CHAPTER 15

Working with Relationships and Associations

At this point in the book, you have worked extensively with entities that are related to one another. You have also experienced a number of scenarios where it was necessary to do extra work to an object because of its associations. For example, when you map stored procedures to entities, you are required to account for any associations the entity was involved in, even when it doesn't make sense to do so. Remember seeing the `ReservationID` of the BreakAwayModel as a parameter of the `DeletePayment` function? You learned that if an `EntitySet` is mapped using a `QueryView`, every other related entity also needs to be mapped using a `QueryView`. In building a WCF service in Chapter 14, you had to do a little extra work to make sure that when inserting new contacts, the context did not also attempt to create a new `Trip` entity.

So much is going on behind the scenes with respect to how the Entity Framework manages relationships that it causes a lot of unexpected behavior, seemingly strange errors, and many rules that we need to follow. The rules are in place to maintain the integrity of these relationships.

It's important to remember that the Entity Data Model (EDM)—because it is based on an Entity Relationship Model (ERM)—is a model of entities *and* relationships, not a model of entities that happen to have relationships. In the EDM, relationships are represented as *associations*, which are actual objects in the model and have the same importance as the model's entities. This is why it's so important to work within the boundaries that keep these relationships in sync with the entities. When you use the Entity Framework to interact with an EDM, its relationships are instantiated as objects, even though you have to do a little digging to see them.

In this chapter, you'll learn how relationships and associations fit into the EDM, how they work both inside and outside the `ObjectContext`, and what it means for a relationship to be an object. With this knowledge, you will be better prepared to manipulate relationships between entities, adding or removing relationships between objects in the way that the Entity Framework expects; you will be able to solve problems that arise

because of relationships; you will even enhance your ability to build more meaningful Entity Data Models. You will also see this knowledge pay off in a big way when working with multi-tiered applications such as web services.

The chapter is divided into two parts. The first part is devoted to teaching you how relationships and associations work in the Entity Framework, from the EDM to the `EntityObject`s. The second part will teach you how to perform a number of critical tasks relating to entity graphs.

Deconstructing Relationships in the Entity Data Model

Many developers new to the Entity Framework have a lot of questions about relationships as they attempt to build Entity Data Models and write code to interact with entities. Having a better understanding of the fundamentals of these associations and how they work will allow you to create more powerful applications and better comprehend the Entity Framework's behavior. First we'll look at the model and then at Object Services.

In the Designer, `Association`s are represented as lines between related entities. The Designer displays the *multiplicity* between the entities. Multiplicity defines how many items a particular end can have. The multiplicity of an end can be defined in one of three ways:

One
: The end must have one item, no less and no more. This is quite often a parent in a parent–child relationship.

Many
: The end can have many items. This is generally a collection of children in a parent–child relationship and it's possible that no items (zero) exist in this collection.

Zero or One
: The end can have either zero items or one item but no more than one. Many of the entity references you have worked with in the BreakAway model have Zero or One ends. For example, the `Customer.PrimaryDestination` field may not have a destination defined, and therefore there will be zero items at that end. If a destination is defined, there can be no more than one.

As you learned in Chapter 2, the common notations for these are 1 (One), * (Many), and 0..1 (Zero or One). The EDM Designer displays the relationships with this notation.

When you hover your mouse pointer over a line representing an `Association`, you can see some additional details of the `Association`, as shown Figure 15-1.

In the Properties window of the `Association`, shown in Figure 15-2, you can see even more details and make modifications if necessary.

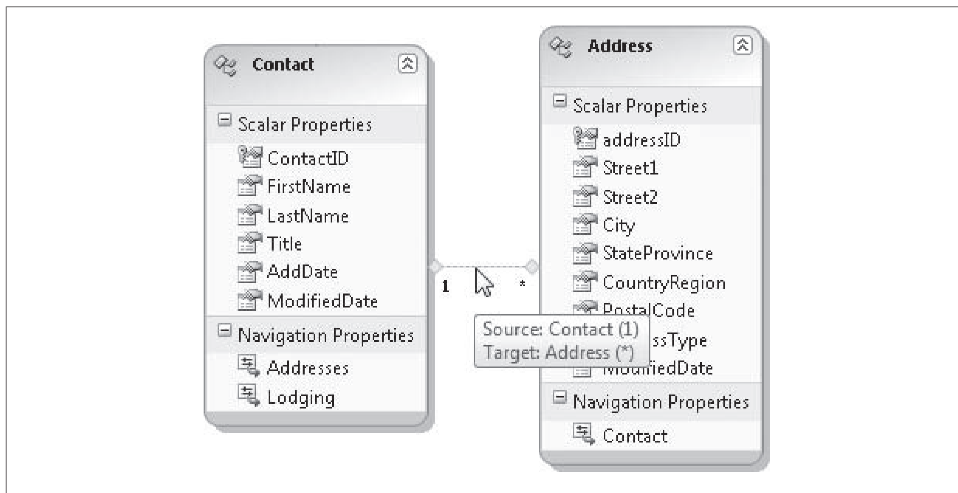


Figure 15-1. An association represented in the Designer

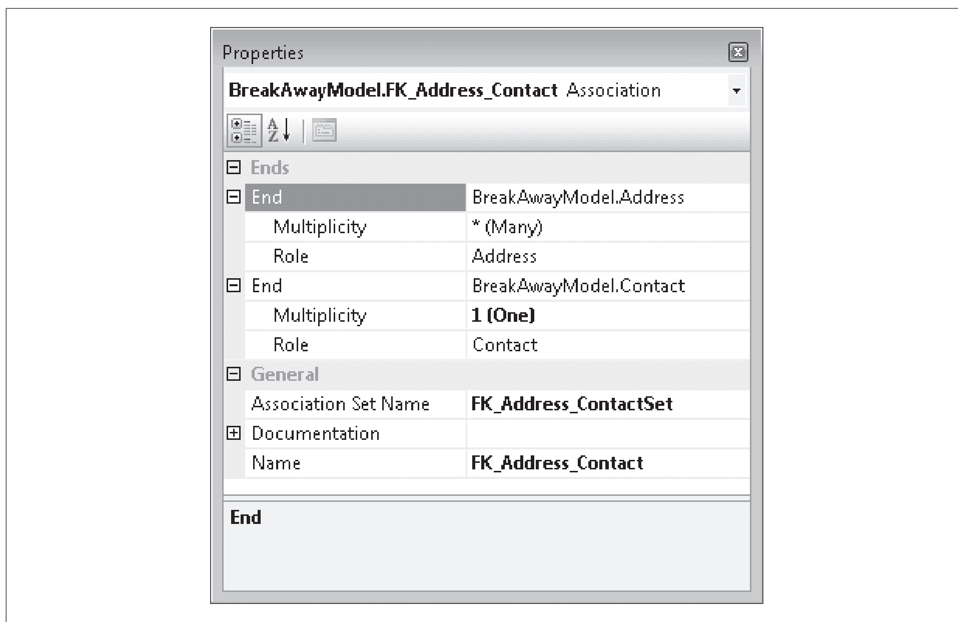


Figure 15-2. The Association's Properties window

By default, the `AssociationSet` has the same name as the `Association`. You may find it helpful to change the name of the `AssociationSet`, as shown in Figure 15-2, so that when you're looking at the EDMX in the XML Editor, it will be obvious whether you are looking at the `Association` or the `AssociationSet`. It is not common, however, to

work with the AssociationSet directly in code, and therefore this is not a change that I have ever made when customizing my own EDMs.

How Did the Entity Data Model Wizard Create the Association?

The EDM Wizard created the FK_Address_Contact association shown in Figure 15-2 when it read the BreakAway database. If you are unfamiliar with how relationships are defined in a database, it will be helpful to read additional details about this.

Figure 15-3 shows a portion of the database diagram for the BreakAway database. The diagram shows the Contact and Address tables as well as a visual representation of the 1:* (One to Many) relationship between Contact and Address. On the Contact side, the symbol for primary key is used because the primary key, ContactID, is used in the definition of the relationship.

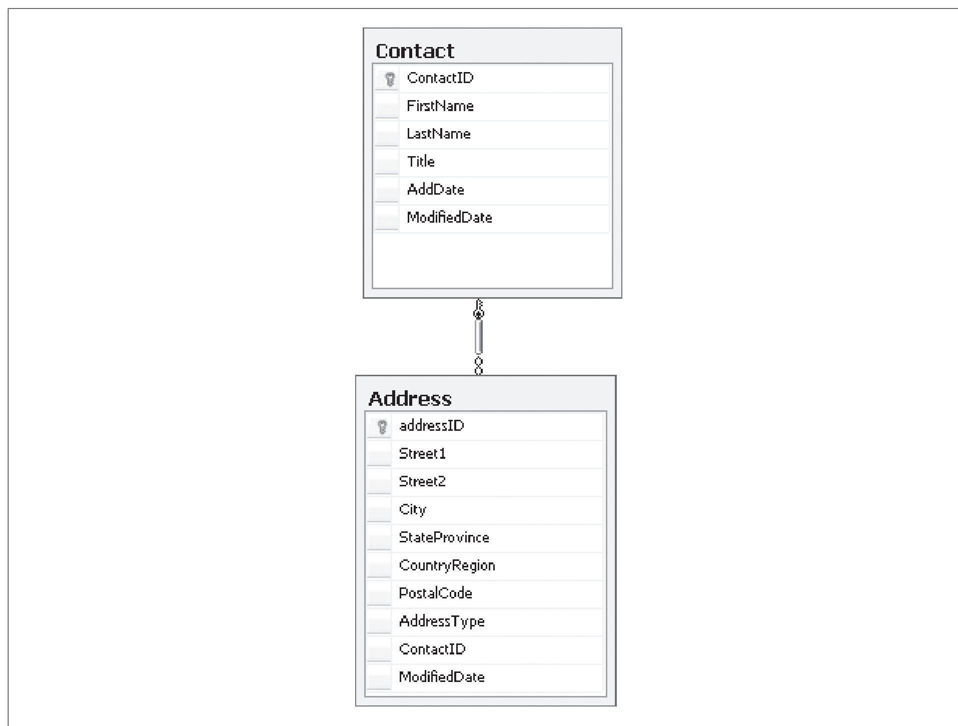


Figure 15-3. A primary key/foreign key relationship defined between the Contact and Address tables in the database

The ContactID field in the Address table is a foreign key. The relationship is known as a *primary key/foreign key relationship*, which is often described and represented as *PK/FK*. This relationship is defined in a foreign key relationship of the Address table named FK_Address_Contact, as shown in Figure 15-4.

