**Associations in EF Code First CTP5: Part 2 – Shared Primary Key Associations**

|  |
| --- |
| In the previous [blog post](http://weblogs.asp.net/manavi/archive/2010/12/11/entity-association-mapping-with-code-first-part-1-one-to-one-associations.aspx) I demonstrated how to map a special kind of one-to-one association—a composition with complex types as the first post in a series about entity association mapping with EF Code First. We argued that the relationships between User and Address are best represented with a complex type mapping and we saw that this is usually the simplest way to represent one-to-one relationships but comes with some limitations.   In today’s blog post I’m going to discuss how we can address those limitations by changing our mapping strategy. This is particularly useful for scenarios that we want a dedicated table for Address, so that we can map both User and Address as entities. One benefit of this model is the possibility for *shared references*— another entity class (let’s say Shipment) can also have a reference to a particular Address instance. If a User has a reference to this instance, as her BillingAddress, the Address instance has to support shared references and needs its own identity. In this case, User and Address classes have a true *one-to-one* association.  **Introducing the Revised Model**  In this revised version, each User *could* have one BillingAddress (Billing Association). Also Shipment has to be delivered to an address so it *always* has one Delivery Address (Delivery Association). Here is the class diagram for this domain model (note the multiplicities on association lines): |
| http://weblogs.asp.net/blogs/manavi/Post02/ClassDiagram.jpg |
| In this model we assumed that the billing address of the user is the same as her delivery address. Now let’s create the association mappings for this domain model. There are several choices, the first being a *One-to-One Primary Key Association*.  **Shared Primary Associations**  Also know as *One-to-One Primary Key Associations*, means two related tables share the same primary key values. The primary key of one table is also a foreign key of the other. Let’s see how we map the primary key associations with Code First.  **How to Implement a One-to-One Primary Key Association with Code First**  First, we start with the POCO classes. As you can see, we've defined *BillingAddress* as a navigation property on User class and another one on Shipment class named *DeliveryAddress*. Both associations are unidirectional since we didn't define related navigation properties on Address class as for User and Shipment. |
| public class User  {      public int UserId { get; set; }      public string FirstName { get; set; }      public string LastName { get; set; }      public virtual Address BillingAddress { get; set; }  }    public class Address  {      public int AddressId { get; set; }      public string Street { get; set; }      public string City { get; set; }      public string PostalCode { get; set; }  }    public class Shipment  {      public int ShipmentId { get; set; }      public DateTime CreatedOn { get; set; }      public string State { get; set; }      public virtual Address DeliveryAddress { get; set; }  }    public class EntityMappingContext : DbContext  {      public DbSet<User> Users { get; set; }      public DbSet<Address> Addresses { get; set; }      public DbSet<Shipment> Shipments { get; set; }  } |
| **How Code First Reads This Object Model: One-to-Many**  Code First reads the model and tries to figure out the multiplicity of the associations. Since the associations are unidirectional, Code First takes this as if one Address has many Users and Many Shipments and will create a *one-to-many* association for each of them. So, what we were hoping for —a one-to-one association, is not inline with the conventions.  **How to Change the Multiplicity to One-to-One by Using the Conventions**  One way to turn our associations to be one-to-one is by making them bidirectional. That is, adding a new navigation property to Address class of type User and another one of type Shipment. By doing that we basically signal Code First that we are looking to have one-to-one associations since for example User has an Address and also Address has a User. Based on the conventions, Code First will change the multiplicity to one-to-one and this will solve the problem.  **Should We Make This Association Bidirectional?**  As always, the decision is up to us and depends on whether we need to navigate through our objects in that direction in the application code. In this case, we’d probably conclude that the bidirectional association doesn’t make much sense. If we call *anAddress.User*, we are saying “give me the user who has this address”, not a very reasonable request. So this is not a good option. Instead we'll keep our object model as it is and will resort to fluent API.  **How to Change the Multiplicity to One-to-One with Fluent API**  The following code is all that is needed to make the associations to be one-to-one. Note how the multiplicities in the UML class diagram (e.g. 1 on User and 0..1 on address) has been translated to the flunet API code by using *HasRequired* and *HasOptional* methods: |
