AUTODESK UNIVERSITY 2015

SD10672

Exploring Entity Framework for AutoCAD Development—And Beyond!

Ben Rand
Job Industrial Services, Inc.

Learning Objectives

- Create an Entity Framework model using Database First and Code First
- Configure database options using Fluent API and DataAnnotations
- Work with Entity Framework objects such as DbContext and DbSet to create, retrieve, update, and delete database records
- Create AutoCAD objects with data queried from a database via the Entity Framework model

Description

This class provides an introduction to Microsoft's Entity Framework, one of the leading object-relational mapper frameworks in use today. With Entity Framework, developers can focus more of their effort on dealing with how objects in their problem domain relate to each other, and less effort worrying about data-persistence code. The class starts by taking a quick look at "Database First" creation of an Entity Framework model. It then moves to the "Code First" style of development, including POCO classes, DbContext configuration. We learn how to consume the resulting Entity Framework model in various different project types (including a Console app and an AutoCAD plug-in application).

Your AU Experts

Ben Rand *began his career in AutoCAD with a part time job while studying English at Brigham Young University. After a year of teaching 6th and 7th grade English, he returned to architecture and AutoCAD. He taught himself to program first in AutoLISP, then VBA to speed up repetitive drafting tasks. He taught as an Autodesk Certified Instructor at Salt Lake Community College, covering AutoCAD, LISP and VBA programming for AutoCAD, as well as 3DS Viz and Max. In 2002, he took a job as CAD Manager for an industrial engineering firm, and has worked in that industry ever since. Ben now spends most of his professional time developing in C#. At AU 2013, he won the overall AUGI Top DAUG prize, not bad for someone who hasn't done production drafting in many years. He is currently an AutoCAD Mentor All-Star, helping AutoCAD and AutoCAD LT newbies during their software trial. Ben is an avid reader, and also enjoys tennis, racquetball, volleyball, running and cycling.*

A large percentage of applications (all?) we developers write require the persistence of data and/or the reading of data from data stores. For example, we might read/write data in a text or XML file, or from a database of some sort, like MySQL, SQLite or SQL Server. Autocad DWG files are essentially databases that store geometric data—with the ability to display that data as a pretty, interactive picture.

In a relational database, tables are used to store records. These records are typically “related” to records in other tables, typically through the use of Primary and Foreign keys. Many common databases use a language of their own called SQL, which stands for Structured Query Language, to read, write and delete data in the various tables. These operations are commonly referred to as “CRUD” operations—Create, Read, Update and Delete.

One of the difficulties faced by developers is “mapping” between the concepts and objects of the database world—tables, records, keys, “CRUD” SQL statements, etc.—and the concepts and objects of the application domain—the objects (classes) and their properties used in the application code. To resolve these difficulties, ORM (object-relational mapping) frameworks have moved to the forefront.

Entity Framework is Microsoft’s entry into the ORM space for .NET. There are, of course, alternatives, such as NHibernate, LLBLGen Pro, and Castle ActiveRecord (or Ruby’s ActiveRecord). While there are distinctions among the various frameworks that are worth exploring, they all serve essentially the same purpose: to allow you to concentrate your development efforts on the problem domain while the framework takes care of translating your problem domain code into database code (“CRUD” SQL statements) so that you don’t have to.

Entity Framework has evolved quickly since its initial release by Microsoft in 2008. Today, there are several ways to work with EF, including database first, model first and code first.

- **Database First**—you have an existing database you want to work with and you want to work with a visual designer to create an Entity Framework model so you can start querying, and updating data. This is probably the quickest way to get started with EF. We’ll take a quick look at this in class.
- **Model First**—you are starting a new database, and want to work in a visual designer. EF converts the model you design to the tables, columns and keys needed to create the database. We won’t spend time with this in class.
- **Code First**—rather than working with a designer, you write POCOs (regular classes) that represent your model. EF translates this model into tables, columns and keys. Because you control all the code, many developers prefer this method of development. While commonly used for new database projects, an existing database can be reverse engineered into POCO classes, and you can take over coding from there.

In this class, we’re going to briefly explore database first, then spend most of the time using the code first approach which seems to be the way many developers are leaning these days, because of the greater control you have as a developer over the code.

Prerequisites

In order to complete the class exercises, you’ll need the following prerequisites installed:

- Microsoft Visual Studio 2015 [Community edition should work, but Professional or higher recommended] (VS 2010 or later) http://bit.ly/1ermhSi

- SQL Server 2014 Express (SQL Server 2005 or later) w/ Tools (SQL Server Management Studio) http://bit.ly/1ge8TTr
- AdventureWorks Sample database http://bit.ly/1S4MKaV
- Autocad 2016 ObjectARX SDK http://autode.sk/1HSTCQK
- Autocad 2016 http://www.autodesk.com/store/autocad

The class code will be presented using C#, but all of this can be done with VB. I was a VBA/VB programmer for many years, but switched over to C# about six years ago. The sample projects will be available in both VB and C# along with the class materials.

Create an Entity Framework model using Database First and Code First

Database First

We're going to use the Database First approach as a brief introduction to EF. This is a good approach for Entity Framework projects where the database already exists, and you want to get up and running as quickly as possible. It is also good if you prefer to work in a more visual environment.

> For this section, it is assumed that you have installed SQL Server Express, and have the AdventureWorks sample database attached. Please see the **Prerequisites** heading for links to download these items.

Start a new Visual Studio Solution

Create a new C# Class Library Project named **AU.AdventureWorks.DatabaseFirst**, but change the Solution name to **AU.AdventureWorks**. Delete the default **Class1.cs** which is added automatically. Right-click **References > Manage NuGet Packages**. Search for **EntityFramework**[1]. Select the package and click **Install**.

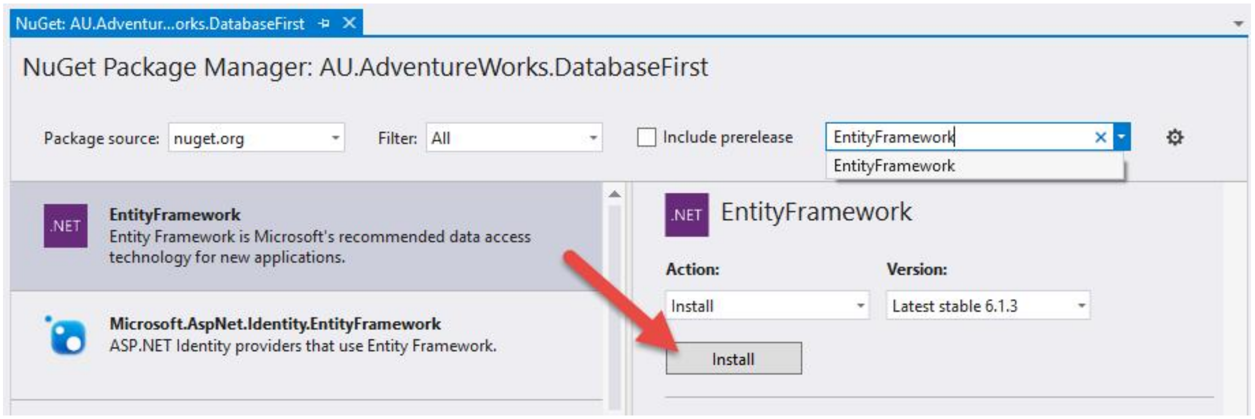

NuGet: AU.Adventur...orks.DatabaseFirst
NuGet Package Manager: AU.AdventureWorks.DatabaseFirst
Package source: nuget.org
Filter: All
Include prerelease
EntityFramework
EntityFramework
EntityFramework
Entity Framework is Microsoft's recommended data access technology for new applications.
Microsoft.AspNet.Identity.EntityFramework
ASP.NET Identity providers that use Entity Framework.
EntityFramework
Action:
Version:
Install
Latest stable 6.1.3
Install

FIGURE 1 LOADING ENTITY FRAMEWORK USING NUGET

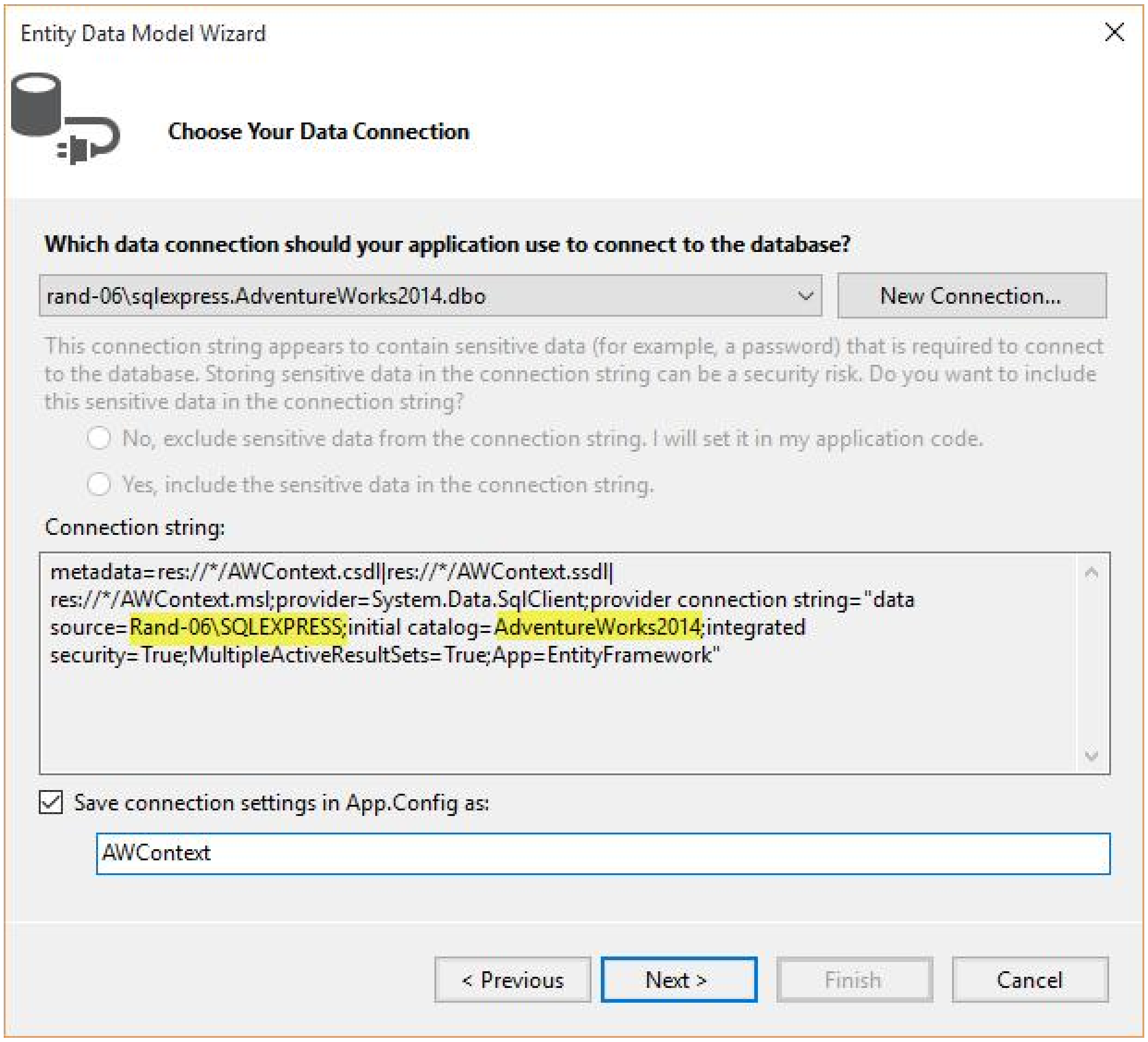
A summary of the changes this NuGet package is displayed, click OK, then accept the license agreement.

Add an Entity Data Model

Right-click the project node, then choose **Add > New Item**. Select **Data**, then choose **ADO.NET Entity Data Model**. Name the model **AWContext**, then click **Add**. Select **EF Designer from database**, then click

[1] EntityFramework 6.1.3 is the latest release at the time of this writing.

Next. Click **New Connection**, then use the wizard to connect to your SQL Server Express instance. Change the Connection settings name to **AWContext**.



Which data connection should your application use to connect to the database?

rand-06\sqlexpress.AdventureWorks2014.dbo

This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

No, exclude sensitive data from the connection string. I will set it in my application code.

Yes, include the sensitive data in the connection string.

Connection string:

```
metadata=res://*/AWContext.csdl|res://*/AWContext.ssdl|
res://*/AWContext.msl;provider=System.Data.SqlClient;provider connection string="data
source=Rand-06\SQLEXPRESS;initial catalog=AdventureWorks2014;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework"
```

☑ Save connection settings in App.Config as: AWContext

FIGURE 2 CONFIGURE THE DATA CONNECTION

The parts of the connection string are as follows:

- `metadata=` sets up the main components of the EF model.
- `provider=` identifies the provider, which is how EF communicates with the database. Support for SQL Server is built in, but providers are also available for other database systems, including Oracle and SQLite.
- `provider connection string`
  - `data source=` identifies the server and SQL Server instance
  - `initial catalog=` identifies the name of the database

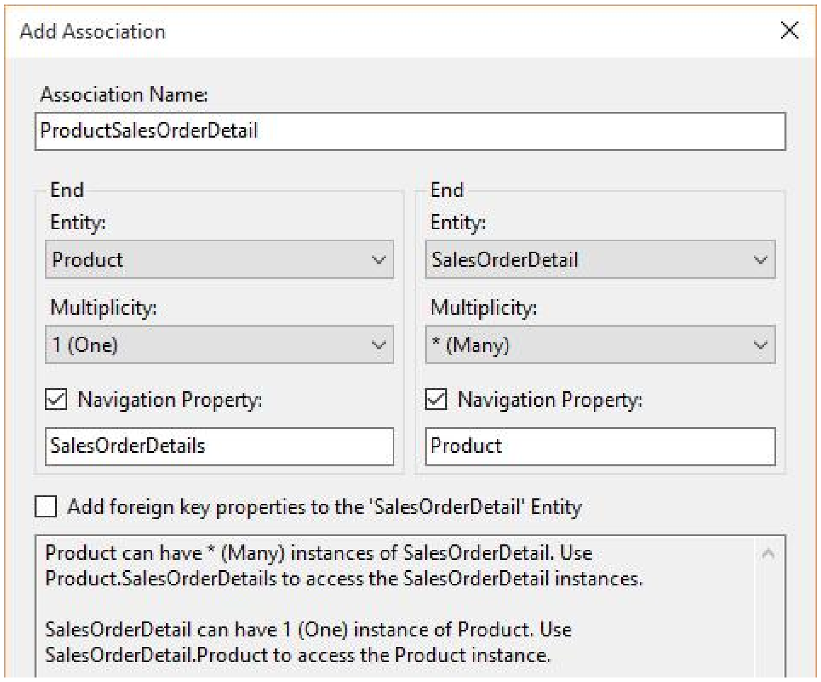
- `integrated security=` identifies how we're going to login to the database. Here I'm using integrated security—my Windows credential.

The next step of the wizard is to select which tables, views and stored procedures we want to include in the model[2]. For our purposes, we're only going to target a select few tables. Expand **Tables > Sales** and select **Sales Order Header** and **Sales Order Detail**. Expand **Production** and select **Product**. Change the name of the model namespace to **AWModel**. Click Finish to build the model.

The wizard generates an EDMX file which is an XML file. However, VS typically displays the EDMX file as a diagram, and you can typically handle most modifications to the EDMX file from the GUI diagram. You'll see the two tables you selected, with a line drawn between them, indicating a relationship. For any given `SalesOrderHeader`, there can be multiple `SalesOrderDetail` records.

Each `SalesOrderDetail` has an associated `Product`, but the database didn't have a Foreign Key relationship set up in the `SalesOrderDetail` table, so EntityFramework was unable to work this relationship out (there's no line connecting `Product` to `SalesOrderDetail`). We'll have to add this to our model.

The `Product` table (the list of Products) is our primary table, so we'll want to start the association *from* that table *to* the `SalesOrderDetail` table. Right-click the **Product** header and choose **Add new > Association**. Be sure that **Product** appears on the left-hand side under Entity, and **SalesOrderDetail** appears on the right-hand side. Uncheck the **Add foreign key properties** check box, as shown below.



Association Name: ProductSalesOrderDetail

End
Entity: Product
Multiplicity: 1 (One)
☑ Navigation Property: SalesOrderDetails

End
Entity: SalesOrderDetail
Multiplicity: * (Many)
☑ Navigation Property: Product

☐ Add foreign key properties to the 'SalesOrderDetail' Entity

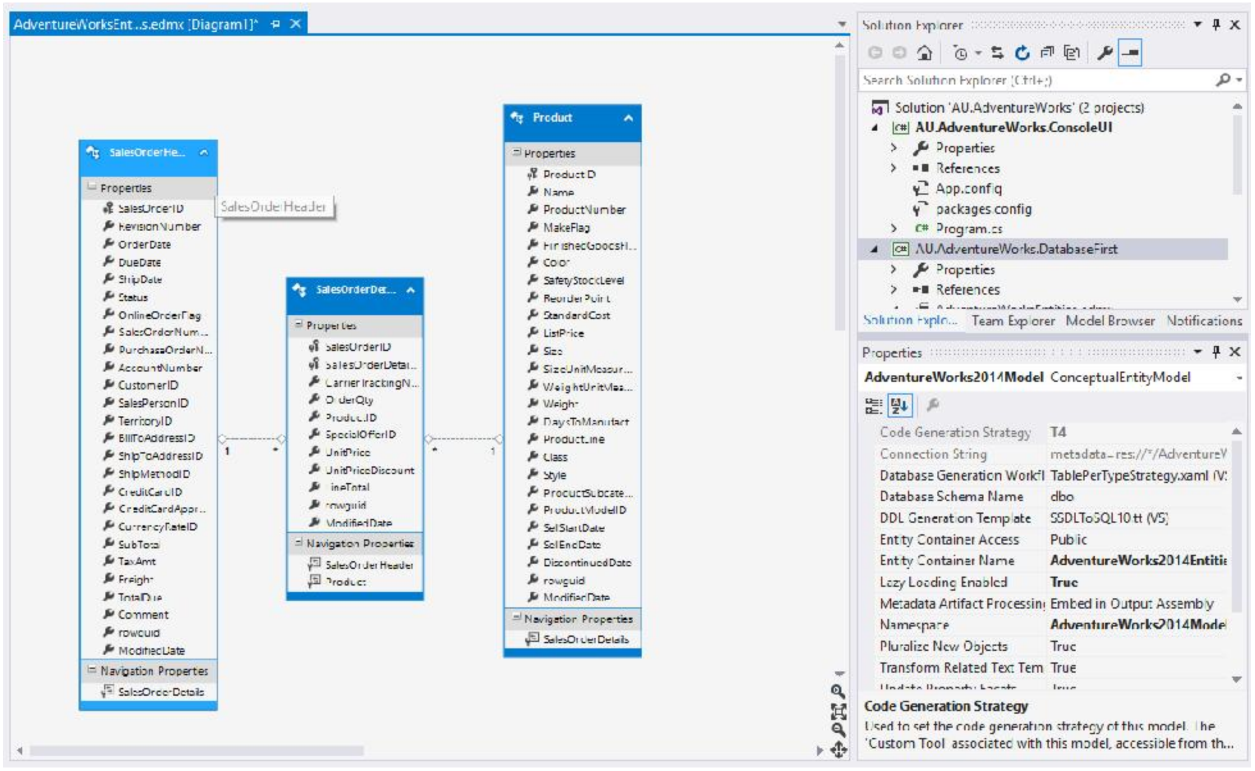
Product can have * (Many) instances of SalesOrderDetail. Use Product.SalesOrderDetails to access the SalesOrderDetail instances.

