Entity Framework’s Code First Workflow

Manually Coding to an Existing Database

The Entity Framework (EF) comes many with differentways of using it referred to as **Workflows**. Some use a **GUI’s** to display, model, and configure the EF context while others use only code. In almost all cases, EF automatically creates code for you, but its code often generates more objects and provide more database access then is needed or wanted! For example, direct access to tables should never be allowed on production-level databases, but the EF assumes this to be the case and automatically creates code allowing that access. As another example, EF 6.1 and above provides automatic stored procedure generation, but these stored procedures must be modified to be ready for production-level deployment.

In this document, we will look at how to create simple, effective classes to provide database processing. We will create this manually so that you will understand the process. You could also generate this code automatically and then modify or remove the unwanted code that EF provides, but you will, of course, need to understand what is needed and what can be safely removed. The following document and demonstration will help with that!

**Tip**: Check out this simple video on EF Workflows: <https://msdn.microsoft.com/en-us/data/jj590134>

# Code First

Microsoft introduced Code First as an alternative to using the GUI based data model. It has been mistakenly thought of as being only used to automatically create a database based of .NET types defined in your code. While it can indeed be used to do that, it can also be used to working with a current database, and it do not generate a lot of additional types and code seen when using the GUI based options.

“When using Code First development you usually **begin** by writing .NET Framework classes that **define your conceptual (domain) model**. In addition to defining the classes, you also need to let DbContext know which types you want to include in the model. To do this, you **define a context class that derives from DbContext and exposes DbSet properties** for the types that you want to be part of the model.” (<https://msdn.microsoft.com/en-us/data/jj679962>)

# Connections

In EF Code First, you can create a connection in **many different ways**. Many of these seem to work by magic, but actually, it is because there are using the EF Code First defaults!

**TIP**: You can find this example here: <https://msdn.microsoft.com/en-us/data/jj592674>

To understand how it works, consider this example from Microsoft’s website. Here we have a connection string configuration. This would be placed in the applications/assembly’s app.config or web.config file.

<configuration>   
  <**connectionStrings**>   
    <add name="BloggingCompactDatabase"   
          providerName="System.Data.SqlServerCe.4.0"   
          connectionString="Data Source=Blogging.sdf"/>   
  </**connectionStrings**>   
</configuration>

Note that the name for the connection string was “BloggingCompactDatabase.” To map this connection string to an EF Database Context object you would make a class with the same name and configure it like this:

public class BloggingContext : DbContext   
{   public BloggingContext(): base("name=BloggingCompactDatabase")  {  }  }

You can also leave off the name of the parameter so this also works:

public class BloggingContext : DbContext   
{   public BloggingContext(): base("BloggingCompactDatabase")  {  }  }

And so does this, if the connection string’s name matches the name of the class!

public class BloggingCompactDatabase: DbContext   
{   }

# The Automatic Mapping Feature

EF will try to define commands automatically by creating a **class** with the **same name as a table** using technology Microsoft calls the “Fluent API.” <https://msdn.microsoft.com/en-us/data/jj591617>

* EF looks for a table in the connected database that matches.
* EF infers the Primary Key by looking for columns named ID or <tablename>ID
* EF infers the PK column uses the identity (auto number) option if the data type is Int
* EF infers Foreign Key relationships and creates Navigation properties

Here is an example from MS website: <https://msdn.microsoft.com/en-us/data/jj679962>

public class SchoolEntities : DbContext   
{  public DbSet<Department> Departments { get; set; } }   
   
public class Department   
{   
    // Primary key   
    public int DepartmentID { get; set; }   
    public string Name { get; set; }   
   
    // Navigation property   
    public virtual ICollection<Course> Courses { get; set; }   
}

public class Course   
{   
    // Primary key   
    public int CourseID { get; set; }   
   
    public string Title { get; set; }   
    public int Credits { get; set; }   
   
    // Foreign key   
    public int DepartmentID { get; set; }   
   
    // Navigation properties   
    public virtual Department Department { get; set; }   
}

## Turing off the Automatic Table feature

Using tables directly has NEVER been considered a “**Best Practice**” so it is not surprising that you can opt-out of using this feature explicitly, by using the **Ignore<T>()** method.