| protected override void OnModelCreating(ModelBuilder modelBuilder)  {      modelBuilder.Entity<User>().HasOptional(u => u.BillingAddress)                                 .WithRequired();        modelBuilder.Entity<Shipment>().HasRequired(u => u.DeliveryAddress)                                     .WithOptional();  } |
| Also it worth mentioning that in CTP5, when we are mapping a one-to-one association with fluent API, we don't need to specify the foreign key as we would do when mapping a one-to-many association with *HasForeignKey* method. Since EF only supports one-to-one primary key associations it will automatically create the relationship in the database based on the primary keys and we don't need to state the obvious as we did in CTP4. |
| **Database Schema**  The mapping result for our object model is as follows (note the Identity column): |
| http://weblogs.asp.net/blogs/manavi/Post02/DbSchema.jpg |
| **Referential Integrity**  In relational database design the referential integrity rule states that each non-null value of a foreign key must match the value of some primary key. But wait, how does it even applies here? All we have is just three primary keys referencing each other. Who is the primary key and who is the foreign key? The best way to find the answer of this question is to take a look at the properties of the relationships in the database that has been created by Code First: |
| http://weblogs.asp.net/blogs/manavi/Post02/RefIntg1.jpg |
| http://weblogs.asp.net/blogs/manavi/Post02/RefIntg2.jpg |
| As you can see, Code First adds a foreign key constraint which links the primary key of the Addresses table to the primary key of the Users table and adds another foreign key constraint that links the primary key of the Shipments table to the primary key of the Addresses table. The foreign key constraint means that a user has to exist for a particular address but not the other way around. In other words, the database guarantees that an Addresses row’s primary key references a valid Users primary key and a Shipments row’s primary key references a valid Addresses primary key.  **How Code First Determines Principal and Dependent?**  Code First has rules to determine the principal and dependent ends of an association. For one-to-many relationships the many end is always the dependent, but it gets a little tricky in one-to-one associations. In one-to-one associations Code First decides based on our object model, and possible data annotations or fluent API that we may have. For example in our case, we wrote this fluent API code to configure User-Address association: |
| modelBuilder.Entity<User>().HasOptional(u => u.BillingAddress).WithRequired(); |
| This reads as "User entity has an optional association with one Address object but this association is required for Address entity."  For Code First this is good enough to make the decision: It marked User as the principal end and Address as the dependent end in the association. Since we have the same fluent API code for the second association between Address and Shipment, it marks Address as the principal end and Shipment as the dependent end in this association as well.   The referential integrity that we saw, is the first result of this Code First's principal/dependent decision.  **Second Result of Code First's Principal/Dependent Decision: Database Identity**  If you take a closer look at the above DB schema, you'll notice that only UserId has a regular identifier generator (aka *Identity* or *Sequence*) and AddressId and ShipmentId does not. This is a very important consequence of the principal/dependent decision for one-to-one associations: *the dependent primary key will become non-Identity by default*. This make sense because they share their primary key values and only one of them can be auto generated and we need to take care of providing valid keys for the rest. |
| **What about Cascade Deletes?**  As we saw, each Address always belongs to one User and each Shipment always delivered to one single Address. We want to make sure that when we delete a User the possible dependent rows on Address and Shipment also get deleted in the database. In fact, this is one of the [Referential Integrity Refactorings](http://www.agiledata.org/essays/databaseRefactoringCatalogReferentialIntegrity.html) which called [*Introduce Cascading Delete*](http://databaserefactoring.com/IntroduceCascadingDelete.html). The primary reason we would apply "Introduce Cascading Delete" is to preserve the referential integrity of our data by ensuring that related rows are appropriately deleted when a parent row is deleted. By default, Code First does not enable cascade delete when it creates a relationship in the database. As always we can override this convention with fluent API: |
| protected override void OnModelCreating(ModelBuilder modelBuilder)  {      modelBuilder.Entity<User>().HasOptional(u => u.BillingAddress)                                 .WithRequired()                                 .WillCascadeOnDelete();        modelBuilder.Entity<Shipment>().HasRequired(u => u.DeliveryAddress)                                     .WithOptional()                                     .WillCascadeOnDelete();  } |