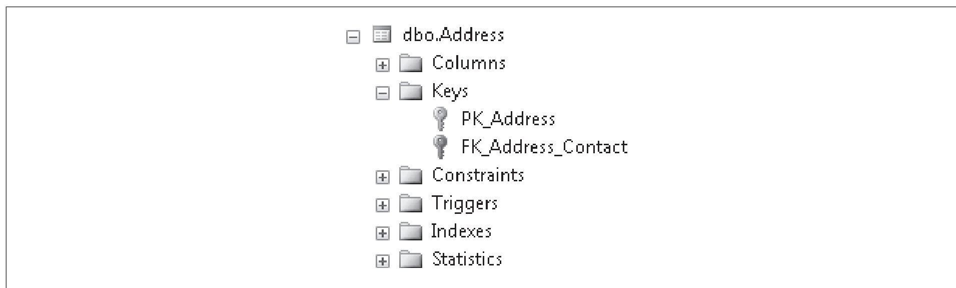


Figure 15-4. The relationship defined by the table that contains the foreign key

The `Contact` table has no knowledge of this relationship; the relationship and all of its rules (known as *constraints*) are contained in the `FK_Address_Contact` key. Figure 15-5 shows the details of this foreign key.

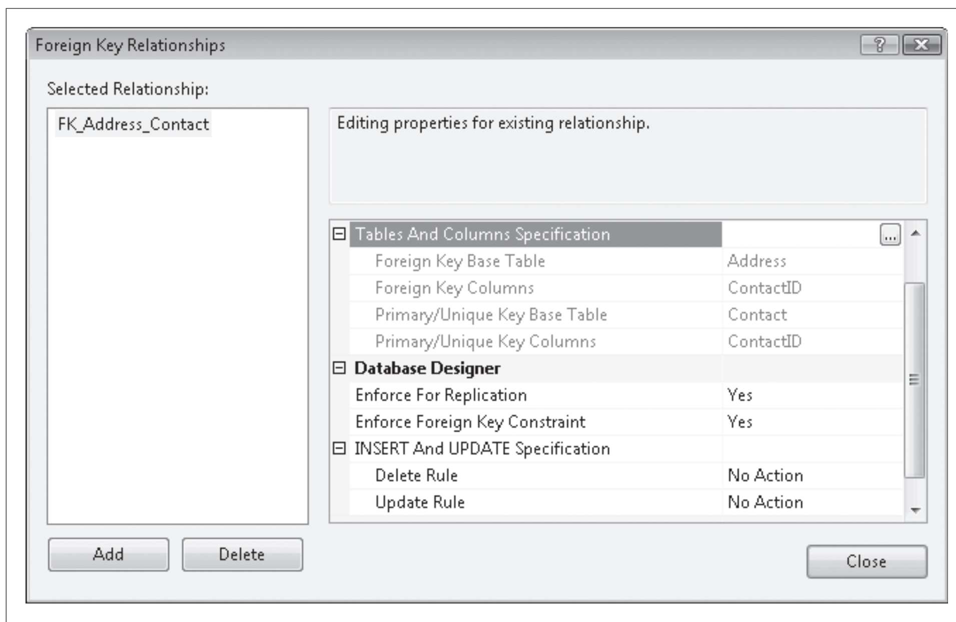


Figure 15-5. SQL Server Management Studio's property editor for defining a relationship between tables

Although a *cascading delete rule* is not being used in this case, you could define such a rule so that anytime a contact is deleted all of its related `Address` records will be deleted automatically. You might expect this to be defined in the `Contact` table, but instead it is defined in the `Address` table's foreign key. To use a cascading delete rule in this case, you would change the Delete Rule in Figure 15-5 from No Action to Cascade.

The EDM Wizard reads all of this information, creates the `FK_Address_Contact` association, and wires it up to the relevant items in the model.

Additional Items Created in the Model

In addition to the association, a number of other items are created in the model as a result of this relationship:

Navigation properties
: Navigation properties are the most visible properties that result from the relationship, and you have used them extensively in this book already.

  The navigation property itself doesn't contain very much information, but it does have a pointer back to the association, which enables the navigation property to return the related entity or collection of entities.

AssociationSets
: Like an `EntitySet`, an `AssociationSet` acts as a container for associations that have been instantiated at runtime. If you have three contacts in the `ObjectContext` along with one or more addresses for those contacts, three instances of the `FK_Address_Contact` association will be in the context as well. Although you will rarely interact with these directly, the `ObjectContext` manages association objects as carefully as it manages entity objects. Because developers don't see these association objects, it is sometimes confusing when the Entity Framework follows particular patterns to maintain the associations. You'll learn more about this later in the chapter.

AssociationSet mapping
: The `EntitySetMapping` element in the model contains no information about navigations or associations, a fact you can confirm by viewing it. Only the scalar properties are mapped. All the relationship information is contained in the `AssociationSetMapping` element for the association that is bound to the entity. You have also seen that you can create or view these mappings in the Designer.

Navigation Properties Are Not Required

Although an association is always bidirectional, navigating with properties doesn't necessarily have to be. An interesting twist on relationships is that you are not required to have a navigation property in your model for every endpoint of a relationship.

As an example, the business logic of your application may define that you will frequently need to navigate from a contact to its reservation, but that you will never have to navigate from a reservation back to the contact, meaning `Reservation.Contact` and `Reservation.ContactReference.Load` will never be required, but `Contact.Reservations` will be useful.

In this case, you could simply delete the `Contact` navigation property from the `Reservation` entity. However, you must do this in the raw XML as the Designer does not support deleting Navigation Properties. This won't impact the association between the two entities, and in an edge case you can always dig into the `ObjectStateManager` to get from the `Reservation` to the `Contact`. The plus side is that when you're coding or

debugging, you won't have the unnecessary `Contact` and `ContactReference` properties constantly in your face.

Understanding How Associations Impact the Native Query

The native query logic you saw when calling `Load` also applies to how initial queries are constructed. As you have seen, even if you do not request related data to be loaded, the `EntityReference` information still needs to be returned. There is a particular query scenario that can result in a surprising native query; however, as you look at it, you'll see that it does make sense after all.

Imagine two entities: `Contact` and `Address`. Each `Contact` can have zero or one related `Address`es (see Figure 15-6).

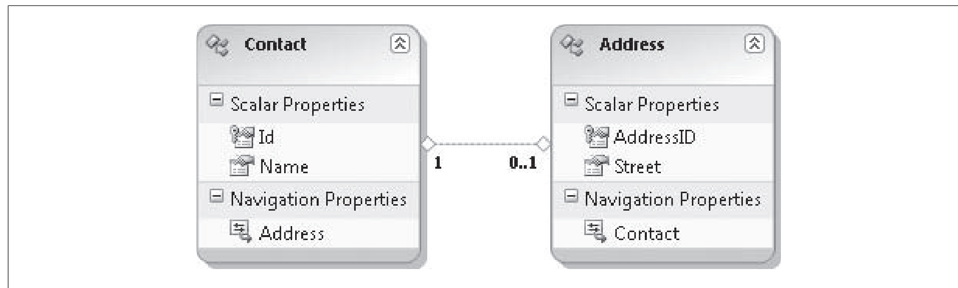


Figure 15-6. Two entities with a 1:0..1 relationship

In the database, `Address` contains a foreign key, `ContactID`, to the `Contact` table.

When you're querying for addresses, you use the `ContactID` field to build the `ContactReference.Entity`. The native query is simply a query of the `Address` table. But when you're querying the `Contact` table, the `Contact` entity needs to build the `AddressReference.EntityKey`. The `AddressID` does not exist in the `Contact` table in the database, however, and therefore an additional query of the `Address` table will be required. Here is a LINQ to Entities query that just requests contacts:

VB
```
From c In context.Contacts Select con
```

C#
```
from c in context.Contacts select c
```

The native T-SQL query, shown in Example 15-1, includes a query of the `Address` table involving all the `Address` table fields to be sure to get whatever fields are required for constructing the `EntityKey`.