“If you want to exclude a type from the model, use the NotMapped attribute or the DbModelBuilder.Ignore fluent API.” (<https://msdn.microsoft.com/en-us/data/jj679962>)

**modelBuilder**.**Ignore**<Department>();

# SQL Commonds Vs Stored Procedures

What is surprising is how hard it is to find examples where stored procedures and views are used along with EF. However, some examples do exist and here is one from Microsoft’s website:

“Entity Framework allows you to query using LINQ with your entity classes. However, there may be times that you **want to run queries using raw SQL directly** against the database. This includes calling stored procedures, which can be helpful for Code First models that currently do not support mapping to stored procedures. The techniques shown in this topic **apply equally to models created with Code First and the EF Designer**.

**…**

**The SqlQuery method on DbSet allows a raw SQL query** to be written that will return entity instances. The returned objects will be tracked by the context just as they would be if they were returned by a LINQ query. For example:

using (var context = new BloggingContext())   
{   
    var blogs = context.Blogs.**SqlQuery**("SELECT \* FROM dbo.Blogs").ToList();   
}

Note that, just as for LINQ queries, the query is not executed until the results are enumerated—in the example above this is done with the call to ToList.

**Care should be taken whenever raw SQL queries are written for two reasons**. **First**, the query should be written to ensure that it only returns entities that are really of the requested type. For example, when using features such as inheritance it is **easy to write a query that will create entities that are of the wrong CLR type**.

**Second**, some types of raw SQL query expose **potential security risks, especially around SQL injection attacks**. Make sure that you use parameters in your query in the correct way to guard against such attacks.” ( <https://msdn.microsoft.com/en-us/data/jj592907>)

# EF vs. ADO.NET

Remember that the **EF is an abstraction layer over the top of ADO.NET** and using it comes with a cost. On a WinForms application, you have one user for the application and performance is not much of an issue, but with a WebSite, it can be. External, or any site with a lot of users, need to be programmed to have the **best performance possible. ADO.NET** still provides you with that at the cost of having to do a lot more manually coding. Take a look at the following Microsoft web page on this topic: <https://msdn.microsoft.com/en-us/data/hh949853>

# Using the ExecuteStoreQuery() Method

“Calling the [ExecuteStoreQuery](https://msdn.microsoft.com/en-us/library/system.data.objects.objectcontext.executestorequery(v=vs.100).aspx) method is equivalent to calling the [ExecuteReader](https://msdn.microsoft.com/en-us/library/system.data.common.dbcommand.executereader(v=vs.100).aspx) method of the [DbCommand](https://msdn.microsoft.com/en-us/library/system.data.common.dbcommand(v=vs.100).aspx) class, only [ExecuteStoreQuery](https://msdn.microsoft.com/en-us/library/system.data.objects.objectcontext.executestorequery(v=vs.100).aspx) returns entities and the [ExecuteReader](https://msdn.microsoft.com/en-us/library/system.data.common.dbcommand.executereader(v=vs.100).aspx) returns property values in the [DbDataReader](https://msdn.microsoft.com/en-us/library/system.data.common.dbdatareader(v=vs.100).aspx).”( <https://msdn.microsoft.com/en-us/library/dd487208(v=vs.100).aspx)>

public IEnumerable<Customer> QueryCustomersView()

{

string strSQLCode = @"Select CustomerID, CustomerName, CustomerTypeID

From **vCustomers**;";

ObjectResult<Customer> objResults;

try

{

objResults = Context.ObjectContext.**ExecuteStoreQuery**<Customer>(strSQLCode);

}

catch (Exception)

{

throw;

}

return objResults.ToList<Customer>();

}

# Using the ExecuteStoreCommand() Method

Using stored procedures provides the best performance and guards against SQL Injection Attacks (one of the top twenty security risks for more than two decades!).

