# Anti-Patterns To Avoid In N-Tier Applications

As a member of the Entity Framework team, I frequently talk to customers about building applications that use the Entity Framework. Probably the topic I get asked about more than anything else is designing n-tier applications. In this article, I will try to set a foundation on which you can build for success in this part of your applications. The majority of the article is devoted to design anti-patterns for n-tier, which are usually the most important issues that I find. This is a topic where there are a lot of options and many issues to consider, so it is important to understand the overall space before making decisions for your particular application. In future articles, I will examine n-tier patterns for success and some of the key APIs and issues specific to the Entity Framework, and provide a sneak peak at features coming in the Microsoft .NET Framework 4 that should make n-tier significantly easier.

Understanding N-tier

Before I dive into the anti-patterns, it is important to have a common understanding of n-tier.

The first point to be clear on is the difference between tiers and layers. A well-designed application will have multiple layers with carefully managed dependencies. Those layers could live in a single tier or be split across multiple tiers. A layer is just an organizational concept in an application, while a tier denotes physical separation or at least a design that will allow physical separation if needed.

Any application that talks to a database has more than one tier unless that database runs in-process, but the application is not called n-tier unless it involves more tiers than just the database and application. Similarly, every ASP.NET application that involves a database is technically n-tier because there is the database, the Web server, and the browser. Unless you introduce Windows Communication Foundation (WCF) or Web services, you would not call that application n-tier since for most purposes the Web server and browser can be thought of as a single client tier. N-tier applications are those that have at a minimum a database tier, a middle tier that exposes a service, and a client tier.

While it does not sound like that big a deal on the surface, it turns out that implementing applications split across multiple tiers is difficult. There are a lot more pitfalls than you think. These pitfalls led Martin Fowler, in his book Patterns of Enterprise Application Architecture (Addison-Wesley, 2002), to make a very strong statement on the subject:

Don't distribute your objects!

Martin calls it the First Law of Distributed Objects.

As with every design rule, though, there are times when the law must be set aside. Your application may have a scalability problem that requires multiple tiers so you can apply more computing resources. Maybe you need to exchange data with a business partner or customer application. It may just be that you have security or infrastructure constraints that divide your application onto multiple computers or prevent one part of your application from talking directly to another part. When you need multiple tiers, you really need them.

While the anti-patterns presented in this article can be applied to a wide range of applications and technologies, the main focus will be creating and consuming custom WCF services that persist data using the Entity Framework.

Not surprisingly, many n-tier anti-patterns are a result of losing focus on the goal of your application. If you do not keep in mind what motivated you to use an n-tier architecture in the first place, or if you neglect critical persistence concerns, then it is all too easy to get in trouble. The following sections will look at some common problems.

Custom Service or RESTful Service?

REST, or Representational State Transfer, is a type of Web service that is rapidly gaining in popularity. So you might ask yourself what the difference is between RESTful services and custom Web services, and why you might choose one type over the other. The key difference between the two types is that REST services are resource-centric while custom services are operation-centric. With REST, you divide your data into resources, give each resource a URL, and implement standard operations on those resources that allow creation, retrieval, update, and deletion (CRUD). With custom services, you can implement any arbitrary method, which means that the focus is on the operations rather than the resources, and those operations can be tailored to the specific needs of your application.

Some services fit very naturally into the REST model—usually when the resources are obvious and much of the service involves management of those resources. Exchange Server, for instance, has a REST API for organizing e-mail and calendar items. Similarly, there are photo-sharing Web sites on the Internet that expose REST APIs. In other cases, the services less clearly match REST operations, but can still be made to fit. Sending e-mail, for example, can be accomplished by adding a resource to an outbox folder. This is not the way you would most naturally think about sending e-mail, but it is not too much of a stretch.

In other cases, though, REST operations just do not fit well. Creating a resource in order to initiate a workflow that drives monthly payroll check printing, for example, would be much less natural than having a specific method for that purpose.

If you can fit your service into the constraints of REST, doing so will buy you a lot of advantages. ADO.NET Data Services in combination with the Entity Framework makes it easy to create both RESTful services and clients to work with them. The framework can provide more functionality to RESTful services automatically because the services are constrained to follow a specific pattern. In addition, RESTful services have a very broad reach because they are so simple and interoperable. They work especially well when you do not know in advance who the clients might be. Finally, REST can be made to scale to handle very large volumes of operations.

For many applications, the constraints of REST are just too much. Sometimes the domain does not divide clearly into a single pattern of resources or the operations involve multiple resources at once. Sometimes the user actions and business logic around them do not map well to RESTful operations, or more precise control is required than can fit into those operations. In these cases, custom services are the way to go.

You can always build an application that has a mix of REST and custom services. Often the ideal solution for an application is a mixture of both.

Anti-Pattern #1: Tight Coupling

Chances are you have heard about the evils of tight coupling. So you always strive to keep your components as loosely coupled as possible, right? Yeah, right.

Loose coupling is more difficult than tight coupling, and often the performance is not as good. You start off with the best of intentions, but end up asking if the benefit is worth the cost. Why introduce an interface and dependency injection when you could just create an instance of the class and call the method directly? Why build an abstraction with custom objects mapped to the database instead of filling a DataTable and passing it around?

To make matters worse, you do not usually feel the pain of tight coupling until much later. In the short term, you gain some efficiency and get the job done, but in the long run evolving the application can become almost impossible.

If you have been building software for any time at all, you probably understand coupling tradeoffs fairly well when it comes to modules within a tier. When you have modules that work together closely, sometimes tight coupling is the right choice, but in other cases, components need to be kept at arm's length from one another so that you can contain the ripple effect of changes to the application.

When it comes to splitting parts of your application into separate tiers, the significance of coupling becomes much greater. The reason for this is simple. Tiers do not always change at the same rate. If you have a service that is consumed by many clients and you cannot guarantee all the clients will upgrade on demand, then you better make sure you can change that service without having to change the clients. If not, you will encounter a problem that is sometimes called shearing rates of change. Imagine your application being pulled in two different directions until it is forcefully ripped apart.

The trick is to identify which parts of the application might have different rates of change and which parts are tightly coupled to each other. First, consider the boundary between the database and the mid-tier. As your application grows, there is a good chance you will need to adjust the database to improve performance, and if you ever share the database between multiple applications, there is a very good chance you will want to evolve one application's mid-tier without changing the other one. Fortunately, using the Entity Framework already helps here because its mapping system provides an abstraction between your mid-tier code and the database. The same questions should be considered between the mid-tier and the client.

A particularly common and painful example of this anti-pattern in action is an architecture that uses table adapters to retrieve data from the database and Web services that exchange DataSets with the client. The table adapter moves the data into a DataSet with the same schema (thus tightly coupling the database to the mid-tier) and then the Web service exchanges that same DataSet with the client (thus tightly coupling the mid-tier to the client). This kind of system is easy to create—there are Visual Studio tools that lead you through the process nicely. But if you build a system that way, changes to any part of the system are likely to ripple to all other parts.

