

[Jon Smith](#)

01 December 2014



162235 views

61

20

[Jon Smith](#)01 December
2014

Using Entity Framework With an Existing Database: Data Access

61

162235 views

20

Pre-existing SQL databases, particularly if complex, can often pose problems for software developers who are creating a new application. The data may be in a format that makes it hard to access or update, and may include functions and procedures. Jon Smith looks at what tools there are in Microsoft's Entity Framework's Data Access technology that can deal with these requirements.

edited: 08/12/2014

In many large-scale projects, software developers are often have to work with existing SQL Server databases with predefined tables and relationships. The problem can be that some predefined databases can have aspects that are awkward to deal with from the software side. As a software developer, my choice of database access tool is Microsoft's [Entity Framework](#) (EF) so I am motivated to see how EF can handle this.

Entity Framework 6 has a number of features to make it fairly straightforward to work with existing databases. In this article I'll detail those steps that I needed to take on the EF side, in order to build a

fully featured web application to work with the AdventureWorks database. I'll actually use the [AdventureWorksLT2012](#) database, which is a cut-down version of the larger AdventureWorks OLTP database. I am using Microsoft's [ASP.NET MVC5 \(MVC\)](#) with the propriety [Kendo UI](#) package for the UI/presentation layer, which I cover in [the next article](#).

At the end, I also mention some other techniques that I didn't need for AdventureWorks, but I have needed on other databases. The aim is to show how you can use EF with pre-existing databases, including ones that need direct access to T-SQL commands and/or Stored Procedures.

Creating the Entity Framework Classes from the existing database

Entity Framework has a well-documented approach, called reverse engineering, [to create the EF Entity Classes and DbContext from an existing database](#). This produces data classes with various Data Annotations to set some of the properties, such as string length and nullability (see the example below built around the CustomerTable), plus a DbContext with an OnModelCreating method to set up the various relationships.

```
namespace DataLayer.GeneratedEf
{
    [Table("SalesLT.Customer")]
    public partial class Customer
    {
        public Customer()
        {
            CustomerAddresses = new HashSet<CustomerAddress>();
            SalesOrderHeaders = new HashSet<SalesOrderHeader>();
        }

        public int CustomerID { get; set; }
```

```
        public bool NameStyle { get; set; }

        [StringLength(8)]
        public string Title { get; set; }

        [Required]
        [StringLength(50)]
        public string FirstName { get; set; }
    }

    [StringLength(50)]
    public string MiddleName { get; set; }
; }

    //more properties left out to short
en the class...
    //Now the relationships

    public virtual ICollection<Customer
Address> CustomerAddresses { get; set; }

    public virtual ICollection<SalesOrd
erHeader> SalesOrderHeaders { get; set; }
}
```

This does a good job of building the classes.

Certainly it is very useful to have the Data Annotations because front-end systems like MVC use these for data validation during input. However I did have a couple of problems:

1. The default code generation template includes the `virtual` keyword on all of the relationships. This enabled [lazy loading](#), which I do not want. (see section 1 below)
2. The table SalesOrderDetail has two keys: one is the SalesOrderHeaderID and one is an identity, SalesOrderDetailID. EF failed on a create and I needed to fix this. (See section 3 below)

I will now describe how I fixed these issues.


1: Removing lazy loading by altering the scaffolding of the EF classes/DbContext

As I said earlier the standard templates enable

'lazy loading'. I have been corrected in my understanding of lazy loading by some readers. The [documentation states](#) that 'Lazy loading is the process whereby an entity or collection of entities is automatically loaded from the database the first time that a property referring to the entity/entities is accessed'. The problem with this is it does not make for efficient SQL commands, as individual SQL SELECT commands are raised for each access to virtual relationships, which is not such a good idea for performance.