SalesOrderDetail can have 1 (One) instance of Product. Use SalesOrderDetail.Product to access the Product instance.

FIGURE 3 ADDING AN ASSOCIATION BETWEEN TABLES

[2] If you're coming new to an established database, selecting all the tables (and/or views) to create an overall diagram can be a good way to learn the relationships between the various tables, provided it is a properly normalized database. But you can also cherry pick only the relevant objects for your application's specific needs.

You'll notice the Association Name and Navigation Properties change automatically. This indicates a relationship such that a given product may be associated with many `SalesOrderDetails`. Conversely, a given `SalesOrderDetail` can have only 1 `Product` associated with it.



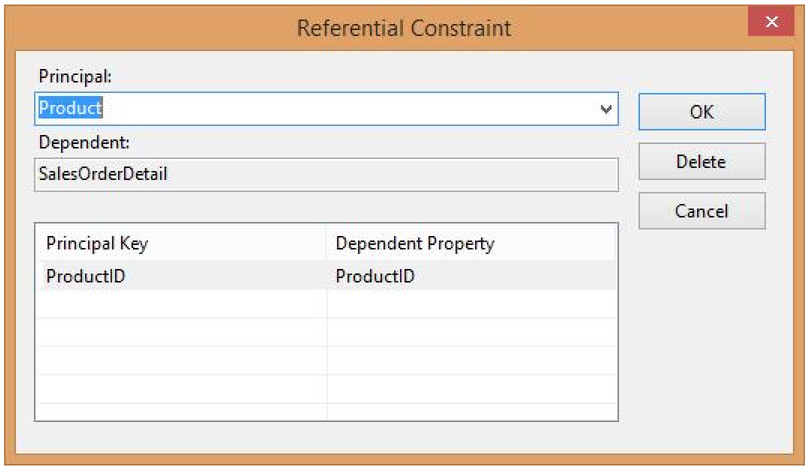
AdventureWorks2014Model ConceptualEntityModel

Code Generation Strategy	T4
Connection String	metadata=res://*/AdventureV
Database Generation Workfl	TablePerTypeStrategy.xaml (V:
Database Schema Name	dbo
DDL Generation Template	SSDLToSQL10.tt (VS)
Entity Container Access	Public
Entity Container Name	AdventureWorks2014Entitie
Lazy Loading Enabled	True
Metadata Artifact Processin	Embed in Output Assembly
Namespace	AdventureWorks2014Mode
Pluralize New Objects	True
Transform Related Text Tem	True

Code Generation Strategy
Used to set the code generation strategy of this model. The 'Custom Tool' associated with this model, accessible from th...

FIGURE 4 A DATABASE FIRST MODEL

Because there isn't a proper foreign key in the database from `SalesOrderDetail` to `Product`, we need to identify this for EF. Double-click the association line between `SalesOrderDetail` and `Product` to bring up the Referential Constraint dialog. Select **Product** as the Principal, then verify that **ProductID** is selected as the **Principal Key** (i.e. from the `Product` table) and as the **Dependent Property** (i.e. from the `SalesOrderDetail` table).



Principal Key	Dependent Property
ProductID	ProductID

FIGURE 5 ADDING AN ASSOCIATION AND REFERENTIAL CONSTRAINT

Click OK, then Save the diagram.

Create a Console Application
At this point, we're ready to start using our new EF model. Let's create a small Console Application, and write some code. In this section, I'm going to make a couple of deliberate errors[3], and work through these issues to illustrate more of how EF works.

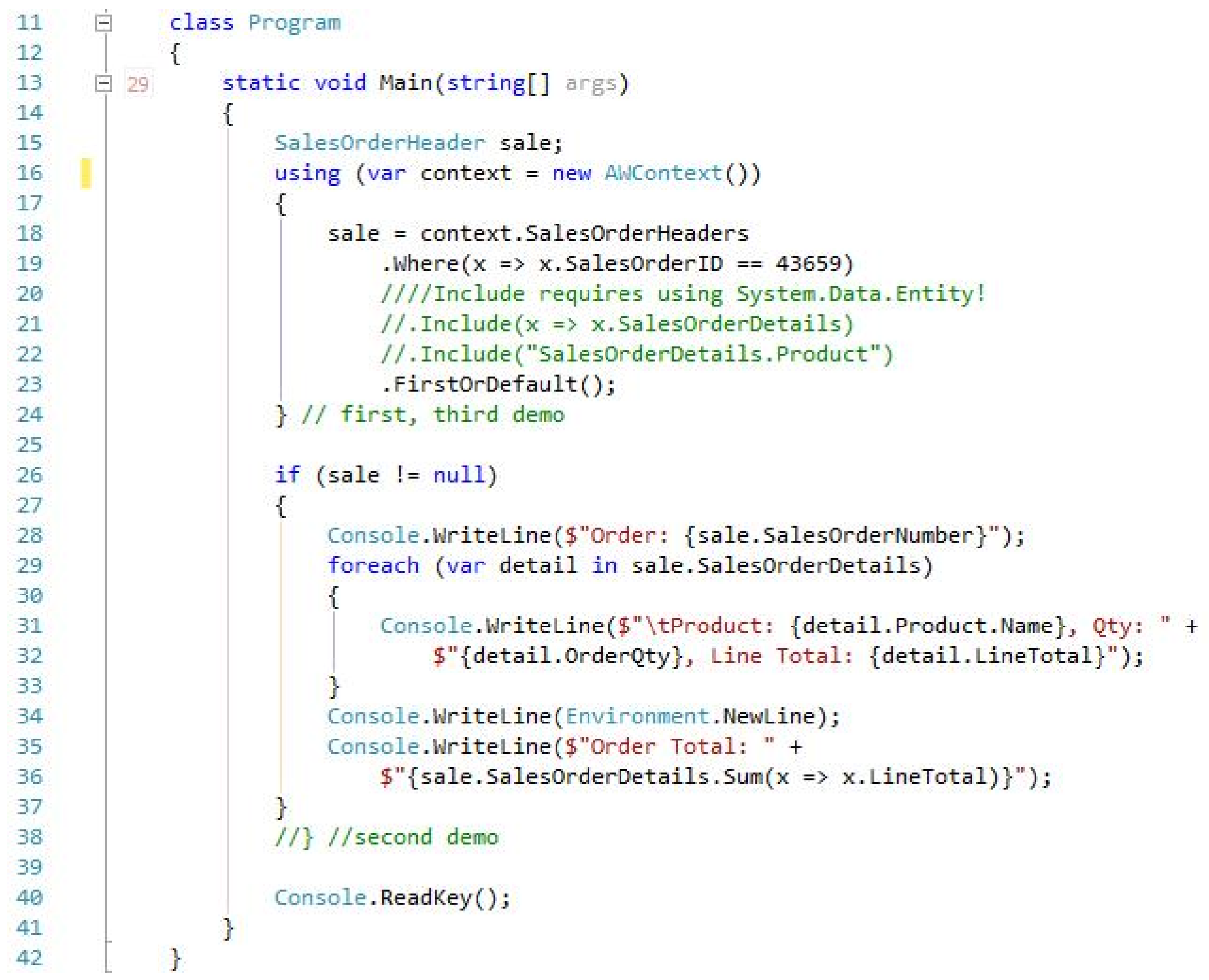
Right-click the solution and choose **Add > New Project**. Select **Console Application** from the project templates, and name it **AU.AdventureWorks.ConsoleUI**. This template creates a class called **Program**, with a **Main** method. We'll start here and build out. Be sure to set this project as the **StartUp Project**.

Add references to the following:

- DatabaseFirst
- EntityFramework (via NuGet).

In the Main method body, add the following code:

[3] When presenting to large groups, it's always a good idea to say you're going to make some mistakes "for learning purposes." That way you're covered when something inevitably goes wrong. :/



```
11  class Program
12  {
13      static void Main(string[] args)
14      {
15          SalesOrderHeader sale;
16          using (var context = new AWContext())
17          {
18              sale = context.SalesOrderHeaders
19                  .Where(x => x.SalesOrderID == 43659)
20                  ////Include requires using System.Data.Entity!
21                  //.Include(x => x.SalesOrderDetails)
22                  //.Include("SalesOrderDetails.Product")
23                  .FirstOrDefault();
24          } // first, third demo
25
26          if (sale != null)
27          {
28              Console.WriteLine($"Order: {sale.SalesOrderNumber}");
29              foreach (var detail in sale.SalesOrderDetails)
30              {
31                  Console.WriteLine($"\tProduct: {detail.Product.Name}, Qty: " +
32                      $"{detail.OrderQty}, Line Total: {detail.LineTotal}");
33              }
34              Console.WriteLine(Environment.NewLine);
35              Console.WriteLine($"Order Total: " +
36                  $"{sale.SalesOrderDetails.Sum(x => x.LineTotal)}");
37          }
38          //} //second demo
39
40          Console.ReadKey();
41      }
42  }
```

FIGURE 6 CODE LISTING 1

Here we should point out several things:

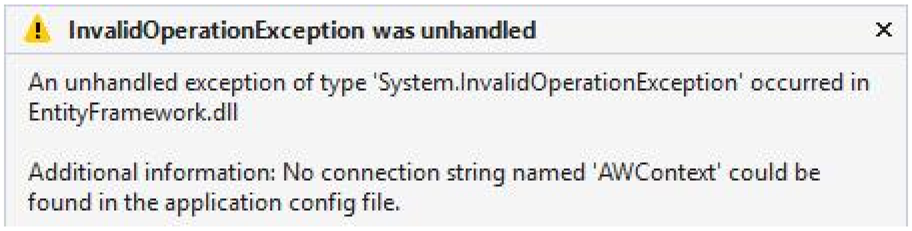
- Line 16: The "context" represents our connection to our EF model (`AWContext`), which takes care of everything we need for connecting to and querying the database.
- Lines 18-23: We use LINQ to query the `SalesOrderHeaders` collection (a `DbSet<SalesOrderHeader>`) looking for a specific record. I prefer using lambda expressions because they're so concise but they took some getting used to. An alternate form is shown below, which some developers may prefer because it's similar to SQL:

```
sale = (from h in context.SalesOrderHeaders
        where h.SalesOrderID == 43659
        select h).FirstOrDefault();
```

- Lines 29-33: We iterate through the `SalesOrderDetails` to extract some of the data we want to print. Note the use of navigation properties in lines 29 and 31 to dig further into the object hierarchy. This is fairly intuitive: a `SalesOrderHeader` has one or more associated `SalesOrderDetails`. Each `SalesOrderDetail` has an associated `Product`.

- Lines 36: We use the Sum extension method to get a sum of the `LineTotal` for all `SalesOrderDetails` associated with the `SalesOrderHeader`. LINQ makes this kind of calculation really easy!

Press F5 to run the application. Right off the bat, we're going to run into an `InvalidExceptionError`, because the ConsoleUI app can't find the connection string to connect to our database. Why is this?

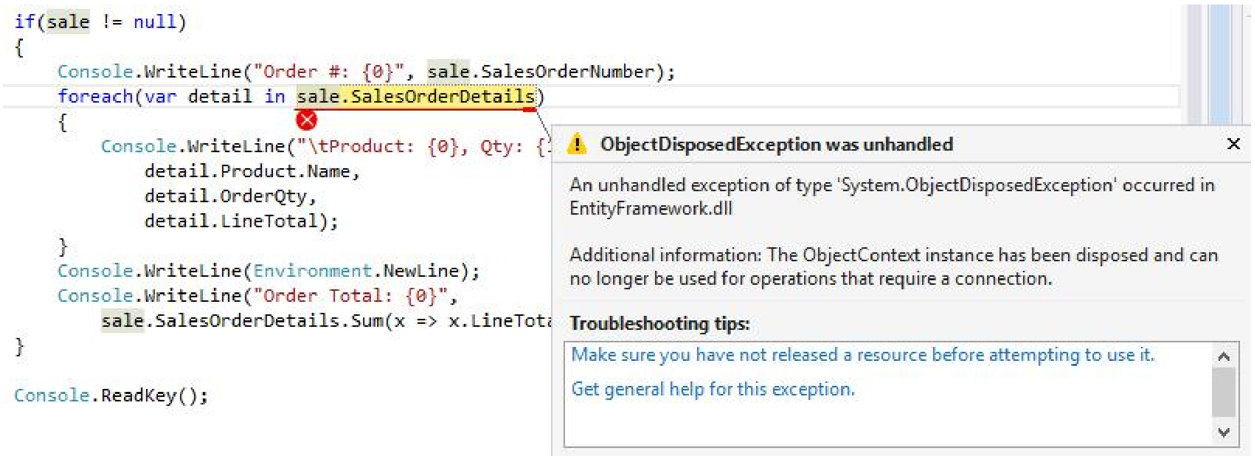


```
InvalidOperationException was unhandled

An unhandled exception of type 'System.InvalidOperationException' occurred in
EntityFramework.dll

Additional information: No connection string named 'AWContext' could be
found in the application config file.
```

FIGURE 7 NO CONNECTION STRING ERROR

The connection string we set up earlier is located in the **DatabaseFirst** project's App.config. Now that **ConsoleUI** is the startup application, this information must come from its own App.config file. Open the DatabaseFirst project's App.config, and copy and paste the entire **`<connectionStrings>`** node to ConsoleUI's App.config file, and run the application again.

At this point, we should see a console window, and an Order # displayed, and then another error is thrown:



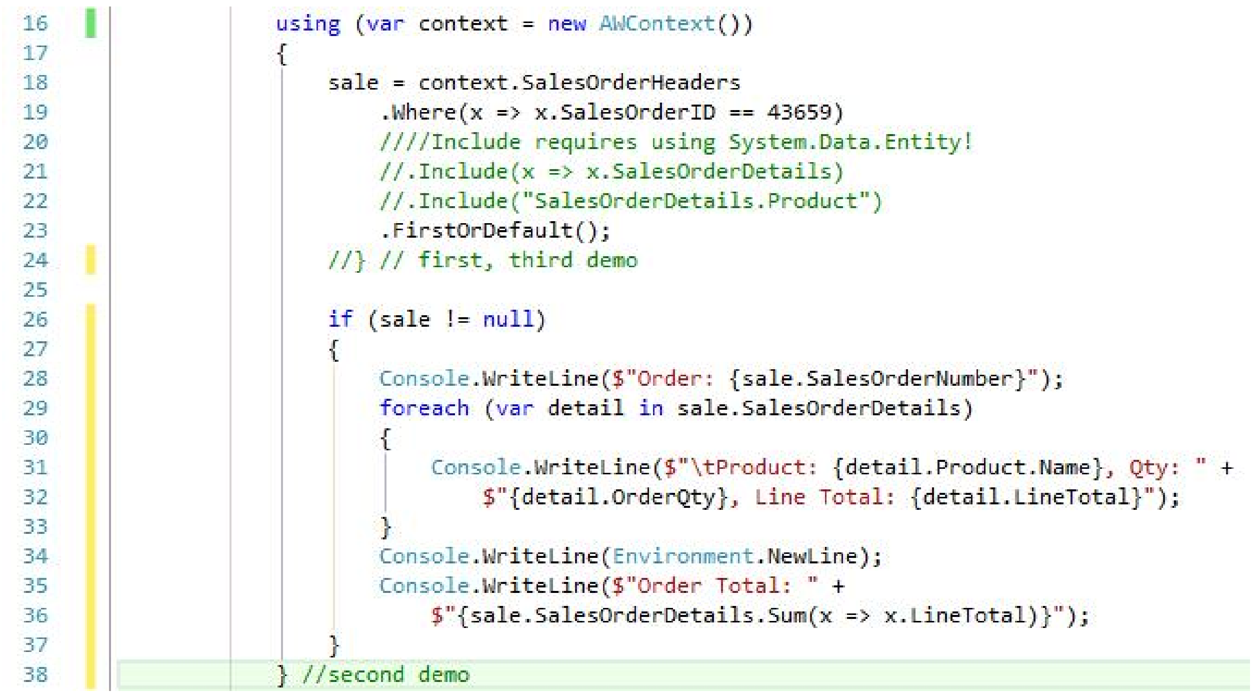
```
if(sale != null)
{
    Console.WriteLine("Order #: {0}", sale.SalesOrderNumber);
    foreach(var detail in sale.SalesOrderDetails)
    {
        Console.WriteLine("\tProduct: {0}, Qty: {
            detail.Product.Name,
            detail.OrderQty,
            detail.LineTotal);
    }
    Console.WriteLine(Environment.NewLine);
    Console.WriteLine("Order Total: {0}",
        sale.SalesOrderDetails.Sum(x => x.LineTota
}

Console.ReadKey();
```

```
ObjectDisposedException was unhandled

An unhandled exception of type 'System.ObjectDisposedException' occurred in
EntityFramework.dll

Additional information: The ObjectContext instance has been disposed and can
no longer be used for operations that require a connection.

Troubleshooting tips:
Make sure you have not released a resource before attempting to use it.
Get general help for this exception.
```

FIGURE 8 WORKING THROUGH AN OBJECTCONTEXT DISPOSED ERROR

We were successful in retrieving data from the database, but only the `SalesOrderHeader` record. When we hit the end of the `using` statement, our connection to the context (and the database) is closed. Later, when we attempt to navigate further into our `SalesOrderHeader` object, the connection is no longer available. By default, EF is *lazy loading* data from the database, which means that it won't retrieve data until it is explicitly requested by the application.