Anti-Pattern #2: Assuming Static Requirements

Speaking of changes to the system, sometimes you design around an assumption that requirements will remain static, but there are two cases where changing requirements have an especially significant impact. One comes from treating the client as trusted, and the other occurs when the mid-tier service assumes the client will be implemented using a particular technology.

While it is unlikely that trust boundaries will change unexpectedly, when it comes to data integrity, security, and trust, the consequences of getting it wrong are just too great. If you perform validation only on the client, for instance, and on the mid-tier trust that the data you receive is OK to send directly to the database without revalidating, the chance that something will eventually go wrong is much larger than you might think. Even knowing the service only runs within your intranet is not enough to keep your information safe. Someone might create another client using the same service or modify the first client to call the service from a different code path that skips validation. Who knows what might happen.

Further, once you have a service, it is more likely that regular code to be used in ways that you did not anticipate than—so much so that the generally accepted wisdom is that you should always validate and enforce some degree of security on the mid-tier even though that may mean validating or performing access control more than once.

The second issue, locking the client into a particular technology, is even more likely to be a problem. Technologies always change. If an application survives long enough, something will happen that forces technology adjustments, and clients are especially susceptible. You may initially design your application as a rich client desktop application and then later find you need to move it to a mobile phone or Silverlight. If that were the case, and you designed your service to exchange DataSets, then major surgery would be needed for the service and all existing clients.

Anti-Pattern #3: Mishandled Concurrency

While there is a tight coupling downside to exchanging DataSets, concurrency is a complex-but-important area that the DataSet handles well. Unfortunately many developers do not understand the nuances of managing concurrency, and to make things worse, a mistake with concurrency is the kind of problem that often only shows up once the application is in production. If you are lucky, it will manifest as an obvious failure. If not, it may cause corruption to your data over a long period of time without being detected.

At its core, concurrency management is fairly simple: guarantee data integrity even if two clients try to modify the same data at roughly the same time. Particularly attentive readers will note that these problems also come up in cases that are unrelated to n-tier, but concurrency issues are particularly relevant to the Entity Framework n-tier designs, because the Entity Framework's handling of n-tier scenarios creates unique concurrency challenges.

For most applications, the concurrency management technique of choice is optimistic concurrency. Even though many clients may access the database simultaneously, the number of times when the exact same entity is modified in conflicting ways is quite small. So you assume everything will work out, but take steps to detect if something goes wrong.

Detection is driven by one or more properties, collectively called the concurrency token, that change whenever any part of the entity changes. When the application reads an entity, it saves the value of the concurrency token. Later, when it wants to write that entity back to the database, it first checks to make sure that the value of the concurrency token in the database is the same now as it was when the entity was originally read. If it is, the update proceeds. If not, the update halts and throws an exception.

The Entity Framework supports optimistic concurrency by transparently tracking the original value of concurrency tokens when entities are queried and checking for conflicts prior to database updates. The problem with n-tier applications is that this process works transparently only as long as a single ObjectContext instance is used to track the entity from the time it is queried until the time SaveChanges is called. If you serialize entities from one tier to another, the recommended pattern is to keep the context around on the mid-tier only long enough for a single service method call. Subsequent calls will spin up a new instance of the context to complete each task. (Creating a new context instance for every service operation is an important recommendation in its own right, by the way. For more information, see Anti-Pattern #4: Stateful Services.)

Once developers begin to learn how the Entity Framework APIs work for this kind of disconnected operation—disconnected in the sense that the entities are disconnected from the context after the query, sent to another tier, and then re-connected when it is time to save—there is a tendency to fall into a nasty pattern:

1. Query the entity and serialize it to the client. At this point, the concurrency token's current value is the same as the original value, and that is the only value sent to the client.
2. The client receives the entity, makes changes, and sends a modified version of the entity back to the mid-tier.
3. Since neither the client nor the service explicitly kept track of the concurrency token or what properties have been modified, the service queries the database to get the current state of the entity into a newly created context, then compares values between the current entity from the database and the one sent back from the client.
4. The service calls SaveChanges, which performs optimistic concurrency checks while persisting the changes.

Did you see the problem? Actually there are two problems.

First, every time an entity is updated, it has to be read from the database twice—once when it is first queried and a second time right before the update—which creates a significant extra load on the system.

Second, and more importantly, the "original value" used by the Entity Framework to check if the entity has been modified in the database comes from the second query instead of the first one. That is, it comes from the query that happens right before the update. So the result is that the optimistic concurrency check made by the Entity Framework will almost never fail. If someone else modifies the entity between the first query and the second one, the system will not detect the conflict because the value used for the concurrency check is from after the other client's modification instead of before it.

There is still a small window (between the second query and the update) when the optimistic concurrency check could detect a problem, so you still have to write your program to handle the exception, but you will not really have protected your data from corruption.

The correct pattern is either to make a copy of the entity on the client and send back both the original version unmodified and the modified version or to write the client in such a way that it does not modify the concurrency token. If the concurrency token is updated by a server trigger or automatically because it is a row version number (probably the best plan anyway), then there is no reason to modify it on the client. The current value of the property can be left untouched and used as storage for the original value.

This is a reasonably sound approach because if a bug in the client causes the value to accidentally be modified, it is highly unlikely that you will get a false success. That bug might cause you to get a false failure, but that is much more acceptable than false success.

To make this approach work, when the mid-tier receives the entity from client, you need to attach it to the context and then go over its properties, manually marking them as modified. In either case, though, you will fix both of the problems with the anti-pattern at once. You will no longer query the database twice, and the concurrency check will be based on the correct value of the token (from the initial query) rather than some later value.

Anti-Pattern #4: Stateful Services

Given the comparative ease of developing client-server solutions, the next anti-pattern comes up when developers try to simplify things by keeping the context around across multiple service calls. This seems nice at first because it sidesteps the concurrency problems. If you keep the context alive on the mid-tier, then it will contain the correct original entity values. When you receive an entity back from the client, you can compare the updated entity with the version of the entity in the context and apply changes as appropriate. When you save the entity, the correct concurrency check will be made and no extra database query is required.

While this approach seems easy on the surface, there are a number of problems lurking. Managing the context lifetime can get tricky quickly. When you have multiple clients calling the services, you have to maintain a separate context for each client or risk collisions between them. And even if you solve those issues, you will end up with major scalability problems.

These scalability problems are not only the result of tying up server resources for every client. In addition you will have to guard against the possibility that a client might start a unit of work, but never complete it, by creating an expiration scheme. Further, if you decide that you need to scale your solution out by introducing a farm with multiple mid-tier server, then you will have to maintain session affinity to keep a client associated with the same server where the unit of work began.

A lot of effort and specialized technology has been expended on addressing these issues when, in fact, the best solution is to avoid them altogether by keeping your mid-tier service implementations stateless. Each time a service call is made, the mid-tier should spin up the necessary resources, handle the call, and then release all resources specific to that call. If some information needs to be maintained for a unit of work that extends across multiple service calls, then that information should be maintained by the client rather than the mid-tier so there is no session affinity, no need to expire unfinished units of work, and no server resources in use for a particular client in between service calls.