Example 15-1. T-SQL when querying an entity with 1:0..1 relationship

```
SELECT
1 AS [C1],
[Extent1].[Id] AS [Id],
[Extent1].[Name] AS [Name],
[Extent2].[Id] AS [Id1],
```

```
[Extent3].[AddressID] AS [AddressID]
FROM   [dbo].[Contact] AS [Extent1]
LEFT OUTER JOIN [dbo].[Employee] AS [Extent2]
  ON [Extent1].[Id] = [Extent2].[Id]
LEFT OUTER JOIN (SELECT
      [addresses].[AddressID] AS [AddressID],
      [addresses].[street] AS [street],
      [addresses].[contactID] AS [contactID]
      FROM [dbo].[addresses] AS [addresses]) AS [Extent3]
        ON [Extent1].[Id] = [Extent3].[contactID]
```

Although this may be considered an edge case, imagine what the query would look like in a large model that contains a number of these types of relationships. This means that what you may think of as a simple query could in fact become something much more complex when it becomes a native database query.

This scenario came to my attention in the MSDN forums where a user with an extreme example of a model with many 0..1 associations attached to a single entity was seeing somewhat unwieldy SQL queries being executed in the database. Even with a simple query of the central entity he reported that the query added "52 unwanted left outer joins and pull[ed] back some 2,800 unwanted columns."

Deconstructing Relationships Between Instantiated EntityObjects

When instantiated objects are joined together in a relationship they are referred to as a *graph*, or an *entity graph*. There is no such thing as unrequited love when working with related entities—every relationship in the Entity Framework is bidirectional. When you load, add, or attach one entity to another entity, you can navigate in both directions. When an address is added to a contact's `Addresses` collection, the contact object becomes accessible through the address's `Contact` property. If these items are attached to an `ObjectContext`, when the relationship is instantiated the `Address.ContactReference` will also be populated. You need to do this on only one end or the other for the complete relationship to be created. It is not necessary to explicitly make the connection in both directions.

Relationships Are First-Class Citizens

Learning to accept relationships as first-class citizens is a challenge for many developers. Associations and the EDM rules that exist because of them are a big point of confusion for many developers who are new to the Entity Framework. Many questions in the forums can be attributed to a lack of understanding of how associations function within the Entity Framework.

The `ObjectContext` tracks relationships independently of entities. In fact, relationships are instantiated as objects. When you add an `Address` to the contact's `Addresses EntityCollection`, a new object is created to represent that relationship. This object has an `EntityState` just as an entity does. When you remove an `Address` from the collection, that relationship object is deleted—it is not removed, but its `EntityState` is set to `Deleted`. The Entity Framework resolves these relationships when `SaveChanges` is called to make the appropriate changes in the database.

Relationship Span

The term *relationship span* is not an official term. You may not even find it in the EDM documentation. But it is often used to describe the rules by which the Entity Framework handles related entities under the covers. When you create a query that traverses relationships, the Entity Framework will know not only how to construct the query, but also how to materialize the related objects. Relationship span defines that the `ObjectContext` will automatically attach an entity when you have joined it to another attached entity.

The "Platinum Rule" About Related Entities That Are Attached or Detached from the ObjectContext

Understanding that the `ObjectContext` manages objects for each relationship that exists between a pair of entities can help you to better understand how the context deals with entities that are detached from the context. Remember that when an entity is detached, the context stops tracking its changes and stops managing its relationships. This is because the context destroys the `ObjectStateEntry` for that entity and any relationships that are dependent on that `ObjectStateEntry`.

The Entity Framework has a "golden rule" that it will never do something you didn't explicitly tell it to do. The Entity Framework's deferred loading, which you learned about in Chapter 4, is a great example of this, whereby the API won't make a trip to the database without your explicit instruction.

Because of this rule, there is one form of nonrequested behavior that may surprise you when detaching an entity from the context. The behavior is the result of how the context manages the relationships between entities. If you detach an entity from the context and that entity is part of a graph, the entity also gets detached from its graph because the context can no longer manage its relationship.

The fact that the context manages the relationship objects mandates that an object graph must be completely in or completely out of the `ObjectContext`. For example, if you have a customer graph that consists of a `Customer` with `Orders` and `OrderDetails` and you detach the `Customer` from the `ObjectContext`, the `Customer` will be disconnected from the rest of the entities in the graph. Because the relationship objects that involved that `Customer` were destroyed along with the `ObjectStateEntry` for that object, this

means you can no longer traverse from the customer, which is not in the context, to the orders, which are in the context.

Conversely, if you have an entity that is not in the `ObjectContext` and you join it to an entity that is in the `ObjectContext`, to follow the rule that the entire graph must be either in or out of the context the detached entity will automatically be attached to the context in order to be part of the graph.

You will see this behavior repeated throughout this chapter as you look at the features and functionality regarding relationships.

Where Does the Platinum Rule Come From?

When I first noticed that an entity had been attached to the `ObjectContext` even though I hadn't explicitly called for this in my code, I was a little taken aback because I had trusted the Entity Framework not to break what I thought of as "the golden rule": that is, it would not do anything I hadn't specifically told it to do. Once I understood why the entities were being automatically attached to the context when I was attaching them to a graph—and why that was necessary—I determined that this must be the Entity Framework's platinum rule, because it overruled the golden rule.

The Relationship Manager and the IRelatedEnd Interface

Along with `ObjectStateManager`, Object Services provides a Relationship Manager to perform the tasks pertaining to relationships between entities being managed by the context. The Relationship Manager keeps track of how entities attached to the `ObjectContext` are related to each other. It's able to do this with the methods and properties that `EntityCollection` and `EntityReference` share through the `IRelatedEnd` interface, which they both implement. `IRelatedEnd`'s methods include `Add`, `Attach`, and `Load`, among others. When these methods are called, or when one entity is simply set to another entity's navigation property (e.g., `myAddress.Contact=myContact`), the Relationship Manager kicks in.

This may sound complex, but it is necessary so that Object Services has a dependable way to manage the many relationships that could exist at any given time. As you create and delete entities, attach and detach entities, and modify relationships, the Relationship Manager is able to keep track of all of this activity. When it comes time to call `SaveChanges`, the Relationship Manager plays a role that is just as important as that of `ObjectStateManager`. All of those updates you witnessed, in which related objects were taken care of automatically, were handled by the Relationship Manager. To have the flexibility that the Entity Framework provides at the coding level, it is necessary to have this complexity at lower levels.

With an understanding of how things are working at the lower levels, interaction with related objects should become much easier to comprehend, anticipate, and implement.

The Relationship Manager provides related objects through late binding

One of the jobs of the Relationship Manager is to "serve up" related entities when they are attached to the `ObjectContext`. When you navigate to a related entity—for example, by requesting `myAddress.Contact`—the Relationship Manager will identify the existing relationship between the `Address` and `Contact` entities, find the correct `Contact` entity in the `ObjectContext`, and return it.

A related object that the `ObjectContext` is not managing is seen as any other property in .NET. A call to `myAddress.Contact` when `myAddress` and its `Contact` are not attached to the context will merely return the `Contact` as a regular property.

Experimenting with Relationship Span

Here's a geeky test so that you can see how some of the plumbing works. Looking at this in detail will give you a better understanding of how the Entity Framework manages relationships and why some of the rules that might not otherwise make sense exist.

Perform a query against the model that retrieves a single `Reservation`, as shown in Example 15-2.

Example 15-2. Retrieving a single entity

VB
```
Dim res=context.Reservations.First
```

C#
```
var res = context.Reservations.First();
```

Grab the new `Unchanged ObjectStateEntries` in the context, as shown in Example 15-3.

Example 15-3. Inspecting the ObjectStateEntries

VB
```
Dim osm = context.ObjectStateManager
Dim oses = osm.GetObjectStateEntries(EntityState.Unchanged)
```

C#
```
var osm = context.ObjectStateManager;
var oses = osm.GetObjectStateEntries(EntityState.Unchanged);
```

Now take a look at these `ObjectStateEntries` in the debugger, shown in Figure 15-7.

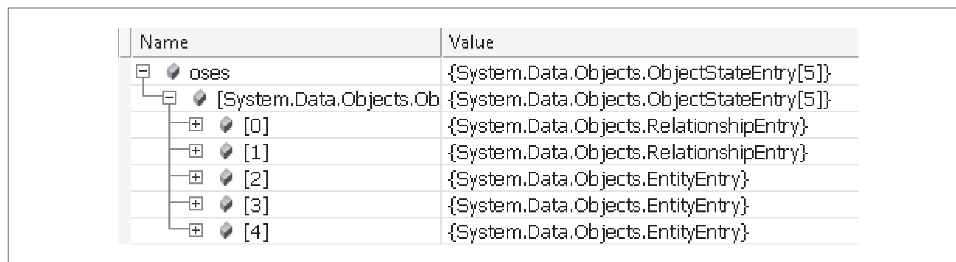


Figure 15-7. ObjectStateEntries in the C# debugger

You queried for a single entity, but three seem to be in the context. In fact, only the first `EntityEntry` is an actual entity. It's the `Reservation` you queried for, as you can see in Figure 15-8.

[2]	{System.Data.Objects.EntityEntry}
Entity	{BAGA.Advanced.Reservation}
EntityKey	"EntitySet=Reservations;ReservationID=1"
EntitySet	{Reservations}
IsKeyEntry	false
IsRelationship	false
ObjectStateManage	{System.Data.Objects.ObjectStateManager}
State	Unchanged
System.Data.IEntity	false
System.Data.IEntity	{System.Data.Objects.ObjectStateManager}
System.Data.Objec	Unchanged
Wrapper	null

Figure 15-8. The ObjectStateEntry for the queried Reservation

The last three `EntityEntries` are placeholders for the `EntityKey`s of the related data—for example, the related `Contact` shown in Figure 15-9.

