for LINQ to SQL and the Entity Framework, converting the query results to DTOs. There is also the possibility of dispensing with tool-generated entities altogether and mapping the DTOs using XML files. The Entity Framework already uses XML mapping files, but in the first version of the Entity Framework it is not yet possible to achieve pure persistence ignorance (PI) using plain old C# objects (POCOs). You can get close to this goal in version 1, but entities are required to implement two interfaces, IEntityWithRelationships and IEntityWithChangeTracking, so you're left with IPOCO (POCO + Interfaces). Expect to see improved support for POCOs and persistence ignorance in subsequent versions of the Entity Framework.

LINQ to SQL does have good support for mapping POCOs to a database schema using an XML mapping file. However, projecting from an L2S.Order or L2E.Order to a DTO.Order gives you an added degree of

flexibility. For example, **Figure 4** shows the GetOrder method of SqlDataProvider, which populates the CustomerName property of DTO.Order from the ContactName property of the L2S.Order's Customer property, in effect flattening the relationship between Customer and Order. I use the same approach to set the DTO.OrderDetail's ProductName property from the L2S.Order_Detail's Product property. In each case, I need to configure the LoadOptions property of the DataContext to eager-load LINQ to SQL entities.

Figure 4 GetOrder Method Returns a DTO.Order Using LINQ to SQL Framework

Copy Code

```
namespace DataAccessLayer
{
    using DataAccessLayer.L2S;
    using DTO = DataTransferObjects;

    public class SqlDataProvider : IDataProvider
    {
        public DTO.Order GetOrder(int orderID)
        {
            using (NorthwindDataContext db = new NorthwindDataContext
                (Settings.Default.NorthwindSqlConnection))
            {
                // Set load options
                DataLoadOptions opts = new DataLoadOptions();
                opts.LoadWith<Order>(o => o.Customer);
                opts.LoadWith<Order>(o => o.Order_Details);
                opts.LoadWith<Order_Detail>(od => od.Product);
                db.LoadOptions = opts;

                // Create a DTO.Order from the L2S.Order
                DTO.Order order =
                    (from o in db.Orders
                     where o.OrderID == orderID
                     select new DTO.Order
                     {
                         OrderID = o.OrderID,
                         CustomerID = o.CustomerID,
                         CustomerName = o.Customer.ContactName,
                         OrderDate = o.OrderDate,
                         Updated = o.Updated.ToArray(),
                         OrderDetails =
                            (from od in o.Order_Details
                             select new DTO.OrderDetail
                             {
                                 OrderID = od.OrderID,
                                 ProductID = od.ProductID,
                                 ProductName = od.Product.ProductName,
                                 Quantity = od.Quantity,
                                 UnitPrice = od.UnitPrice,
                                 Updated = od.Updated.ToArray()
                             }).ToList()
                     }).Single();
                return order;
            }
        }
    }
}
```

Now let's turn to the GetOrder method of EntityDataProvider, which uses LINQ to Entities to return a DTO.Order based on an L2E.Order (see **Figure 5**). Notice the Include operator used to eager-load

Order_Details and Order_Details.Product properties. (For more on LINQ operators, see the Data Points column "Standard Query Operators with LINQ.") The Entity Framework does not provide lazy-loading capability out of the box, as does LINQ to SQL, so you are required to indicate explicitly which related entities you wish to include in your query results. After retrieving the specified order, you can simply use it to create a new DTO.Order.