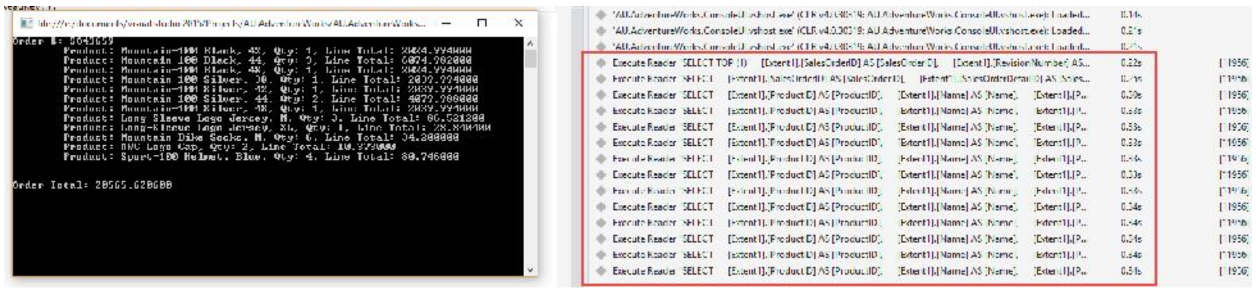
Here, we can either restructure our program a bit to keep the connection open while we do all this processing, or we can *eager load* the data we need. I want to demonstrate one of the important considerations of lazy vs eager loading. Let's make a very small change to our code, shifting the closing bracket of the using statement to just after the closing bracket of the `if` statement. This causes all the processing to happen before the `context` (connection) object is disposed.



```
            using (var context = new AWContext())
            {
                sale = context.SalesOrderHeaders
                    .Where(x => x.SalesOrderID == 43659)
                    ////Include requires using System.Data.Entity!
                    //.Include(x => x.SalesOrderDetails)
                    //.Include("SalesOrderDetails.Product")
                    .FirstOrDefault();
                //} // first, third demo

                if (sale != null)
                {
                    Console.WriteLine($"Order: {sale.SalesOrderNumber}");
                    foreach (var detail in sale.SalesOrderDetails)
                    {
                        Console.WriteLine($"\tProduct: {detail.Product.Name}, Qty: " +
                            $"{detail.OrderQty}, Line Total: {detail.LineTotal}");
                    }
                    Console.WriteLine(Environment.NewLine);
                    Console.WriteLine($"Order Total: " +
                        $"{sale.SalesOrderDetails.Sum(x => x.LineTotal)}");
                }
            } //second demo
```

FIGURE 9 CODE LISTING 1.A

And run the application again:



```
Order #: SO43659
        Product: Mountain-100 Black, 42, Qty: 1, Line Total: 2024.994000
        Product: Mountain-100 Black, 44, Qty: 3, Line Total: 6074.982000
        Product: Mountain-100 Black, 48, Qty: 1, Line Total: 2024.994000
        Product: Mountain-100 Silver, 38, Qty: 1, Line Total: 2039.994000
        Product: Mountain-100 Silver, 42, Qty: 1, Line Total: 2039.994000
        Product: Mountain-100 Silver, 44, Qty: 2, Line Total: 4079.988000
        Product: Mountain-100 Silver, 48, Qty: 1, Line Total: 2039.994000
        Product: Long-Sleeve Logo Jersey, M, Qty: 3, Line Total: 86.521200
        Product: Long-Sleeve Logo Jersey, XL, Qty: 1, Line Total: 28.840400
        Product: Mountain Bike Socks, M, Qty: 6, Line Total: 34.200000
        Product: AWC Logo Cap, Qty: 2, Line Total: 10.373000
        Product: Sport-100 Helmet, Blue, Qty: 4, Line Total: 80.746000


Order Total: 20565.620600
```

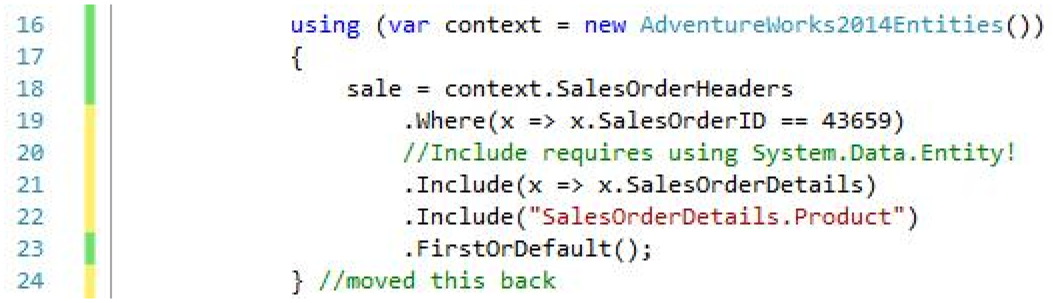
```
'AU.AdventureWorks.ConsoleUI.vshost.exe' (CLR v4.0.30319: AU.AdventureWorks.ConsoleUI.vshost.exe): Loaded...  0.14s
'AU.AdventureWorks.ConsoleUI.vshost.exe' (CLR v4.0.30319: AU.AdventureWorks.ConsoleUI.vshost.exe): Loaded...  0.21s
'AU.AdventureWorks.ConsoleUI.vshost.exe' (CLR v4.0.30319: AU.AdventureWorks.ConsoleUI.vshost.exe): Loaded...  0.21s
Execute Reader "SELECT TOP (1)  [Extent1].[SalesOrderID] AS [SalesOrderID],  [Extent1].[RevisionNumber] AS...  0.22s  [11956]
Execute Reader "SELECT  [Extent1].[SalesOrderID] AS [SalesOrderID],  [Extent1].[SalesOrderDetailID] AS [Sales...  0.29s  [11956]
Execute Reader "SELECT  [Extent1].[ProductID] AS [ProductID],  [Extent1].[Name] AS [Name],  [Extent1].[P...  0.30s  [11956]
Execute Reader "SELECT  [Extent1].[ProductID] AS [ProductID],  [Extent1].[Name] AS [Name],  [Extent1].[P...  0.33s  [11956]
Execute Reader "SELECT  [Extent1].[ProductID] AS [ProductID],  [Extent1].[Name] AS [Name],  [Extent1].[P...  0.33s  [11956]
Execute Reader "SELECT  [Extent1].[ProductID] AS [ProductID],  [Extent1].[Name] AS [Name],  [Extent1].[P...  0.33s  [11956]
Execute Reader "SELECT  [Extent1].[ProductID] AS [ProductID],  [Extent1].[Name] AS [Name],  [Extent1].[P...  0.34s  [11956]
Execute Reader "SELECT  [Extent1].[ProductID] AS [ProductID],  [Extent1].[Name] AS [Name],  [Extent1].[P...  0.33s  [11956]
Execute Reader "SELECT  [Extent1].[ProductID] AS [ProductID],  [Extent1].[Name] AS [Name],  [Extent1].[P...  0.34s  [11956]
Execute Reader "SELECT  [Extent1].[ProductID] AS [ProductID],  [Extent1].[Name] AS [Name],  [Extent1].[P...  0.34s  [11956]
Execute Reader "SELECT  [Extent1].[ProductID] AS [ProductID],  [Extent1].[Name] AS [Name],  [Extent1].[P...  0.34s  [11956]
Execute Reader "SELECT  [Extent1].[ProductID] AS [ProductID],  [Extent1].[Name] AS [Name],  [Extent1].[P...  0.34s  [11956]
Execute Reader "SELECT  [Extent1].[ProductID] AS [ProductID],  [Extent1].[Name] AS [Name],  [Extent1].[P...  0.34s  [11956]
Execute Reader "SELECT  [Extent1].[ProductID] AS [ProductID],  [Extent1].[Name] AS [Name],  [Extent1].[P...  0.34s  [11956]
```

FIGURE 10 LAZY LOADING DATA

Success! We now see all the detail and associated product information in the console. However, notice on the left something else that VS 2015[4] reveals: each time our program iterates over the next `SalesOrderDetail`, EF executes another SELECT statement to return the data from the database. That means that there's quite a bit of traffic back and forth from the database. Is there a better way?

Let's go back and make some more modifications:

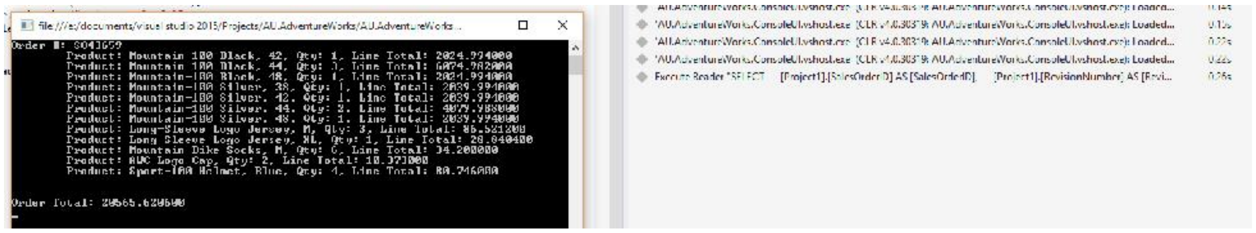
[4] This may only be available in the Enterprise version of VS. I hadn't seen it in VS 2013 Professional (or prior). Needless to say, this is *really* helpful information to know.



```
            using (var context = new AdventureWorks2014Entities())
            {
                sale = context.SalesOrderHeaders
                    .Where(x => x.SalesOrderID == 43659)
                    //Include requires using System.Data.Entity!
                    .Include(x => x.SalesOrderDetails)
                    .Include("SalesOrderDetails.Product")
                    .FirstOrDefault();
            } //moved this back
```

FIGURE 11 LISTING 1.B

Here, we're using the `.Include` extension method to eager load more data along with our results. Because of the navigation defined in our EF model, EF will only query the `SalesOrderDetails` that actually belong to the `SalesOrderHeader` that we ask for. Run the application again. Note above that we're including both the `SalesOrderDetails` and the `Product` associated with each detail. I've used two different ways to specify these navigations.



file:///e:/documents/visual studio 2015/Projects/AU.AdventureWorks/AU.AdventureWorks...

```
Order #: SO43659
        Product: Mountain 100 Black, 42, Qty: 1, Line Total: 2024.994000
        Product: Mountain 100 Black, 44, Qty: 3, Line Total: 6074.982000
        Product: Mountain-100 Black, 48, Qty: 1, Line Total: 2024.994000
        Product: Mountain-100 Silver, 38, Qty: 1, Line Total: 2039.994000
        Product: Mountain-100 Silver, 42, Qty: 1, Line Total: 2039.994000
        Product: Mountain-100 Silver, 44, Qty: 2, Line Total: 4079.988000
        Product: Mountain-100 Silver, 48, Qty: 1, Line Total: 2039.994000
        Product: Long-Sleeve Logo Jersey, M, Qty: 3, Line Total: 86.521200
        Product: Long Sleeve Logo Jersey, XL, Qty: 1, Line Total: 28.840400
        Product: Mountain Bike Socks, M, Qty: 6, Line Total: 34.200000
        Product: AWC Logo Cap, Qty: 2, Line Total: 10.373000
        Product: Sport-100 Helmet, Blue, Qty: 4, Line Total: 80.746000

Order Total: 20565.620600
```

```
'AU.AdventureWorks.ConsoleUI.vshost.exe' (CLR v4.0.30319: AU.AdventureWorks.ConsoleUI.vshost.exe): Loaded...   0.14s
'AU.AdventureWorks.ConsoleUI.vshost.exe' (CLR v4.0.30319: AU.AdventureWorks.ConsoleUI.vshost.exe): Loaded...   0.15s
'AU.AdventureWorks.ConsoleUI.vshost.exe' (CLR v4.0.30319: AU.AdventureWorks.ConsoleUI.vshost.exe): Loaded...   0.22s
'AU.AdventureWorks.ConsoleUI.vshost.exe' (CLR v4.0.30319: AU.AdventureWorks.ConsoleUI.vshost.exe): Loaded...   0.22s
Execute Reader "SELECT    [Project1].[SalesOrderID] AS [SalesOrderID],    [Project1].[RevisionNumber] AS [Revi...   0.26s
```

FIGURE 12 EAGER LOADING DATA

This time, notice that not only do we avoid the error we got before (because all the data is already included), but there's also only *one* SELECT statement sent to the database. To be sure, a bigger chunk of data will be returned in the request, but overall we've decreased a lot of back-and-forth from the database, and sped up our application quite a bit. It's important to note that neither lazy loading nor eager loading is inherently bad or preferable. Which you use will be very situation dependent; they both have their place.

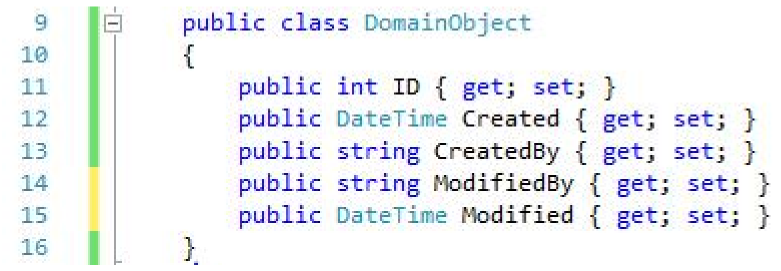
We've just had a brief introduction to the Database First approach to using Entity Framework. This can be a good way to get started with EF, or if you're inheriting an existing database and probably won't be changing the structure so much—though using Database First certainly does not mean you can't.

Creating a Code First Model

We're now going to switch gears to try out the Code First approach. Rather than having code "mysteriously" generated for you by EF, many developers favor this approach because it provides so much control over the model entities. It's also a highly useful approach for "greenfield" database development, and you almost never have to touch the database itself.

In this section, we're going to create a fairly simple model based on a common scenario you might encounter in the Oil & Gas industry. We'd like to keep track of a list of projects, each of which can have one or more associated items of equipment.

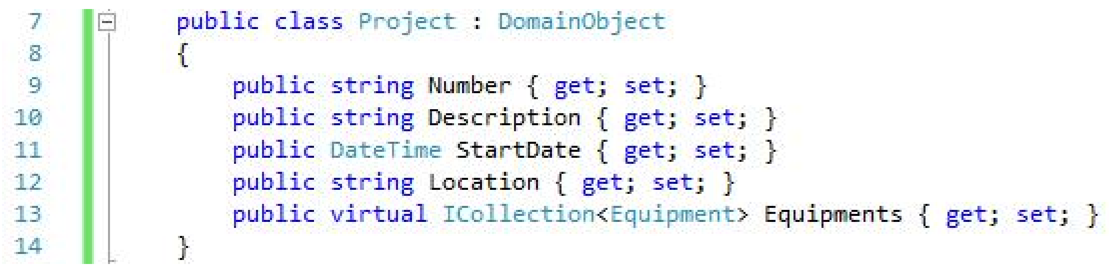
Close the AU.AdventureWorks solution and start a new C# Class Library project called **AU.CodeFirst.Model**, and change the solution name to **AU.CodeFirst**. Delete the default Class1 file, then add a new class called **DomainObject**. Because this is a brand new, from scratch database, and we have the run of the place, I want all of my model objects to have some common properties. So I'm going to have most of my classes inherit from `DomainObject`. This way I don't have to repeat a lot of common code. Add the following class:



```
public class DomainObject
{
    public int ID { get; set; }
    public DateTime Created { get; set; }
    public string CreatedBy { get; set; }
    public string ModifiedBy { get; set; }
    public DateTime Modified { get; set; }
}
```

FIGURE 13 LISTING 2

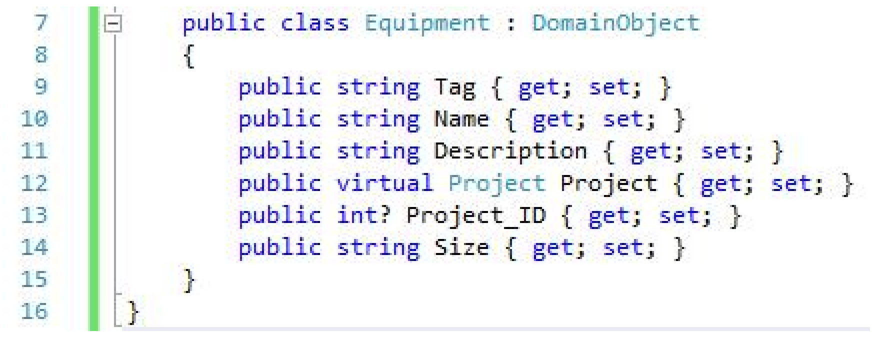
Next, we'll add our `Project` class:



```
public class Project : DomainObject
{
    public string Number { get; set; }
    public string Description { get; set; }
    public DateTime StartDate { get; set; }
    public string Location { get; set; }
    public virtual ICollection<Equipment> Equipments { get; set; }
}
```

FIGURE 14 LISTING 3

Pretty straightforward. Note the inheritance from `DomainObject`. Also note the `Equipments` property, an `ICollection<Equipment>`. It's marked with the `virtual` keyword which is needed for EF to provide lazy loading of navigable properties. Thus, each Project can have a collection of Equipment associated with it. Next, we'll add the `Equipment` class:



```
public class Equipment : DomainObject
{
    public string Tag { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public virtual Project Project { get; set; }
    public int? Project_ID { get; set; }
    public string Size { get; set; }
}
}
```

FIGURE 15 LISTING 4

The main thing to note here are the `Project` and `Project_ID` properties. The `Project` property is used to create a navigation property from each piece of equipment back to the project it belongs to. The `Project_ID` field will be used to create the Foreign Key in the database[5].

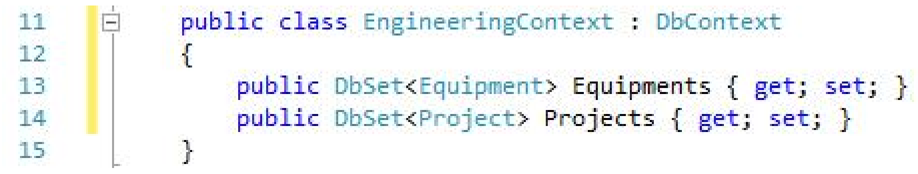
[5] Although it's not strictly required to specify a Foreign Key, EF will in fact create an FK field in the database for you. It is highly recommended that you include it in your model so that you can retrieve objects by ID values (integers) rather than having to retrieve the actual object (i.e. a Project) first.

Creating a context object
In order to make this into a fully functional EF model, we need to create a class that inherits from a special Entity Framework class, `DbContext`. This class acts as the glue that binds our model together, and is the engine[6] that makes it all work.

