|  |
| --- |
|  |
| ADO.NET Entity Framework II |
| Working with POCOs |
|  |
|  |
|  |

|  |
| --- |
|  |

# 

POCOs with the Entity Framework

# Overview

In this lab, we will be exploring the POCO capabilities of ADO.NET Entity Framework. We will create a new entity data model and use the model to create a database and plain old CLR objects.

# Part I – Creating the Model

1. In Visual Studio use File -> New Project to create a new Windows Console application in the before directory of this lab. Name the application **ClassManager**.
2. When asked to create a unit test project, select No (we’ll focus on EF for this lab).
3. Right-click the project and select Add -> New Item.
4. Under the Data templates, select ADO.NET Entity Data Model. Name the model **Classes**.
5. In the “Choose Model Contents” dialog, select Empty Model.
6. Right-click on the Entity Model design surface and select Add -> Entity.
7. Name the new entity **Class** and select OK.

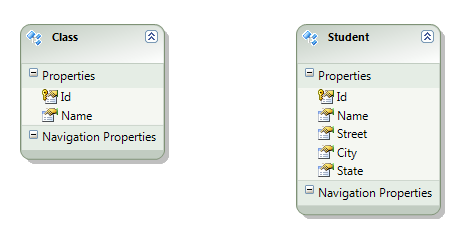
*The default is to give the new entity an Id property of type Int32 – leave this setting in place.*

1. Right-click the Class entity and select Add -> Scalar Property.
2. Give the new property the name **Name**.

*The default data type for a new entity is String – leave this as the default.*

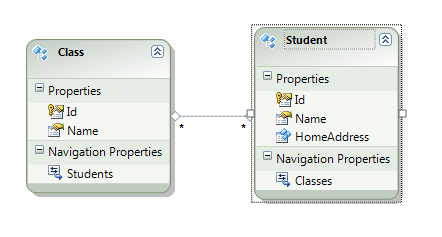
1. Right-click the design surface and create a Student entity. The Student should have the following properties (use the Properties window):

* Id (Int32 – Key field generated by default).
* Name (String, Max Length 25)
* Street (String, Max Length 25)
* City (String, Max Length 25)
* State (String, Max Length 2)



Instead of storing City, State, and Street in each Student object, let’s create a complex Address property to hold this information.

1. In the designer, hold down the shift key to select the Street, City, and State properties. Right-click the select properties and select **Refactor Into New Complex Type**. Give the complex type the name **Address**.
2. Change the name of the new property on the Student class to **HomeAddress**.
3. Right-click the design surface and select Add -> Association.
4. Make the association a **Many to Many relationship** between Students and Classes and click OK.



1. Right-click the white space on the design surface and select Properties.
2. Change the Entity Container Name to ClassDB.

# Part II – Creating the Database

1. Right-click the design surface again and select “Generate Database from Model”.
2. On “Choose your data connection” click “New Connection”.
3. Enter your server name, and enter **Classes** as the database name. When asked to create the database, answer “yes”.
4. Click next, then finish.

*You should see a .sql file appear in Solution Explorer.*

1. Double-click the Classes.edmx.sql file (it should open in a new query window in Visual Studio).

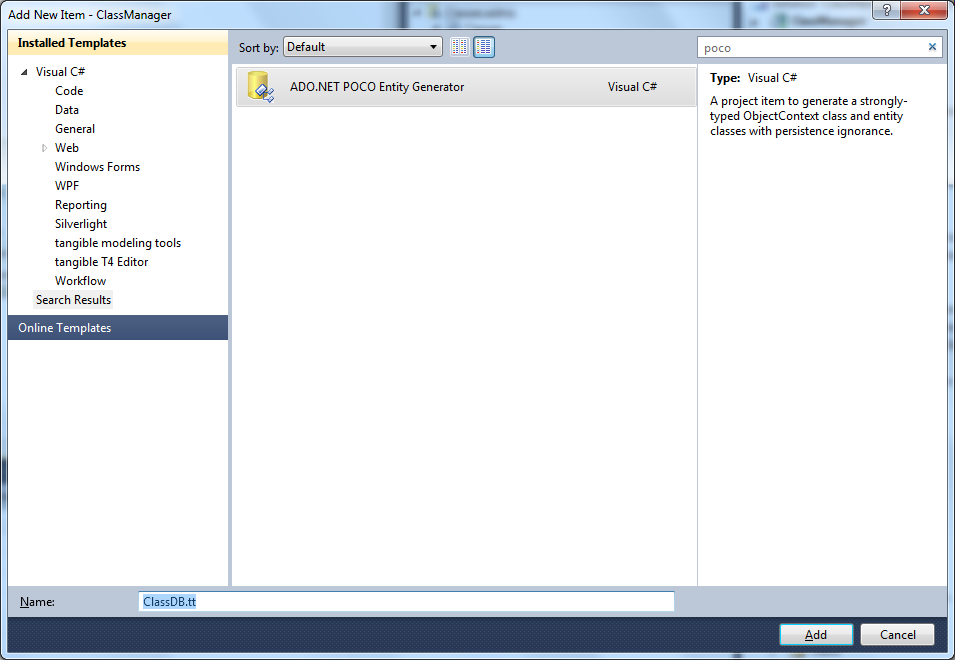
*Take a moment to browse through the DDL inside. Notice the DDL will create three tables – the extra table is a “join” table between classes and students.*