Figure 5 GetOrder Method Returns a DTO.Order Using LINQ to Entities

Copy Code

```
namespace DataAccessLayer
{
    using DataAccessLayer.L2E;
    using DTO = DataTransferObjects;

    public class EntityDataProvider : IDataProvider
    {
        public DTO.Order GetOrder(int orderID)
        {
            using (NorthwindEntities db = new NorthwindEntities
                (Settings.Default.NorthwindEntityConnection))
            {
                // Get the specified L2E.Order
                Order nwOrder =
                    (from o in db.OrderSet
                        .Include("Order_Details")
                        .Include("Order_Details.Product")
                     where o.OrderID == orderID
                     select o).First();

                // Create a DTO.Order from the L2E.Order
                DTO.Order order = new DTO.Order
                {
                    CustomerID = nwOrder.CustomerID,
                    CustomerName = (from c in db.CustomerSet
                                    where c.CustomerID == nwOrder.CustomerID
                                       select c).First().ContactName,
                    OrderID = nwOrder.OrderID,
                    OrderDate = nwOrder.OrderDate,
                    Updated = nwOrder.Updated,
                    OrderDetails =
                        (from od in nwOrder.Order_Details
                         select new DTO.OrderDetail
                         {
                             OrderID = od.OrderID,
                             ProductID = od.ProductID,
                             ProductName = od.Product.ProductName,
                             Quantity = od.Quantity,
                             UnitPrice = od.UnitPrice,
                             Updated = od.Updated
                         }).ToList()
                };
                return order;
            }
        }
    }
}
```

Persisting Individual Objects

LINQ to SQL and the Entity Framework take different approaches to persisting disconnected entities. Let's say, for example, you would like to create a new order that has one or more order details. As shown in **Figure 6**, LINQ to SQL requires at least two lines of code: one to insert the order by calling InsertOnSubmit and another to insert the order's detail items using InsertAllOnSubmit. The Entity Framework, on the other hand, only requires that you add the order to the OrderSet of the ObjectContext, which will automatically add the entire object graph (see **Figure 7**). Both LINQ to SQL and the Entity Framework do the right thing by inserting the parent order first, obtaining the value for OrderID, which is an identity column in the database, then inserting child order details.

Figure 6 Creating an Order Using LINQ to SQL

Copy Code

```
public DTO.Order CreateOrder(DTO.Order order)
{
    // Insert new order
    using (NorthwindDataContext db = new NorthwindDataContext
        (Settings.Default.NorthwindSqlConnection))
    {
        // Create an L2S.Order
        Order nwOrder = new Order
        {
            OrderID = order.OrderID,
            CustomerID = order.CustomerID,
            OrderDate = order.OrderDate,
            Updated = order.Updated
        };

        // Add order details
        nwOrder.Order_Details.AddRange(
            from od in order.OrderDetails
            select new Order_Detail
            {
                OrderID = od.OrderID,
                ProductID = od.ProductID,
                Quantity = od.Quantity,
                UnitPrice = od.UnitPrice,
                Updated = od.Updated
            });

        // Insert order and order details separately
        db.Orders.InsertOnSubmit(nwOrder);
        db.Order_Details.InsertAllOnSubmit(nwOrder.Order_Details);
        db.SubmitChanges();

        // Return the updated order
        return GetOrder(nwOrder.OrderID);
    }
}
```

Figure 7 Creating an Order Using LINQ to Entities

Copy Code

```
public DTO.Order CreateOrder(DTO.Order order)
{
    using (NorthwindEntities db = new NorthwindEntities
        (Settings.Default.NorthwindEntityConnection))
    {
        // Create an L2S.Order
        Order nwOrder = new Order
        {
```

```
            OrderID = order.OrderID,
            CustomerID = order.CustomerID,
            OrderDate = order.OrderDate,
            Updated = order.Updated
        };

        // Add order details
        foreach (DTO.OrderDetail od in order.OrderDetails)
        {
            Order_Detail detail = new Order_Detail
            {
                Order = nwOrder,
                Product = (from p in db.ProductSet
                           where p.ProductID == od.ProductID
                           select p).First(),
                Quantity = od.Quantity,
                UnitPrice = od.UnitPrice,
                Updated = od.Updated
            };
            nwOrder.Order_Details.Add(detail);
        }

        // Add order and order details
        db.AddToOrderSet(nwOrder);
        db.SaveChanges();

        // Return the updated order
        return GetOrder(nwOrder.OrderID);
    }
}
```

Before looking at deleting or updating entities, I need to discuss how to deal with concurrency issues. LINQ to SQL requires you to attach a disconnected entity prior to calling DeleteOnSubmit. Doing so prompts LINQ to SQL to perform optimistic concurrency checks, causing deletes to fail with a ChangeConflictException if a user tries to delete the item after another user has changed it. There really isn't any easy way to turn off this behavior, because LINQ to SQL will throw an InvalidOperationException if you attempt to invoke DeleteOnSubmit without first having called Attach. The Entity Framework, on the other hand, doesn't force you to handle change conflicts when deleting entities (see the code download accompanying this article). **Figure 8** shows the code for updating an order using LINQ to SQL. Notice that the Attach method accepts an asModified parameter of type bool. Setting this parameter to true indicates to LINQ to SQL that you wish to use a timestamp column for concurrency management. Doing so will cause LINQ to SQL to include the current timestamp value in the WHERE clause for the SQL Update statement. If the record was updated by another user, the update will not affect any rows and LINQ to SQL will throw a ChangeConflictException.