Anti-Pattern #5: Two Tiers Pretending to be Three

Another anti-pattern I encounter fairly often is also an attempt to simplify this process. Usually it shows up as a request something like, "Why can't you make the Entity Framework serialize queries across tiers?" followed almost immediately by, "Oh, and while you are at it, can you support initiating updates from another tier as well?"

These are probably features Microsoft could add to the Entity Framework, but if you stop and think about them for a minute in light of the other issues I have discussed, you would have to question whether this is a good idea.

If you could create an Entity Framework ObjectContext on the client tier, execute any Entity Framework query to load entities into that context, modify those entities, and then have SaveChanges push an update from the client through the mid-tier to the database server—if you could do all that, then why have the mid-tier at all? Why not just expose the database directly?

Remember Fowler's First Law of Distributed Objects. Keep in mind that the only time this kind of architecture makes sense is when you really, really need it. If you really need it, then you need better security or the ability to scale out with multiple servers, or some other thing that I suggest will not really be solved by introducing a mid-tier that is simply a thin proxy for the database. You might use this technique to subvert a restriction placed on you by a corporate policy, but it is hardly capturing the spirit of an n-tier application. My suggestion is to either invest in building an n-tier application to meet a particular need or, if you can get away with it, avoid n-tier altogether.

Anti-Pattern #6: Undervaluing Simplicity

This brings me to the last n-tier anti-pattern. In the name of avoiding all the anti-patterns discussed previously, it is easy to decide that you need to create the most carefully architected, multi-tier, fully separated, re-validating, super design that you can come up with. Then you can spend all your time building infrastructure and none of your time actually delivering value to your users.

It is important to think over your goals and consider whether you are going to need the investment n-tier requires. Simple is good. Sometimes a two-tier app is just the thing. Sometimes you need more tiers than that, but everything is under your control and trusted or you have an AJAX, Silverlight, or click-once client that auto-deploys so that you do not have to worry about shearing rates of change.

If you can make the problem simpler, do so. Put in all the effort for the full solution if you must, but by the same token make sure you put in enough effort to do the job in a way that meets your goals.

# N-Tier Application Patterns

In my previous article, I described a foundation on which you can build successful n-tier applications, focusing mainly on anti-patterns to avoid. There are many issues to consider before making decisions about the design of an n-tier application. In this article, I examine n-tier patterns for success and some of the key APIs and issues specific to the Entity Framework. I also provide a sneak peak at features coming in the Microsoft .NET Framework 4 that should make n-tier development significantly easier.

Change Set

The idea behind the change set pattern is to create a serializable container that can keep the data needed for a unit of work together and, ideally, perform change tracking automatically on the client. The container glues together the parts of the unit of work in a custom format, so this approach also tends to be quite full-featured and is easy to use on the mid-tier and on the client.

DataSet is the most common example of this pattern, but other examples exist, such as the EntityBag sample I wrote some time ago as an exploration of this technique with the Entity Framework. Both examples exhibit some of the downsides of this pattern. First, the change set pattern places significant constraints on the client because the wire format tends to be very specific to the change set and hard to make interoperable. In practice, the client must use .NET with the same change set implementation used on the mid-tier. Second, the wire format is usually quite inefficient. Among other things, change sets are designed to handle arbitrary schemas, so overhead is required to track the instance schema. Another issue with change set implementations such as DataSet, but not necessarily endemic to the pattern, is the ease with which you can end up tightly coupling two or more of the tiers, which causes problems if you have different rates of change. Finally, and probably of most concern, is how easy it is to abuse the change set.

In some ways, this pattern automates and submerges critical concerns that should be at the forefront of your mind when designing your solution. Precisely because it is so easy to put data into the change set, send it to the mid-tier, and then persist, you can do so without verifying on the mid-tier that the changes you are persisting are only of the type that you expect. Imagine that you have a service intended to add an expense report to your accounting system that ends up also modifying someone's salary.

The change set pattern is best used in cases where you have full control over client deployment so that you can address the coupling and technology requirement issues. It is also the right choice if you want to optimize for developer efficiency rather than runtime efficiency. If you do adopt this pattern, be sure to exercise the discipline to validate any changes on the mid-tier rather than blindly persisting whatever changes arrive.

DTOs

At the opposite end of the spectrum from change sets are Data Transfer Objects, or DTOs. The intent of this pattern is to separate the client and the mid-tier by using different types to hold the data on the mid-tier and the data on the client and in the messages sent between them.

The DTO approach requires the most effort to implement, but when implemented correctly, it can achieve the most architectural benefits. You can develop and evolve your mid-tier and your client on completely separate schedules because you can keep the data that travels between the two tiers in a stable format regardless of changes made on either end. Naturally, at times you'll need to add some functionality to both ends, but you can manage the rollout of that functionality by building versioning plus backward and forward compatibility into the code that maps the data to and from the transfer objects. Because you explicitly design the format of the data for when it transfers between the tiers, you can use an approach that interoperates nicely with clients that use technologies other than .NET. If necessary, you can use a format that is very efficient to send across the wire, or you can choose, for instance, to exchange only a subset of an entity's data for security reasons.

The downside to implementing DTOs is the extra effort required to design three different sets of types for essentially the same data and to map the information between the types. You can consider a variety of shortcuts, however, like using DTOs as the types on the client so that you have to design only two types instead of three; using LINQ to Objects to reduce the code that must be written to move data between the types; or using an automatic mapping library, which can further reduce the code for copying data by detecting patterns such as properties with the same name on more than one type. But there is no way around the fact that this pattern involves more effort than any of the other options—at least for initial implementation.

This is the pattern to consider when your solution becomes very large with very sophisticated requirements for interoperability, long-term maintenance, and the like. The longer the life of a project, the more likely that DTOs will pay off. For many projects, however, you might be able to achieve your goals with a pattern that requires less effort.

Simple Entities

Like the change set pattern, the simple entities pattern reuses the mid-tier entity types on the client, but unlike change sets, which wrap those entities in a complex data structure for communication between tiers, simple entities strives to keep the complexity of the data structure to a minimum and passes entity instances directly to service methods. The simple entities pattern allows only simple property modification to entity instances on the client. If more complex operations are required, such as changing the relationships between entities or accomplishing a combination of inserts, updates, and deletes, those operations should be represented in the structure of the service methods.

The beauty of the simple entities approach is that no extra types are required and no effort has to be put into mapping data from one type to another. If you can control deployment of the client, you can reuse the same entity structures (either the same assemblies or proxies), and even if you have to work with a client technology other than .NET, the data structures are simple and therefore easy to make interoperable. The client implementation is typically straightforward because minimal tracking is required. When properties must be modified, the client can change them directly on an entity instance. When operations involving multiple entities or relationships are required, special service methods do the work.

The primary disadvantage of this pattern is that more methods are usually required on the service if you need to accomplish complex scenarios that touch multiple entities. This leads to either chatty network traffic, where the client has to make many service calls to accomplish a scenario or special-purpose service methods with many arguments.