For that reason I do not use Lazy Loading so I want to turn it off. However if this isn't an issue for you then you can leave it in. Lazy Loading can make handling relationships easier for the software, although in my second article I will show you a method that specifically selects each data column it needs and therefore does not need Lazy Loading

Now you could hand-edit each generated class so as to remove the 'virtual', but what happens if, or rather when, the database changes? The problem is that you would then have to re-import the database and so lose all your edits, which you or your colleague might have forgotten about by then, and suddenly your whole web application slows down. No, the common rule with generated code is not to edit it. In this case the answer is to change the code that is generated during the creation of the classes and `DbContext`.



Note: You can turn off lazy loading via the EF Configuration class too, but I prefer to remove the virtual keyword as it ensures that lazy loading is definitely off.

2: Altering the code that Reverse

Engineering produces

The generation of the EF classes and `DbContext` is done using some t4 templates, referred to as scaffolding. By default the reverse engineering of the database uses some internal scaffolding, but you can import the scaffolding and change it.

There is a very clear explanation of how to [import the scaffolding](#) using NuGet, so I'm not going to repeat it.

Once you have installed the `EntityFramework.CodeTemplates` you will find two files called `Content.cs.t4` and `EntityType.cs.t4`, which control how the `DbContext` and each entity class respectively are built. Even if you aren't familiar with t4 (a great tool) then you can understand what it does – its a code generator and anything not surround by `<# #>` is standard text. I found the word 'virtual' in the `EntityType.cs.t4` and deleted it. I also removed the word 'virtual' from the `Content.cs.t4` file on the declaration of the `DbSet<>`.

You may want to alter the scaffolding more extensively, perhaps by adding a `[Key]` attribute on primary keys for some reason. All is possible, but you must dig into the .t4 code in more depth.

One warning about using importing scaffolding – Visual Studio threw a nasty error message when I first tried to import using the

`EntityFramework.CodeTemplates` scaffolding (see [stackoverflow](#) entry). It took a bit of finding but it turns out if you have [Entity Framework Power Tools Beta 4](#) installed then they clash. If you have Entity Framework Power Tools installed then you need to disable it and restart Visual Studio before you can import/reverse engineer a database. I hope that gets fixed as

Entity Framework Power Tools is very useful.

Note: There are two other methods to reverse engineer an existing database:

EntityFramework Reverse POCO Code First Generator by Simon Hughes. This is Visual Studio extension recommended by the EF Guru, Julia Lerman, in one of her [MSDN magazine articles](#). I haven't tried it, but if Julia recommends it then it must be good.

Entity Framework Power Tools Beta 4 can also reverse engineer a database. Its quicker, only two clicks, but its less controllable. I don't suggest you use this.

3: Fixing a problem with how the two keys are defined in the SalesOrderDetail table

The standard definition for the SalesOrderDetail table key parts are as follows

```
[Table("SalesLT.SalesOrderDetail")]
public partial class SalesOrderDetail
{
    [Key]
    [Column(Order = 0)]
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int SalesOrderID { get; set; }

    [Key]
    [Column(Order = 1)]
    public int SalesOrderDetailID { get; set; }

    //other properties left out for clarity
    ...
}
```

You can see it marks the first as not database-generated, but it does not mark the second as an

Identity key. This caused problems when I tried to create a new `SalesOrderDetail` so that I could add a line item to an order. I got the SQL error:

```
Cannot insert explicit value for identity column in table 'SalesOrderDetail' when IDENTITY_INSERT is set to OFF.
```

That confused me for a bit, as other two-key items had worked, such as `CustomerAddress`. I tried a few things but as it looked like an EF error I tried telling EF that the `SalesOrderDetailID` was an Identity key by using the attribute ...

```
[DatabaseGenerated(DatabaseGeneratedOption.Identity)].
```

That fixed it!

The best solution would be to edit the scaffolding again to always add that attribute to identity keys. That needed a bit of work and the demo was two days away so in the meantime I added the needed attribute using the [MetadataType attribute](#) and a 'buddy' class. This is a generally useful feature so I use this example to show you how to do this in the next section.