Figure 8 Updating an Order Using LINQ to SQL

Copy Code

```
public DTO.Order UpdateOrder(DTO.Order order)
{
    using (NorthwindDataContext db = new NorthwindDataContext
        (Settings.Default.NorthwindSqlConnection))
    {
        // Create an L2S.Order
        Order nwOrder = new Order
        {
            OrderID = order.OrderID,
            CustomerID = order.CustomerID,
            OrderDate = order.OrderDate,
            Updated = order.Updated
```

```
        };

        // Attach and save order
        db.Orders.Attach(nwOrder, true);
        db.SubmitChanges();

        // Return the updated order
        return GetOrder(order.OrderID);
    }
}
```

This strategy relieves the client from needing to preserve original values and submit them along with the update, which would need to be passed to one of the other overloads of the Attach method. It doesn't matter what you name the column, so long as it has a timestamp data type. When you add a table with a timestamp column to the LINQ to SQL designer in Visual Studio, the UpdateCheck attribute will be set to Never for all entity fields and LINQ to SQL will use the timestamp to check for change conflicts.

Updating entities and managing concurrency with the Entity Framework is presently a bit more involved than with LINQ to SQL, because it requires the entity's original values, even if you are using a timestamp column for concurrency (see **Figure 9**). (This is likely to improve in the next version of the Entity Framework.) Because the client has not provided the original order, you need to retrieve the order from the database and set its Updated property to the value provided by the client, which is the only "original" value you really care about. Be sure to commit the change to Updated, either by detaching and re-attaching the order to the object context or by calling AcceptAllChanges. Then call ApplyPropertyChanges to update the original order with values from the updated order. There is one more step required in order for the Entity Framework to use the timestamp column for concurrency checking: you will need to set the ConcurrencyMode for the Updated property on the Order entity from None to Fixed (see **Figure 10**).

Figure 9 Updating an Order Using LINQ to Entities

Copy Code

```
public DTO.Order UpdateOrder(DTO.Order order)
{
    using (NorthwindEntities db = new NorthwindEntities
        (Settings.Default.NorthwindEntityConnection))
    {
        // Get original order using key from updated order
        EntityKey key = db.CreateEntityKey("OrderSet",
            new Order { OrderID = order.OrderID });
        Order origOrder = db.GetObjectByKey(key) as Order;

        // Set Updated to match DTO.Order - (for concurrency)
        origOrder.Updated = order.Updated;

        // Commit change to the Updated property
        db.Detach(origOrder);
        db.Attach(origOrder);

        // Create a NW order for updating the original order
        Order updatedOrder = new Order
        {
            OrderID = order.OrderID,
            OrderDate = order.OrderDate,
            CustomerID = order.CustomerID,
            Updated = order.Updated
        };

        // Apply changes to the original order
        db.ApplyPropertyChanges("OrderSet", updatedOrder);
```

```
        // Persist changes
        db.SaveChanges();

        // Return the updated order
        return GetOrder(order.OrderID);
    }
}
```