The simple entities approach is especially effective when you have relatively simple clients or when the scenarios are such that operations are homogenous. Consider, for example, the implementation of an e-commerce system in which the vast majority of operations involve creating new orders. You can design your application-interaction patterns so that modifications to information like customer data are performed in separate operations from creating new orders. Then the service methods you need are generally either queries for read-only data, modifications to one entity at a time without changing much in the way of relationships, or inserting a set of related entities all at once for a new order. The simple entities pattern works fairly well with this kind of scenario. When the overall complexity of a solution goes up, when your client becomes more sophisticated, or when network performance is so critical that you need to carefully tune your wire format, other patterns are more appropriate.

Self-Tracking Entities

The self-tracking entities pattern is designed to build on the simple entities pattern and achieve a good balance between the various concerns to create a single pattern that works in many scenarios. The idea is to create smart entity objects that keep track of their own changes and changes to related entities. To reduce constraints on the client, these entities are plain-old CLR objects (POCO) that are not tied to any particular persistence technology—they just represent the entities and some information about whether they are unchanged, modified, new, or marked for deletion.

Because the entities are self-tracking, they have many of the ease-of-use characteristics of a change set, but because the tracking information is built into the entities themselves and is specific to their schema, the wire format can be more efficient than with a change set. In addition, because they are POCO, they make few demands on the client and interoperate well. Finally, because validation logic can be built into the entities themselves, you can more easily remain disciplined about enforcing the intended operations for a particular service method.

There are two primary disadvantages for self-tracking entities compared to change sets. First, a change set can be implemented in a way that allows multiple change sets to be merged if the client needs to call more than one service method to retrieve the data it needs. While such an implementation can be accomplished with self-tracking entities, it is harder than with a change set. Second, the entity definitions themselves are complicated somewhat because they include the tracking information directly instead of keeping that information in a separate structure outside the entities. Often this information can be kept to a minimum, however, so it usually does not have much effect on the usability or maintainability of the entities.

Naturally, self-tracking entities are not as thoroughly decoupled as DTOs, and there are times when more efficient wire formats can be created with DTOs than with self-tracking entities. Nothing prevents you from using a mix of DTOs and self-tracking entities, and, in fact, as long as the structure of the tracking information is kept as simple as possible, it is not difficult to evolve self-tracking entities into DTOs at some later date if that becomes necessary.

Implementing N-Tier with the Entity Framework

Having reviewed your options and decided that you need an n-tier application, you can select a pattern and a client technology knowing what pitfalls to avoid. Now you're ready to get rolling. But where does the Entity Framework (EF) fit into all this?

The EF provides a foundation for addressing persistence concerns. This foundation includes a declarative mapping between the database and your conceptual entities, which decouples your mid-tier from the database structure; automatic concurrency checks on updates as long as appropriate change-tracking information is supplied; and transparent change tracking on the mid-tier. In addition, the EF is a LINQ provider, which means that it is relatively easy to create sophisticated queries that can help with mapping entities to DTOs.

The EF can be used to implement any of the four patterns described earlier, but various limitations in the first release of the framework (shipped as part of Visual Studio 2008 SP1/.NET 3.5 SP1) make patterns other than the simple entities pattern very difficult to implement. In the upcoming release of the EF in Visual Studio 2010/.NET 4, a number of features have been added to make implementing the other patterns easier. Before we look at the future release, though, let's look at what you can do with the EF now by using the simple entities pattern.

Concurrency Tokens

The first step you need to take before looking at any aspects of n-tier development is to create your model and make sure that you have concurrency tokens. You can read about the basics of building a model elsewhere. There are some great tutorials, for instance, available in the [Entity Framework section](http://msdn.microsoft.com/data/) of the MSDN Data Platform Developer Center.

The most important point for this discussion, however, is to make sure that you have specified concurrency tokens for each entity. The best option is to use a row version number or an equivalent concept. A row's version automatically changes whenever any part of the row changes in the database. If you cannot use a row version, the next best option is to use something like a time stamp and add a trigger to the database so that the time stamp is updated whenever a row is modified. You can also perform this sort of operation on the client, but that is prone to causing subtle data corruption problems because multiple clients could inadvertently come up with the same new value for the concurrency token. Once you have an appropriate property configured in the database, open the Entity Designer with your model, select the property, and set its Concurrency Mode in the Properties pane to Fixed instead of the default value None. This setting tells the EF to perform concurrency checks using this property. Remember that you can have more than one property in the same entity with Concurrency Mode set to Fixed, but this is usually not necessary.

Serialization

After you have the prerequisites out of the way, the next topic is serialization. You need a way to move your entities between tiers. If you are using the default entity code generated by the EF and you are building a Windows Communication Foundation (WCF) service, your work is done because the EF automatically generates DataContract attributes on the types and DataMember attributes on the persistable properties of your entities. This includes navigation properties, which means that if you retrieve a graph of related entities into memory, the whole graph is serialized automatically. The generated code also supports binary serialization and XML serialization out of the box, but XML serialization applies only to single entities, not to graphs.

Another important concept to understand is that while the default-generated entities support serialization, their change-tracking information is stored in the ObjectStateManager (a part of the ObjectContext), which does not support serialization. In the simple entities pattern, you typically retrieve unmodified entities from the database on the mid-tier and serialize them to the client, which does not need the change-tracking information. That code might look something like this:

public Customer GetCustomerByID(string id)

{

using (var ctx = new NorthwindEntities())

{

return ctx.Customers.Where(c => c.CustomerID == id).First();

}

}

When it comes time to perform an update, however, the change-tracking information must be managed somehow, and that leads to the next important part of the EF you need to understand.

Working with the ObjectStateManager

For two-tier persistence operations, the ObjectStateManager does its job automatically for the most part. You don't have to think about it at all. The state manager keeps track of the existence of each entity under its control; its key value; an EntityState value, which can be unchanged, modified, added, or deleted; a list of modified properties; and the original value of each modified property. When you retrieve an entity from the database, it is added to the list of entities tracked by the state manager, and the entity and the state manager work together to maintain the tracking information. If you set a property on the entity, the state of the entity automatically changes to Modified, the property is added to the list of modified properties, and the original value is saved. Similar information is tracked if you add or delete an entity. When you call SaveChanges on the ObjectContext, this tracking information is used to compute the update statements for the database. If the update completes successfully, deleted entities are removed from the context, and all other entities transition to the unchanged state so that the process can start over again.

When you send entities to another tier, however, this automatic tracking process is interrupted. To implement a service method on the mid-tier that performs an update by using information from the client, you need two special methods that exist on the ObjectContext for just this purpose: Attach and ApplyPropertyChanges.

The Attach method tells the state manager to start tracking an entity. Normally, queries automatically attach entities, but if you have an entity that you retrieved some other way (serialized from the client, for example), then you call Attach to start the tracking process. There are two critical things about Attach to keep in mind.

First, at the end of a successful call to Attach, the entity will always be in the unchanged state. If you want to eventually get the entity into some other state, such as modified or deleted, you need to take additional steps to transition the entity to that state. In effect, Attach tells the EF, "Trust me. At least at some point in the past, this is how this entity looked in the database." The value an entity's property has when you attach it will be considered the original value for that property. So, if you retrieve an entity with a query, serialize it to the client, and then serialize it back to the mid-tier, you can use Attach on it rather than querying again. The value of the concurrency token when you attach the entity will be used for concurrency checks. (For more information about the danger of querying again, see my description of the anti-pattern Mishandled Concurrency in the June issue of *MSDN Magazine* at [Anti-Patterns To Avoid In N-Tier Applications](http://msdn.microsoft.com/en-us/magazine/dd882522.aspx).)