“Using **parameterized commands helps guard against SQL injection attacks**, in which an attacker "injects" a command into a SQL statement that compromises security on the server. Parameterized commands guard against a SQL injection attack by guaranteeing that values received from an external source are passed as values only, and not part of the SQL statement. “(<https://msdn.microsoft.com/en-us/library/system.data.objects.objectcontext.executestorecommand(v=vs.100).aspx>)

ExecuteStoreCommand() is used to execute command that do not return a result set. It is similar to calling the ExecuteNonQuery() method in ADO.NET, and as such is used to call transactional stored procedures.

public int DelCustomer(int CustomerID){

var objParms = ParameterFactories.CustomerParameterFactory(CustomerID: CustomerID);

string strSQLCode = @"Exec @RC = pDelCustomer @CustomerID = "

+ CustomerID.ToString() + ";";

try{

this.Context.ObjectContext.**ExecuteStoreCommand**(strSQLCode, objParms["RC"],

objParms["CustomerID"]);

if ((int)objParms["RC"].Value < 0){

throw new Exception("Error reported in Stored Procedure: "+

objParms["RC"].Value.ToString()); }

}//end try

catch(Exception){throw;}

return (int)objParms["RC"].Value;

}//end method

# EF Code First Demo

Next, we will look at a demonstration of how you manually program both the database and EF to use stored procedures and views.

## Setting up the Database

In this example I am going to create a simple database with tables, stored procedures, and views.

### Creating the Database

The following code checks to see if the database is already in existence and if so deletes it before trying to create a new version of it. This is commonly done in demonstrations, since it allows the presenter or student to “reset the database” at will. The Use statement is important! Since the SQL Database Engine can manage multiple files at once, it needs to know which one you want to work with.

Use Master;

Go

**If Exists** (Select Name from SysDatabases Where Name = 'EFCodeFirstDemo')

**Drop** Database EFCodeFirstDemo;

Go

**Create Database EFCodeFirstDemo;**

Go

Use EFCodeFirstDemo;

Go

### Creating a Table

The following code creates a new Customers table using SQL Server’s Identity option. This option automatically generates a new CustomerID each time a row is added to the table. We will see later how you can capture this newly generated ID for use in your application.

---------------- Customers ---------------------------

Create **Table** **Customers**

(CustomerID int Primary Key **Identity**

,CustomerName nVarchar(100)

,CustomerTypeID int

);

Go

### Setting Permission

Users, and application developers, should never be provided with direct access to tables of a database. This has been considered a “best practice” for more than two decades! To stop this from happening we will deny access directly to the table for either querying or transactional processing using the following command.

**Deny** Select, Insert, Update, Delete **On** Customers **To** Public;

Go

### Creating Transactional Stored Procedures

With access to the table blocked we need to create an abstraction layer that will provide public access. We do so by creating a set of stored procedures and views. We will start with creating an Insert stored procedure. In this example Note the following:

* The @NewRowID parameter works like a C# output parameter and provides access to the result of SQL Sever’sSCOPE\_IDENTITY() function
* The SCOPE\_IDENTITY() returns the newly added integer generated by the table’s Identity option
* The @RC variable is used to indicate the status of the stored procedure statements, conventionally returning positive integers for successful completion and negative integers for unsuccessful completion
* Execute permission is grated publicly, which allows an insert to take place on the Customers table, but only though the stored procedure

Create **Procedure** **pInsCustomer**

(@CustomerName nVarchar(100), @CustomerTypeID int, **@NewRowID int Out**)

AS

Begin -- Body

Declare @RC int = 0;

Begin Try

Insert into Customers (CustomerName, CustomerTypeID)

Values (@CustomerName, @CustomerTypeID);

Select **@NewRowID = SCOPE\_IDENTITY()**;

Set @RC = 100;

End Try

Begin Catch

**IF** (**ERROR\_NUMBER() = 547**) Set @RC = **-200** -- Violation of Foreign Key

**ELSE** Set @RC = **-100**; -- Unspecified Error

End Catch

Return @RC;

End -- Body

Go

Grant Exec on pInsCustomer to Public;

Go

**NOTE**: In production level stored procedures additional code would be added to include more validation and error handling, however this example provides a sample of the main features of stored procedure creation.

### Creating Query-able Stored Procedures and Views

In additional to transactional statements (insert, update, and delete), we will need to have way to query the table. This can be done using views or stored procedures. Stored procedures have parameters and protect from SQL injection attacks, but views provide a way to perform Ad-Hoc queries when necessary, while still maintaining an abstraction layer. Here is an example of both.