Add a new C# Class Library project to the solution named **AU.CodeFirst.DataAccess**. Add references to the following:

- AU.CodeFirst.Model
- EntityFramework (via NuGet)

Next, add a new class called **EngineeringContext** with the following code:



```
11    public class EngineeringContext : DbContext
12    {
13        public DbSet<Equipment> Equipments { get; set; }
14        public DbSet<Project> Projects { get; set; }
15    }
```

FIGURE 16 LISTING 5

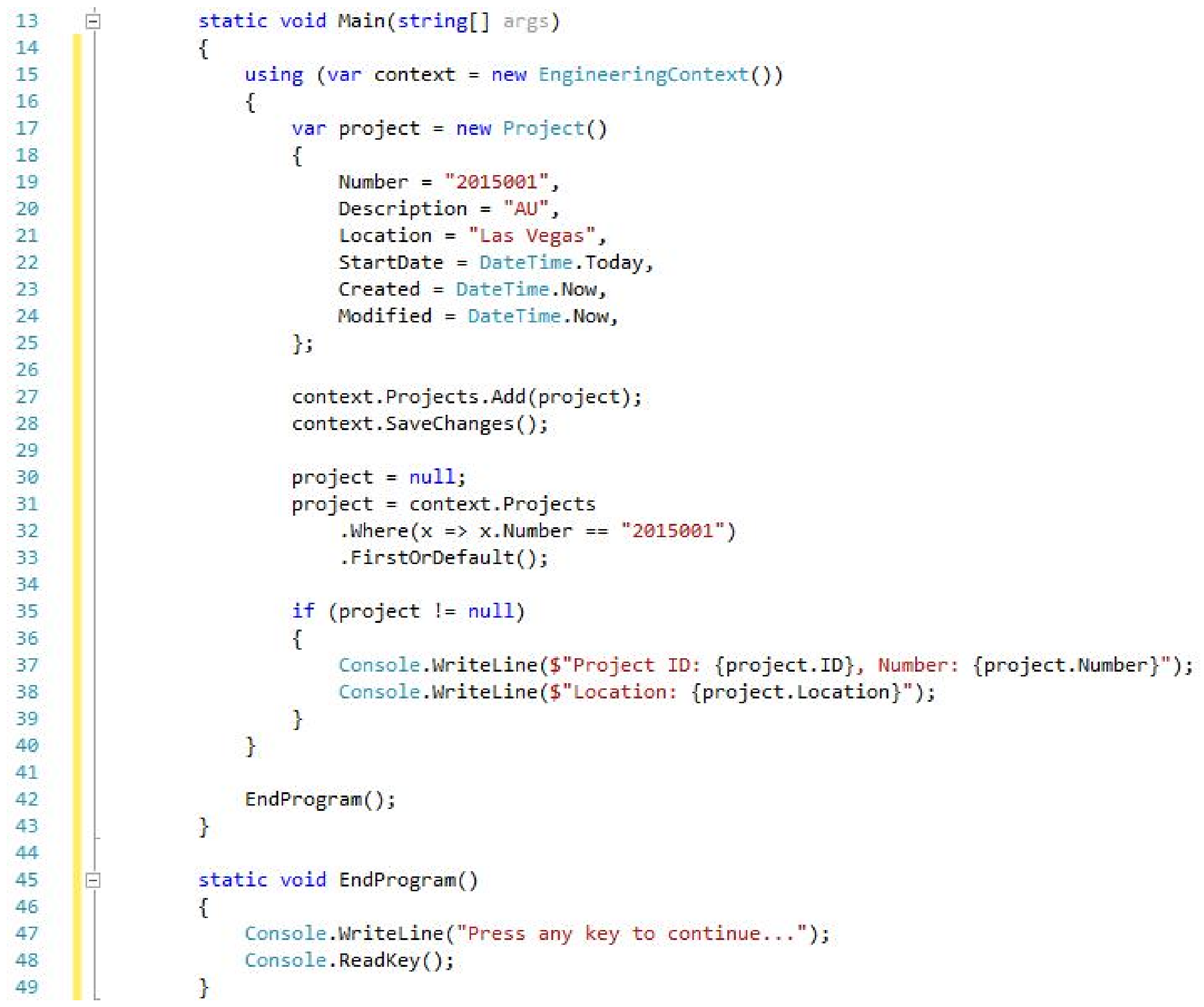
- Line 11: The class inherits from `DbContext`, an EF class.
- Lines 13-14: Use `DbSet<T>` to expose the projects and equipment objects from our model.

We're almost there, but we can't run the application, nor do we have a database yet. Let's add a console app to the solution, and try and get things running. Name the project **AU.CodeFirst.ConsoleUI** and set it as the Startup Project. Add references to the following:

- AU.CodeFirst.DataAccess
- AU.CodeFirst.Model
- EntityFramework (via NuGet).

Edit the `Program.Main` method as shown below:

[6] Hmm, a gluey-engine. Maybe I should think of a better metaphor.

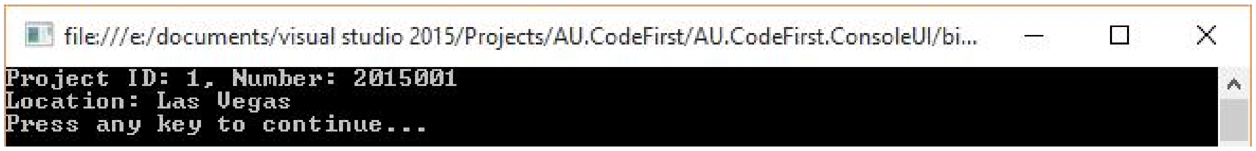


```
13          static void Main(string[] args)
14          {
15              using (var context = new EngineeringContext())
16              {
17                  var project = new Project()
18                  {
19                      Number = "2015001",
20                      Description = "AU",
21                      Location = "Las Vegas",
22                      StartDate = DateTime.Today,
23                      Created = DateTime.Now,
24                      Modified = DateTime.Now,
25                  };
26
27                  context.Projects.Add(project);
28                  context.SaveChanges();
29
30                  project = null;
31                  project = context.Projects
32                      .Where(x => x.Number == "2015001")
33                      .FirstOrDefault();
34
35                  if (project != null)
36                  {
37                      Console.WriteLine($"Project ID: {project.ID}, Number: {project.Number}");
38                      Console.WriteLine($"Location: {project.Location}");
39                  }
40              }
41
42              EndProgram();
43          }
44
45          static void EndProgram()
46          {
47              Console.WriteLine("Press any key to continue...");
48              Console.ReadKey();
49          }
```

FIGURE 17 LISTING 6

- Line 15: The context code is essentially the same as what we used in Database First style development. Once we have a model and DbContext derived class (however we choose to get there), any “consuming” code is the same.
- Line 17-25: Create a new Project object to add to our database.
- Line 27: Add the project to the context’s `Projects` property.
- Line 28: Save the changes to the database.
- Lines 31-33: Retrieve the project by `Number`.
- Lines 37-38: Display the information, including the project’s `ID` value, which comes from the database.

Run the application and you should see some evidence—note the `ID` number, which we did NOT specify in code—that we are retrieving data from the database:


file:///e:/documents/visual studio 2015/Projects/AU.CodeFirst/AU.CodeFirst.ConsoleUI/bi...

```
Project ID: 1, Number: 2015001
Location: Las Vegas
Press any key to continue...
```

FIGURE 18 READING DATA FROM THE DATABASE

So we saved data to a database, and read it back, which is great. But it raises a question: what database did our application just write to and read from? We haven't done anything to configure this anywhere in our solution.

Configure database options using Fluent API and DataAnnotations

One of the nice things about using CodeFirst on a new database project is that EF does a lot of work for you in the background to minimize your effort to get your project up and running. Starting in VS 2010, Microsoft introduced something called LocalDB, which is an even lighter version of SQL Server Express. If you don't have SQL Server Express (or higher) installed on your development computer, VS will just use LocalDB to host a local version of any databases EF creates[7].

If we look in SQL Server Object Explorer, you should see a (localdb) instance[8]. Drill down into Databases and you should see a database named AU.CodeFirst.DataAccess.EngineeringContext, and if you drill into Tables, you should see three tables: dbo.__MigrationHistory, dbo.Projects and dbo.Equipments. We'll talk about the MigrationHistory table more as we get further into CodeFirst Migrations.

Expand **Equipments > Columns**, and **Projects > Columns** to see what EF has done for you.

You should see the `ID` columns marked as PK (Primary Key), any string properties are set as `nvarchar(max)`. You'll also note that Equipments has a `ProjectId` or a `Project_ID1` column which is a Foreign Key. Lots of work done for us, and it's a good starting point, but most of us like more control than this…and what's up with EF not making `Project_ID` our FK?

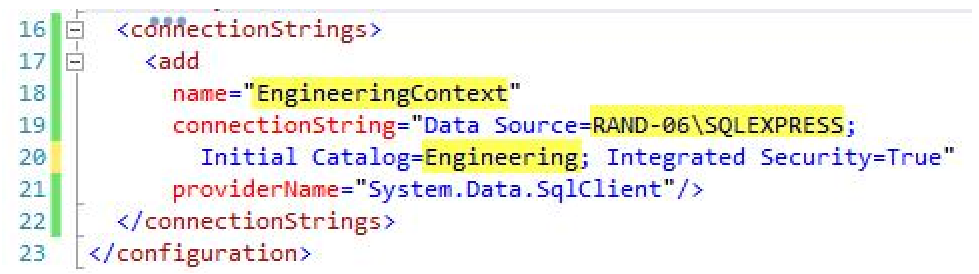
Configuring the Database Name

`DbContext` has several conventions for getting to the database name. If we don't provide something, EF will provide something for us[9]. In this case, EF creates a database corresponding to our solution name + the name of our context class, thus `AU.CodeFirst.DataAccess.EngineeringContext`.

`DbContext` has a constructor that accepts a connection string name. And, by default it will look for a connection string named the same as the DbContext-derived class (`EngineeringContext` in our case). Open **App.config** in the AU.CodeFirst.ConsoleUI project.

Copy the following code into the `<configuration>` element:

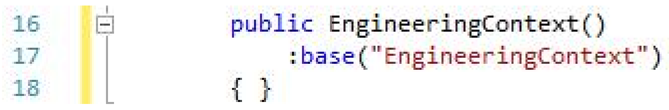
[7] Of course, we'll eventually need to have the database on an instance of SQL Server Express (or higher) for production use. But that task *can* be pushed down the road a bit.
[8] If you have some flavor of SQL Server installed on your development laptop, VS will use that in favor of localdb, so you may need to add a connection to your SQL Server instance in SQL Server Object Explorer.
[9] "If you don't have a lawyer, one will be appointed for you…"



```
  <connectionStrings>
    <add
      name="EngineeringContext"
      connectionString="Data Source=RAND-06\SQLEXPRESS;
        Initial Catalog=Engineering; Integrated Security=True"
      providerName="System.Data.SqlClient"/>
  </connectionStrings>
</configuration>
```

FIGURE 19 LISTING 7[10]

We'll also need this in the DataAccess project's App.config, for Migrations, so paste it in that file as well. I've highlighted the name of our connection string, the SQL Server instance name, and the name of the database, `Engineering`. We need to update our `EngineeringContext` class to use the new connection string. Add this constructor:



```
        public EngineeringContext()
            :base("EngineeringContext")
        { }
```

FIGURE 20 LISTING 8

This tells EF to look in the startup application's App.config file and search for a `connectionString` named `EngineeringContext`. It uses the connection information to connect to the appropriate SQL Server instance and create (if necessary) the database specified as the `Initial Catalog`.

Run the application again, and then check SQL Server Object Explorer. We should see a new Engineering database appear.

Using Data Annotations to configure database options

When we looked at the database schema EF created for us, we saw that all the string properties were made into `nvarchar(max)` columns. These require a lot more storage space than we're likely to need for most string values. We may also want to specify that some columns are required, and we may want different naming for some of the tables and columns than the conventions EF uses to derive these.

Fortunately, EF gives us plenty of control over all this configuration, using either DataAnnotations or the Fluent API. Which you choose is largely a matter of preference[11].

First, let's look at using DataAnnotations to indicate required fields, and to limit string column sizes. In the **AU.CodeFirst.Model** project, add a reference to:

- System.ComponentModel.DataAnnotations

Open the **Equipment** class, and add the following using statements[12]:

```
using System.ComponentModel.DataAnnotations;
```

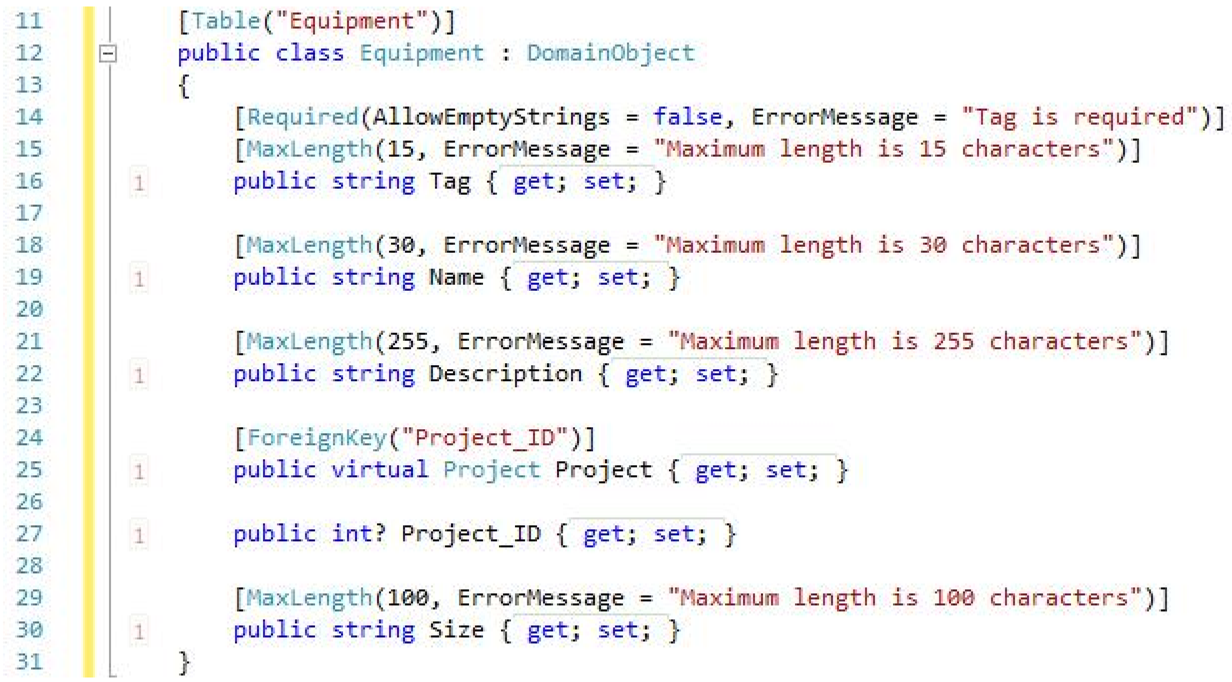
[10] You will need to input the name of your SQL Express instance or (localdb) for the `Data Source`.

[11] There are some things you can only do with the Fluent API, but by and large you can do most configuration using either DataAnnotations or Fluent API. You can even use a mixture as I will demonstrate in the code.

[12] This is usually strike one in the argument against DataAnnotations, you have to drag one more reference into otherwise "clean" POCO classes. I'm not that much of a purist…if you need it you need it, but it's still something to consider.

```
using System.ComponentModel.DataAnnotations.Schema;
```

Next, make the following modifications:



```
[Table("Equipment")]
public class Equipment : DomainObject
{
    [Required(AllowEmptyStrings = false, ErrorMessage = "Tag is required")]
    [MaxLength(15, ErrorMessage = "Maximum length is 15 characters")]
    public string Tag { get; set; }

    [MaxLength(30, ErrorMessage = "Maximum length is 30 characters")]
    public string Name { get; set; }

    [MaxLength(255, ErrorMessage = "Maximum length is 255 characters")]
    public string Description { get; set; }

    [ForeignKey("Project_ID")]
    public virtual Project Project { get; set; }

    public int? Project_ID { get; set; }

    [MaxLength(100, ErrorMessage = "Maximum length is 100 characters")]
    public string Size { get; set; }
}
```

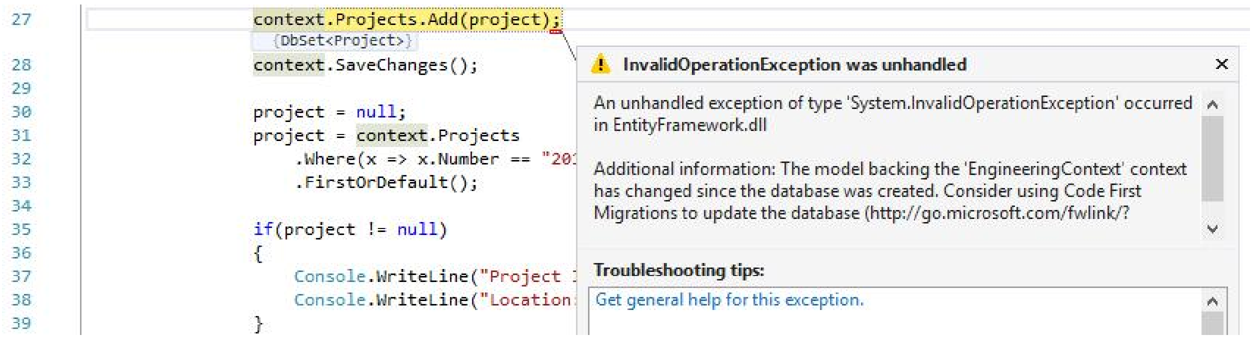
FIGURE 21 LISTING 9

Here, I've used only a few of the many DataAnnotations[13] available, `Table`, `Required` and `MaxLength`. Others include `DataType`, `Range`, `RegularExpression`, `CreditCard` and `Phone`. These annotations are used to determine the SQL code required to generate table and column names, data types and sizes, and constraints. They can also serve to validate object data before the data gets sent to the database.

Code First Migrations

Let's run our project again and see what happens.

[13] Strike two against DataAnnotations is that our previously clean POCO class is now quite a bit more difficult to read, with all the annotations getting in the way. Really it's a matter of preference and coding style. Wait until you've tried Fluent API, and see what you prefer. Ultimately, you can mix and match, but that could be confusing when the code enters the maintenance phase, or if you have conflicting configurations.



```
        context.Projects.Add(project);
          {DbSet<Project>}
        context.SaveChanges();

        project = null;
        project = context.Projects
            .Where(x => x.Number == "20
            .FirstOrDefault();

        if(project != null)
        {
            Console.WriteLine("Project 
            Console.WriteLine("Location
        }
```

InvalidOperationException was unhandled

An unhandled exception of type 'System.InvalidOperationException' occurred in EntityFramework.dll

Additional information: The model backing the 'EngineeringContext' context has changed since the database was created. Consider using Code First Migrations to update the database (http://go.microsoft.com/fwlink/?

Troubleshooting tips:

Get general help for this exception.

FIGURE 22 A MODEL CHANGE ERROR

Wait a minute. The first time we ran the project, EF created the database for us, no argument. We just made some changes to our model (adding DataAnnotations), which definitely require some changes to the database schema, but why didn't EF just do it for us?

EF is trying to protect you from yourself a bit here. Database schema changes are *potentially* dangerous. You could be dropping columns or tables (i.e. LOSING DATA) and so you need to be cautious about these sorts of changes[14]. When your application starts up, Entity Framework initializes the model and compares it against the database schema to make sure they're in sync[15].

> I have been using EF since around 2008, most of the time developing directly against production databases. While this is definitely NOT a best practice recommendation, I have never *inadvertently* lost data because of something I did with EF schema changes. However, the potential is there and the danger is real. Be careful making schema changes and be sure you understand what EF is doing. If you're not sure, create a sample project to test unfamiliar functionality first. Make a backup of a production database and restore it to a development server to test/develop against.

The error message suggests using Code First Migrations. This is a terrific tool that helps migrate changes from your EF model to the database schema[16]. In Visual Studio, go to **View > Other Windows > Package Manager Console** (referred to as PMC going forward). PMC uses PowerShell to give you access to lots of cool functionality. It's kind of like Autocad's command-line, so Autocad devs might feel right at home.