The second thing to know about Attach is that if you attach an entity that is part of a graph of related entities, the Attach method will walk the graph and attach each of the entities it finds. This occurs because the EF never allows a graph to be in a mixed state, where it is partially attached and partially not attached. So if the EF attaches one entity in a graph, it needs to make sure that the rest of the graph becomes attached as well.

The ApplyPropertyChanges method implements the other half of a disconnected entity modification scenario. It looks in the ObjectStateManager for another entity with the same key as its argument and compares each regular property of the two entities. When it finds a property that is different, it sets the property value on the entity in the state manager to match the value from the entity passed as an argument to the method. The effect is the same as if you had performed changes directly on the entity in the state manager when it was being tracked. It is important to note that this method operates only on "regular" properties and not on navigation properties, so it affects only a single entity, not an entire graph. It was designed especially for the simple entities pattern, where a new copy of the entity contains all the information you need in its property values—no extra tracking information is required for it to function.

If you put the Attach and ApplyPropertyChanges methods together to create a simple service method for updating an entity, the method might look something like this:

public void UpdateCustomer(Customer original, Customer modified)

{

using (var ctx = new NorthwindEntities())

{

ctx.Attach(original);

ctx.ApplyPropertyChanges(modified.EntityKey.EntitySetName, modified);

ctx.SaveChanges();

}

}

While these methods make implementation of the service easy, this kind of service contract adds some complication to the client which now needs to copy the entity before modifying it. Many times, this level of complexity is more than you want or need on the client. So, instead of using ApplyPropertyChanges, you can attach the modified entity and use some lower-level APIs on the ObjectStateManager to tell it that the entity should be in the modified state and that every property is modified. This approach has the advantage of reducing the data that must travel from the client to the mid-tier (only one copy of the entity) at the expense of increasing the data that is updated in the database in some scenarios (every property will be updated even if the client modified only some because there is no way to tell which properties were modified and which were not). **Figure 1** shows what the code for this approach would look like.

http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 1 Update Service Method

public void UpdateCustomer(Customer modified)

{

using (var ctx = new NorthwindEntities())

{

ctx.Attach(modified);

var stateEntry = ctx.ObjectStateManager.GetObjectStateEntry(modified);

foreach (var propertyName in stateEntry.CurrentValues

.DataRecordInfo.FieldMetadata

.Select(fm => fm.FieldType.Name))

{

stateEntry.SetModifiedProperty(propertyName);

}

}

ctx.SaveChanges();

}

Expanding the service to include methods for adding new customers and deleting customers is also straightforward. **Figure 2** shows an example of this code.

http://i.msdn.microsoft.com/Global/Images/clear.gif Figure 2 Add and Delete Service Methods

public void AddCustomer(Customer customer)

{

using (var ctx = new NorthwindEntities())

{

ctx.AddObject("Customers", customer);

ctx.SaveChanges();

}

}

public void DeleteCustomer(Customer customer)

{

using (var ctx = new NorthwindEntities())

{

ctx.Attach(customer);

ctx.DeleteObject(customer);

ctx.SaveChanges();

}

}

This approach can be extended to methods that change relationships between entities or perform other operations. The key concept to remember is that you need to first get the state manager into something like the state it would have been in originally if you had queried the database, then make changes to the entities for the effect you want, and then call SaveChanges.

Patterns Other Than Simple Entities in .NET 3.5 SP1

If you decide to use the first release of the EF to implement one of the other patterns, my first suggestion is to read the next section, which explains how .NET 4 will make things much easier. If your project needs one of the other patterns before .NET 4 is released, however, here are a few things to think about.

The change set pattern can certainly be implemented. You can see a sample of this pattern that was written to work with one of the prerelease betas of the EF at [code.msdn.com/entitybag/](http://code.msdn.com/entitybag/). This sample has not been updated to work with the 3.5 SP1 version of the EF, but the work required to do that is fairly easily. One key step you might want to adopt even if you choose to build a change set implementation from scratch is to create an ObjectContext on the client with only the conceptual model metadata (no mapping, storage model, or real connection to the database is needed) and use that as a client-side change tracker.

DTOs are also possible. In fact, implementing DTOs is not that much more difficult with the first release of the EF than it will be in later releases. In either case, you have to write your own code or use an automatic mapper to move data between your entities and the DTOs. One idea to consider is to use LINQ projections to copy data from queries directly into your DTOs. For example, if I created a CustomerDTO class that has just name and phone properties, I could then create a service method that returns a set of CustomerDTOs like this:

public List<CustomerDTO> GetCustomerDTOs()

{

using (var ctx = new NorthwindEntities())

{

var query = from c in ctx.Customers

select new CustomerDTO()

{

Name = c.ContactName,

Phone = c.Phone

};

return query.ToList();

}

}

Unfortunately, self-tracking entities is the hardest pattern to implement in the SP1 release for two reasons. First, the EF in .NET 3.5 SP1 does not support POCO, so any self-tracking entities that you implement have a dependency on the 3.5 SP1 version of .NET, and the serialization format will not be as suitable for interoperability. You can address this by hand writing proxies for the client, but they will be tricky to implement correctly. Second, one of the nice features of self-tracking entities is that you can create a single graph of related entities with a mix of operations—some entities can be modified, others new, and still others marked for deletion—but implementing a method on the mid-tier to handle such a mixed graph is quite difficult. If you call the Attach method, it will walk the whole graph, attaching everything it can reach. Similarly, if you call the AddObject method, it will walk the whole graph and add everything it can reach. After either of those operations occurs, you will encounter cases in which you cannot easily transition some entities to their intended final state because the state manager allows only certain state transitions. You can move an entity from unchanged to modified, for instance, but you cannot move it from unchanged to added. To attach a mixed graph to the context, you need to shred your graph into individual entities, add or attach each one separately, and then reconnect the relationships. This code is very difficult.

API Improvements in .NET 4

In the upcoming release of the EF, which will ship with Visual Studio 2010 and .NET 4, we have made a number of improvements to ease the pain of implementing n-tier patterns—especially self-tracking entities. I'll touch on some of the most important features in the following paragraphs.

POCO The EF will support complete persistence ignorance for entity classes. This means that you can create entities that have no dependencies on the EF or other persistence-related DLLs. A single entity class used for persisting data with the EF will also work on Silverlight or earlier versions of .NET. Also, POCO helps isolate the business logic in your entities from persistence concerns and makes it possible to create classes with a very clean, interoperable serialization format.

Improved N-Tier Support APIs Working with the ObjectStateManager will be easier because we have relaxed the state transition constraints. It will be possible to first add or attach an entire graph and then walk over that graph changing entities to the right state. You will be able to set the original values of entities, change the state of an entity to any value, and change the state of a relationship.