Adding new DataAnnotations to EF Generated classes

Being able to add attributes to properties in already generated classes is a generally useful thing to do. I needed it to fix the key problem (see section 1 above), but you might want to add some `DataAnnotations` to help the UI/presentation layer such as marking properties with their datatype, e.g. `[DataType(DataType.Date)]`. The process for doing this is given in the Example section of this [link to the MetadataType attribute](#). I will show you my example of adding the missing Identity attribute.

The process requires me to add a partial class in

another file (see later for more on this) and then

add the

```
[MetadataType(typeof(SalesOrderDetailMetaData))]
```

attribute to the property `SaledOrderDetailID`

in a new class, sometimes called a 'buddy' class .

See below:

```
[MetadataType(typeof(SalesOrderDetailMetaDa
ta))]  
public partial class SalesOrderDetail : IMo
difiedEntity  
{  
}  
  
public class SalesOrderDetailMetaData  
{  
    [DatabaseGenerated(DatabaseGeneratedOpt
ion.Identity)]  
    public int SalesOrderDetailID { get; se
t; }  
}
```

The effect is to apply those attributes to the existing properties. That fixed my problem with EF creating new `SalesOrderDetail` properly and I was away.

What happens when the database changes?

Having sorted the scaffolding as discussed above then just repeat step 1, 'Creating the Entity Framework Classes from the existing database'.

There are a few things you need to do before, during and after the re-import.

1. You should remember/copy the name of the `DbContext` so you use the same name when you re-import. That way it will recompile properly without major name changes.
2. Because you are using the same name as the existing `DbContext` you must delete the previous `DbContext` otherwise the re-

importing process will fail. If it's easier you can delete all the generated files as they are replaced anyway. That is why I suggest you put them in a separate directory with no other files added.

3. When re-importing by default the process will add the connection string to your `App.Config` file again. I suggest you un-tick that otherwise you end up with lots of connection strings (minor point, but can be confusing).
4. If you use source control (I really recommend you do) then a quick compare of the files to check what has changed is worthwhile.

Adding new properties or methods to the Entity classes

In my case I wanted to add some more properties or methods to the class? Clearly I can't add properties that change the database – I would have to talk to the DBA to change the database definition and import the new database schema again. However in my case I wanted to add properties that accessed existing database properties to produce more useful output, or to have an [intention revealing name](#), like `HasSalesOrder`.

You can do this because the scaffolding produces 'partial' classes, which means I can have another file which adds to that class. To do this it must: have the same namespace as the generated classes

The class is declared as `public partial <same class name>`.

I recommend you put them in a different folder to

the generated files. That way they will not be overwritten by accident when you recreate the generated files (note: the namespace must be the original namespace, not that of the new folder). Below I give an example where I added to the customer class. Ignore for now the `IModifiedEntity` interface (dealt with later in this article) and `[Computed]` attribute, which I will cover in [the next article](#).

```
public partial class Customer : IModifiedEntity
{
    [Computed]
    public string FullName { get { return Title + " " + FirstName + " " + LastName + " " + Suffix; } }

    /// <summary>
    /// This is true if any sales orders. We use this to decide if a 'Customer' has actually bought anything
    /// </summary>
    [Computed]
    public bool HasSalesOrders { get { return SalesOrderHeaders.Any(); } }
}
```

Note that you almost certainly will want to add to the `DbContext` class (I did – see section 4 below). This is also defined as a partial class so you can use the same approach. Which leads me on to...

Dealing with properties best dealt with at the Data Layer

In the AdventureWorks database there are two properties called 'ModifiedDate' and 'rowguid'. In the AdventureWorks Lite database these were not generated in the database. Therefore the software needs to update ModifiedDate on create or update and set the rowguid on create.

Many databases have properties like this and, if

not handled by the database, they are best dealt with at Data/Infrastructure layer. With EF this can be done by providing a partial class and overriding the `SaveChanges()` method to handle the specific issues your database needs. In the case of AdventureWorks I added an `IModifiedEntity` interface to each partial class that has `ModifiedDate` and `rowguid` property.