| **What If Both Ends are Required?**  We saw that the only reason Code First could figure out principal and dependent in our 1:1 associations was because our fluent API code clearly specified one end as Required and the other as Optional. But what if both endpoints are the same in terms of being required in the association? For example what if in our domain model, User always has one Address and Address always has one User (required on both end)? The answer is that ultimately, this scenario need to be configured by fluent API and the interesting point is that fluent API is designed in a way that will *force* you to explicitly specify who is dependent and who is principal in such cases that this cannot be inferred by Code First.   To illustrate the idea, let's see how we can configure mapping for this User-Address association (Required/Required) with fluent API: |
| modelBuilder.Entity<User>().HasRequired(u => u.BillingAddress).WithRequiredDependent(); |
| So we invoke WithRequiredDependent() after HasRequired() method. To see the reason, we need to take a look at the RequiredNavigationPropertyConfiguration type which is returned by HasRequired(): |
| public class RequiredNavigationPropertyConfiguration<TEntityType, TTargetEntityType>  {      public DependentNavigationPropertyConfiguration<TEntityType, TTargetEntityType> WithMany();      public CascadableNavigationPropertyConfiguration WithOptional();      public CascadableNavigationPropertyConfiguration WithRequiredDependent();      public CascadableNavigationPropertyConfiguration WithRequiredPrincipal();  } |
| As you can see, if you want to go another Required after HasRequired() method, you have to either call WithRequiredDependent() or WithRequiredPrincipal() since there is no WithRequired() method on this RequiredNavigationPropertyConfiguration class which is returned by HasRequired() method. |
| Both WithRequired and WithOptional methods return a CascadableNavigationPropertyConfiguration type which has a WillCascadeOnDelete() method. Now if we run the code and check the database, we'll see that cascade delete on both relationships are switched on. |
| **Working with the Model**  Here is an example for adding a new user along with its billing address. EF is smart enough to use the newly generated UserId for the AddressId as well: |
| using (var context = new EntityMappingContext())  {      Address billingAddress = new Address()      {          Street = "Yonge St.",          City   = "Toronto"      };      User morteza = new User()      {          FirstName      = "Morteza",          LastName       = "Manavi",          BillingAddress = billingAddress      };        context.Users.Add(morteza);      context.SaveChanges();  } |
| The following code is an example of adding a new Address and Shipment for an existing User (assuming that we have a User with UserId=2 in the database): |
| using (var context = new EntityMappingContext())  {      Address deliveryAddress = new Address()      {          AddressId = 2,          Street    = "Main St.",          City      = "Seattle"      };      Shipment shipment = new Shipment()      {          ShipmentId      = 2,          State           = "Shipped",          CreatedOn       = DateTime.Now,          DeliveryAddress = deliveryAddress      };        context.Shipments.Add(shipment);      context.SaveChanges();  } |
| **Limitations of This Mapping**  There are two important limitations to associations mapped as shared primary key:   * **Difficulty in saving related objects:** The main difficulty with this approach is ensuring that associated instances are assigned the same primary key value when the objects are saved. For example, when adding a new Address object, it's our responsibility to provide a unique AddressId that is also valid (a User can be found with such a value as UserId.) * **Multiple addresses for User is not possible:** With this mapping we cannot have more than one Address for User. At the beginning of this post, when we introduce our model, we assumed that the user has the same address for billing and delivery. But what if that's not the case? What if we also want to add a Home address to User for the deliveries? In the current setup, each row in the User table has a corresponding row in the Address table. Two addresses would require an additional address table, and this mapping style therefore wouldn’t be adequate.   **Summary**  In this post we learned about one-to-one associations which shared primary key is just one way to implement it. Shared primary key associations aren’t uncommon but are relatively rare. In many schemas, a one-to-one association is represented with a foreign key field and a unique constraint. In the next posts we will revisit the same domain model and will learn about other ways to map one-to-one associations that does not have the limitations of the shared primary key association mapping.  **References** |