Foreign Key Property Support The first release of the EF supports modeling relationships only as completely separate from entities, which means that the only way to change relationships is through the navigation properties or the RelationshipManager. In the upcoming release, you'll be able to build a model in which an entity exposes a foreign key property that can be manipulated directly.

T4-Based Code Generation The final important change to the EF will be the use of the T4 template engine to allow easy, complete control over the code that is generated for entities. This is important because it means Microsoft can create and release templates that generate code for a variety of scenarios and usage patterns, and you can customize those templates or even write your own. One of the templates we will release will produce classes that implement the self-tracking entities pattern with no custom coding required on your part. The resulting classes allow the creation of very simple clients and services.

More to Learn I hope this article has given you a good survey of the design issues involved in creating n-tier applications and some specific hints for implementing those designs with the Entity Framework. There is certainly a lot more to learn, so I encourage you to take a look at the [Application Architecture Guide from the patterns & practices group](http://codeplex.com/AppArchGuide/) and the [Entity Framework FAQ](http://blogs.msdn.com/dsimmons/pages/entity-framework-faq.aspx).

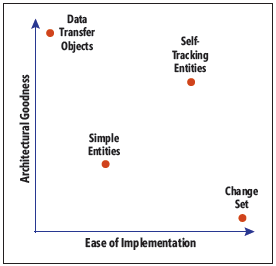
# Building N-Tier Apps with EF4

This article is the third in a series about n-tier programming with the Entity Framework (see [msdn.microsoft.com/magazine/dd882522.aspx](http://msdn.microsoft.com/en-us/magazine/dd882522.aspx) and [msdn.microsoft.com/magazine/ee321569.aspx](http://msdn.microsoft.com/en-us/magazine/ee321569.aspx)), specifically about building custom Web services with the Entity Framework (EF) and Windows Communication Foundation (WCF). (In some situations, a REST-based service or some other approach is appropriate, but in these articles, I’ve focused on custom Web services.) The first article described a number of important design considerations and antipatterns. In the second article, I wrote about four patterns that can be used successfully in an n-tier application. That article also included code samples that illustrate how the first release of the Entity Framework (EF 3.5 SP1) can be used to implement what I call the Simple Entities pattern. In this article, I’ll look at some features coming in the second release of the Entity Framework (EF4) and how you use them to implement the Self-Tracking Entities and Data Transfer Objects (DTOs) n-tier patterns.

While Simple Entities is usually not the preferred pattern for n-tier applications, it is the most viable option in the first release of the EF.  EF4, however, significantly changes the options for n-tier programming with the framework.  Some of the key new features include the following:

1. New framework methods that support disconnected operations, such as ChangeObjectState and ChangeRelationshipState, which change an entity or relationship to a new state (added or modified, for example); ApplyOriginalValues, which lets you set the original values for an entity; and the new ObjectMaterialized event, which fires whenever an entity is created by the framework.
2. Support for Plain Old CLR Objects (POCO) and foreign key values on entities. These features let you create entity classes that can be shared between the mid-tier service implementation and other tiers, which may not have the same version of the Entity Framework (.NET 2.0 or Silverlight, for example). POCO objects with foreign keys also have a straightforward serialization format that simplifies interoperability with platforms like Java. The use of foreign keys also enables a much simpler concurrency model for relationships.
3. T4 templates to customize code generation. These templates provide a way to generate classes implementing the Self-Tracking Entities or DTOs patterns.

The Entity Framework team has used these features to implement the Self-Tracking Entities pattern in a template, making that pattern a lot more accessible, and while DTOs still require the most work during initial implementation, this process is also easier with EF4. (The Self-Tracking Entities template and a few other EF features are available as part of a Web download feature community technology preview (CTP) rather than in the Visual Studio 2010/.NET 4 box. The samples in this article assume that Visual Studio 2010/.NET 4 and the feature CTP are installed.) With these new capabilities, one way to evaluate the four patterns I’ve described (Simple Entities, Change Set, Self-Tracking Entities and DTOs) is in terms of a trade-off between architectural goodness (separation of concerns/loose coupling, strength of contract, efficient wire format and interoperability) and ease of implementation and time to market. If you plot the four patterns on a graph that represents this trade-off, the result might look something like **Figure 1**.

  
Figure 1 **Comparing N-Tier Patterns with EF4**

The right pattern for a particular situation depends on a lot of factors. In general, DTOs provide many architectural advantages at a high initial implementation cost. Change Set exhibits few good architectural characteristics but is easy to implement (when it’s available for a particular technology—for example, the DataSet in traditional ADO.NET).

I recommend a pragmatic/agile balance between these concerns by starting with Self-Tracking Entities and moving to DTOs if the situation warrants it. Often, you can get up and running quickly with Self-Tracking Entities and still achieve many important architectural goals. This approach represents a much better trade-off than Change Set or Simple Entities, either of which I would recommend only if you have no other viable options. DTOs, on the other hand, are definitely the best choice as your application becomes larger and more complex or if you have requirements that can’t be met by Self-Tracking Entities, like different rates of change between the client and the server. These two patterns are the most important tools to have in your toolbox, so let’s take a look at each of them.

## Self-Tracking Entities

To use this pattern with the Entity Framework, start by creating an Entity Data Model that represents your conceptual entities and map it to a database. You can reverse engineer a model from a database you have and customize it, or you can create a model from scratch and then generate a database to match (another new feature in EF4). Once this model and mapping are in place, replace the default code generation template with the Self-Tracking Entities template by right-clicking the entity designer surface and choosing Add Code Generation Item.

Next, choose the Self-Tracking Entities template from the list of installed templates. This step turns off default code generation and adds two templates to your project: one template generates the ObjectContext, and the other template generates entity classes. Separating code generation into two templates makes it possible to split the code into separate assemblies, one for your entity classes and one for your context.

The main advantage of this approach is that you can have your entity classes in an assembly that has no dependencies on the Entity Framework. This way, the entity assembly (or at least the code that it generates) and any business logic you have implemented there can be shared by the mid-tier and the client if you want. The context is kept in an assembly that has dependencies on both the entities and the EF. If the client of your service is running .NET 4, you can just reference the entity assembly from the client project. If your client is running an earlier version of .NET or is running Silverlight, you probably want to add links from the client project to the generated files and recompile the entity source in that project (targeting the appropriate CLR).

Regardless of how you structure your project, the two templates work together to implement the Self-Tracking Entities pattern. The generated entity classes are simple POCO classes whose only feature beyond basic storage of entity properties is to keep track of changes to the entities—the overall state of an entity, changes to critical properties such as concurrency tokens, and changes in relationships between entities. This extra tracking information is part of the DataContract definition for the entities (so when you send an entity to or from a WCF service, the tracking information is carried along).

On the client of the service, changes to the entities are tracked automatically even though the entities are not attached to any context. Each generated entity has code like the following for each property. If you change a property value on an entity with the Unchanged state, for instance, the state is changed to Modified:

[DataMember]

public string ContactName

{

get { return \_contactName; }

set

{

if (!Equals(\_contactName, value))

{

\_contactName = value;

OnPropertyChanged("ContactName");

}

}

}

private string \_contactName;

Similarly, if new entities are added to a graph or entities are deleted from a graph, that information is tracked. Since the state of each entity is tracked on the entity itself, the tracking mechanism behaves as you would expect even when you relate entities retrieved from more than one service call. If you establish a new relationship, just that change is tracked—the entities involved stay in the same state, as though they had all been retrieved from a single service call.