[3]	{System.Data.Objects.EntityEntry}
Entity	null
EntityKey	"EntitySet=Contacts;ContactID=547"
EntitySet	{Contacts}
IsKeyEntry	true
IsRelationship	false
ObjectStateManage	{System.Data.Objects.ObjectStateManager}
State	Unchanged

Figure 15-9. An ObjectStateEntry that is a placeholder for an entity even though the entity is not in the cache

Even if a related entity doesn't exist in memory, the `ObjectContext` needs to be aware of any relationships that an existing entity (the `Reservation`) might have. That's because of the rule (created to cover all scenarios) that states that when a reservation is deleted, the relationship to its contact must also be deleted. This makes sense when both entities have been pulled into the `ObjectContext`, but not when only the reservation is in there.

While designing the Entity Framework, its creators decided it was safer to have an all-encompassing rule so that unexpected edge cases wouldn't result in errors. However, to satisfy the rule that the relationship must be deleted, the relationship must first exist. The pseudoentities were created during the query so that the relationships could be created without developers having to pull down additional data to satisfy the rule.

This is why these extra entries exist—one for the `Contact`, one for the related `Trip`, and one for the `UnpaidReservation` entity (which you created in Chapter 13). If the true

entity is brought into the context at some point, the `Entity` value of the entry will be updated. A private property for these entries denotes them as `EntityKeyEntries`; however, you won't see that name displayed in the debugger.

As you read through this chapter, this knowledge will help you better understand some of the rules and behavior surrounding relationships.

Understanding Navigation Properties in Entity Objects

On their own, navigation properties are almost meaningless. They are completely dependent on an association to provide access to related data. The navigation property does nothing more than define which association endpoint defines the start of the navigation and which defines the end.

A navigation property is not concerned with whether the property leads to an entity or an `EntityCollection`. The multiplicity in the association determines that, and the results are visible in the object instances where the navigation property is resolved. The property is resolved either as an entity plus an `EntityReference`, as with the `Contact` and `ContactReference` properties of the `Address` entity in Figure 15-10, or as an `EntityCollection`, as with the `Addresses` property of the `Contact` entity. You can also see this by looking in the generated classes for the model. The Entity Framework still needs to make this determination as it is materializing objects from query results, and then populate the objects correctly.

EntityReference properties

Navigation properties that return `Entity` and `EntityReference` properties need to be addressed together because they come as a pair, even though both may not be populated. When the navigation property points to the "one" or "zero or one" side of relationship of a 1:1 or 1:* relationship, that property is resolved as two public properties. One property contains the actual entity, and the other property contains a reference to the entity. This reference contains the related entity's `EntityKey`, which is relative to a foreign key in a database. The `EntityKey` provides just enough information about that entity to be able to identify it when necessary. When you execute a query that returns addresses, the `ContactID` from the `Addresses` table is used to construct the `EntityKey` for `ContactReference`. Even if the contact does not exist in memory, the `ContactReference` property provides the minimal information about the `Contact` that is related to the `Address`.

EntityReference.Value. The value object of an `EntityReference` shows up in two places. The first is the navigation property (`Reservation.Customer`) and the second is within the `EntityReference` property (`Reservation.CustomerReference`). You can see this in Figure 15-11, where the `Customer` is loaded into the `ObjectContext` and therefore is hooked up with the `Reservation` entity.

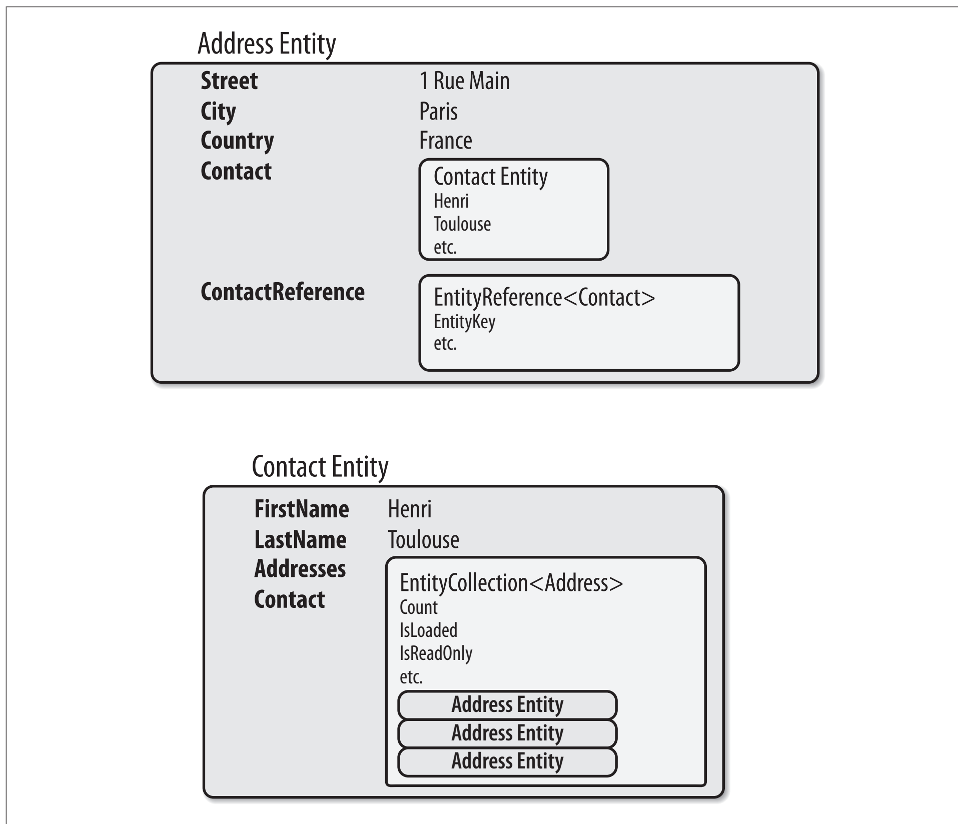


Figure 15-10. Resolving navigation properties as entities, EntityReferences and EntityCollections

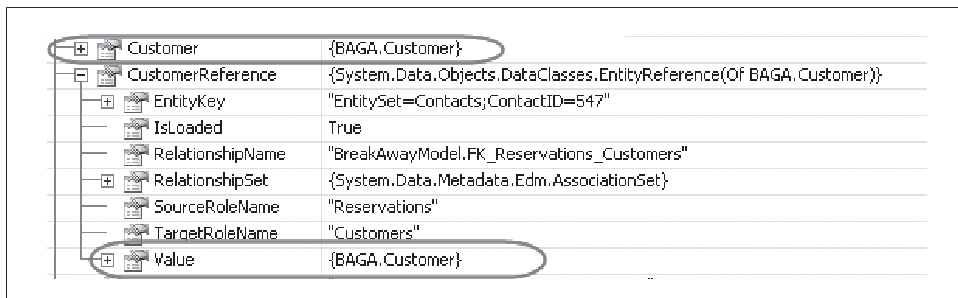


Figure 15-11. The Customer property of the Reservation entity as the actual property and as the value of the EntityReference property, CustomerReference

What if there is no EntityReference. In many scenarios, the "one" side of a relationship is required, such as the constraint in the BreakAway model that says a reservation can't exist without a related customer. However, in some relationships the target is 0..1, meaning the related end isn't required (e.g., if a customer's preferred activity does not

need to be specified). In the case where there is nothing on that end, the navigation property (`Customer.PrimaryActivity`) will be null. The `EntityReference` will exist, but its `EntityKey` and `Value` will be null (`Nothing` in VB), as shown in Figure 15-12 and Figure 15-13.

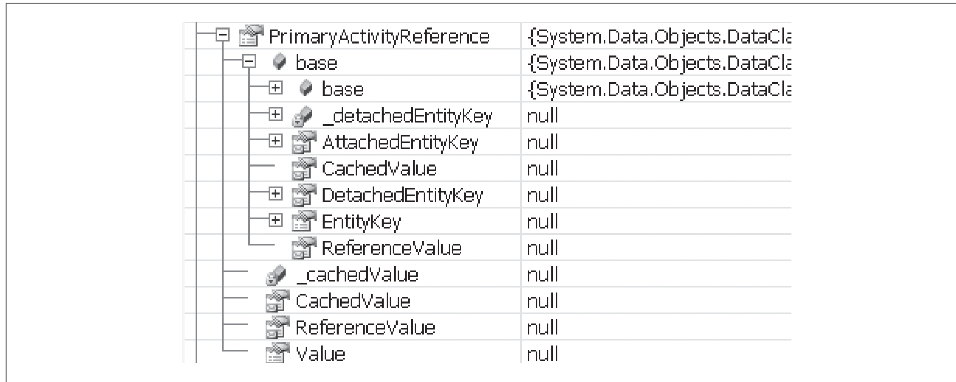


Figure 15-12. An unpopulated EntityReference, which will have no EntityKey and Value

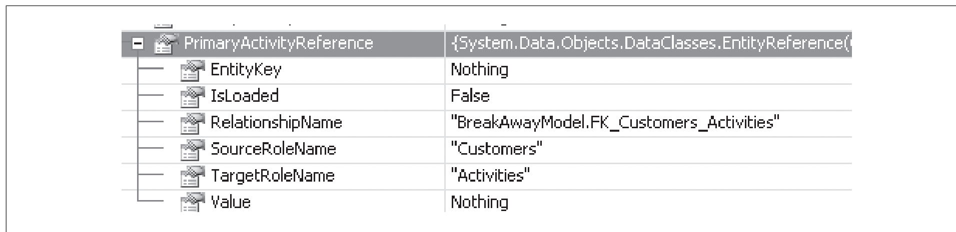


Figure 15-13. The VB debug view of the empty EntityReference

EntityCollection properties

The other type of `IRelatedEnd` is an `EntityCollection`. `EntityCollection` is a generic class that is a container for entities of a particular type and is used by a navigation property to point to the "many" end of an association. `Contact.Addresses` is an `EntityCollection` of `Address` types.