Create **View vCustomers**

AS

**Select CustomerID, CustomerName, CustomerTypeID**

**From Customers;**

Go

**Grant Select** on vCustomers to Public;

Go

Create **Procedure pSelCustomer**

(@CustomerID int = null, @CustomerName nVarchar(100) = null, @CustomerTypeID int = null)

AS

Begin -- Body

Declare @RC int = 0;

Begin Try

If (@CustomerID is Null AND @CustomerName is Null AND @CustomerTypeID is Null)

***Select CustomerID, CustomerName, CustomerTypeID***

***From vCustomers***

Else

Select CustomerID, CustomerName, CustomerTypeID

From vCustomers

Where CustomerID = @CustomerID

Or CustomerName = @CustomerName

Or CustomerTypeID = @CustomerTypeID

Set @RC = 100;

End Try

Begin Catch

Set @RC = -100;

End Catch

Return @RC;

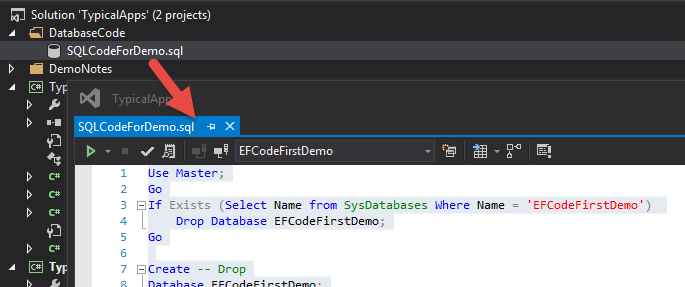
End -- Body

Go

**Grant Exec** on pSelCustomer to Public;

Go

**Note**: You will find the complete script for creating the database and its object in the file **SQLCodeForDemo.sql** included in this demonstration.



## Creating the Context Model Types

With the database object in place, we need to create .NET objects that represent its objects. We will use classes to represent a unit of work associated with each database entity (table).

### Create Types for Tables

In this demonstration we have only two tables; Customers and CustomerTypes and we will create a class for each of these. Each of the tables have both stored procedures and views, and we will create one or more methods for each stored procedure and view. We will also create interfaces as an abstraction layer within .NET. Like our abstraction layer in the database (stored procedures and views), this protects our application from database changes to its structure over time and is consider a “best practice” when creating applications.

We will use “Repository Pattern” to organize our code. We start by creating an interface that represent a unit of work.

#region Interfaces

public interface ICustomerRepository

{

//Table Data

IEnumerable<Customer> **Customers** { get; set; }

int **CustomerID** { get; set; }

string **CustomerName** { get; set; }

int **CustomerTypeID** { get; set; }

//Transaction Processing

int **InsCustomer**(string CustomerName, int CustomerTypeID, out int NewRowID);

int **UpdCustomer**(int CustomerID, string CustomerName, int CustomerTypeID);

int **DelCustomer**(int CustomerID);

//Query Processing

IEnumerable<Customer> **SelCustomer**(int CustomerID = 0

, string CustomerName = ""

, int CustomerTypeID = 0);

IEnumerable<Customer> **QueryCustomersView**();

}

#endregion

**Tip:** Check out this link on the Repository Pattern: <http://www.codeproject.com/Articles/770156/Understanding-Repository-and-Unit-of-Work-Pattern>

Next we create a concrete class to implement the interface. Here is an example of part of that code.

#region Concrete Classes

public class **Customer** : **ICustomerRepository**

{

IObjectContextAdapter Context;

public Customer() { **Context = new EFDbContext();** }

#region Support for table data in Customers[…]

#region Support for table Transaction processing

public int InsCustomer(string CustomerName, int CustomerTypeID, out int NewRowID)

{

int RC; //Used to trap the Stored Procedure's return code

IParameterFactory objParms = new CustomersParameterFactory(CustomerName: CustomerName,

CustomerTypeID: CustomerTypeID);

string strSQLCode = @"Exec @RC = pInsCustomer " +

" @CustomerName = '" + CustomerName + "'" + // Don't forget the SINGLE Quotes!!!