# Using Entity Framework 4.1 Code First with an existing database

By [**Caprica1**](http://www.codeproject.com/script/Membership/View.aspx?mid=3007561) | 19 Apr 2011 | [Technical Blog](http://www.codeproject.com/script/Articles/Types.aspx?#Technical%20Blog)

[All-Topics](http://www.codeproject.com/search.aspx?aidlst=301)[EF](http://www.codeproject.com/search.aspx?aidlst=1396)

This post focuses on using the Entity Framework 4.1 RC Fluent API with an existing database. When I say an existing database, what I really mean is that you&#8217;re not letting EF Code First auto generate the database for you.  So it could be that you like to create your entities and then manually

See Also

* [More like this](http://www.codeproject.com/search.aspx?q=Using+Entity+Framework+4.1+Code+First+with+an+existing+database)
* [More by this author](http://www.codeproject.com/script/Articles/MemberArticles.aspx?amid=3007561)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Article | [Browse Code](http://www.codeproject.com/script/Articles/ViewDownloads.aspx?aid=184133) | | [Stats](http://www.codeproject.com/script/Articles/Statistics.aspx?aid=184133) | [Revisions (2)](http://www.codeproject.com/script/Articles/ListVersions.aspx?aid=184133) | | [Alternatives](http://www.codeproject.com/script/Articles/ListAlternatives.aspx?aid=184133) |
| http://s.codeproject.com/script/Ratings/Images/stars-fill.png  http://s.codeproject.com/script/Ratings/Images/stars-empty.png | | 4.11 (2 votes) | | |

[Discuss](http://www.codeproject.com/Articles/184133/Using-Entity-Framework-4-1-Code-First-with-an-exis#_comments)3

**A Technical Blog article**. [View original blog here.](http://agilenet.wordpress.com/2011/04/11/entity-framework-4-1-rc-with-an-existing-database/)[[^](http://agilenet.wordpress.com/2011/04/11/entity-framework-4-1-rc-with-an-existing-database/)]

Top of Form

This post focuses on using the Entity Framework 4.1 RC Fluent API with an existing database.

When I say an existing database, what I really mean is that you’re not letting EF Code First auto generate the database for you.  So it could be that you like to create your entities and then manually craft the database.  Or you may have a separate team of developers who like to control and manage the database.

This article refers to the fluent API which is part of the Code First approach to the using the Entity Framework.  Code First can create the database for you based on your entities and entity relationships, but as I mention above you may not want Code First to do this.

Using EF Code First in this way is seen a core aspect of Code First by the EF team at Microsoft.  So far I have to say I really like this approach, no large xml files to worry about, instead it’s all controlled through code.  Additionally the entities are plain old CLR objects (POCO).  This means that your entities (and thus the assembly they are contained in) don’t need to reference the Entity Framework.

The following example shows how to create an entity model, then manually create your database and then map those entities to your database.  Finally it shows using the DatabaseContext to save and retrieve entities.

**Create the entities**

Here we have a company class and a country class.  Each company has a country associated with it.  There is no association going the other way:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/184133/Using-Entity-Framework-4-1-Code-First-with-an-exis)

public class Company

{

public int Id { get; set; }

public string CompanyName { get; set; }

public Country HomeCountry { get; set; }

}

public class Country

{

public int Id { get; set; }

public string Code { get; set; }

public string Name { get; set; }

}

**Create the database**

Next we create the database.  We could let EF do this for us, but here we want full control over the new database (or it could be an existing database you’re connecting to):

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/184133/Using-Entity-Framework-4-1-Code-First-with-an-exis)

CREATE DATABASE CodeFirstCustom;

GO

Use CodeFirst

create table Companies (

Id int identity(1,1) not null,

HomeCountryId int not null,

Name varchar(20) not null,

constraint PK\_Companies primary key clustered (Id))

create table Countries (

Id int identity(1,1) not null,

Code varchar(4) not null,

Name varchar(20) not null,

constraint PK\_Countries primary key clustered (Id))

alter table Companies

add constraint FK\_Company\_HomeCountry foreign key (HomeCountryId)

references Countries (Id) on delete no action

**Mapping the entities to the database**

So we now have the entities and the database.  Next the fun bit, mapping the two together using the fluent API which is part of EF Code First.