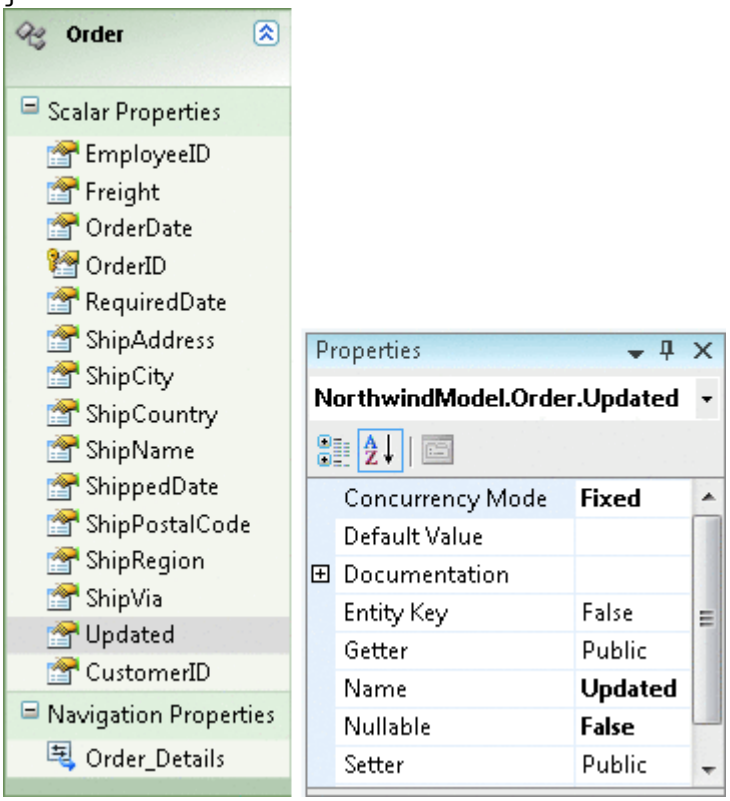

Figure 10 **Concurrency Mode of Updated Set to Fixed**

Tracking Changes across Service Boundaries

If your goal is to persist changes for multiple entities in the same transaction with one call to the service, you need a way to determine the change state of each entity. That is why each DTO in the sample application contains a TrackingState property. Assuming this property has been set by the client, the DAL can then apply the appropriate LINQ to SQL or the Entity Framework API for persisting inserts, updates, and deletes for disconnected entities. This is what allows you to pass an array of DTOs to a service operation for batch updating, teasing out inserted, updated, or deleted order details with a simple LINQ to Objects query. For example, to obtain only order details that have been added to an order, you can query the collection for items with a TrackingState of Created:

Copy Code

```
List<Order_Detail> insertedDetails =
    (from od in order.OrderDetails
     where od.TrackingState == TrackingInfo.Created
     select new Order_Detail
     {
         OrderID = od.OrderID,
         ProductID = od.ProductID,
         Quantity = od.Quantity,
         UnitPrice = od.UnitPrice,
         Updated = od.Updated == null ? new byte[0] : od.Updated
     }).ToList();
```

You need to convert a null value for Updated to an empty byte array because timestamp columns in the database table are ordinarily not nullable. Once you have the list of inserted order details, you can use the convenient LINQ to SQL API for adding new entities by passing a collection of order details:

Copy Code

```
db.Order_Details.InsertAllOnSubmit(insertedDetails);
```

As you might guess, there are matching AttachAllOnSubmit and DeleteAllOnSubmit methods. When you call SubmitChanges on the DataContext, LINQ to SQL will make sure to commit these changes in the correct order (parent-children for inserts, children-parent for deletes) and wrap everything in a transaction. You can take the same approach to persisting order details using LINQ to Entities. See the download for the full code listing of UpdateOrder for both LINQ to SQL and the Entity Framework.

It should be obvious by now that the DAL is relying on the client to track change state for each DTO. The most convenient mechanism for tracking changes to objects on the client is a generic collection. In order for the collection to update the TrackingState property, it should constrain the type argument to implement an interface. (You could also use Reflection to get and set the TrackingState property, but a generic type constraint will provide both type safety and better performance.) Enter the ITrackable interface, which has just one member, the TrackingState property:

Copy Code

```
public interface ITrackable
{
    TrackingInfo TrackingState { get; set; }
}
```

As I mentioned at the beginning, there are two types of the change tracking collection, one for use with Windows Forms that extends BindingList<T> and another for WPF applications that extends ObservableCollection<T>. But rather than duplicate change-tracking logic in both these collections, I have centralized it in a class called ChangeTrackingHelper<T>, which derives from Collection<T> and constrains T to implement ITrackable (see **Figure 11**). It also constrains T to implement INotifyPropertyChanged so that it can handle each item's PropertyChanged event and set the TrackingState property to Updated.