", @CustomerTypeID = " + CustomerTypeID.ToString() +

", @NewRowID = @NewRowID Out;";

try

{

this.Context.ObjectContext.ExecuteStoreCommand(strSQLCode

, objParms.Parmeters["RC"]

, objParms.Parmeters["CustomerName"]

, objParms.Parmeters["CustomerTypeID"]

, objParms.Parmeters["NewRowID"]

);

//Get the new row ID created by the table's Identity feature

if (objParms.Parmeters["NewRowID"].Value is DBNull)

{ NewRowID = 0; } //if the insert has failed, then set this to an arbitrary number

else { NewRowID = (int)objParms.Parmeters["NewRowID"].Value; } //else send it back as output

//Trap or return the Stored Procedure's return code

RC = (int)objParms.Parmeters["RC"].Value;

if ( RC < 0)

{throw new Exception("Error reported in Stored Procedure: " +

objParms.Parmeters["RC"].Value.ToString()); }

}

catch (Exception)

{

throw;

}

return (int)objParms.Parmeters["RC"].Value;

}

## Supporting Classes

In addition to classes that support our tables, we will create two other classes. The first is for configuring the EF database context object which allows us to connect to the database and refine what EF will automatically do for you when its functionality is invoked. The second one will use a “Factory Pattern” to create the parameter we will need for our queries and stored procedure calls.

### Creating The EF Context

For this will need to do two things; create the connection string in the configuration file and create a class that inherits from DBContext.

#### Adding a Connection

<connectionStrings>

<add name="EFDbContext" connectionString="Data Source=(localdb)\v11.0;Initial Catalog=EFCodeFirstDemo;Integrated Security=True" providerName="System.Data.SqlClient"/>

</connectionStrings>

#### Creating the Context Class

//For EF and ADO.NET

using System.Data.Entity;

using System.Data.Entity.Infrastructure;

using System.Data;

using System.Data.Entity.Core.Objects;

namespace TypicalClassLibrary

{

public class EFDbContext : DbContext

{

//Set up the connection using the Constructor

public EFDbContext(): base("name=EFDbContext")

{

//Stop EF from automatically mapping to a missing "Customer" table

DbModelBuilder objDBMB = new DbModelBuilder();

objDBMB.Ignore<Customer>();

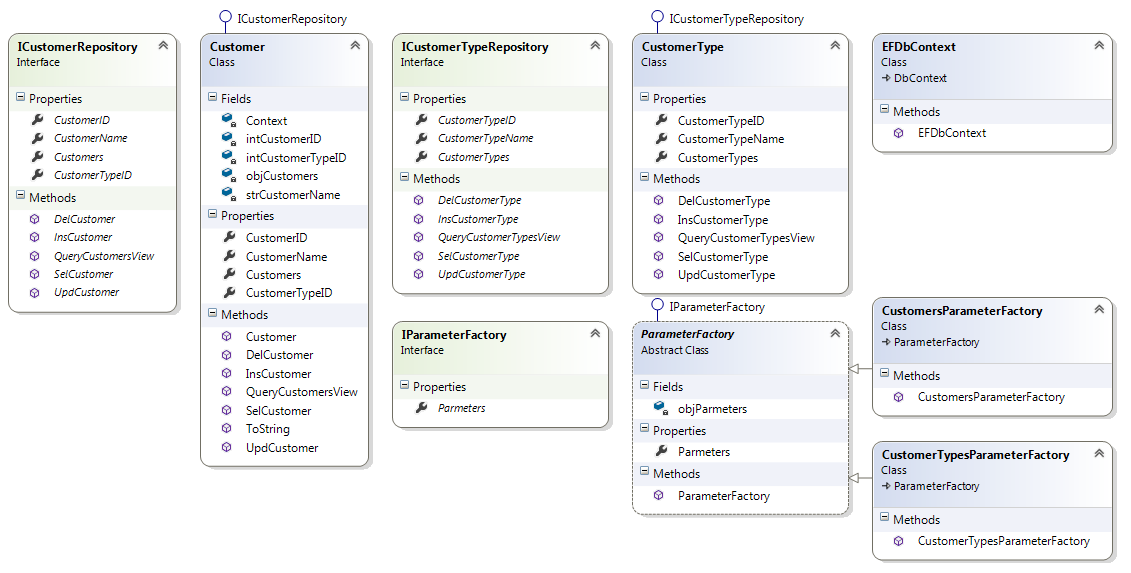
//objDBMB.Ignore<CustomerType>();//I would add a line for each table in the DB

}

}//end class

}//end namespace

At this point we have the following .NET structures in place:



### Creating a Parameter Factory

Each table with its associated stored procedures and views will need a set of ADO.NET Parameters. Instead of redundantly creating the same parameters in each method, let’s create a static class that will allow us to configure the parameters at once.