Firstly we create a configuration class for each entity which maps between the entity and the database:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/184133/Using-Entity-Framework-4-1-Code-First-with-an-exis)

public class CompanyConfiguration: EntityTypeConfiguration<Company>

{

public CompanyConfiguration(): base()

{

HasKey(p => p.Id);

Property(p => p.Id).

HasColumnName("Id").

HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity).

IsRequired();

Property(p => p.CompanyName).

HasColumnName("Name").

IsRequired();

HasRequired(x => x.HomeCountry).

WithMany().

Map(x => x.MapKey("HomeCountryId"));

ToTable("Companies");

}

}

public class CountryConfiguration: EntityTypeConfiguration<Country>

{

public CountryConfiguration(): base()

{

HasKey(p => p.Id);

Property(p => p.Id)

.HasColumnName("Id")

.HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity)

.IsRequired();

Property(p => p.Code)

.HasColumnName("Code")

.IsRequired();

Property(p => p.Name)

.HasColumnName("Name")

.IsRequired();

ToTable("Countries");

}

}

The most difficult part of the above is understanding the syntax for defining the relationship between the Company and Country entity.  We actually wouldn’t need to do this if we had named the FK column HomeCountry\_ID (as this is what EF will use by default).  In our case though we don’t use underscores so we have to tell the fluent API what the name of the column is.  So within the configuration class for the Company we say HasRequired(x => x.HomeCountry) – this tells EF that a home country must be set.  We then define the relationship going back the other way, in this case we say WithMany() – however, in our example there is no relation going the other way, so we leave this empty.  Finally we use Map(x => x.MapKey(“HomeCountryId”)) to specify the foreign key name.

It should be noted that an alternative to the Fluent API is to use attributes on your entity.  I like to keep the entities clean though, even from attributes.  I’ll talk about attributes in a future post though.

**Creating the context**

It is the context that we code against to add and remove entities to and from the database.  We create our context by creating a class which inherits from DbContext:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/184133/Using-Entity-Framework-4-1-Code-First-with-an-exis)

public class MyContext: DbContext

{

public DbSet<Company> Companies { get; set; }

public DbSet<Country> Countries { get; set; }

public MyContext(string connectionString): base(connectionString)

{

Database.SetInitializer<MyContext>(null);

}

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Configurations.Add(new CountryConfiguration());

modelBuilder.Configurations.Add(new CompanyConfiguration());

base.OnModelCreating(modelBuilder);

}

}

Above I have created a couple of properties using DbSet for our entities that we wish to expose.

In the constructor we then tell EF Code First not to attempt to create a database as we are using our own.  We then override the OnModelCreating method and load our configuration classes.

That’s it, we can then start using the context.

**Using the context**

In this example, we’re starting off with an empty database:

http://www.codeproject.com/images/minus.gifCollapse | [Copy Code](http://www.codeproject.com/Articles/184133/Using-Entity-Framework-4-1-Code-First-with-an-exis)

class Program

{

private const string ConnectionString = @"Server=.\sql2005;Database=CodeFirst;integrated security=SSPI;";

static void Main(string[] args)

{

// Firstly, create a new country record.

Country country = new Country();

country.Code = "UK";

country.Name = "United Kingdom";

// Now create an instance of the context.