Figure 11 Change Tracking Helper Class

Copy Code

```
internal class ChangeTrackingHelper<T> : Collection<T>
    where T : ITrackable, INotifyPropertyChanged
{
    Collection<T> deletedItems = new Collection<T>();

    // Listen for changes to each item
    private bool tracking;
    public bool Tracking
    {
        get {  return tracking; }
        set
        {
            foreach (T item in this)
            {
                if (value) item.PropertyChanged += OnItemChanged;
                else item.PropertyChanged -= OnItemChanged;
            }
            tracking = value;
        }
    }

    void OnItemChanged(object sender, PropertyChangedEventArgs e)
    {
        if (!Tracking) return;
        ITrackable item = (ITrackable)sender;
        if (e.PropertyName != "TrackingState")
        {
```

```
            // Mark item as updated
            if (item.TrackingState == TrackingInfo.Unchanged)
                item.TrackingState = TrackingInfo.Updated;
        }
    }

    // Mark item as created and listen for property changes
    protected override void InsertItem(int index, T item)
    {
        if (!Tracking) return;
        item.TrackingState = TrackingInfo.Created;
        item.PropertyChanged += OnItemChanged;
        base.InsertItem(index, item);
    }

    // Mark item as deleted and cache it
    protected override void RemoveItem(int index)
    {
        if (!Tracking) return;
        T item = this.Items[index];
        if (item.TrackingState != TrackingInfo.Created)
        {
            item.TrackingState = TrackingInfo.Deleted;
            item.PropertyChanged -= OnItemChanged;
            deletedItems.Add(item);
        }
        base.RemoveItem(index);
    }

    public ChangeTrackingHelper<T> GetChanges()
    {
        // Create a collection with changed items
        ChangeTrackingHelper<T> changes = new ChangeTrackingHelper<T>();
        foreach (T existing in this)
        {
            if (existing.TrackingState != TrackingInfo.Unchanged)
                changes.Add(existing);
        }

        // Append deleted items
        foreach (T deleted in deletedItems)
            changes.Add(deleted);
        return changes;
    }
}
```

When an item is added to the collection, its TrackingState is set to Created. When an item is removed, its TrackingState is set to Deleted and it is cached in a collection of deleted items. The user indicates that tracking should begin on a collection by setting the Tracking property to true, at which point the collection will subscribe to each item's PropertyChanged event. The GetChanges method creates a new ChangeTrackingHelper collection, comprised only of items marked as Created, Updated, or Deleted.

Configuring the Client

Now that we have a set of generic change tracking collections, it's time to use them! I've placed these in a separate class library, so all you have to do is reference it from the client application. Nevertheless, you need to take some additional steps so that the collection can use classes generated by svcutil.exe when you add a service reference to the client project. Classes such as Customer, Product, Order, and OrderDetails will

already possess the TrackingState property, but they will not implement the ITrackable interface required by ChangeTrackingHelper<T>.
Furthermore, the TrackingInfo enum needs to be placed in a namespace where the change tracking collection can find it, instead of within the nested namespace where svcutil.exe will place it when you add a service reference. Lastly, it would be nice if arrays of objects returned by service operations would appear as change tracking collections in your client application. This would set the data type for Order.OrderDetails to ChangeTrackingCollection<T> or ChangeTrackingList<T> instead of an array of OrderDetail, which would save you some extra work when you're sending changed order details to the service.
Here are the steps you should follow to set up a Windows Forms or WPF client application to use a change tracking collection to track entity changes and send them to the service. Be sure to follow these steps in the order given:

1. From a Windows Forms or WPF client application, set a reference to the ClientChangeTracker class library project or assembly, where the ITrackable interface and change tracking collections are located.
2. With the WCF service application running, add a service reference pointing to the metadata endpoint or Web Services Description Language (WSDL) URL. It is important that you reference the ClientChangeTracker assembly prior to adding the service reference. By default, a service reference will reuse types in all referenced assemblies, and because ClientChangeTracker contains the TrackingInfo enum with DataContract and EnumMember serialization attributes, svcutil.exe will reuse it instead of copying it into the Reference.cs code file created by the service reference.
3. After setting the service reference, add partial classes for each DTO class created by the service reference. For example, if your service operations expose Customer, Product, Order, and OrderDetail objects, these will be included in the service reference. If you show all files for the project and expand the service reference and Reference.svcmap node, you'll see these classes located in Reference.cs. Make note of the namespace in which they are located (which should match what you specified when adding the service reference). Now add a class file to the project and change the namespace to reflect the namespace in Reference.cs. Within this namespace, add public partial classes for each DTO class and implement the ITrackable interface:

   Copy Code

   ```
   namespace WpfClient.OrderService
   {
       public partial class Customer : ITrackable { }

       public partial class Product : ITrackable { }

       public partial class Order : ITrackable { }

       public partial class OrderDetail : ITrackable { }
   }
   ```

   Note the absence of the TrackingState property. You won't have to insert it here, because the code-generated classes already contain this property—it is part of the data contract for each DTO. You may also insert additional code in these classes to make it easier to use them in your application, for example, constructors to initialize the class (which are called when you create the class using new, but not when WCF creates them). See the code download for an example.

4. Lastly, you will probably want to configure the service reference to use ChangeTrackingCollection<T> or ChangeTrackingList<T> as the collection type instead of System.Array. This means that when a service operation returns an Order array, it will materialize on the client as ChangeTrackingCollection<Order> or ChangeTrackingList<Order>. The OrderDetails property of each Order will likewise show up as a change tracking collection type. How cool is that? To make this magic happen, you'll need to open the Reference.svcmap file (show all project files and open the service reference to see it) and dive into the XML. For a WPF client, configure the CollectionMappings element as follows:

Copy Code
```
<CollectionMappings>
    <CollectionMapping TypeName="ChangeTracker.ChangeTrackingCollection'1"
Category="List" />
</CollectionMappings>
```
For a Windows Forms client, you should replace ChangeTracker.ChangeTrackingCollection`1 with Change-Tracker.ChangeTrackingList`1.

1. Now, with the service application still running, right-click the service reference and select "Update Service Reference" and recompile the client application. If all goes well, the project will build successfully and you'll be on your way.

After performing these steps, all that's left is to write code for the client application to retrieve some DTOs, bind them to the UI where the user can update them, and send them back to the service, which will persist them to the database. Retrieving objects from the service is simply a matter of creating an instance of the service proxy and invoking operations to get Customers, Products, and Orders. For example, if the service has a GetOrder method that accepts an int for orderId, then your client code might look like this:

Copy Code
```
using (OrderServiceClient proxy = new OrderServiceClient())
{
   CurrentOrder = proxy.GetOrder(orderId);
}
```
If you've set up data binding properly, the order should appear in the user interface, together with the order details. **Figure 12** shows what the WPF client looks like for the sample application. The "Get Order" button allows the user to retrieve an existing order, choosing from a list of customer orders. The "New Order" button simply creates a new order, and the "Delete Order" button calls DeleteOrder on the service proxy, passing the current order.