# Testing the Code

We now need to test the code before we try to develop a real application that uses it. To do this we will create a simple Console application. After we create the project, we need to add a reference to the Class Library and configure the connection string in the Console’s app.config file.

<connectionStrings>

<add name="EFDbContext"

connectionString="Data Source=(localdb)\v11.0;Initial Catalog=EFCodeFirstDemo;Integrated Security=True"

providerName="System.Data.SqlClient" />

</connectionStrings>

Now we add code to test the methods of our classes.

static void Main(string[] args)

{

try

{

**IObjectContextAdapter** Context = new EFDbContext();

**ICustomerRepository** objCustomer = new Customer();

int intNewRowID;

int intRC = 0;

Console.WriteLine("--Test insert sproc: ");

intRC = objCustomer.InsCustomer("New Data", 2, out intNewRowID);

Console.WriteLine("RC = {0} and New Row ID = {1}", intRC.ToString(), intNewRowID.ToString());

Console.WriteLine("\n" + "--Test select view");

foreach (var row in objCustomer.QueryCustomersView())

{ Console.WriteLine(row.ToString()); }

Console.WriteLine("\n" + "--Test delete sproc: ");

intRC = objCustomer.DelCustomer(intNewRowID);

Console.WriteLine("RC = {0}", intRC.ToString());

Console.WriteLine("\n" + "--Test select sproc with all rows: ");

foreach (var row in objCustomer.SelCustomer(0))

{ Console.WriteLine(row.ToString()); }

Console.WriteLine("\n" + "--Test update sproc: ");

intRC = objCustomer.UpdCustomer(2, "Customer 2", 2);

Console.WriteLine("RC = {0}", intRC.ToString());

Console.WriteLine("\n" + "--Test select sproc with 1 row: ");

foreach (var row in objCustomer.SelCustomer(2))

{ Console.WriteLine(row.ToString()); }

Console.Write("\n" + "--Test the Return Code Error handling: " + "\n");

intRC = objCustomer.InsCustomer("New Data", 123, out intNewRowID);

}

catch (Exception ex) {

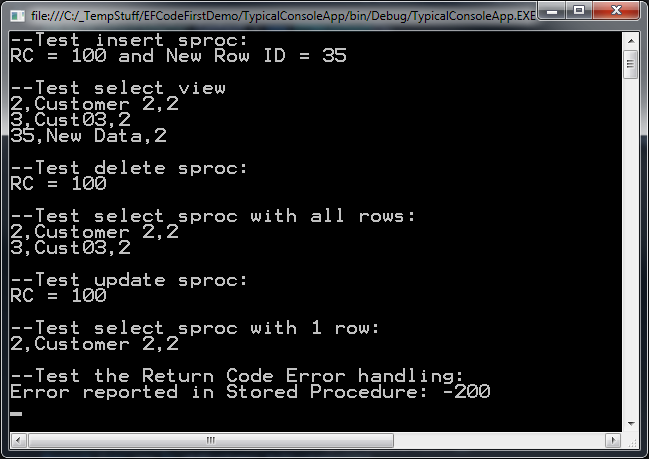
Console.WriteLine(ex.Message.ToString());

}

Console.Read();

}//end Main

Running this code should produce this result.



# **Demo**: Using EF’s Code First Workflow

Your instructor will now demo how to create a database and a class library to process and query its data.

# Lab 1

In this lab, you will create a new Context Model, Parameter Factory, and Concrete Classes for the Customer Type table and its supporting stored procedures and views.

1. ***Open*** *the* ***Code File*** *called CustomerTypes.cs in the TypicalClassLibrary project of the EFCodeFirstDemo Solutions.*
2. **Review the interface** representing a single row of data, a set of data rows, transaction processing (Insert, Update, and Delete), and query processing.
3. **Create** a new **Parameter Factory** called CustomerTypeParameterFactory and add code to work with the CustomerType database objects’ parameters.
4. **Create** a **class** to process a single of data called CustomerType and added code to implement ICustomerType the interface.
5. **Create** a **class** to implement the ICustomerTypes interface.
6. **Modify** the **EFContext**.cs file to block the automatic code generation for the CustomerTypes table.

objDBMB.Ignore<CustomerType>();

1. **Test** the new classes and methods in the Console application.