Then I added the code below to the `AdventureWorksLt2012 DbContext` to provide the functionality required by this database.

```
public partial class AdventureWorksLt2012 :
IGenericServicesDbContext
{
    /// <summary>
    /// This has been overridden to handle
    ModifiedDate and rowguid
    /// </summary>
    /// <returns></returns>
    public override int SaveChanges()
    {
        HandleChangeTracking();
        return base.SaveChanges();
    }

    /// <summary>
    /// This handles going through all the
    entities that have
    /// changed and seeing if we need to do
    anything.
    /// </summary>
    private void HandleChangeTracking()
    {
        foreach (var entity in ChangeTracke
r.Entries()
                .Where(e => e.State == EntitySta
te.Added
                || e.State == EntityState.Mo
dified))
        {
            UpdateTrackedEntity(entity);
        }
    }

    /// <summary>
    /// Looks at everything that has change
    d and
    /// applies any further action if requi
    red.
    /// </summary>
    /// <param id="entityEntry"></param>
    /// <returns></returns>
    private static void UpdateTrackedEntity
(DbEntityEntry entityEntry)
    {
        var trackUpdateClass = entityEntry.
```

```
Entity as IModifiedEntity;
    if (trackUpdateClass == null) return;
    trackUpdateClass.ModifiedDate = DateTime.UtcNow;
    if (entityEntry.State == EntityState.Added)
        trackUpdateClass.rowguid = Guid.NewGuid();
}
```

The `IModifiedEntity` interface is really simple:

```
//This interface is added to all the database entities
//that have a modified date and rowGuid. Save Changes uses this
// to find entities that need the date updating, or a new rowguid added
public interface IModifiedEntity
{
    DateTime ModifiedDate { get; set; }
    Guid rowguid { get; set; }
}
```

Using SQL Store Procedures

Some databases rely on [SQL Stored Procedures](#) (SPs) for insert, update and delete of rows in a table. AdventureWorksLT2012 did not, but if you need to that EF 6 has added a neat way of linking to stored procedures. It's not trivial, but you can find [good information here](#) on how to get EF to use SPs for Insert, Update and Delete operations.

Clearly if the database needs SPs for CUD (Create, Update and Delete) actions then you need to use them, and there are plenty of advantages in doing so. In the absence of stored procedures, it is easy from the software point of view to use EFs CUD actions and EFs CUD have some nice features. For instance, EF has an in-memory copy of the original values and uses this for working out what has changed. The benefit is that the EF updates are efficient – you update one property and only that cell in a row is updated. The more subtle benefit is tracking changes and handling SQL security, i.e. if

you use SQL column-level security (Grant/Deny) then if that property is unchanged we do not trigger a security breach. This is a bit of an esoteric feature, but I have used it and it works well.

Other things you could do

This is all I had to do to get EF to work with an existing database, but there are other things I have had to use in the past. Here is a quick run through of other items:

Using Direct SQL commands

Sometimes it makes sense to bypass EF and use a SQL command, and EF has all the commands to allow you to do this. The EF documentation has a page on this here which gives a reasonable overview, but I recommend Julia Lerman's book '[Programming Entity Framework: DbContext](#)' which goes into this in more detail (note: this book is very useful but it covers an earlier version of EF so misses some of the latest commands like the use of SPs in Insert, Update and Delete).