MyContext myContext = new MyContext(ConnectionString);

// Set the state of the new entity as added (note that this also adds it to the context).

myContext.Entry(country).State = EntityState.Added;

myContext.SaveChanges();

Console.WriteLine("Saved Country");

// Now insert a Company record

Company company = new Company();

company.CompanyName = "AccessUK";

// Assign our new Country instance to the Company instance

// (we could just use the instance we created above, but I want to show how

// easy it is to query the context)

company.HomeCountry = myContext.Countries.First(e => e.Code == "UK");

// This is another way of adding an entity to the conext.

myContext.Companies.Add(company);

// Save the new company.

myContext.SaveChanges();

Console.WriteLine("Saved Company");

// Retrieve the company we just saved

Company savedCompany = myContext.Companies.First(e => e.CompanyName == "AccessUK");

Console.WriteLine("Retrieved Company");

Console.ReadLine();

}

}

In the above example, I create a new Company instance, save it to the database then create a new Company to which I assign the Country.  I then finish by retrieving the Company instance.

The full example created above can be found here (use an SVN client to download it):

Bottom of Form

[Entity Framework 4.1 Code-First Made Easy](http://geekswithblogs.net/BarrelOfMonkeys/archive/2011/08/10/entity-framework-4.1-code-first-made-easy.aspx)

No matter which ORM or data access technology you use, at some point you just need to do some [CRUD](http://en.wikipedia.org/wiki/Create,_read,_update_and_delete), and you need to do it fast.

In this post, I’m going to show you how to get your CRUD on with the latest version of Entity Framework in no time flat. In fact, I’d be surprised if the code to get your first CRUD operation done takes you more than a few minutes. The one thing the EF team did right with Code-First is make it really easy, but most of the “getting started” videos and documentation is packed with overviews and information overload you don’t need to get started. One doesn’t need to understand the principles of [General Relativity](http://en.wikipedia.org/wiki/General_relativity) in order to fully appreciate the humor in [cartoon gravity](http://www.youtube.com/watch?v=_d8ROhH3_vs&t=0m15s).

# Audience

I’m assuming you know how to create a database, build a connection string, write LINQ queries, and read C# code. If any of those things are foreign to you, then I’d suggest learning those topics first.

# Prerequisites

First of all, you need Visual Studio 2010. Next, you’ll need to install the bits for [Entity Framework 4.1](http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=26825), affectionately known as “Magical Unicorn Edition”.

# References

For this example, I created a simple Console Application project called CodeFirstSample and added references to three libraries:

* System.Data.Entity: this is the base Entity Framework library.
* EntityFramework: this is the Entity Framework 4.1 library.
* System.ComponentModel.DataAnnotations: this library will allow us to map CLR types to the database, among other things.
* System.Transactions: provides transactional support.

# Customer Class

First, I start with an innocent-looking class:

1: namespace CodeFirstSample

2: {

3: using System;

4:

5: public class Customer

6: {

7: public Guid CustomerId { get; set; }

8:

9: public string Name { get; set; }

10: }

11: }

Not surprisingly, I have a SQL Server database table that is quite similar (though not exact, so I can demonstrate mapping):

1: create database CodeFirstSample;

2: go

3:

4: use CodeFirstSample;

5: go

6:

7: create table dbo.Customer

8: (

9: customer\_id uniqueidentifier primary key

10: ,name nvarchar(128) not null

11: );

12: go

Easy as pie so far, right? Well, you could make it easier by skipping creating the database; code-first will create it for us if it doesn’t already exist (and presuming we’re a sysadmin on the server). This is handy functionality if your database is trivial in design.

Next, we need to tell Entity Framework how to translate the columns in our database table to the properties in our Customer class. We do that by changing our class to the following (changes highlighted):

1: namespace CodeFirstSample

2: {

3: using System;

4: using System.ComponentModel.DataAnnotations;

5:

6: public class Customer

7: {

8: [Key,Column("customer\_id",TypeName="uniqueidentifier",Order=0)]

9: public Guid CustomerId { get; set; }

10:

11: [Column("name",TypeName="nvarchar")]

12: [Required,StringLength(128)]

13: public string Name { get; set; }

14: }

15: }

Entity Framework Code-First uses attributes in the Data Annotations library for mapping columns and keys.