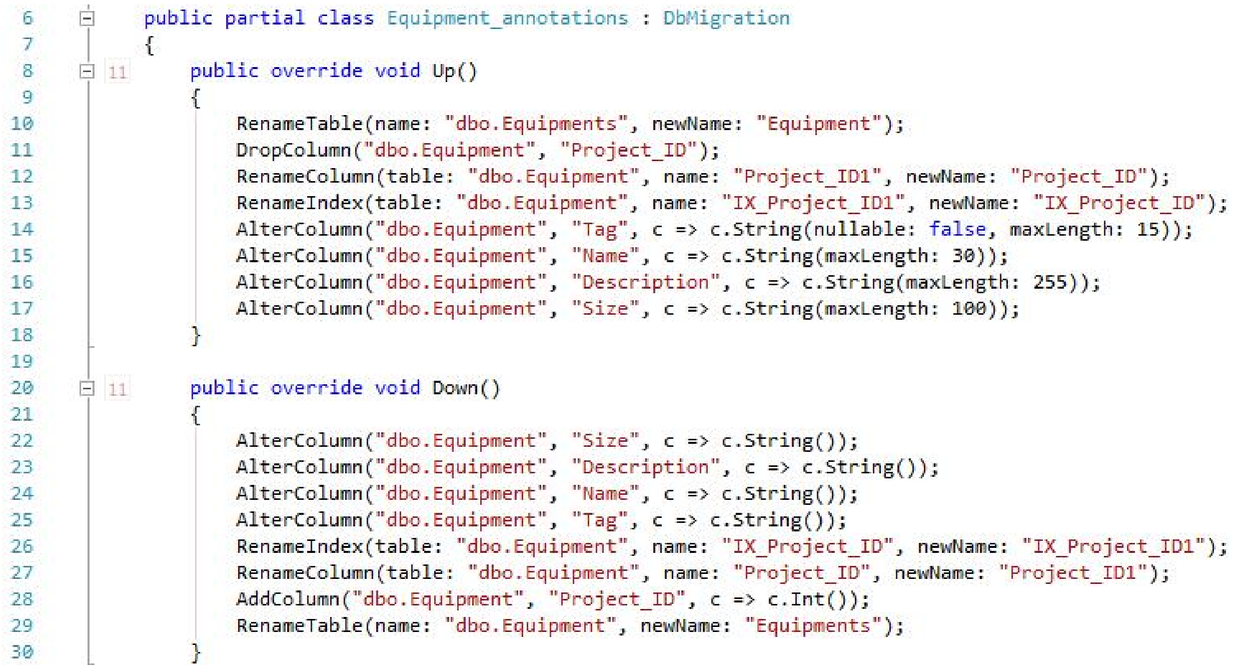
In the PMC, set the **Default Project** to **AU.CodeFirst.DataAccess**. This is important because it's where the DbContext derived class lives. At the command prompt, type **Enable-Migrations** and press Enter. Several things will happen:

- A **Migrations** folder is created. This folder will store migration classes that will be created along the way, enabling you to apply and even roll back schema changes.
- A **Configuration** class is added. This class configures certain settings about how EF functions, as well as provides a means for seeding data into the database for development purposes.
- An **InitialCreate** class. Because our database already exists, the `InitialCreate` class is added to provide a base point and to give us a roll-back capability if we wanted to discard a given change.

[14] EF has some other built-in protections if proposed schema changes will result in data loss.
[15] This might sound time-consuming, but EF uses some tricks to speed this up and in practice it's pretty fast.
[16] You *could* go make all the changes in the database yourself, but it would be very tedious and difficult to get exactly right.

The cool thing about the Migration classes from a non-SQL developer standpoint is that it isn't SQL, it's just code. EF will convert it to SQL code, but the generated migration code is pretty easy to get the gist of, and even make modifications to (if necessary)[17].

We're going to baby step our way into this. At the PM Console, type **`Add-Migration Equipment_annotations`** and press Enter. This will create a new **Equipment_annotations** class for us with the following migration code:



```
public partial class Equipment_annotations : DbMigration
{
    public override void Up()
    {
        RenameTable(name: "dbo.Equipments", newName: "Equipment");
        DropColumn("dbo.Equipment", "Project_ID");
        RenameColumn(table: "dbo.Equipment", name: "Project_ID1", newName: "Project_ID");
        RenameIndex(table: "dbo.Equipment", name: "IX_Project_ID1", newName: "IX_Project_ID");
        AlterColumn("dbo.Equipment", "Tag", c => c.String(nullable: false, maxLength: 15));
        AlterColumn("dbo.Equipment", "Name", c => c.String(maxLength: 30));
        AlterColumn("dbo.Equipment", "Description", c => c.String(maxLength: 255));
        AlterColumn("dbo.Equipment", "Size", c => c.String(maxLength: 100));
    }

    public override void Down()
    {
        AlterColumn("dbo.Equipment", "Size", c => c.String());
        AlterColumn("dbo.Equipment", "Description", c => c.String());
        AlterColumn("dbo.Equipment", "Name", c => c.String());
        AlterColumn("dbo.Equipment", "Tag", c => c.String());
        RenameIndex(table: "dbo.Equipment", name: "IX_Project_ID", newName: "IX_Project_ID1");
        RenameColumn(table: "dbo.Equipment", name: "Project_ID", newName: "Project_ID1");
        AddColumn("dbo.Equipment", "Project_ID", c => c.Int());
        RenameTable(name: "dbo.Equipment", newName: "Equipments");
    }
```

FIGURE 23 LISTING 10

Two methods are overridden: `Up` and `Down`. `Up` is used for new modifications to the database, `Down` is used to roll them back. Notice that the "Equipments" table will now be renamed to "Equipment", thanks to the `Table` annotation. And several existing columns will be altered, changing their size, data type (in some cases), and whether or not they are required. Note especially the dropping of the `Project_ID` column and the renaming of `Project_ID1` to `Project_ID`. This is what the `ForeignKey` attribute does to realign the foreign key the way we originally wanted it.

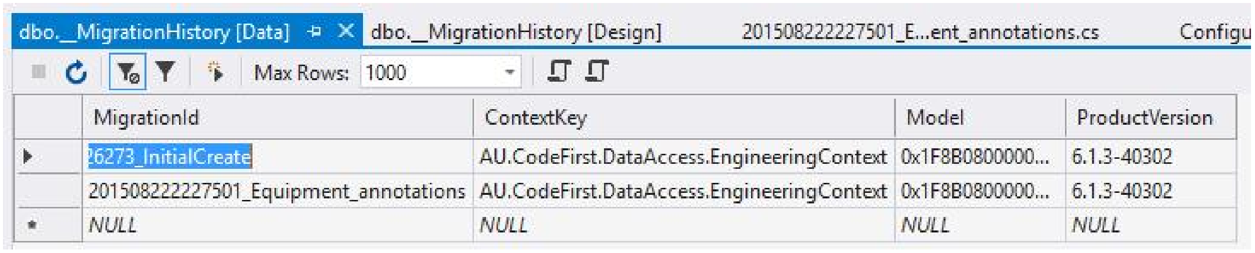
> It's important to READ the migration code before proceeding to the next step. I catch many problems in my model configuration by reading this code first. After you fix the configuration error(s), type **`Add-Migration <name_of_migration> –force`** to regenerate the migration file.

After reading the migration code, type **`Update-Database`** at the PM Console, and press Enter. If you want to see the actual SQL code[18], type **`Update-Database –verbose`**. Assuming there aren't any errors

[17] Be careful about making modifications to migration classes. Usually, if there's a problem it's because of an underlying issue in your model configuration. The underlying issues should be addressed, then the migration class can be regenerated. There are *some* valid cases where you would edit a migration class, but in my experience they're fairly rare.

[18] If you want to output the actual SQL code so that a DBA can run the script, use `update-database –script`.

from SQL Server, the migration applies the changes to the database. Information about the migration is stored in the __MigrationHistory table.


dbo.__MigrationHistory [Data] | dbo.__MigrationHistory [Design] | 201508222227501_E...ent_annotations.cs | Configu

Max Rows: 1000

	MigrationId	ContextKey	Model	ProductVersion
▶	26273_InitialCreate	AU.CodeFirst.DataAccess.EngineeringContext	0x1F8B0800000...	6.1.3-40302
	201508222227501_Equipment_annotations	AU.CodeFirst.DataAccess.EngineeringContext	0x1F8B0800000...	6.1.3-40302
*	NULL	NULL	NULL	NULL

FIGURE 24 MIGRATIONHISTORY TABLE

EF uses the information in this table to compare the existing database to any changes you may have made in your model, in order to compute the migration files. Leave this table to EF.

Many-to-Many Relationships
A design change has been requested for our application. We need to expand our model to include contractors, which can be associated with a project. Of course, one project can have multiple contractors, and one contractor can work on multiple projects. This leads us to a more complicated "many to many" relationship.

We also want all our model objects to support a Comments field, where a user could enter whatever comment they wanted to associate with that record. Let's make those changes before proceeding to our discussion of Fluent API.

In the **AU.CodeFirst.Model** project, open the **DomainObject** class and add the following property to it:

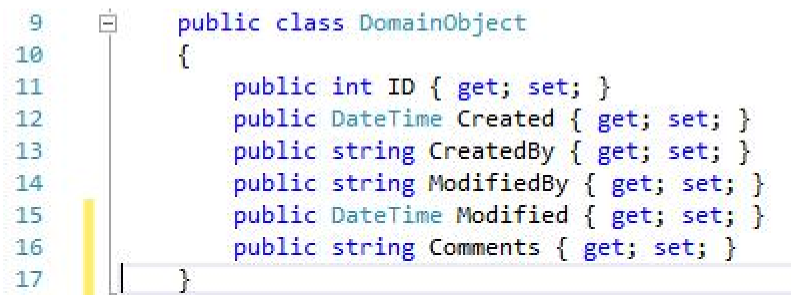

```
public class DomainObject
{
    public int ID { get; set; }
    public DateTime Created { get; set; }
    public string CreatedBy { get; set; }
    public string ModifiedBy { get; set; }
    public DateTime Modified { get; set; }
    public string Comments { get; set; }
}
```

FIGURE 25 LISTING 11

Next, add the following class:

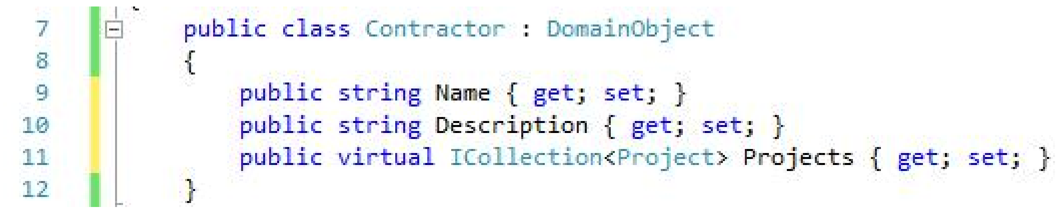
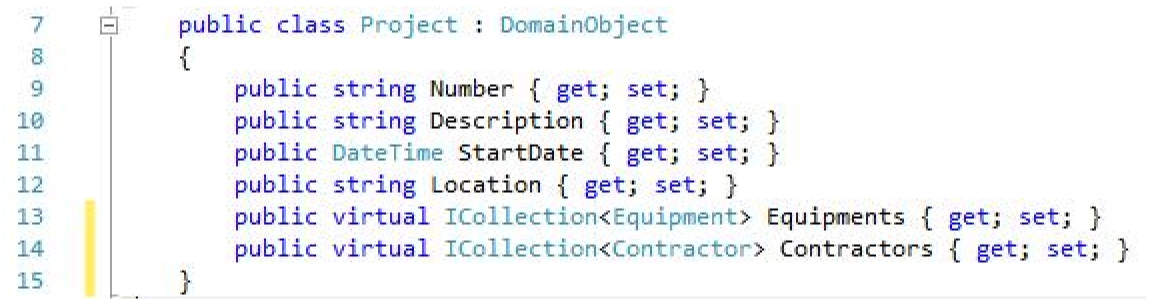

```
public class Contractor : DomainObject
{
    public string Name { get; set; }
    public string Description { get; set; }
    public virtual ICollection<Project> Projects { get; set; }
}
```

FIGURE 26 LISTING 12

Add the following property to the Project class:

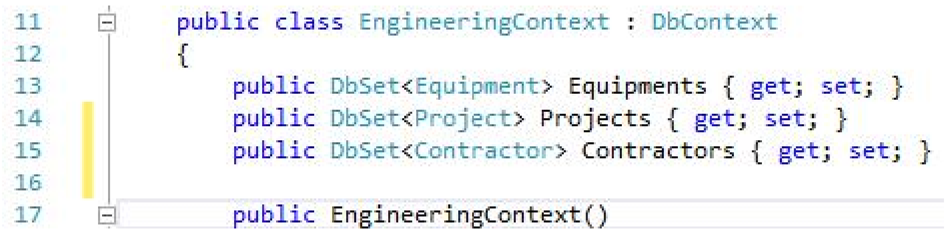


```
public class Project : DomainObject
{
    public string Number { get; set; }
    public string Description { get; set; }
    public DateTime StartDate { get; set; }
    public string Location { get; set; }
    public virtual ICollection<Equipment> Equipments { get; set; }
    public virtual ICollection<Contractor> Contractors { get; set; }
}
```

FIGURE 27 LISTING 13

A project can have many contractors, and a contractor can have many projects.

We also need to update our EngineeringContext, to add the new `DbSet<Contractor>` property. Add the following to the EngineeringContext class:

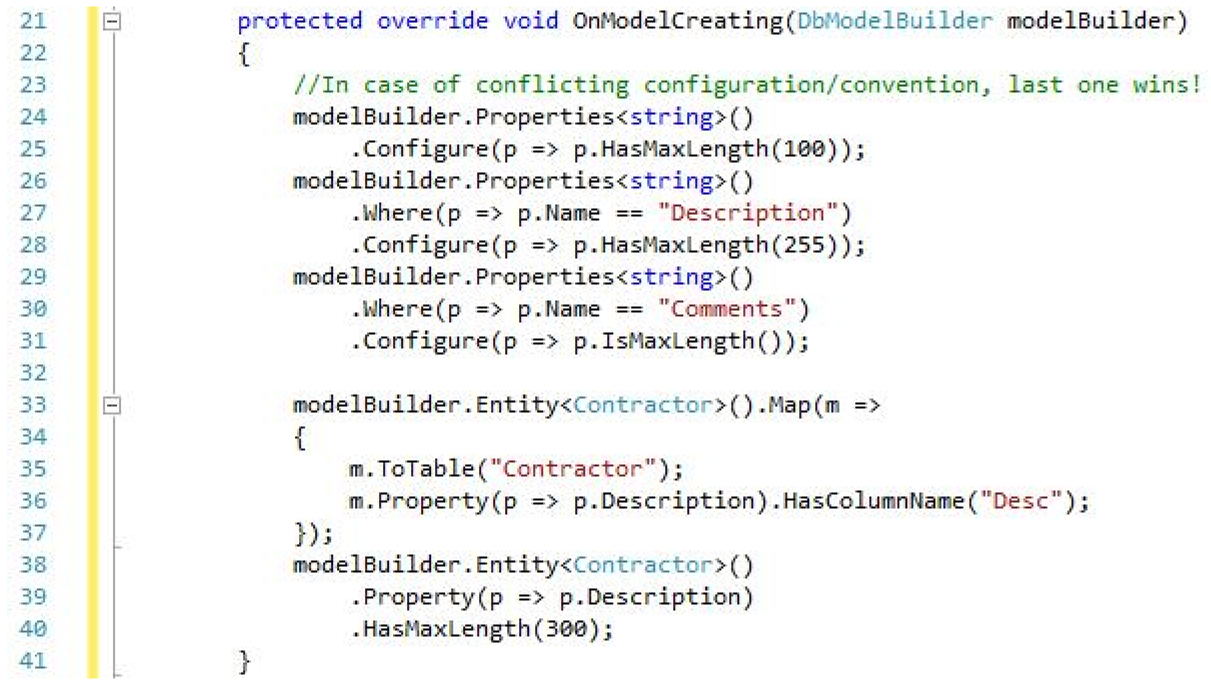


```
public class EngineeringContext : DbContext
{
    public DbSet<Equipment> Equipments { get; set; }
    public DbSet<Project> Projects { get; set; }
    public DbSet<Contractor> Contractors { get; set; }

    public EngineeringContext()
```

FIGURE 28 LISTING 14

Using the Fluent API to configure database options
Instead of DataAnnotations, we'll now use Fluent API to specify configuration options. Rather than explicitly configuring the length of every string field to avoid unwanted `nvarchar(max)` fields, we can configure a more reasonable default. Many of our (eventual) tables might have similar fields, like "Description". So, we'll give all fields named "Description" a different default from the standard default for string fields.

The key method for configuration of the EF model is the `OnModelCreating` method of our `DbContext`-derived class. We'll override this method in the **EngineeringContext** class in the **AU.CodeFirst.DataAccess** project. Add the following code:



```
21          protected override void OnModelCreating(DbModelBuilder modelBuilder)
22          {
23              //In case of conflicting configuration/convention, last one wins!
24              modelBuilder.Properties<string>()
25                  .Configure(p => p.HasMaxLength(100));
26              modelBuilder.Properties<string>()
27                  .Where(p => p.Name == "Description")
28                  .Configure(p => p.HasMaxLength(255));
29              modelBuilder.Properties<string>()
30                  .Where(p => p.Name == "Comments")
31                  .Configure(p => p.IsMaxLength());
32
33              modelBuilder.Entity<Contractor>().Map(m =>
34              {
35                  m.ToTable("Contractor");
36                  m.Property(p => p.Description).HasColumnName("Desc");
37              });
38              modelBuilder.Entity<Contractor>()
39                  .Property(p => p.Description)
40                  .HasMaxLength(300);
41          }
```

FIGURE 29 LISTING 15

Note the following:

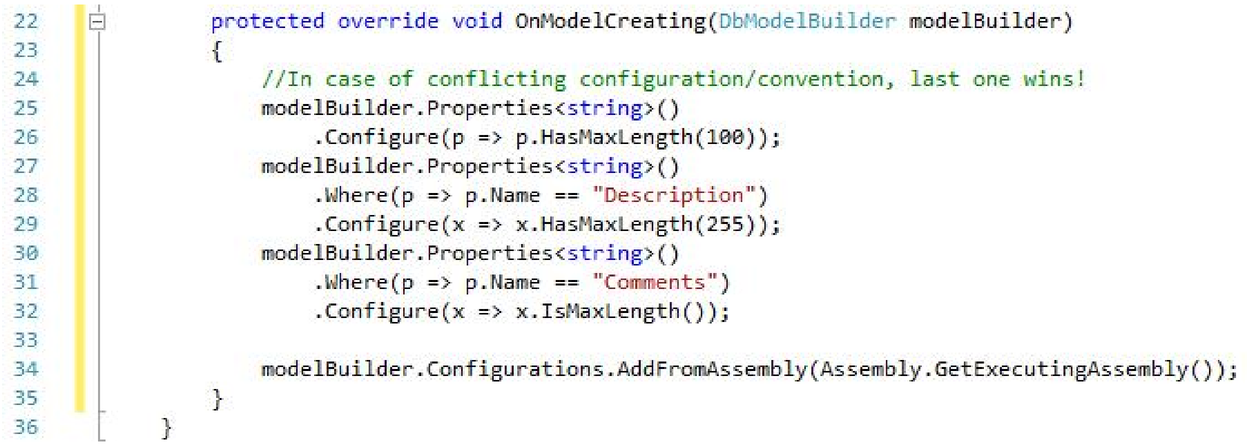
- Lines 24-25: Here we specify that ANY string property in the model defaults to a maximum length of 100 characters, unless some later configuration specifies otherwise.
- Lines 26-28: Next, we make a more specific configuration where only string properties named "Description" default to a maximum length of 255 characters.
- Lines 29-31: We add another specific configuration for any string field named "Comments," which be max length in the database.
- Lines 33-37: Configure the Contractor entity to Map to a table named "Contractor" and redirect the "Description" field in our class to a column named "Desc" in the database[19].
- Lines 38-40: Change the max length of the Description property for the Contractor entity to have a length of 300 characters. This later (and more specific) configuration trumps the previous "Description" configuration.