The context template adds a new method, ApplyChanges, to the generated context. ApplyChanges attaches a graph of entities to the context and sets the information in the ObjectStateManager to match the information tracked on the entities. With the information that the entities track about themselves and ApplyChanges, the generated code handles both change tracking and concurrency concerns, two of the most difficult parts of correctly implementing an n-tier solution.

As a concrete example, **Figure 2** shows a simple ServiceContract that you could use with Self-Tracking Entities to create an n-tier order submission system based on the Northwind sample database.

Figure 2 **A Simple Service Contract for the Self-Tracking Entities Pattern**

[ServiceContract]

public interface INorthwindSTEService

{

[OperationContract]

IEnumerable<Product> GetProducts();

[OperationContract]

Customer GetCustomer(string id);

[OperationContract]

bool SubmitOrder(Order order);

[OperationContract]

bool UpdateProduct(Product product);

}

The GetProducts service method is used to retrieve reference data on the client about the product catalog. This information is usually cached locally and isn’t often updated on the client. GetCustomer retrieves a customer and a list of that customer’s orders.  The implementation of that method is quite simple, as shown here:

public Customer GetCustomer(string id)

{

using (var ctx = new NorthwindEntities())

{

return ctx.Customers.Include("Orders")

.Where(c => c.CustomerID == id)

.SingleOrDefault();

}

}

This is essentially the same code that you would write for an implementation of this kind of method with the Simple Entities pattern. The difference is that the entities being returned are self-tracking, which means that the client code for using these methods is also quite simple, but it can accomplish much more.

To illustrate, let’s assume that in the order submission process you want not only to create an order with appropriate order detail lines but also to update parts of the customer entity with the latest contact information. Further, you want to delete any orders that have a null OrderDate (maybe the system marks rejected orders that way). With the Simple Entities pattern, the combination of adding, modifying and deleting entities in a single graph would require multiple service calls for each type of operation or a very complicated custom contract and service implementation if you tried to implement something like Self-Tracking Entities in the first release of the EF. With EF4, the client code might look like **Figure 3**.

Figure 3 **Client Code for the Self-Tracking Entities Pattern**

var svc = new ChannelFactory<INorthwindSTEService>(

"INorthwindSTEService")

.CreateChannel();

var products = new List<Product>(svc.GetProducts());

var customer = svc.GetCustomer("ALFKI");

customer.ContactName = "Bill Gates";

foreach (var order in customer.Orders

.Where(o => o.OrderDate == null).ToList())

{

customer.Orders.Remove(order);

}

var newOrder = new Order();

newOrder.Order\_Details.Add(new Order\_Detail()

{

ProductID = products.Where(p => p.ProductName == "Chai")

.Single().ProductID,

Quantity = 1

});

customer.Orders.Add(newOrder);

var success = svc.SubmitOrder(newOrder);

This code creates the service, calls the first two methods on it to get the product list and a customer entity, and then makes changes to the customer entity graph using the same sort of code you would write if you were building a two-tier Entity Framework application that talks directly to the database or were implementing a service on the mid-tier. (If you aren’t familiar with this style of creating a WCF service client, it automatically creates a client proxy for you without creating proxies for the entities, since we are reusing the entity classes from the Self-Tracking entities template. You could also use the client generated by the Add Service Reference command in Visual Studio if you want.) But here, there is no ObjectContext involved. You are just manipulating the entities themselves. Finally, the client calls the SubmitOrder service method to push the changes up to the mid-tier.

Of course, in a real application the client’s changes to the graph would probably have come from a UI of some sort, and you would add exception handling around the service calls (especially important when you have to communicate over the network), but the code in **Figure 3** illustrates the principles. Another important item to notice is that when you create the order detail entity for the new order, you set just the ProductID property rather than the Product entity itself. This is the new foreign key relationship feature in action. It reduces the amount of information that travels over the wire because you serialize only the ProductID back to the mid-tier, not a copy of the product entity.

It’s in the implementation of the SubmitOrder service method that Self-Tracking Entities really shines:

public bool SubmitOrder(Order newOrder)

{

using (var ctx = new NorthwindEntities())

{

ctx.Orders.ApplyChanges(newOrder);

ValidateNewOrderSubmission(ctx, newOrder);

return ctx.SaveChanges() > 0;

}

}

The call to ApplyChanges performs all the magic. It reads the change information from the entities and applies it to the context in a way that makes the result the same as if those changes had been performed on entities attached to the context the whole time.

## Validating Changes

Something else you should notice in the SubmitOrder implementation is the call to ValidateNewOrderSubmission. This method, which I added to the service implementation, examines the ObjectStateManager to make sure that only the kinds of changes we expect in a call to SubmitOrder are present.

This step is really important because by itself, ApplyChanges pushes whatever changes it finds in an entire graph of related objects into the context. Our expectation that the client will only add new orders, update the customer and so on doesn’t mean that a buggy (or even malicious) client would not do something else. What if it changed the price on a product to make an order cheaper or more expensive than it should be? The details of how the validation is performed are less important than the critical rule that you should always validate changes before saving them to the database. This rule applies regardless of the n-tier pattern you use.

 A second critical design principle is that you should develop separate, specific service methods for each operation. Without these separate operations, you do not have a strong contract representing what is and isn’t allowed between your two tiers, and properly validating your changes can become impossible. If you had a single SaveEntities service method instead of a SubmitOrder and a separate UpdateProduct method (only accessible by users authorized to modify the product catalog), you could easily implement the apply and save part of that method, but you would be unable to validate properly because you would have no way to know when product updates are allowed and when they are not.

## Data Transfer Objects

The Self-Tracking Entities pattern makes the n-tier process easy, and if you create specific service methods and validate each one, it can be fairly sound architecturally. Even so, there are limits to what you can do with the pattern. When you run into those limits, DTOs are the way out of the dilemma.

In DTOs, instead of sharing a single entity implementation between the mid-tier and the client, you create a custom object that’s used only for transferring data over the service and develop separate entity implementations for the mid-tier and the client. This change provides two main benefits: it isolates your service contract from implementation issues on the mid-tier and the client, allowing that contract to remain stable even if the implementation on the tiers changes, and it allows you to control what data flows over the wire. Therefore, you can avoid sending unnecessary data (or data the client is not allowed to access) or reshape the data to make it more convenient for the service. Generally, the service contract is designed with the client scenarios in mind so that the data can be reshaped between the mid-tier entities and the DTOs (maybe by combining multiple entities into one DTO and skipping properties not needed on the client), while the DTOs can be used directly on the client.