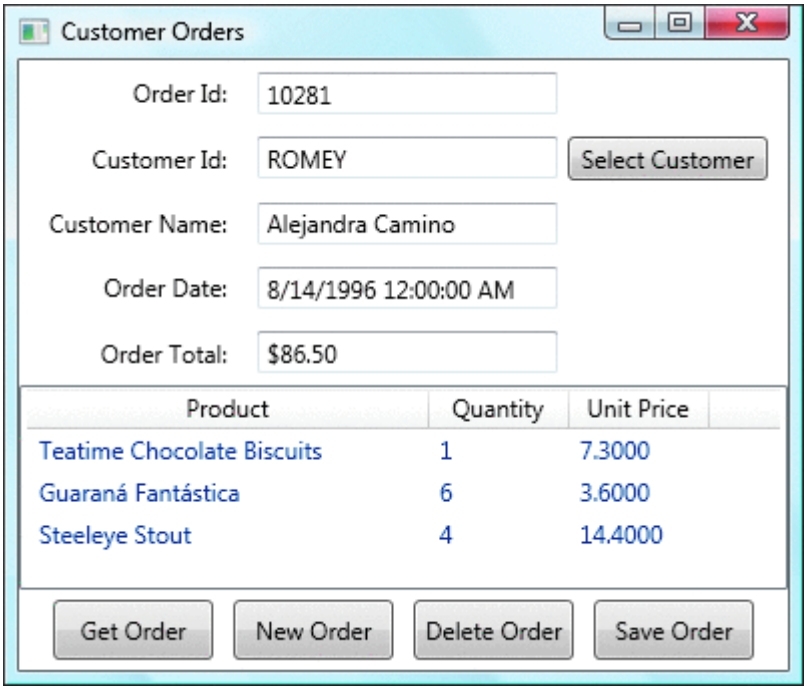

Figure 12 **Sample WPF Client App**

The "Save Order" button will call the service CreateOrder operation to persist a newly created order, or it will call the UpdateOrder operation to save changes to an existing order. Both these operations return an order object, which will contain the current Updated property (for concurrency management), and a new OrderId property for inserted orders (the Order table in Northwind uses an identity column). After obtaining the updated order, you will start change tracking and set data binding to use the order.

However, before saving an existing order, you have a little more work to do. We only want to pass order details that have been created, updated, or deleted and not waste network bandwidth by sending details that are unchanged. ChangeTrackingCollection<T> and ChangeTrackingList<T> make this relatively easy by exposing a GetChanges method, which will return only order details that have been inserted, modified or removed. The code to invoke UpdateOrder should look like **Figure 13**.

Figure 13 Invoking UpdateOrder

Copy Code

```
Order changedOrder = new Order
{
    OrderID = CurrentOrder.OrderID,
    CustomerID = CurrentOrder.CustomerID,
    OrderDate = CurrentOrder.OrderDate,
    Updated = CurrentOrder.Updated,
    TrackingState = CurrentOrder.TrackingState,
    OrderDetails = CurrentOrder.OrderDetails.GetChanges()
};

using (OrderServiceClient proxy = new OrderServiceClient())
{
    CurrentOrder = proxy.UpdateOrder(changedOrder);
}
```

Because I've configured the service reference to use ChangeTrackingCollection<T> as the collection type, the OrderDetails property of the Order class will be of type ChangeTrackingCollection<OrderDetail>, and I can call GetChanges on it directly to get just the order details that were added, modified or removed, passing those on to the UpdateOrder operation. And because ChangeTrackingCollection<T> extends ObservableCollection<T>, it supports data binding features conducive to WPF applications (for example, implementing INotifyCollectionChanged so that the UI can update an items control when items are added or

removed from the collection). Similarly, because ChangeTrackingList<T> extends BindingList<T>, it serves as a good data source for a Windows forms DataGridView, which leverages interfaces implemented by BindingList<T> to enhance the user experience.

Moving Forward

You now have a flexible service-oriented application architecture where the client is completely ignorant of how objects will be persisted. As tempting as it might be to pass tool-generated entities across service boundaries, we have seen that DTOs allow more control over the structure and shape of your object model. In addition, building a data access layer will decouple the application from whatever persistence technology you may be using, allowing you to replace a data access implementation without affecting other parts of the application.

LINQ to SQL and the ADO.NET Entity Framework represent a quantum leap forward in the ORM space and go a long way toward reducing the impedance mismatch between objects and relational data. They support various scenarios, including traditional client-server and service-oriented architectures. However, updating the database using DTOs requires more work on the part of the developer, especially if you use a timestamp value to detect conflicts among concurrent users. Nevertheless, incorporating change state into the data contract between client and service allows you to attach entities that have been disconnected from their original context. Lastly, we have seen how a generic change tracking collection can do the heavy lifting of managing change state on the client, allowing you to process batch updates with a single round trip to the service.

For more on the Entity Framework, see "ADO.NET: Achieve Flexible Data Modeling with the Entity Framework" by Elisa Flasko.

Anthony Sneed is an instructor for the developer training company DevelopMentor, where he authors and teaches courses on the .NET Framework 3.5, LINQ, and the Entity Framework. In his spare time he enjoys cooking his world-famous chili and producing family videos. You can reach him at tony@tonysneed.com or visit his blog at blog.tonysneed.com