Using EntityTypeConfiguration Classes

As your model grows, `OnModelCreating` is going to get big in a hurry, since you will likely have at least some configuration to do for each class in the model. I don't particularly like having long, complex methods like this, preferring instead to move configurations for each entity to their own separate configuration class. I only use `OnModelCreating` for the "global" configurations, and to call the `Configurations.AddFromAssembly` method[20]. Let's pare that method down to this:

[19] This isn't something I'd typically do on purpose, UNLESS the database already existed and had a naming convention I didn't like. Then I might choose a better name as the property name in my model class, but still map it to the existing column name in the database.

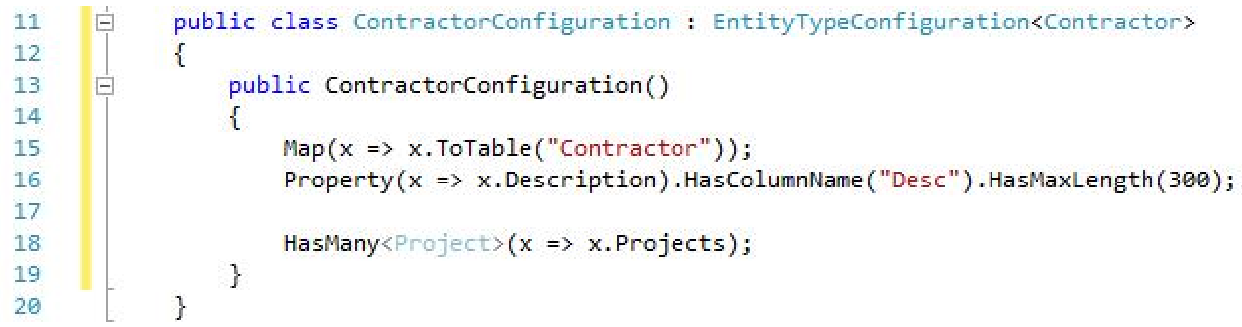
[20] This method is new to EF 6, so it isn't there in earlier versions of EF.



```
22          protected override void OnModelCreating(DbModelBuilder modelBuilder)
23          {
24              //In case of conflicting configuration/convention, last one wins!
25              modelBuilder.Properties<string>()
26                  .Configure(p => p.HasMaxLength(100));
27              modelBuilder.Properties<string>()
28                  .Where(p => p.Name == "Description")
29                  .Configure(x => x.HasMaxLength(255));
30              modelBuilder.Properties<string>()
31                  .Where(p => p.Name == "Comments")
32                  .Configure(x => x.IsMaxLength());
33
34              modelBuilder.Configurations.AddFromAssembly(Assembly.GetExecutingAssembly());
35          }
36      }
```

FIGURE 30 LISTING 15.A

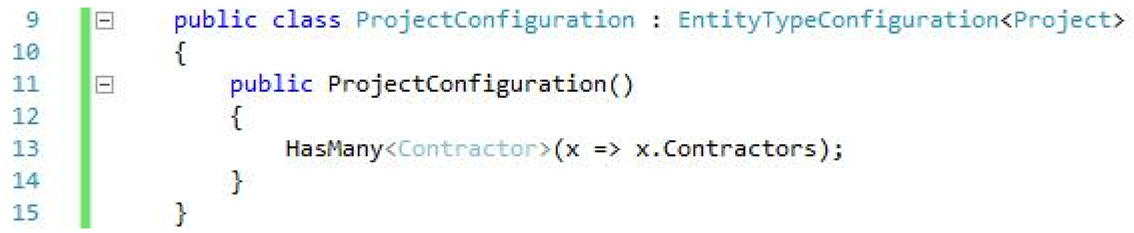
Next, we'll create separate configuration classes, derived from EF's `EntityTypeConfiguration<T>` class. Because there will be (potentially) many of these, I'll create a separate folder for them. Right-click the AU.CodeFirst.DataAccess project and choose **Add New Folder**. Name the folder **Configurations**. Add the following class to the new folder:



```
11      public class ContractorConfiguration : EntityTypeConfiguration<Contractor>
12      {
13          public ContractorConfiguration()
14          {
15              Map(x => x.ToTable("Contractor"));
16              Property(x => x.Description).HasColumnName("Desc").HasMaxLength(300);
17
18              HasMany<Project>(x => x.Projects);
19          }
20      }
```

FIGURE 31 LISTING 16

You only need to provide a constructor in this class, with the particulars of the configuration you want for that type of object. You'll note that the code is pretty similar, though there are some syntactical differences from what we saw in lines 33-40 of Listing 15. While we're at it, we need a configuration class for the Project class as well:



```
 9      public class ProjectConfiguration : EntityTypeConfiguration<Project>
10      {
11          public ProjectConfiguration()
12          {
13              HasMany<Contractor>(x => x.Contractors);
14          }
15      }
```

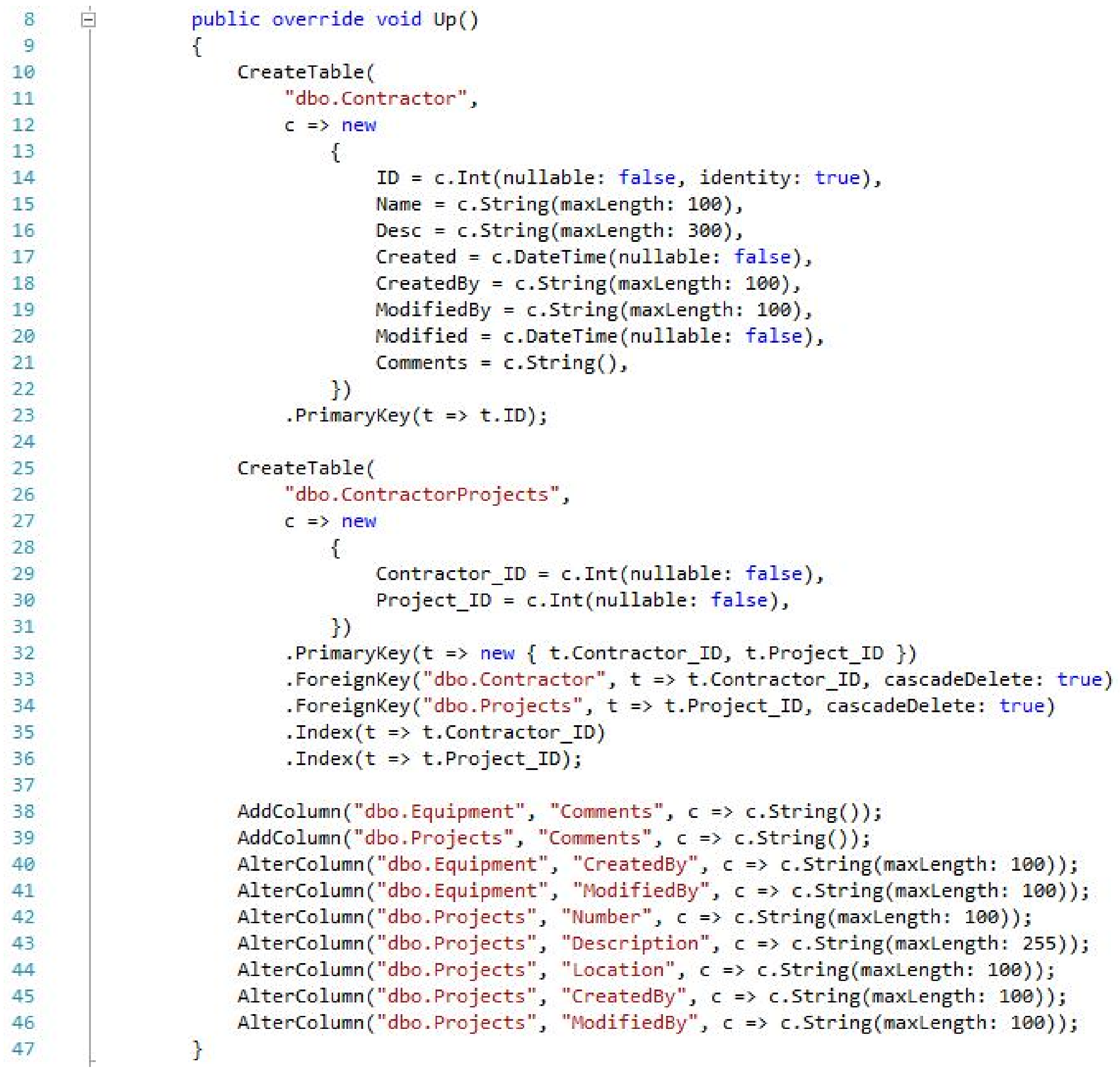
FIGURE 32 LISTING 17

- Line 13: Note the `HasMany` statement in both configuration classes (line 18 of Listing 16). This is how we tell EF that a Project has many Contractors and a Contractor has many projects. We'll see the effect of this shortly.

We've done quite a lot of modification to our model. It's high time we create a migration and update the database. In the PM Console, type:

```
Add-Migration Configuration_and_Contractor
```

The configuration class is generated and opened. There's quite a lot going on, but in the interest of space, I'll only show the Up method here:



```
 8          public override void Up()
 9          {
10              CreateTable(
11                  "dbo.Contractor",
12                  c => new
13                      {
14                          ID = c.Int(nullable: false, identity: true),
15                          Name = c.String(maxLength: 100),
16                          Desc = c.String(maxLength: 300),
17                          Created = c.DateTime(nullable: false),
18                          CreatedBy = c.String(maxLength: 100),
19                          ModifiedBy = c.String(maxLength: 100),
20                          Modified = c.DateTime(nullable: false),
21                          Comments = c.String(),
22                      })
23                  .PrimaryKey(t => t.ID);
24
25              CreateTable(
26                  "dbo.ContractorProjects",
27                  c => new
28                      {
29                          Contractor_ID = c.Int(nullable: false),
30                          Project_ID = c.Int(nullable: false),
31                      })
32                  .PrimaryKey(t => new { t.Contractor_ID, t.Project_ID })
33                  .ForeignKey("dbo.Contractor", t => t.Contractor_ID, cascadeDelete: true)
34                  .ForeignKey("dbo.Projects", t => t.Project_ID, cascadeDelete: true)
35                  .Index(t => t.Contractor_ID)
36                  .Index(t => t.Project_ID);
37
38              AddColumn("dbo.Equipment", "Comments", c => c.String());
39              AddColumn("dbo.Projects", "Comments", c => c.String());
40              AlterColumn("dbo.Equipment", "CreatedBy", c => c.String(maxLength: 100));
41              AlterColumn("dbo.Equipment", "ModifiedBy", c => c.String(maxLength: 100));
42              AlterColumn("dbo.Projects", "Number", c => c.String(maxLength: 100));
43              AlterColumn("dbo.Projects", "Description", c => c.String(maxLength: 255));
44              AlterColumn("dbo.Projects", "Location", c => c.String(maxLength: 100));
45              AlterColumn("dbo.Projects", "CreatedBy", c => c.String(maxLength: 100));
46              AlterColumn("dbo.Projects", "ModifiedBy", c => c.String(maxLength: 100));
47          }
```

FIGURE 33 LISTING 18

Key things to note:

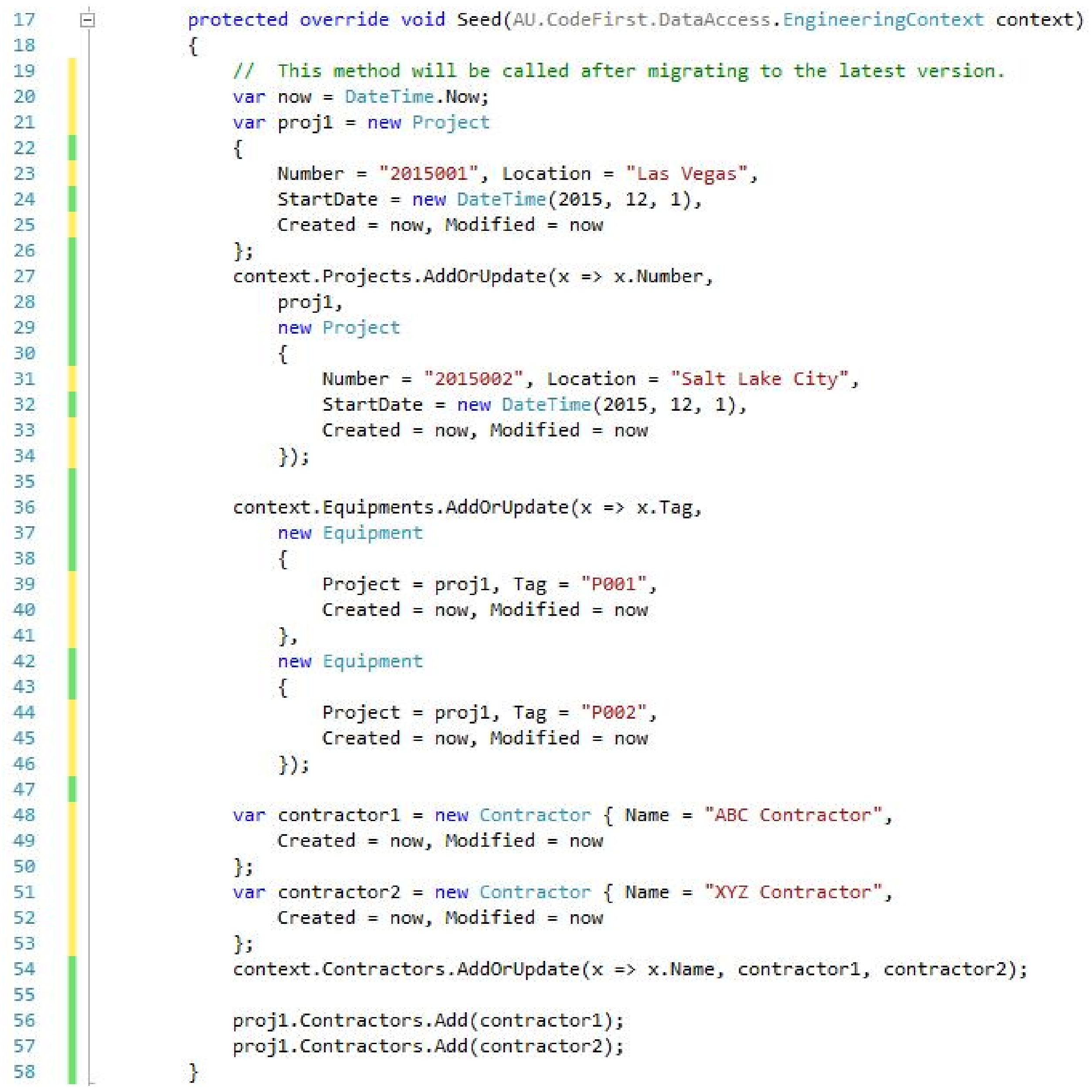
- Lines 10-23: A new **Contractor** table will be added. Note the "Desc" column, which will have a max length of 300 (not 255).

- Lines 15, 18, 19: Other string columns have a max length of 100.
- Line 21: The Comments field is max length.
- Lines 25-36: A new **ContractorProjects** table will be created. Note that we didn't create a POCO for this—it's purely a many-to-many mapping table. This is a common convention used in databases.
- Lines 38-39: New **Comments** columns will be added to the Equipment and Projects tables, thanks to inheritance from our base `DomainObject` class.
- Lines 40-46: Existing string columns in the Equipment and Projects tables modified to a max length of 100.

Go ahead and run `update-database` from the PMC.

Seeding the database
When developing a new database, it's often helpful to have "seed" data in place for testing with "real" data. The `DbMigrationsConfiguration` class, from which our `Configuration` class is derived, features a `Seed` method we can override just for this purpose. Open the `Configuration` class in the **DataAccess** project. Some commented out sample code is placed there for you as a guide. We'll seed a couple of projects, contractors, and equipment into the new, empty database.

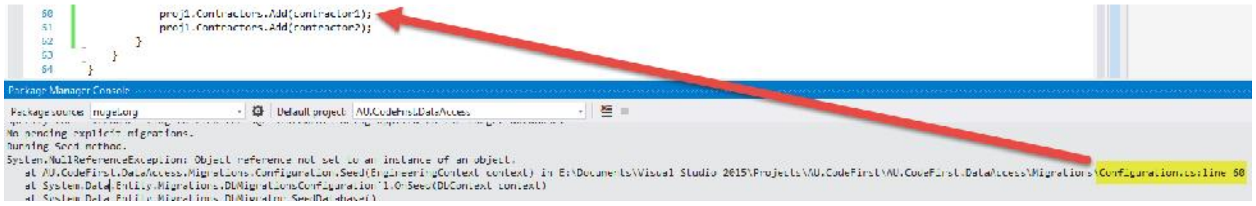
Add the following code to the override of the Seed method:



```
        protected override void Seed(AU.CodeFirst.DataAccess.EngineeringContext context)
        {
            //  This method will be called after migrating to the latest version.
            var now = DateTime.Now;
            var proj1 = new Project
            {
                Number = "2015001", Location = "Las Vegas",
                StartDate = new DateTime(2015, 12, 1),
                Created = now, Modified = now
            };
            context.Projects.AddOrUpdate(x => x.Number,
                proj1,
                new Project
                {
                    Number = "2015002", Location = "Salt Lake City",
                    StartDate = new DateTime(2015, 12, 1),
                    Created = now, Modified = now
                });

            context.Equipments.AddOrUpdate(x => x.Tag,
                new Equipment
                {
                    Project = proj1, Tag = "P001",
                    Created = now, Modified = now
                },
                new Equipment
                {
                    Project = proj1, Tag = "P002",
                    Created = now, Modified = now
                });

            var contractor1 = new Contractor { Name = "ABC Contractor",
                Created = now, Modified = now
            };
            var contractor2 = new Contractor { Name = "XYZ Contractor",
                Created = now, Modified = now
            };
            context.Contractors.AddOrUpdate(x => x.Name, contractor1, contractor2);

            proj1.Contractors.Add(contractor1);
            proj1.Contractors.Add(contractor2);
        }
```

FIGURE 34 LISTING 19

We didn't make a change to our model, so we only need to run `Update-Database` again, to try and get our seed data into the database. In PMC, go ahead and run `Update-Database` … and you'll get yet another error. This time it's a `NullReferenceException`, traced to the line of the Seed method where we tried to add contractors to a project.