`EntityCollection` is not based on other `Collection` classes in .NET. In fact, it implements an Entity Framework interface in the `System.Data.Objects.DataClasses` namespace, called `IRelatedEnd`.

You have worked with `EntityCollection`s in many of the examples in this book already. Although you'll probably work with `EntityCollection` most frequently through a navigation property, it is also possible to instantiate an `EntityCollection` in memory and work with it directly.

You cannot attach an `EntityCollection` to an entity through its navigation property—for example, with:

```
MyContact.Addresses=myAddressEntityCollection
```

The `EntityCollection` *is* the navigation property (when the target is the "many" side of an association). You need to insert items into the collection itself, which you can do explicitly or by setting the `EntityReference` on the child object, which you'll see next. You can also remove items from the collection as needed.

Referential Integrity and Constraints

It is possible to place constraints in both the database and the EDM to ensure the integrity of the data. The point at which these constraints are checked in the pipeline varies.

Many developers approach the Entity Framework with the assumption that these constraints will be checked when their code is manipulating the entities. For example, if you delete an order, when you call `ObjectContext.DeleteObject` it would be nice to have the Entity Framework tell you "Hey, there are still line items for this order. You can't do that."

Constraints that are not checked until they hit the database

Unfortunately, many constraint checks must be deferred to the database, because it is common for some dependent data not to be loaded into your application. Even if the Entity Framework did alert you to those details and you removed them as well, what if the database contains more order detail data that you hadn't loaded for some reason? This would make you feel uncomfortable deleting the order and sending that instruction to the database, which would throw an error anyway because of the other details that are still present.

Other constraint checks that, for the same reason, can't be made on the client side are for uniqueness, primary keys, and foreign keys. An `EntityReference` might contain a key value for data that is not loaded into the `ObjectContext` but that does exist in the database, so only the database can provide that check.

In Chapter 4, you saw how database constraints were specified in the Store Schema Definition Layer (SSDL) portion of the model, which only declares the existence of the constraints in the database. It is also possible to apply additional constraints on the Conceptual Schema Definition Layer (CSDL) side, but these are generally meant to validate the model and won't have much impact on updates.

This is why it's important to catch exceptions and deal with them when you call `SaveChanges`.

Constraints that the Entity Framework checks when SaveChanges is called

The model itself can perform checks on the multiplicity of associations; however, this does not happen until SaveChanges is called.

For example, the 1:* relationship between Contact and Address means you can have one and only one contact related to an Address—not five and not zero.

However, it is possible to create a new Address in code without assigning a Contact or ContactReference, as shown in Example 15-4. When you add the Address to the context, the Entity Framework can't possibly know whether you plan to create a relationship.

Example 15-4. Code that will not incur a constraint check

VB
```
Dim add = Address.CreateAddress(0, "Home", Now())
context.AddToAddresses(add)
```

C#
```
var add = Address.CreateAddress(0, "Home", Now());
context.AddToAddresses(add);
```

But when it comes time to call SaveChanges, after any custom SavingChanges logic has been implemented, it's pretty obvious that if there is no relationship by now, there never will be. That's when an UpdateException is thrown before the command is even created to send to the database. The UpdateException provides the following explicit message, which is great for debugging and for logging:

```
Entities in 'BAEntities.Addresses' participate in the 'FK_Address_ContactSet'
relationship.
0 related 'Contact' were found. 1 'Contact' is expected.
```

You should be able to catch a scenario like this before your code goes into production. Otherwise, you'll either want to check for this constraint yourself in business rules (e.g., in SavingChanges) or, as a last resort, catch the exception and either deal with the orphaned address or ask the user to do something about it.

We will explore UpdateException along with other Entity Framework exceptions in Chapter 18.

Deletes and Cascading Deletes

Most databases support cascading deletes, which are used to automatically (and unquestionably) delete all of the children of a parent when that parent record is deleted. Cascading deletes are supported in the Entity Framework as well, but with caveats.

The biggest caveat is that the Entity Framework can perform cascading deletes only on objects that are in memory. This could cause problems if a database constraint is enforcing referential integrity, and if the database does not also have a cascading delete

set up on the same set of data. In this case, if the database still contains children, an error will be thrown.

Cascading deletes in the database

Referring back to Figure 15-5, a cascading delete in the database is defined in the `Key` definitions of the "child" or dependent table. If data in the related table is deleted, all of the related rows of the dependent table will be automatically deleted.

Cascading deletes in the EDM

You can also define a cascading delete in the EDM. The Designer does not support this, and therefore you need to do it manually in the XML Editor. This facet is added to the `Association` definition in the CSDL.

The EDM Wizard can identify cascading deletes in a database. Microsoft's sample database, AdventureWorksLT, defines cascading deletes in many relationships, including the relationship between `SalesOrderDetail`s and its "parent," `SalesOrderHeader`s. The wizard recognizes and includes that cascading delete in the `Association` definition, as shown in Example 15-5.

Example 15-5. An Association's OnDelete action set to Cascade as defined in the CSDL

```
<Association Name="FK_SalesOrderDetail_SalesOrderHeader_SalesOrderID">
  <End Role="SalesOrderHeader"
       Type="AdventureWorksLTModel.SalesOrderHeader" Multiplicity="1">
    <OnDelete Action="Cascade" />
  </End>
  <End Role="SalesOrderDetail"
       Type="AdventureWorksLTModel.SalesOrderDetail" Multiplicity="*"/>
  <ReferentialConstraint>
    <Principal Role="SalesOrderHeader">
      <PropertyRef Name="SalesOrderID" />
    </Principal>
    <Dependent Role="SalesOrderDetail">
      <PropertyRef Name="SalesOrderID" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

The following code spins up a new `ObjectContext` and queries for the first `SalesOrderHeader` along with its `SalesOrderDetail`s. In this case, two `SalesOrderDetail`s are returned along with the `SalesOrderHeader`. Then the `SalesOrderHeader` is deleted from the context, after which the `ObjectStateManager` is queried for all objects whose state is `Deleted`:

VB

```
Using context As New AWLTEntities
  Dim ord = context.SalesOrderHeader.Include("SalesOrderDetail").First
  context.DeleteObject(ord)
  Dim oses = context.ObjectStateManager.GetObjectStateEntries _
             (EntityState.Deleted)
```

```
  context.SaveChanges
End Using
```

C#

```
using (AWLTEntities context = new AWLTEntities ())
{
  var ord = context.SalesOrderHeader
                 .Include("SalesOrderDetail").First();
  context.DeleteObject(ord);
  var oses = context.ObjectStateManager
                 .GetObjectStateEntries(EntityState.Deleted);
  context.SaveChanges();
}
```

In the `ObjectStateEntries` that are returned, you can see that three entities are deleted: the `Order` entity and the two `Detail` entities. At the same time the context deleted the entities, it also needed to delete all of the association instances (`RelationshipEntries`) that existed. Two of those seven `RelationshipEntries` are for relationships between the order and each detail record. The rest are for other associations that exist within those entities, such as Detail to Product, Order to Customer, Order to Billing, and Shipping Address.

When `SaveChanges` is called, explicit commands are sent to the database to delete the `SalesOrderHeader` and the two `SalesOrderDetail`s, so although the database will perform a cascading delete, no more detail records will be available to delete. However, if another `SalesOrderDetail` for this order is added to the database in the meantime, the database's cascading delete will delete it.

Recommendation: Cascade in both the model and the database, or in neither

Although you can add the `OnDelete` action to an association when the cascade is not defined in the database, this is not recommended because doing so will create incorrect expectations on the part of the developer, and unpredictable results. The recommendation is to use the `OnDelete` action in the database as a reflection of its definition in the database. If it exists in the database, it should exist in the model. If it does not exist in the database, it should not exist in the model. You can, of course, ignore the recommendations as long as your code is prepared for the possible repercussions.

Defining Relationships Between Entities

Now that you have had a tour through the plumbing of how relationships work in the Entity Data Model and in instantiated objects, it's time to see how you can impact relationships in your code.

Anytime you set one entity as the property of another (for example, `Reservation.Customer=aCustomer`) or add an entity to an `EntityCollection` property of an entity (for example, `Reservations.Payments.Add(aNewPayment)`) you are defining a relationship. In turn, the `ObjectContext` will create a new relationship object (technically, a `RelationshipEntry` to represent this relationship.

You can create relationships between entities in a number of ways. You saw several of them in the applications you built in previous chapters.

The CLR Way: Setting a Navigation Property

The simplest way to create relationships between entities is to create the relationship the CLR way—by setting one entity as the property of another entity:

```
MyAddress.Contact = myContact
```

If you are starting with the child entity (e.g., `MyAddress`), this is pretty obvious. But if you are starting with the contact and you want to add the address to its collection, there's no reason not to switch your perspective and simply set the address's `Contact` property. It's simpler and has the same effect.

This covers the most common scenarios, though you should be aware of the following subtleties:

- If both objects are detached, no relationship object will be created. You are simply setting a property the CLR way.
- If both objects are attached, a relationship object will be created.
- If only one of the objects is attached, the other will become attached (thanks to the "platinum rule") and a relationship object will be created. If that detached object is new, when it is attached to the context its `EntityState` will be `Added`.

Setting an EntityReference Using an EntityKey

If you want to populate an `EntityReference` but you do not have an entity in hand, you can create the `EntityReference` using only an `EntityKey`. This requires that you know the value of the `EntityReference`'s key. And in fact, you did this when creating reservations in some of the example applications earlier in the book.

This allows you to create a foreign key for an entity without having the related data in memory. For example, in the customization you made to `SavingChanges`, you used an `EntityReference` to set the default `CustomerType` of a new `Customer`. Example 15-6 shows the relevant code from that method.

Example 15-6. Defining an EntityReference with an EntityKey

VB
```
cust.CustomerTypeReference.EntityKey =
 New EntityKey("BAEntities.CustomerTypes", "CustomerTypeID", 1)
```

C#
```
cust.CustomerTypeReference.EntityKey =
 new EntityKey("BAEntities.CustomerTypes", "CustomerTypeID", 1);
```

This is especially handy, because although you can set default values for scalar properties in the model, it's not possible to set a default value for a navigation property. By setting the `CustomerTypeReference.EntityKey`, you can define the `CustomerType` without

having to load the actual CustomerType object into memory. When the Customer is saved, the CustomerTypeReference is resolved, and in the database the Customer record has its CustomerTypeID foreign key value set to 1.

Synchronizing EntityReference.Value and Its EntityKey

The ObjectContext automatically synchronizes the value of the EntityReference (when the entity exists in the ObjectContext) and the EntityKey of the EntityReference property. When both entities are attached, the EntityKey and Value will stay in sync. If you change one, the other will automatically change. However, you should be aware of the following scenarios regarding related entities that are not attached to the context:

- If both entities are detached, the EntityReference Value and EntityKey will not be synchronized because that is a function only the ObjectContext can perform automatically. It is possible to create an EntityReference whose EntityKey represents one entity and whose value represents another entity, however.
- When attaching an entity to the ObjectContext that has an EntityReference, the value (when it is not null) will take precedence, and if the Value and EntityKey are out of sync, the EntityKey will be updated to match the Value.

Loading, Adding, and Attaching Navigation Properties

There are a number of ways to populate navigation properties using methods provided by the IRelatedEnd interface, which EntityCollection and EntityReference implement. Depending on the type (collection or reference) from which you call the methods, the Entity Framework performs slightly different methods in the background to achieve the goal.

Rules, rules, rules

A lot of rules dictate when you can Load, Add, or Attach navigation properties. When you are unfamiliar with how relationships function in the Entity Framework, you will be more likely to unknowingly break these rules and hit an exception. As you settle into a better understanding of relationships, all of these rules that you are about to learn on the following pages will become second nature and you will no longer think of them as rules that you need to remember, but as features of the Entity Framework. However, looking at them now, along with the reason behind each rule, will help you get to that better level of understanding.

EntityReference.Load and EntityCollection.Load

Examples:

```
Contact.Addresses.Load
Address.ContactReference.Load
```

Although we discussed the `Load` method earlier in the book, here we will dig a little deeper.

You can call `Load` from entities that are attached to the `ObjectContext`. Calling `Load` will cause a query to be created and executed to retrieve the related data that `Load` requested. If you have a `Contact` and call `Contact.Addresses.Load`, all of the addresses belonging to that contact will be retrieved from the database.

One condition allows you to call `Load` from detached entities: when the entity was created as a result of a `NoTracking` query. Example 15-7 calls `Load` on a `Detached` entity.

Example 15-7. Calling Load on a detached entity

VB

```
Dim query = context.Reservations
query.MergeOption = Objects.MergeOption.NoTracking
Dim res = query.First
res.CustomerReference.Load()  '<--succeeds
```

C#

```
var query = context.Reservations;
query.MergeOption= System.Data.Objects.MergeOption.NoTracking;
var res = query.First();
res.CustomerReference.Load(); //<--succeeds
```

However, if you detach an entity from the context, calling `Load` will cause an exception to be thrown, as shown in Example 15-8.

Example 15-8. A failed attempt to call Load on a detached entity

VB

```
Dim query = context.Reservations
Dim res = query.First
context.Detach(res)
res.CustomerReference.Load()  '<--InvalidOperationException
```

C#

```
var query = context.Reservations;
var res = query.First();
context.Detach(res);
res.CustomerReference.Load();  //<--InvalidOperationException
```

Load provides behind-the-scenes query creation and execution

Calling the `Load` method on an `EntityCollection` or `EntityReference` will force the Entity Framework to create and execute a query against the data store to retrieve the related entities. This is why the code in Example 15-7 will fail. `Load` requires an `ObjectContext` to create and execute the query.

For example, `Contact.Addresses.Load` will cause a query to automatically run against the data store for all of the contact's addresses. The only overload for `Load` is to include `MergeOptions`, but you cannot provide a filter to control which of the contact's addresses are to be loaded from the database. If you have a conditional mapping defined for the `Address` entity, it will be honored.

> **Lazy Loading Versus Deferred Loading**
>
> Many data access frameworks, including LINQ to SQL, support implicit lazy loading by default. The Entity Framework does not. Instead, it supports explicit lazy loading. What's the difference? Implicit lazy loading does not require the developer to explicitly call a `Load` method. If the related data is not loaded at the time it is requested, it will be automatically retrieved for you. The Entity Framework, on the other hand, requires that you explicitly load the data. The thinking behind the Entity Framework's `Load` method requirement is the now familiar rule that the Entity Framework will not do something you did not ask it to do. Implicitly making connections and round trips to the database is costly. There seems to be a semantic difference in how some developers use the term *lazy loading*. You may hear people say that the Entity Framework supports lazy loading, but in that case they really mean explicit lazy loading, also known as *deferred loading*.
>
> Although version 1 of the Entity Framework doesn't support implicit lazy loading, the ADO.NET team's MSDN Code Gallery project provides a code example that shows how you can implement lazy loading in the Entity Framework. Also, optional lazy loading is targeted for the next version of the Entity Framework. You can read more on Product Manager Tim Mallalieu's blog post from June 24, 2008, at *http://blogs.msdn.com/timmall/*.

Understanding why Load cannot be called on Added entities

Because of the way `Load` creates and executes queries, there are other scenarios where you cannot call `Load` besides not being able to call it from a `Detached` entity. One of these may not make sense, but with some detective work, you can come to understand the reason. The scenario is that you cannot call `Load` on objects whose `EntityState` is `Added`.

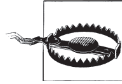
The short answer to "but why?" is that `Load` requires an `EntityKey` with an appropriate ID. The temporary `EntityKey` assigned to `Added` entities is insufficient. This makes sense if you want to load the children of an entity, but it does not seem to make sense if you wanted to load an `EntityReference` for a new entity. As an example, consider a new `Reservation` where you have identified the `Reservation`'s `Customer` using `CustomerReference.EntityKey`. If you have the `Customer`'s `EntityKey`, it seems very logical that you should be able to load the customer. Example 15-9 attempts to do this.

Example 15-9. A failed attempt to call Load on an entity whose EntityState is Added

VB
```
Dim newAddress = Address.CreateAddress(0, "Home", Now())
context.AddToAddresses(newAddress)
newAddress.ContactReference.EntityKey = _
   New EntityKey("BAEntities.Contacts", "ContactID", 100, Nothing)
newAddress.ContactReference.Load()  '<--InvalidOperationException
```

C#
```
var newAddress = Address.CreateAddress(0, "Home",Now);
context.AddToAddresses(newAddress);
newAddress.ContactReference.EntityKey =
```

```
 new EntityKey("BAEntities.Contacts", "ContactID", 100);
newAddress.ContactReference.Load(); //<--InvalidOperationException
```



If you receive an exception on the call to `CreateAddress` telling you it does not take three arguments, the problem is that `Address.TimeStamp` needs a default value in the model. Go back to the model and change the `Default` for `Address.TimeStamp` to `0x123`. If you enter it in the Properties window, don't put any quotes around this value.

The call to `Load` will throw an exception explaining that you can't call `Load` on an `Added` entity (with some additional details). The reason is that the query to load the data is not constructed the way you might expect.

If you were writing the query, you would probably use SQL like this:

```
SELECT * from Contact WHERE ContactID=100
```

But the Entity Framework does not construct `Load` queries in this way.

The Entity Framework needs to have a single way to construct `Load` queries for both `EntityReference` and `EntityCollection`. The common denominator between loading `EntityReference`s and `EntityCollection`s is the `EntityKey` of the calling entity; therefore, the calling entity needs to exist in the database as well. Look at the native T-SQL query constructed to load the reservation of a payment based on the code in Example 15-10.

Example 15-10. Using Load on an EntityReference

VB
```
Dim firstPayment = context.Payments.First
firstPayment.ReservationReference.Load()
```

C#
```
var firstPayment = context.Payments.First();
firstPayment.ReservationReference.Load();
```

Even though the `Payment` already has the `EntityKey` of the `Reservation` it needs, the native query shown in Example 15-11 is based on only the `PaymentID` value.

Example 15-11. The T-SQL that results from the Load method in Example 15-9