Key Attribute: informs Entity Framework that this field is (or is part of) the primary key.

Column Attribute consists of the following arguments:

* Name: specifies the column name in the database to which the property will be mapped.
* TypeName: optional argument that specifies the database data type of the column.
* Order: if the column is part of the primary key, specifies the order of the column in the primary key as a zero-based index (the IntelliSense documentation on this field is very unhelpful). This is only required if you have a composite (multi-column) primary key.

That’s all for our Customer class.

# DbContext

In EF 4.1, the DbContext is where all the magic happens. We’ll create a derived type called EfContext and tell it that our Customer class is part of its model:

1: namespace CodeFirstSample

2: {

3: using System.Data.Entity;

4:

5: class EfContext : DbContext

6: {

7: public EfContext( string connection ) : base( connection ) {}

8:

9: protected override void OnModelCreating( DbModelBuilder modelBuilder )

10: {

11: modelBuilder.Entity<Customer>().ToTable( "dbo.Customer" );

12: base.OnModelCreating(modelBuilder);

13: }

14: }

15: }

See that highlighted line? That’s all it takes to map our customer to a table. There are other ways this can be done, but doing it in the OnModelCreating method is foolproof, and if you’ll explore its features, you’ll find a plethora of other goodies it can accomplish.

# App.Config

Next, we add a simple configuration file to our project:

1: <?xml version="1.0" encoding="utf-8" ?>

2: <configuration>

3: <connectionStrings>

4: <clear/>

5: <add

6: name="EfSample"

7: providerName="System.Data.SqlClient"

8: connectionString="Data Source=localhost;Initial Catalog=CodeFirstSample;Integrated Security=SSPI;"/>

9: </connectionStrings>

10: </configuration>

# Using it

Next, well… there’s not a next. We’re done. It can’t be that simple, can it?

Yes, it can be. As long as you know how to use LINQ, you technically don’t need to learn any other data mapping techniques in order to perform CRUD operations with EF 4.1. And just to prove it, here’s some sample code that covers most all the bases:

1: var id = Guid.NewGuid();

2:

3: // insert

4: using ( var db = new EfContext( "name=EfSample" ) )

5: {

6: var customers = db.Set<Customer>();

7: customers.Add( new Customer { CustomerId = id, Name = "John Doe" } );

8:

9: db.SaveChanges();

10: }

11:

12: // update

13: using ( var db = new EfContext( "name=EfSample" ) )

14: {

15: var customers = db.Set<Customer>();

16: var c = customers.Single( x => x.CustomerId == id );

17:

18: // look ma, no SQL injection!

19: c.Name = "'; drop table dbo.Customer;--";

20: db.SaveChanges();

21: }

22:

23: // delete - with transaction

24: using ( var db = new EfContext( "name=EfSample" ) )

25: using ( var ts = new System.Transactions.TransactionScope() )

26: {

27: var customers = db.Set<Customer>();

28: var c = customers.Single( x => x.CustomerId == id );

29:

30: customers.Remove( c );

31: db.SaveChanges();

32: ts.Complete();

33: }

34:

35: // insert - with rollback

36: using ( var db = new EfContext( "name=EfSample" ) )

37: using ( var ts = new System.Transactions.TransactionScope() )

38: {

39: var customers = db.Set<Customer>();

40: customers.Add( new Customer { CustomerId = id, Name = "Jane Doe" } );

41:

42: db.SaveChanges();

43: //ts.Complete(); // no completion will cause rollback

44: }

Note how the connection information is specified. It’s in the form “name={config}” where {config} is the name you gave the connection string in the App.config file. This comes in handy if you need to use connect to multiple databases of the same schema.

Not only are these CRUD operations easy, they’re also fully parameterized and resilient to most attempts at SQL Injection (assuming you follow [guidelines](http://msdn.microsoft.com/en-us/library/cc716760.aspx)).