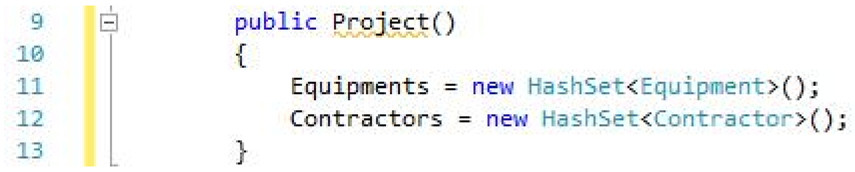


```
            proj1.Contractors.Add(contractor1);
            proj1.Contractors.Add(contractor2);
        }
    }
}
```

```
No pending explicit migrations.
Running Seed method.
System.NullReferenceException: Object reference not set to an instance of an object.
   at AU.CodeFirst.DataAccess.Migrations.Configuration.Seed(EngineeringContext context) in E:\Documents\Visual Studio 2015\Projects\AU.CodeFirst\AU.CodeFirst.DataAccess\Migrations\Configuration.cs:line 60
   at System.Data.Entity.Migrations.DbMigrationsConfiguration`1.OnSeed(DbContext context)
   at System.Data.Entity.Migrations.DbMigrator.SeedDatabase()
```

FIGURE 35 NULLREFERENCEEXCEPTION

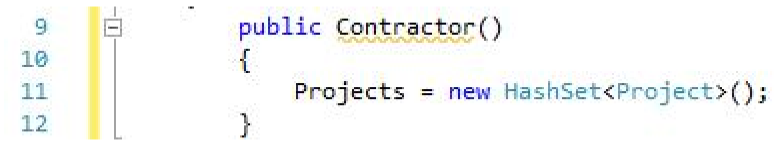
The problem is that the `Contractors` property in our model hasn't been initialized[21], so we need to safeguard against this by updating our Project class to initialize the collections. Add this constructor to the `Project` class.



```
        public Project()
        {
            Equipments = new HashSet<Equipment>();
            Contractors = new HashSet<Contractor>();
        }
```

FIGURE 36 LISTING 20

We should also add a constructor to the Contractor class:



```
        public Contractor()
        {
            Projects = new HashSet<Project>();
        }
```

FIGURE 37 LISTING 21

In PMC, run `Update-Database` again. We can use SQL Server Object Explorer to view the records that have been seeded into the database for us.

The most interesting aspect of this has to do with the many-to-many relationship between projects and contractors. In our seed code, we added `contractor1` and `contractor2` to `proj1.Contractors`. What this does in the database is add two records to the ContractorProjects table, linking each contractor to the project. The great thing about this is that the navigation from the Project table to the ContractorProjects table to the Contractor table happens automagically in the background, thanks to the navigation properties we setup during configuration.

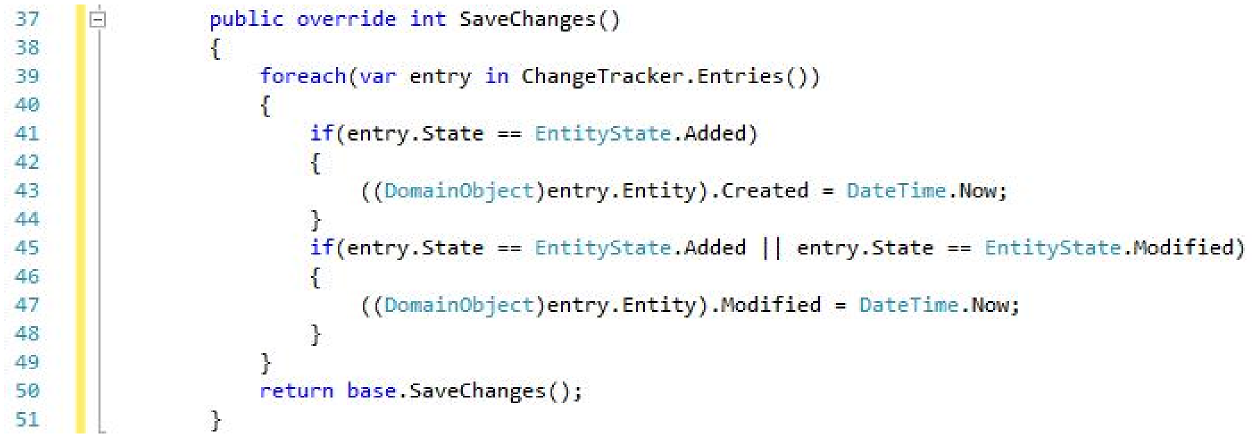
Work with Entity Framework objects such as DbContext and DbSet to create, retrieve, update, and delete database records

The `Created` and `Modified` date fields are intended to mark when a record is first created, and then the date and time of the last modification to the record. It would be nice if we could find a "centralized" spot for this. We'll now expand our understanding of `DbContext` a bit, by overriding `SaveChanges` to fill this information in any time we're about to save changes to the database.

[21] In practice you might not run into this problem for a while, particularly if you're working with an existing database already populated with data. Most of the time you'll query and modify existing data (so EF fills the DbSet). Still, it's good to know how to counter this problem, and the fix is pretty simple.

Because all the objects in our model conveniently derive from `DomainObject`, and thus inherit the `Created` and `Modified` properties, we can check entities before they're saved and fill in the necessary date values.

Open the `EngineeringContext` class in the **AU.CodeFirst.DataAccess** project, and override the `SaveChanges` method. Add the following code to this method:



```
        public override int SaveChanges()
        {
            foreach(var entry in ChangeTracker.Entries())
            {
                if(entry.State == EntityState.Added)
                {
                    ((DomainObject)entry.Entity).Created = DateTime.Now;
                }
                if(entry.State == EntityState.Added || entry.State == EntityState.Modified)
                {
                    ((DomainObject)entry.Entity).Modified = DateTime.Now;
                }
            }
            return base.SaveChanges();
        }
```

FIGURE 38 LISTING 22

`DbContext` utilizes something called `ChangeTracker` to keep track of the "state" of entities. These states include `Detached`[22], `Unchanged`, `Added`, `Modified`, and `Deleted`. In the code above, we'll check all entries. For those whose state is `Added`, we'll update the `Created` date. For those whose state is either `Added` or `Modified`, we'll update the `Modified` field. Note that we need to cast the `entry.Entity` object to a `DomainObject` to access the `Created` and `Modified` properties. And finally, we call the base `SaveChanges` methods.

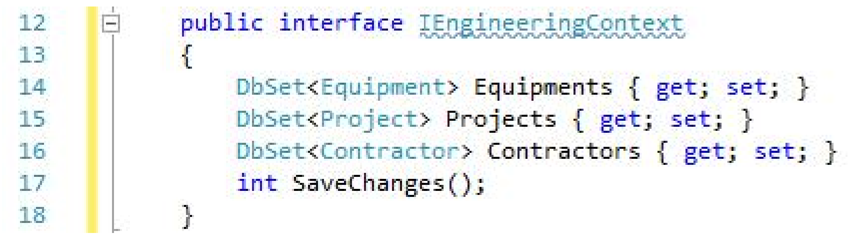
Refactoring towards Abstraction

One thing about our code in the ConsoleUI project bothers me. The client code that makes use of `EngineeringContext` (to this point our console app), isn't going to be of much use to us when we switch to our Autocad project. It's also not going to be very easy to write unit tests. Although we're not going to get into a big discussion of unit testing (maybe a class for next year?), I want to at least take you a few steps in the direction of making our code more testable and reusable.

The first step is to provide an abstraction of our `EngineeringContext`, which we can do by creating an interface. An interface is a contract that says any implementer of the interface promises to provide certain methods or properties. Other code can be written to this *abstraction* rather than the concrete implementation of a class. This is key to writing not just testable code, but code that is easy to maintain and extend.

In the **AU.CodeFirst.DataAccess** project, open the `EngineeringContext` class. Position your cursor somewhere in the `EngineeringContext` part of the public class line and press CTRL+. then choose **Extract Interface**. This creates the following interface code:

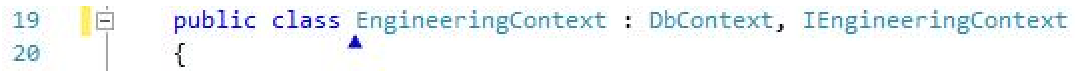
[22] Detached is a special state wherein objects can be retrieved into the context, but their state is not tracked and changes will not be persisted. You can think of this as a "read-only" state.



```
public interface IEngineeringContext
{
    DbSet<Equipment> Equipments { get; set; }
    DbSet<Project> Projects { get; set; }
    DbSet<Contractor> Contractors { get; set; }
    int SaveChanges();
}
```

FIGURE 39 LISTING 23

and modifies `EngineeringContext`:



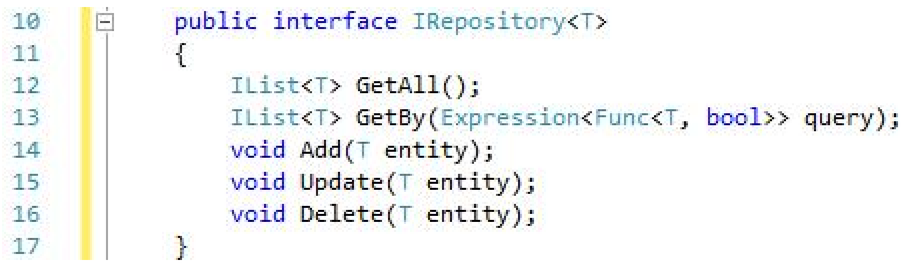
```
public class EngineeringContext : DbContext, IEngineeringContext
{
```

FIGURE 40 LISTING 24

The next step is to create a repository class[23]. This repository will act as the main layer of interaction for code in our UI layer (whatever that might be: a console app, Autocad plugin or web site). It will contain all the code that interacts directly with an `IEngineeringContext` implementer (i.e. our `EngineeringContext` class). We need our repository class to do a few key things:

- get all records for a particular type
- get only records meeting certain criteria
- add or update a record
- delete a record

This leads us to the following interface, which we can setup using *generics*.

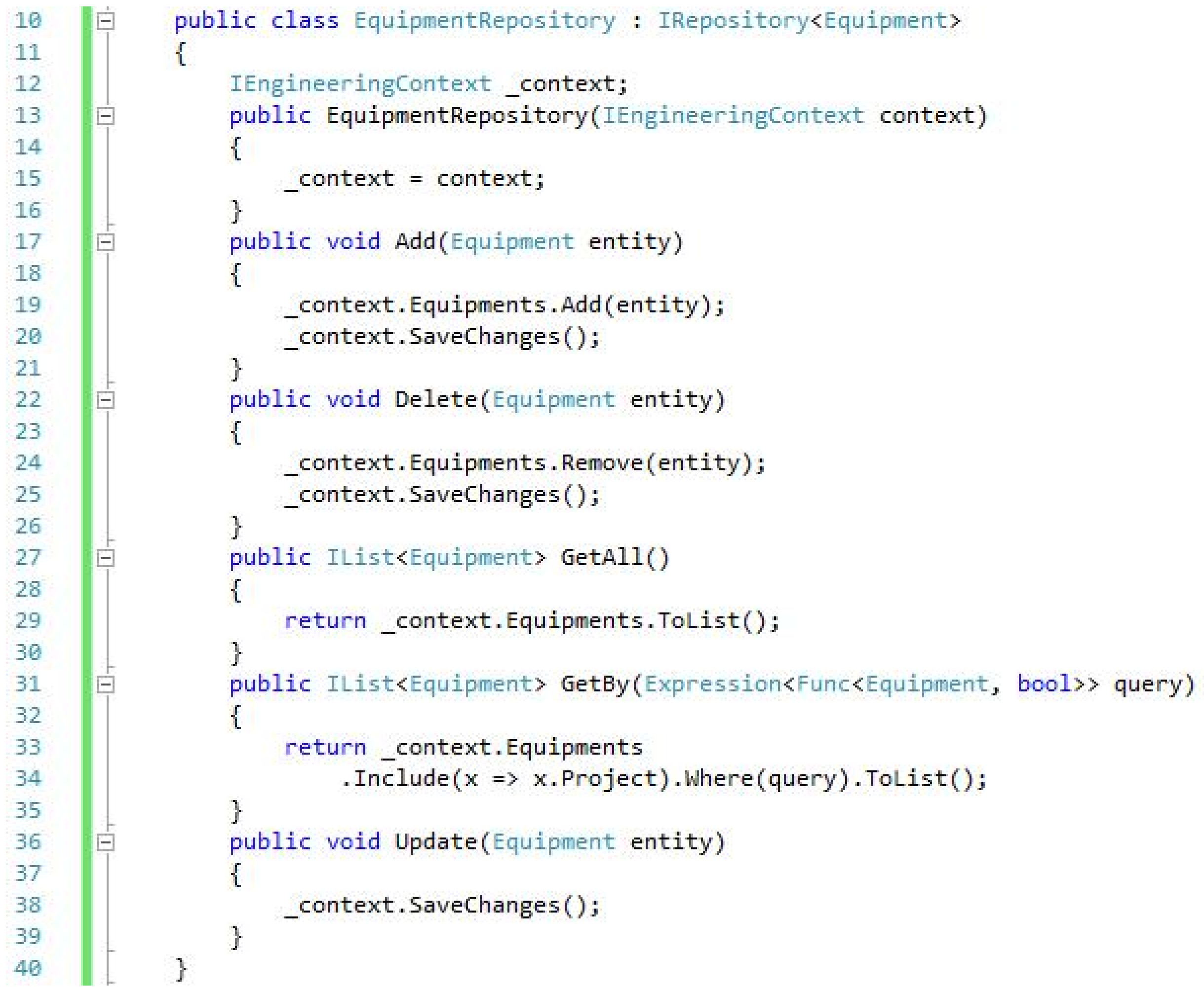


```
public interface IRepository<T>
{
    IList<T> GetAll();
    IList<T> GetBy(Expression<Func<T, bool>> query);
    void Add(T entity);
    void Update(T entity);
    void Delete(T entity);
}
```

FIGURE 41 LISTING 25

We'll now create an implementer of this interface for interacting with Equipment.

[23] See important note on the following page, this isn't a complete Repository pattern implementation.



```
public class EquipmentRepository : IRepository<Equipment>
{
    IEngineeringContext _context;
    public EquipmentRepository(IEngineeringContext context)
    {
        _context = context;
    }
    public void Add(Equipment entity)
    {
        _context.Equipments.Add(entity);
        _context.SaveChanges();
    }
    public void Delete(Equipment entity)
    {
        _context.Equipments.Remove(entity);
        _context.SaveChanges();
    }
    public IList<Equipment> GetAll()
    {
        return _context.Equipments.ToList();
    }
    public IList<Equipment> GetBy(Expression<Func<Equipment, bool>> query)
    {
        return _context.Equipments
            .Include(x => x.Project).Where(query).ToList();
    }
    public void Update(Equipment entity)
    {
        _context.SaveChanges();
    }
}
```

FIGURE 42 LISTING 26

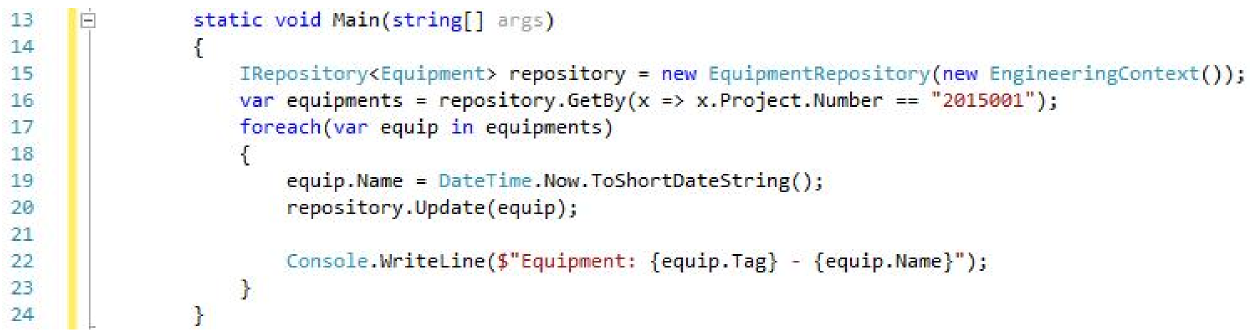
This implementation[24] demonstrates basic interaction with `DbContext` for all CRUD operations (Create, Read, Update and Delete). Of particular interest is the `GetBy` method which allows a caller to specify a criteria expression used to evaluate which items will be returned. You'll see how this is used momentarily.

> Please keep in mind that this is not a complete implementation. I'm trying to keep the example as short as possible and have left out a lot of other important considerations. Please see the list of references at the end for more in-depth resources than what I've provided here. This very simple implementation should suffice for what we want to eventually accomplish in our Autocad plugin.

Notice that this repository *depends* on a class that implements `IEngineeringContext` (an abstraction). And this dependency is *injected* into the repository class via its constructor. These are critical aspects of making your code testable and modifiable.

[24] We would also eventually need similar repositories for our Project and Contractor classes as well.

Let's write a quick test in our ConsoleUI project before proceeding to our final Autocad project. Change the `Project.Main` method as follows:



```
        static void Main(string[] args)
        {
            IRepository<Equipment> repository = new EquipmentRepository(new EngineeringContext());
            var equipments = repository.GetBy(x => x.Project.Number == "2015001");
            foreach(var equip in equipments)
            {
                equip.Name = DateTime.Now.ToShortDateString();
                repository.Update(equip);

                Console.WriteLine($"Equipment: {equip.Tag} - {equip.Name}");
            }
        }
```

FIGURE 43 LISTING 27

Run the console app to see how we're retrieving the equipment, then updating the Name and saving the changes to the database. The interaction with `EngineeringContext` is almost completely hidden away in the implementation of our `IRepository` interface: `EquipmentRepository`. This is perfect as we can now reuse this in a variety of application types, including our Autocad plug-in.

Create AutoCAD objects with data queried from a database via the Entity Framework model

We've created a database from scratch using Code First, and seen how to create, update, delete and read data. To wrap things up, we want to do something more practical as Autodesk developers. Let's write an Autocad plug-in that can draw something for us (a table), using information retrieved via EF.

Create an Autocad DLL

Add a new C# class library project to your solution called **AU.CodeFirst.Autocad**. Make this the Startup project.

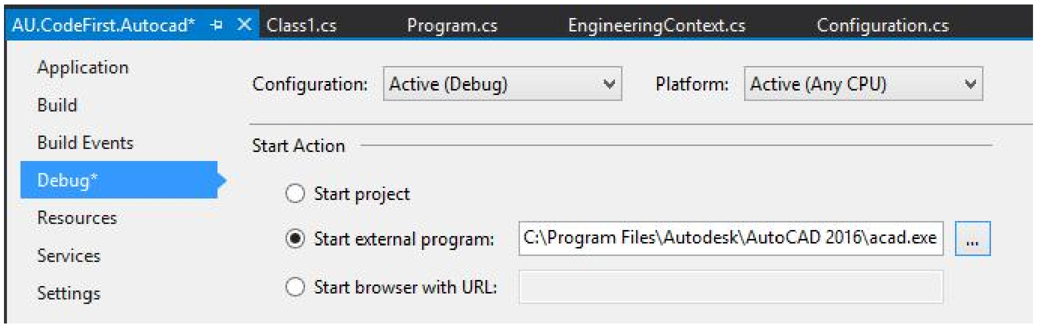
Add references to the following:

- EntityFramework (via NuGet)
- AU.CodeFirst.DataAccess
- AU.CodeFirst.Model
- System.Configuration
- AcMgd, AcDbMgd, AcCoreMgd (browse to the **ObjectArx 2016\inc**[25] folder. Be sure to set the **CopyLocal** property of these three references to **false**.