1. Right-click the editing window and select Execute SQL (Ctrl+Shift+E).
2. When prompted to connect, establish a connection to your local database.

*If the command does not execute successfully, ask your instructor for help*

# Part III – Generating Code

1. Return to the Classes.edmx file.
2. Right-click the design surface and select Add -> Code Generation Item.
3. Search for “POCO”. If no results are found, click the “Online Templates” section. You should find (or be able to download) the ADO.NET POCO Entity Generator. Name the file **“Classes.tt”** before clicking Add.



1. Examine the Classes.Context.cs file (underneath the Classes.Context.tt file). Notice the constructor of ClassDB enables lazy loading.
2. Examine the Student.cs file underneath the Classes.tt file. Notice all properties are marked as virtual.

# Part IV – Using The Model

1. In Program.cs, instantiate a new instance of ClassDB.
2. Pass the new ClassDB context object to a new method named AddStudents, and a new method named AddClasses.

static void Main(string[] args)

{

var ctx = new ClassDB();

AddStudents(ctx);

AddClasses(ctx);

}

private static void AddClasses(ClassDB ctx)

{

}

private static void AddStudents(ClassDB ctx)

{

}

1. Inside AddStudents, create three student objects and add them to the object context.

ctx.Students.AddObject(

new Student

{

Name = "Poonam",

HomeAddress =

{

Street = "Potomac",

City = "Columbia",

State = "MD"

}

});

ctx.Students.AddObject(

new Student

{

Name = "Joy",

HomeAddress =

{

Street = "Oak",

City = "Hagerstown",

State = "MD"

}

});

ctx.Students.AddObject(

new Student

{

Name = "Alex",

HomeAddress =

{

Street = "Main",

City = "Hagerstown",

State = "MD"

}

});

1. Inside AddClasses, create three classes and add them to the object context.

ctx.Classes.AddObject(new Class {Name = "Physics"});

ctx.Classes.AddObject(new Class { Name = "Art" });

ctx.Classes.AddObject(new Class { Name = "Music" });

1. Back in the Main method, use SaveChanges on the context to persist the new objects.

var ctx = new ClassDB();

AddStudents(ctx);

AddClasses(ctx);

**ctx.SaveChanges();**

1. Add another method to the Program class that takes a ClassDB object as a parameter and has the name AssociateStudentsWithClasses.
2. Inside the method, use LINQ queries and the objects you retrieve to add **all** students to the Physics class. Add two students to the Music class, and 1 student to the Art class. Persist all changes.

private static void AssociateStudentsWithClasses(ClassDB ctx)

{

var students = ctx.Students.ToList();

var physics = ctx.Classes.First(c => c.Name == "Physics");

foreach(var student in students)

{

physics.Students.Add(student);

}

var music = ctx.Classes.First(c => c.Name == "Music");

foreach(var student in students.Take(2))

{

music.Students.Add(student);

}

var art = ctx.Classes.First(c => c.Name == "Art");

art.Students.Add(students.First());

ctx.SaveChanges();

}

1. Run the program and ensure it executes without error.

# Part V – Extra Credit

1. Modify the program so it first deletes all entities in the database.

2. Write a basic “report” that writes to the console a list of all classes, and the students inside each class.

# Conclusion

Congratulations! You’ve used a model first approach with the Entity Framework and worked with classes that have a many to many relationship.