These benefits, however, come at the price of having to create and maintain one or two more layers of objects and mapping. To extend the order submission example, you could create a class just for the purpose of submitting new orders. This class would combine properties of the customer entity with properties from the order that are set in the new order scenario, but the class would leave out properties from both entities that are computed on the mid-tier or set at some other stage in the process. This makes the DTO as small and efficient as possible. The implementation might look like this:

public class NewOrderDTO

{

public string CustomerID { get; set; }

public string ContactName { get; set; }

public byte[] CustomerVersion { get; set; }

public List<NewOrderLine> Lines { get; set; }

}

public class NewOrderLine

{

public int ProductID { get; set; }

public short Quantity { get; set; }

}

Okay, this is really two classes—one for the order and one for the order detail lines—but the data size is kept as small as possible. The only seemingly extraneous information in the code is the CustomerVersion field, which contains the row version information used for concurrency checks on the customer entity. You need this information for the customer entity because the entity already exists in the database. For the order and detail lines, those are new entities being submitted to the database, so their version information and the OrderID aren’t needed—they are generated by the database when the changes are persisted.

The service method that accepts this DTO uses the same lower-level Entity Framework APIs that the Self-Tracking Entities template uses to accomplish its tasks, but now you need to call those APIs directly rather than let the generated code call them for you. The implementation comes in two parts. First, you create a graph of customer, order and order detail entities based on the information in the DTO (see **Figure 4**).

Figure 4 **Creating a Graph of Entities**

var customer = new Customer

{

CustomerID = newOrderDTO.CustomerID,

ContactName = newOrderDTO.ContactName,

Version = newOrderDTO.CustomerVersion,

};

var order = new Order

{

Customer = customer,

};

foreach (var line in newOrderDTO.Lines)

{

order.Order\_Details.Add(new Order\_Detail

{

ProductID = line.ProductID,

Quantity = line.Quantity,

});

}

Then you attach the graph to the context and set the appropriate state information:

ctx.Customers.Attach(customer);

var customerEntry = ctx.ObjectStateManager.GetObjectStateEntry(customer);

customerEntry.SetModified();

customerEntry.SetModifiedProperty("ContactName");

ctx.ObjectStateManager.ChangeObjectState(order, EntityState.Added);

foreach (var order\_detail in order.Order\_Details)

{

ctx.ObjectStateManager.ChangeObjectState(order\_detail,

EntityState.Added);

}

return ctx.SaveChanges() > 0;

The first line attaches the entire graph to the context, but when this occurs, each entity is in the Unchanged state, so first you tell the ObjectStateManager to put the customer entity in the Modified state, but with only one property, ContactName, marked as modified. This is important because you don’t actually have all the customer information—just the information that was in the DTO. If you marked all properties as modified, the Entity Framework would try to persist a bunch of nulls and zeroes to other fields in the customer entity.

Next you change the state of the order and each of its order details to Added, and then you call SaveChanges.

Hey, where’s the validation code? In this case, because you have a very specific DTO for your scenario, and you are interpreting that object as you map the information from it into your entities, you perform the validation as you go along. There’s no way this code could inadvertently change the price of a product because you never touch the product entity. This is another benefit of the DTO pattern, but only in a roundabout way. You still have to do the validation work; the pattern just forces one level of validation. In many cases, your code needs to include additional validation of the values or other business rules.

One other consideration is properly handling concurrency exceptions. As I mentioned earlier, the version information for the customer entity is included in the DTO, so you are set up to properly detect concurrency issues if someone else modifies the same customer. A more complete sample would either map this exception to a WCF fault so that the client could resolve the conflict, or it would catch the exception and apply some sort of automatic policy for handling the conflict.

If you wanted to extend the sample further by adding another operation, like the ability to modify an order, you would create another DTO specifically for that scenario, with just the right information for it.  This object would look something like our NewOrderDTO, but it would have the OrderID and Version properties for the order and order details entities as well as each property you want to allow the service call to update. The service method implementation would also be similar to the SubmitOrderDTO method shown earlier—walking through the DTO data, creating corresponding entity objects and then setting their state in the state manager before saving the changes to the database.

If you were to implement the order update method both with Self-Tracking Entities and Data Transfer Objects, you would find that the Self-Tracking Entities implementation reuses the entities and shares almost all the same service implementation code between it and the new order submission method—the only difference would be the validation code, and even some of that might be shared. The DTO implementation, however, requires a separate Data Transfer Object class for each of the two service methods, and the method implementations follow similar patterns but have very little if any code that can be shared.

## Tips from the Trenches

Here are some tips for what  to watch out for and understand.

* **Make certain to reuse the Self-Tracking Entity template’s generated entity code on your client.** If you use proxy code generated by Add Service Reference in Visual Studio or some other tool, things look right for the most part, but you will discover that the entities don’t actually keep track of their changes on the client.
* **Create a new ObjectContext instance in a Using statement for each service method so that it is disposed of before the method returns.** This step is critical for scalability of your service. It makes sure that database connections are not kept open across service calls and that temporary state used by a particular operation is garbage collected when that operation is over. The Entity Framework automatically caches metadata and other information it needs in the app domain, and ADO.NET pools database connections, so re-creating the context each time is a quick operation.
* **Use the new foreign key relationships feature whenever possible.**  It makes changing relationships between entities much easier. With independent associations—the only type of relationship available in the first release of the Entity Framework—concurrency checks are performed on relationships independently of the concurrency checks performed on entities, and there is no way to opt out of these relationship concurrency checks. The result is that your services must carry the original values of relationships and set them on the context before changing relationships. With foreign key relationships, though, the relationship is simply a property of the entity, and if the entity passes its concurrency check, no further check is needed. You can change a relationship just by changing the foreign key value.
* **Be careful of EntityKey collisions when attaching a graph to an ObjectContext.** If, for instance, you are using DTOs and parts of your graph represent newly added entities for which the entity key values have not been set because they will be generated in the database, you should call the  AddObject method to add the whole graph of entities first and then change entities not in the Added state to their intended state rather than calling the Attach method and then changing Added entities to that state. Otherwise, when you first call Attach, the Entity Framework thinks that every entity should be put into the Unchanged state, which assumes that the entity key values are final. If more than one entity of a particular type has the same key value (0, for example), the Entity Framework will throw an exception. By starting with an entity in the Added state, you avoid this problem because the framework does not expect Added entities to have unique key values.
* **Turn off automatic lazy loading (another new EF4 feature) when returning entities from service methods.** If you don’t, the serializer will trigger lazy loading and try to retrieve additional entities from the database, which will cause more data than you intended to be returned (if your entities are thoroughly connected, you might serialize the whole database), or, more likely, you will receive an error because the context will be disposed of before the serializer tries to retrieve the data. Self-Tracking Entities does not have lazy loading turned on by default, but if you are creating a DTOs solution, this is something to watch out for.

## And in the End

The .NET 4 release of the Entity Framework makes the creation of architecturally sound n-tier applications much easier. For most applications, I recommend starting with the Self-Tracking Entities template, which simplifies the process and enables the most reuse. If you have different rates of change between service and client, or if you need absolute control over your wire format, you should move up to a Data Transfer Objects implementation. Regardless of which pattern you choose, always keep in mind the key principles that the antipatterns and patterns represent—and never forget to validate your data before saving