Finally, copy the `connectionStrings` section of one of the other App.Config files we've modified before into the new project's App.Config[26].

And finally, right-click the project, choose Properties > Debug and set the Start Action to launch Autocad, as shown below:

[25] By default the ObjectArx SDK installs to C:\ObjectArx 2016. Of course, if you are using an earlier version of the SDK, the version number may differ (i.e. ObjectArx 2015).
[26] There's a serious gotcha coming up with this…stay tuned.


AU.CodeFirst.Autocad* | Class1.cs | Program.cs | EngineeringContext.cs | Configuration.cs

Application
Build
Build Events
Debug*
Resources
Services
Settings

Configuration: Active (Debug) Platform: Active (Any CPU)

Start Action

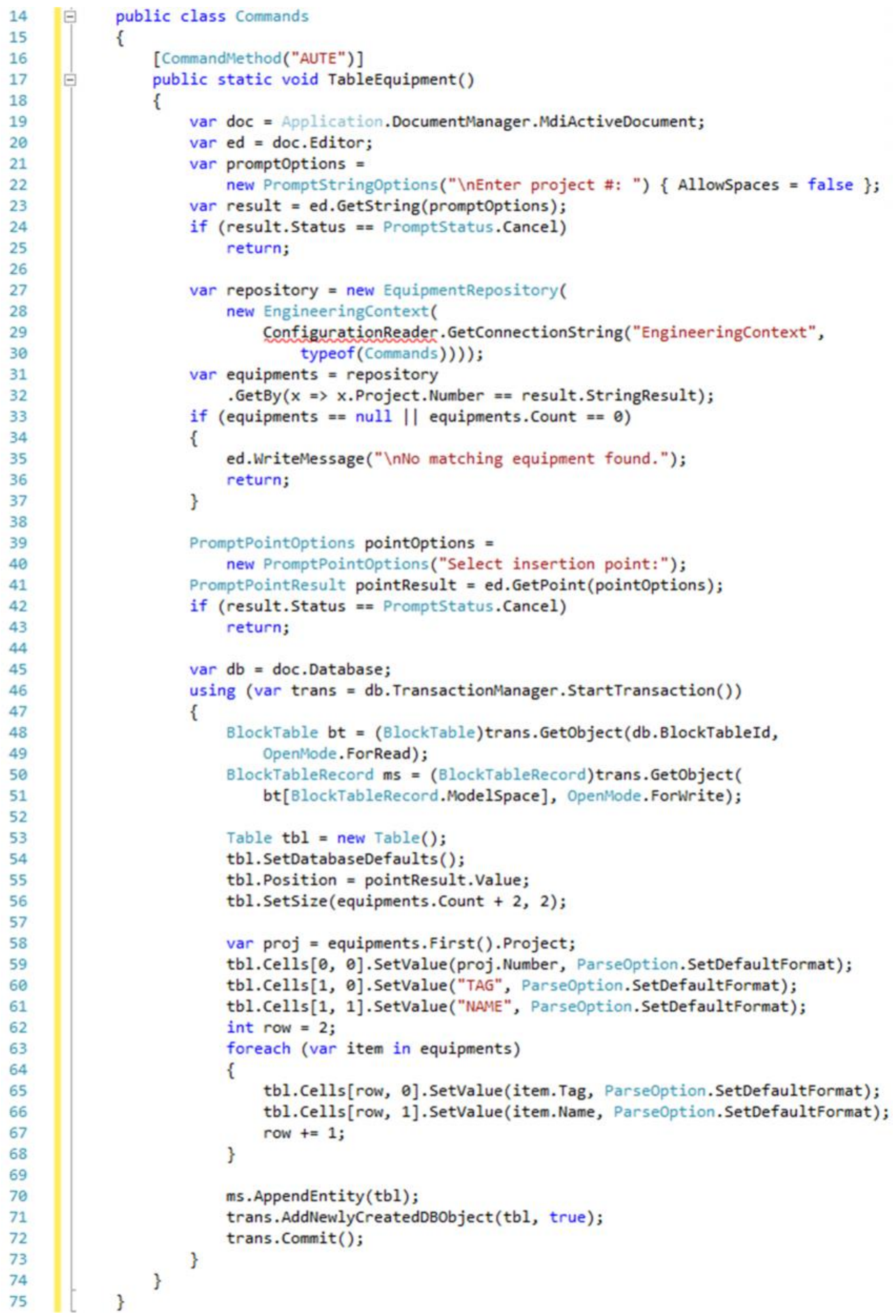
Start project [✓ Start external program: C:\Program Files\Autodesk\AutoCAD 2016\acad.exe] Start browser with URL:

FIGURE 44 DEBUGGING INTO AUTOCAD

The main thing we want to demonstrate here is how to leverage the EF model we've created to create objects in Autocad. We'll retrieve data from the database and use it to create a table of the Equipment data belonging to a user-specified project. While this example is going to be pretty simple, the sky is really the limit here.

Add a class file to your project named **Commands**, and add the code in the listing below. For sake of space, I've included the Line notes ahead of the listing this time. This is a much longer method than I'd ordinarily like, but the main gist of it is as follows.

- Lines 20-24: Prompt the user for a project #.
- Lines 26-32: Retrieve the equipment associated with that project.
- Lines 39-42: Prompt the user for an insertion point.
- Lines 45-73: Create a table with the project number in the table's "header" row, and each equipment's tag and name values in the "data" rows.



```
public class Commands
{
    [CommandMethod("AUTE")]
    public static void TableEquipment()
    {
        var doc = Application.DocumentManager.MdiActiveDocument;
        var ed = doc.Editor;
        var promptOptions =
            new PromptStringOptions("\nEnter project #: ") { AllowSpaces = false };
        var result = ed.GetString(promptOptions);
        if (result.Status == PromptStatus.Cancel)
            return;

        var repository = new EquipmentRepository(
            new EngineeringContext(
                ConfigurationReader.GetConnectionString("EngineeringContext",
                    typeof(Commands))));
        var equipments = repository
            .GetBy(x => x.Project.Number == result.StringResult);
        if (equipments == null || equipments.Count == 0)
        {
            ed.WriteMessage("\nNo matching equipment found.");
            return;
        }

        PromptPointOptions pointOptions =
            new PromptPointOptions("Select insertion point:");
        PromptPointResult pointResult = ed.GetPoint(pointOptions);
        if (result.Status == PromptStatus.Cancel)
            return;

        var db = doc.Database;
        using (var trans = db.TransactionManager.StartTransaction())
        {
            BlockTable bt = (BlockTable)trans.GetObject(db.BlockTableId,
                OpenMode.ForRead);
            BlockTableRecord ms = (BlockTableRecord)trans.GetObject(
                bt[BlockTableRecord.ModelSpace], OpenMode.ForWrite);

            Table tbl = new Table();
            tbl.SetDatabaseDefaults();
            tbl.Position = pointResult.Value;
            tbl.SetSize(equipments.Count + 2, 2);

            var proj = equipments.First().Project;
            tbl.Cells[0, 0].SetValue(proj.Number, ParseOption.SetDefaultFormat);
            tbl.Cells[1, 0].SetValue("TAG", ParseOption.SetDefaultFormat);
            tbl.Cells[1, 1].SetValue("NAME", ParseOption.SetDefaultFormat);
            int row = 2;
            foreach (var item in equipments)
            {
                tbl.Cells[row, 0].SetValue(item.Tag, ParseOption.SetDefaultFormat);
                tbl.Cells[row, 1].SetValue(item.Name, ParseOption.SetDefaultFormat);
                row += 1;
            }

            ms.AppendEntity(tbl);
            trans.AddNewlyCreatedDBObject(tbl, true);
            trans.Commit();
        }
    }
}
```

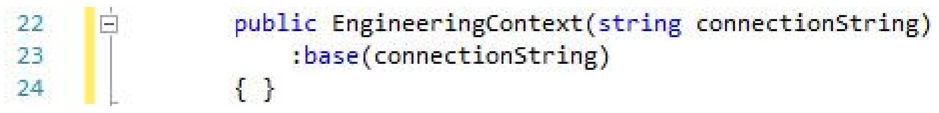
FIGURE 45 LISTING 28

The first time I ran this I expected everything to go smoothly, since this worked just fine in the console app. But when I ran it inside Autocad, I ran into an unexpected problem.

Because our DLL is running *inside* Autocad, it's not our dll's App.Config that's being used…it's Autocad's! Our plugin can't locate the database, since the `connectionString` we need is not in Autocad.exe.config, and all sorts of whackiness[27] ensues.

I really don't want to go mucking about in the Autocad.exe.config file, nor should you, so we need to add a way for our application to read our dll's config file, locate the correct `connectionString`, and supply that data to our `EngineeringContext`. Look back at listing 28, lines 29-30 and you'll see the call to the `ConfigurationReader` class we're about to add.

To make this work, we need to add another constructor to our `EngineeringContext` class first:



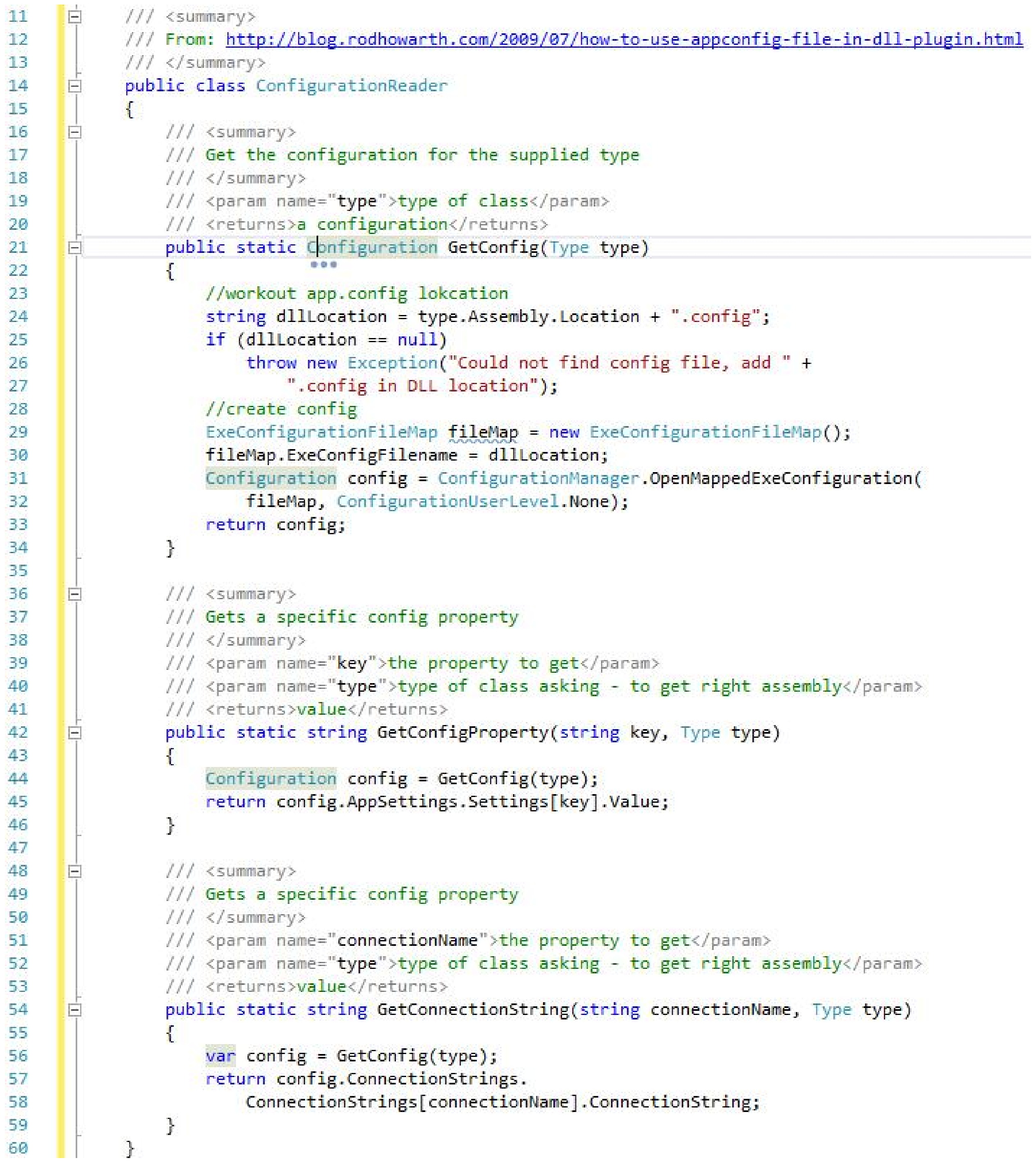
```
        public EngineeringContext(string connectionString)
            :base(connectionString)
        { }
```

FIGURE 46 LISTING 29

And then in our Autocad project, add a class called **ConfigurationReader**. I found this solution on the Internet (thank you Rod Howarth[28]!) and added the `GetConnectionString` method.

- Lines 21-34: Locate the dll's config file and read it into a `System.Configuration.Configuration` object.
- Lines 54-59: Locate the ConnectionStrings section, and get the named (i.e. "EngineeringContext") ConnectionString.

[27] I prefer "whackiness" to "Autocad crashed and burned a horrible death of the fatal execution variety."
[28] See Rod Howarth reference in the Resources section at the end of the document.



```csharp
/// <summary>
/// From: http://blog.rodhowarth.com/2009/07/how-to-use-appconfig-file-in-dll-plugin.html
/// </summary>
public class ConfigurationReader
{
    /// <summary>
    /// Get the configuration for the supplied type
    /// </summary>
    /// <param name="type">type of class</param>
    /// <returns>a configuration</returns>
    public static Configuration GetConfig(Type type)
    {
        //workout app.config lokcation
        string dllLocation = type.Assembly.Location + ".config";
        if (dllLocation == null)
            throw new Exception("Could not find config file, add " +
                ".config in DLL location");
        //create config
        ExeConfigurationFileMap fileMap = new ExeConfigurationFileMap();
        fileMap.ExeConfigFilename = dllLocation;
        Configuration config = ConfigurationManager.OpenMappedExeConfiguration(
            fileMap, ConfigurationUserLevel.None);
        return config;
    }

    /// <summary>
    /// Gets a specific config property
    /// </summary>
    /// <param name="key">the property to get</param>
    /// <param name="type">type of class asking - to get right assembly</param>
    /// <returns>value</returns>
    public static string GetConfigProperty(string key, Type type)
    {
        Configuration config = GetConfig(type);
        return config.AppSettings.Settings[key].Value;
    }

    /// <summary>
    /// Gets a specific config property
    /// </summary>
    /// <param name="connectionName">the property to get</param>
    /// <param name="type">type of class asking - to get right assembly</param>
    /// <returns>value</returns>
    public static string GetConnectionString(string connectionName, Type type)
    {
        var config = GetConfig(type);
        return config.ConnectionStrings.
            ConnectionStrings[connectionName].ConnectionString;
    }
}
```

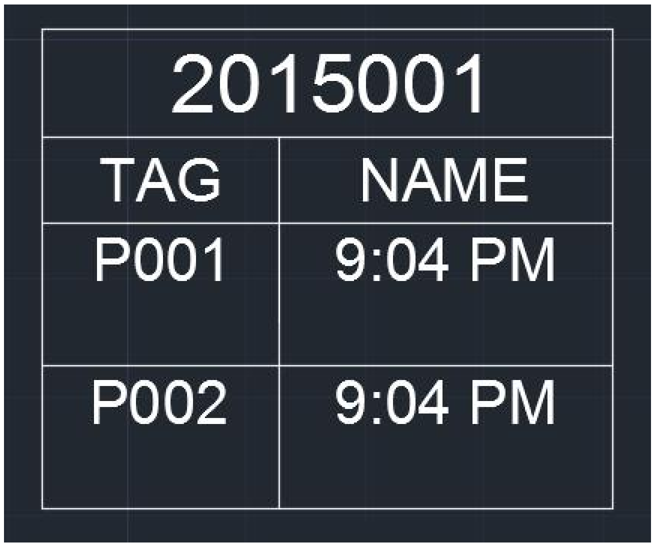
FIGURE 47 LISTING 30

At this point, we can run our solution in Debug mode, and Autocad should fire up. Start a new drawing, then use the NETLOAD command and browse to the solution folder, and under **AU.CodeFirst.Autocad\bin\Debug**, load the **AU.CodeFirst.Autocad.dll**. At the command prompt, type **AUTE** and press Enter.

At the prompt, enter **2015001** and press Enter.

At the prompt, pick an insertion point.

You should be rewarded with a beautiful baby table.



2015001	
TAG	NAME
P001	9:04 PM
P002	9:04 PM

FIGURE 48 A TABLE IS BORN

Summary

The goal of this class was to give you an introduction to using Entity Framework. We examined how to use the Database First approach to quickly get started with an existing database. We then created a brand new database using the Code First approach.

We looked at some of the ways we can customize the configuration of an EF model, including specifying a database name and server location, using data annotations and the Fluent API to configure schema options, as well as seeding our development database with sample data.

We took some first steps towards layering our application, and abstracting away from our DbContext derived class, which makes our solution more reusable, testable and modifiable.

And finally, we created an Autocad plugin that extracted data from the database via EF, which was used to draw a table object. Although I picked Autocad because that's where much of my experience lies, this should be easily extendable to other Autodesk applications that support .NET plugins.

This was a lot of ground to cover, and I hope it wasn't too much—it's a big topic and I wanted to cover enough to make EF useable. I also hope the problem-solving approach was valuable as it better represents a true "life in the trenches with EF" experience, at least as I've lived it. Please see the Resources section at the end of this document for more in-depth study.

Resources

- Google “Julie Lerman”, the end[29].
  - Several courses on Pluralsight
  - Book: *Programming Entity Framework: Code First* http://amzn.to/1SODx34
  - Book: *Programming Entity Framework: DbContext* http://amzn.to/1RUrYql
  - Book: *Programming Entity Framework: Building Data Centric Apps with ADO.NET Entity Framework* http://amzn.to/1QBiOk7
- “Data Points: Code First Goodies in Entity Framework 6.” MSDN Magazine, January 2014: https://msdn.microsoft.com/en-us/magazine/Dn519921.aspx
- Entity Framework Documentation: https://msdn.microsoft.com/en-us/data/ee712907
- Entity Framework Tutorial: http://www.entityframeworktutorial.net/code-first/entity-framework-code-first.aspx
- Autodesk “My First Plug-in Training”. http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=18162797
- Rod Howarth, How to use an app.config file in a DLL plugin (External Command): http://blog.rodhowarth.com/2009/07/how-to-use-appconfig-file-in-dll-plugin.html

[29] Just kidding, but she is a very prolific author and THE expert on Entity Framework.