For certain types of reads SQL makes a lot of sense. For instance in my GenericSecurity library I need to read the current SQL security setup (see below). I think you will agree it makes a lot of sense to do this with a direct SQL read rather than defining multiple data classes just to build the command.

```
var allUsers = db.Database.SqlQuery<SqlUserA  
ndRoleRow>(  
    @"select mp.name as UserName, rp.name as Rol  
eName, mp.type as UserType  
from sys.database_role_members drm  
join sys.database_principals rp on (drm.role  
_principal_id = rp.principal_id)  
join sys.database_principals mp on (drm.membe  
r_principal_id = mp.principal_id)  
ORDER BY UserName");
```

For SQL commands such as create, update and delete is less obvious, but I have used it in some cases. For these you use the `SqlCommand` method, see example from Microsoft below:

```
using (var context = new BloggingContext())
{
    context.Database.SqlCommand(
        "UPDATE dbo.Blogs SET Name = 'Another Name' WHERE BlogId = 1");
}
```

Neither of these example had parameters, but if you did need any parameters then `SqlQuery` and `SqlCommand` methods can take parameters, which are checked to protect against a SQL injection attack. The [Database.SqlQuery Method](#) documentation shows this.

One warning on `SqlCommands`. Once you have run a `SqlCommand` then EF's view of the database, some of which is held in memory, is out of date. If you are going to close/dispose of the `DbContext` straight away then that isn't a problem. However if the command is followed by other EF accesses, read or write, then you should use the EF 'Reload' command to get EF back in track. See my [stackoverflow answer](#) here for more on this.

SQL Transaction control

When using EF to do any database updates using the `.SaveChanges()` function then all the changes are done in one transaction, i.e. if one fails then none of the updates are committed. However if you are using raw SQL updates, or a combination of EF and SQL updates, you may well need these to be done in one transaction. Thankfully EF version 6 introduced commands to allow you to control transactions.

I used these commands in my EF code to work

with SQL security. I wanted to execute a set of SQL commands to set up SQL Security roles and grant/deny access, but if any one failed I wanted to roll back. The code to execute a sequence of sql commands and rollback if any single command fails is given below:

```
using (var dbContextTransaction = db.Database.BeginTransaction())
{
    try
    {
        foreach (var text in sqlCommands)
        {
            db.Database.ExecuteSqlCommand(text);
        }
        dbContextTransaction.Commit();
    }
    catch (Exception ex)
    {
        dbContextTransaction.Rollback();
        //report the error in some way
    }
}
```

You can also use the same commands in a mixed SQL commands and EF commands. See this [EF documentation](#) for an example of that.

Conclusion

There were a few issues to sort out but all of them were fixable. Overall, getting EF to work with an existing database was fairly straightforward, once you know how. The problem I had with multiple keys (see section 1) was nasty, but now I, and you, know about it we can handle it in the future.

I think the AdventureWorks Lite database is complex enough to be a challenge: with lots of relationships, composite primary keys, computed columns, nullable properties etc. Therefore getting EF to work with AdventureWorks is a good test of EFs capability to work with existing SQL databases. While the AdventureWorks Lite database did not need any raw SQL queries or

Stored Procedures other projects of mine have used these, and I have mentioned some of these features at the end of the article to complete the picture.

In fact version 6 of EF added a significance amount of extra features and commands to make mixed EF/SQL access very possible. The more I dig into things the more goodies I find in EF 6. For instance EF 6 brought in [Retry logic for Azure](#), [Handling transaction commit failures](#), [SQL transaction control](#), [improved sharing connections between SQL and EF](#), plus a number of other things. Have a good look around the [EF documentation](#) – there is a lot there.

So, no need to hold back on using Entity Framework on your next project that has to work with an existing SQL database. You can use it in a major role as I did, or now you have good connection sharing just use it for the simple CRUD cases that do not need heavy T-SQL methods.

My [second article](#) carried on this theme by looking at the challenges of displaying and updating this data at the user interface end. I talk about various methods to develop a good the user experience quickly while still keeping a reasonable database performance

I have now set up a [live web site](#) running the web application that prompted this article. It is fairly basic in appearance as I was concentrating on the features, not the style, but you might find it interesting. I also have made the source code available on [GitHub](#) in case you might like to see how this application was built.

My open-source GenericServices library is



now available on [NuGet](#) with
documentation on the GenericServices'
[GitHub Wiki](#).