```
SELECT 1 AS [C1],
[Extent2].[ReservationID] AS [ReservationID],
[Extent2].[ReservationDate] AS [ReservationDate],
[Extent2].[ContactID] AS [ContactID],
[Extent2].[EventID] AS [EventID]
FROM  [dbo].[Payments] AS [Extent1]
INNER JOIN [dbo].[Reservations] AS [Extent2]
    ON [Extent1].[ReservationID] = [Extent2].[ReservationID]
WHERE ([Extent1].[ReservationID] IS NOT NULL)
   AND ([Extent1].[PaymentID] = @EntityKeyValue1)

Parameter: @EntityKeyValue1=1
```

The query is doing the following, in the order shown:

1. Using the `Payment`'s `PaymentID` value to locate the `ReservationID`
2. Performing a `JOIN` on the `Reservation` table based on the common `ReservationID`
3. Returning the appropriate fields from the `Reservation` table to map to the `Reservation` entity

What happens when you load children? In Example 15-12, `Load` is called on an existing parent (`Reservation`) in order to load its children (`Payments`).

Example 15-12. Using Load to load an EntityCollection

VB
```
Dim firstRes = context.Reservations.First
firstRes.Payments.Load()
```

C#
```
var firstRes = context.Reservations.First();
firstRes.Payments.Load();
```

When `Load` is called, you again can see, in the native query shown in Example 15-13, that the calling entity's key ID is used for the query. In this case, that is the `ReservationID` and its value is `1`.

Example 15-13. The T-SQL that results from calling EntityCollection.Load

```
SELECT AS [C1],
[Extent1].[PaymentID] AS [PaymentID],
[Extent1].[PaymentDate] AS [PaymentDate],
[Extent1].[Amount] AS [Amount],
[Extent1].[ModifiedDate] AS [ModifiedDate],
[Extent1].[PaidinFull] AS [PaidinFull],
[Extent1].[ReservationID] AS [ReservationID]
FROM [dbo].[Payments] AS [Extent1]
WHERE ([Extent1].[ReservationID] IS NOT NULL)
AND ([Extent1].[ReservationID] = @EntityKeyValue1)

Parameter:
@EntityKeyValue1=1
```

Now this query seems more natural—it looks into the `Payments` table for all payments with `ReservationID=1`.

Because of the need to construct both queries (as a result of `EntityReference.Load` and `EntityCollection.Load`) using the ID of the calling entity, the calling entity needs to be in the database already. In the case of an `Added` entity, there's a very good chance that the calling entity does not yet exist in the data store. So, it's a reasonable rule to completely disallow `Load` on `Added` entities.

In the course of deciphering this rule, you now have gained a lot of insight into the logic behind `Load` queries. For some of us, this is truly geeky fun. Others might want a short reprieve before continuing.

Now let's move on to other ways of defining relationships between entities.

EntityCollection.Add

Example:

```
Contact.Addresses.Add(myNewAddress)
```

You use `Add` to add items to an `EntityCollection`. You can use `Add` only on `EntityCollection`. It isn't valid with `EntityReference`.

Again, because of the varying states of an entity, you can use `Add` in a few different ways.

Adding new entities that are detached

You can use `Add` to add a new entity that is not yet attached to the `ObjectContext`. `Add` will first add the entity to the `ObjectContext` and will then add it to the `EntityCollection` of the calling entity. This requires that the entity have no `EntityKey`; otherwise, it will throw an exception. Any entity with an `EntityKey` is presumed to have come from the data store, and therefore it can't be treated as a new record.

For instance, the code in Example 15-14 queries for a `Customer` from the database, creates a new `Address` in memory, and then adds the `Address` to the `Customer`'s `Addresses` collection.

Example 15-14. Adding a new, detached entity to another entity

VB

```
Using context As New BAEntities
  Dim con = (From c In context.Contacts.Include("Addresses")).First
  Dim newAdd = Address.CreateAddress(0, "home", Now);)
   With newAdd
      .Street1 = "1 Main"
      .City = "Hamburg"
      .StateProvince = "NY"
    End With
  con.Addresses.Add(newAdd)
End Using
```

C#

```
using (BAEntities context = new BAEntities())
{
  var con = (from c in context.Contacts.Include("Addresses") select c)
            .First();
  var newAdd = Address.CreateAddress(0, "home", Now);
  newAdd.Street1 = "1 Main";
  newAdd.City = "Hamburg";
  newAdd.StateProvince = "NY";
  con.Addresses.Add(newAdd);
}
```

When this is complete, `newAdd.Contact` returns the `Contact` entity and `newAdd.ContactReference` is populated with an `EntityKey` and the `Value`. The `newAdd`

Figure 15-14. A new entity that was added to the EntityCollection of a detached entity

object instance is attached to the context with a temporary `EntityKey` and an `EntityState` of `Added`.

Adding new or existing entities that are attached

You can use `Add` to add entities that are already attached to the `ObjectContext`, regardless of whether or not they are new.

Adding entities to the EntityCollection of a detached object

If the calling entity is detached from the context, `Add` will be treated as a CLR method, as shown in Example 15-15.

Example 15-15. Adding to the EntityCollection of a detached entity

VB

```
Dim firstRes = context.Reservations.First
context.Detach(firstRes)
Dim newPayment = New Payment
newPayment.Amount = 100
firstRes.Payments.Add(newPayment)
```

C#

```
var firstRes = context.Reservations.First();
context.Detach(firstRes);
var newPayment = new Payment();
newPayment.Amount = 100;
firstRes.Payments.Add(newPayment);
```

In this case, the `newPayment` entity will be added to the `Payments` collection and you'll be able to navigate from the `Payment` to the `Reservation` (`newPayment.Reservation`). But because the `ObjectContext` is not involved, no `Relationship` object is created, and therefore `newPayment.ReservationReference.EntityKey` will be null, as shown in Figure 15-14.

Attach and Remove

Examples:

```
Contact.Addresses.Attach(myAddress)
Address.ContactReference.Attach(contact)
```

In addition to `ObjectContext.Attach`, an `Attach` method exists for `IRelatedEnd`, which you can use for `EntityCollection`s and `EntityReference`s.

Using the `Attach` method, you can define relationships between entities that already exist in the `ObjectContext` but that have not been connected automatically.

`EntityCollection.Attach` is more commonly used than `EntityReference.Attach`. Although it is technically possible to use the latter, it is not generally necessary because it is so much easier to merely assign the `EntityReference` value.

`EntityCollection.Remove` is used to remove an entity from a collection. It will stay in the `ObjectContext`, but will no longer be related to a parent in that particular association. There is no `Remove` method for `EntityReference`.

When removing entities from an `EntityCollection`, be careful not to orphan entities. If you leave a child entity without a required parent, you will get an `EntityReferenceException` when you call `SaveChanges`.

Use Attach only when a relationship exists in the data store

`EntityCollection.Attach` is necessary in only a few edge cases, which occur when you are attempting to reconstruct a graph from data that may have otherwise lost its `EntityReference.EntityKey`. This could happen, for example, if you are serializing groups of data to move across tiers. Although binary serialization and Windows Communication Foundation (WCF) serialization maintain graphs, XML serialization (used in ASMX Web Services) does not, and therefore it is necessary to transmit the data in a different format. If you have the Entity Framework in the client and service, chances are the `EntityReference.EntityKey` property is intact. But if your client application is not using the Entity Framework (as in the examples in this chapter), the `EntityReference`s won't exist. This is a scenario in which `EntityCollection.Attach` would help you rebuild the graph from preexisting data when it gets back to the service. An entity in this state—for example, an `Address` that has an `EntityKey` because it came from the data store, but whose `Contact`, `ContactReference.EntityKey`, and `ContactReference.Value` properties are null—is the correct target of `Attach`.

Other than this, you'll encounter a lot of rules if you try to use `Attach` where it doesn't belong. Here are a few places where you might think you want to use `Attach`, but you will run into trouble. Most of these problems occur when you are not following the main purpose of `Attach`, which is to connect entities that are already attached to the `ObjectContext` and are not new.

- You cannot use Attach to attach another EntityCollection; however, it is possible to attach an IEnumerable (discussed shortly). Otherwise, you need to attach one entity at a time.
- You cannot use Attach when the EntityState of either end is Detached. ObjectContext is required to work out the relationship. Do an ObjectContext.Attach first.
- You cannot use Attach when the ends are in different contexts.
- You cannot use Attach to attach entities whose EntityState is Added. You'll want to use Add in this case.
- You cannot use Attach to attach entities whose EntityState is Deleted. It's not possible to create a RelationshipEntry for a deleted entity.
- You cannot use Attach to reassign entities to another graph.

Attach Versus Add

Whether you are calling Attach or Add from ObjectContext or from IRelatedEnd, it's important to understand the difference between these two methods.

Add will create an ObjectStateEntry with EntityState=Added. If you are using the ObjectContext.Add method, the EntityState is impacted on the entity that you are adding. If you are calling the EntityCollection.Add method, it is the relationship whose EntityState is affected. When SaveChanges is called the entity that was added to the context will be inserted in the database and the relationship that was created will be persisted in the database.

Attach will create an ObjectStateEntry whose EntityState=Unchanged. If you are calling the ObjectContext.Attach method, the resulting ObjectStateEntry will have a clean slate. SaveChanges will do nothing with that entity. If you are calling EntityCollection.Attach, a RelationshipEntry will be created and it will have a clean slate. Nothing will happen with respect to this relationship when SaveChanges is called.

When you use the CLR properties to set the entity value, Add and, if necessary, Remove are called under the covers.

Moving an Entity to a New Graph

Often you will want to move an entity from one graph to another, perhaps for the simple reason that an end user applied a payment to the wrong reservation.

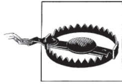
In such a case, you can simply reassign the Reservation property:

```
myPayment.Reservation=myReservation
```

or add the payment to the Payments EntityCollection of the new reservation:

```
myReservation.Payments.Add(myPayment)
```

The ObjectContext will resolve the existing relationship. If you were to inspect the relationships in the context after the change, you would find a RelationshipEntry linking the payment to the original reservation and another RelationshipEntry linking the payment to the new reservation. The EntityState of the first entry would be Deleted, while the EntityState of the second entry would be Added.



Remember that Attach will not work here. Attach is for creating a relationship in the context to reflect a relationship that already exists in the database. In the same way that ObjectContext.Attach renders an entity's EntityState as Unchanged, this Attach will render the relationship's EntityState as Unchanged. Therefore, the relationship modification you performed will not be applied during the call to SaveChanges.

Learning a Few Last Tricks to Make You a Relationship Pro

If you spend any amount of time in the MSDN Forums for the Entity Framework, you may recognize two questions that are asked frequently. The first is "How can I filter the children that are returned when I use Load?" and the second is "How can I get the foreign key value of a navigation property?"

Now that you have learned so much about relationships in the Entity Framework, you will be able to understand the solutions to both of these FAQs.

The first is solved with a little-known method called CreateSourceQuery. The second is solved by digging down into the EntityReference to get at its properties. Here is how to perform both tricks.

Using CreateSourceQuery to Enhance Deferred Loading

Not only can you attach an entity to an IRelatedEnd, but you can also attach an IEnumerable when you are calling EntityCollection.Attach. An EntityCollection is not an IEnumerable, which is why you can't just attach another EntityCollection. But if you want to attach a number of entities at once, you can wrap them in something as simple as a list or the results of another query.

Additionally, you can use this as an alternative to EntityCollection.Load because it will give you some flexibility regarding what you are loading. You can use CreateSourceQuery to create queries on the fly for an Attach method, though you'll use it a bit differently for EntityCollection.Attach and EntityReference.Attach.

The most efficient way to create an IEnumerable for attaching or loading entities into an EntityCollection is to use the CreateSourceQuery method. Like the Load method, CreateSourceQuery will take care of the query creation and execution for you, leveraging the existing ObjectContext.

As an example, if you have a customer in the ObjectContext and you want to get that customer's reservations, you could call the following:

```
myCust.Reservations.Load()
```

This would load all of the reservations for that customer.

However, if you want to filter those reservations, you can use `CreateSourceQuery` instead, as shown in the following code:

VB
```
Dim cust=context.Contacts.OfType(Of Customer).First
Dim sq = cust.Reservations.CreateSourceQuery() _
 .Where(Function(res) res.ReservationDate > New Date(2008, 1, 1))
cust.Reservations.Attach(sq)
```

C#
```
var cust=context.Contacts.OfType<Customer>().First();
var sq = cust.Reservations.CreateSourceQuery()
 .Where(r => r.ReservationDate > new DateTime(2008, 1, 1));
cust.Reservations.Attach(sq);
```

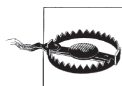
The query will execute when the `CreateSourceQuery` is called. Now only the subset of reservations for that customer will be loaded.

You can also use `CreateSourceQuery` to filter on types. In the following code, `Attach` is being used with an `EntityReference`, which will not take `IQueryable`. Instead, you need to pass in an object, which you can get using the `FirstOrDefault` query method. Since `Attach` will throw an exception if you attempt to pass in a null, you need to test for null before calling `Attach`:

VB
```
For Each a In addresses
  Dim sq = a.ContactReference.CreateSourceQuery() _
            .OfType(Of Customer).FirstOrDefault
  If Not sq Is Nothing Then
    a.ContactReference.Attach(sq)
  End If
Next
```

C#
```
var addresses = context.Addresses.Take(5);
foreach (var a in addresses)
{
  var sq = a.ContactReference.CreateSourceQuery()
            .OfType<Customer>().FirstOrDefault();
  if (sq != null)
    a.ContactReference.Attach(sq);
}
```

With this code, only customers will be loaded.



Watch those resources. Just like `Load`, the preceding query will be run whether or not the contact is a customer, so you may end up with a lot of wasted trips to the database. Consider your data, your resources, and your application's needs. In some scenarios, you may find yourself better off making one big query for customers and not doing this explicit lazy loading.

Getting a Foreign Key Value

Foreign keys do not exist in an EDM. Foreign keys from a database table are replaced with navigation properties when an entity is created. Because you can traverse relationships in the EDM without having to perform `JOIN`s, the need for foreign keys is greatly reduced. However, at times you may want to have access to a foreign key value such as `Address.ContactID`.

Although the value is not exposed directly, if you have the entity or the `EntityReference`, you can get to that value. For example, if the `Address.Contact` property is populated, you can simply request `Address.Contact.ContactID`.

If the `ContactReference` property is populated, you could drill into the `EntityKey` and extract the value. Don't forget that an `EntityKey` is composed of a collection of key/value pairs. Although a `Contact` entity may have only one key property (`ContactID`), in plenty of cases multiple properties are combined to make an `EntityKey`, just as you can use multiple fields in a database table to create a primary key.

Example 15-16 shows how to retrieve the `ContactID` from an `Address` entity.

Example 15-16. Retrieving the ContactID from an Address entity

VB
```
address.ContactReference.EntityKey.EntityKeyValues _
 .Where(Function(k) k.Key = "ContactID")
```

C#
```
address.ContactReference.EntityKey.EntityKeyValues
 .Where(k=> k.Key == "ContactID")
```

An extension method that can return a foreign key value for any `EntityReference` would be handy here. A simple approach requires that the developer knows about the `EntityReference`, and this particular method will presumptuously return the value of the first key, as shown in Example 15-17.

Example 15-17. Returning the foreign key value from an EntityReference property

VB
```
<Extension()> _
Private Function ForeignKey(ByVal entRef As EntityReference) As Int32
  Dim keyval As Int32
  If Int32.TryParse _
    (entRef.EntityKey.EntityKeyValues(0).Value.ToString, keyval) Then
      Return keyval
  Else
    Return Nothing
  End If
End Function
```

C#
```
public static class extension
{
  public static int ForeignKey(this EntityReference entRef)
  {
    int keyval = 0;
    if (int.TryParse(entRef.EntityKey.EntityKeyValues[0]
                        .Value.ToString(),
```

```
                  out keyval))
        return keyval;
      else
        return 0;
    }
  }
```

Now you can call the extension method like so:

```
address.ContactReference.ForeignKey()
```

To write a method that extends the entity and does not require that you first drill into the `EntityReference` calls for some tricky use of `MetadataWorkspace`, which you will learn about in Chapter 17.

In Chapter 20, you will see examples of customizing entities with foreign key fields. For example, you will add a `PrimaryActivityID` property to the `Customer` entity to simplify the job of the UI developer.

Summary

As you can tell by now, relationships are central to the EDM and to how the Entity Framework functions. There are a lot of nuances and subtleties to understand, and some overall rules that are critical to have under your belt. And yet we've covered only the tip of the iceberg in this chapter. You will continue to encounter complexities in your code where relationships are involved, but having this deep understanding of how relationships work and how they relate to the rest of the Entity Framework means you have a lot of problem-solving tools at your disposal. We will continue to cover relationships, and how to solve other challenges that occur in various scenarios, throughout the remainder of the book.

CHAPTER 16

Making It Real: Connections, Transactions, Performance, and More

In previous chapters, you worked with bits and pieces of code and built small examples, but you did not build a real-world application. As such, you may be wondering how the Entity Framework addresses the everyday concerns of software developers. How do you control connections? Is there any connection pooling? Are database calls transactional? What about security? How's the performance? This chapter will address these and many of the additional questions developers ask after learning the basics of the Entity Framework.

EntityConnection and Database Connections in the Entity Framework

One of the benefits of using the Entity Framework is that it removes the need to write code to set up a database connection. Given that a connection string is available to the Entity Framework, most typically as part of the `EntityConnectionString` defined in a *.config* file, the Entity Framework will automatically set up, open, and close the database connection for you. Compared to typical ADO.NET code where you need to instantiate; define; and in many cases explicitly open a connection, instantiate and define a command, execute the command, and then explicitly close the connection, letting the `ObjectContext` handle all of this in the background as part of the query pipeline is certainly convenient. And this is the benefit you get in the default query scenarios. But oftentimes, you'll want more control over how and when connections are being made. To be able to do that, let's take a look at how the `EntityConnection` and `DbConnection` relate to each other. We'll also see how to programmatically force them to work the way you want if, in fact, the default behavior doesn't meet your needs.