



Entity Framework and Repository Pattern

Estimated time for completion: 60 minutes

Overview:

In this lab you will take advantage of Entity Framework 4 support for POCO's to completely decouple application logic from the data access implementation, which will enable you to test the application logic without the use of a database.

Goals:

- See the benefit of decoupling data access from application logic.
- Learn how to use POCO's with Entity Framework.
- Construct unit tests for application logic that doesn't require a database.

Lab Notes:

Download and install the T4 POCO generator from

<http://visualstudiogallery.msdn.microsoft.com/en-us/23df0450-5677-4926-96cc-173d02752313>

Run the AcmeCorp.sql script to create the database. You may need to change the -S flag to point to your sql server instance e.g. -S.\SQLEXPRESS

```
osql -E -S. -I AcmeCorp.sql
```

Part 1: Refactor to POCOs

In this part of the lab you will create a new class library to contain the domain entities replacing the domain entities automatically generated by Entity Framework.

Open the project AcmeCorp.sln. Compile and run the project you will be transported back in time to the days of VT100 and a simple office purchase order system. You will be shown some open purchase orders you can add line items to these orders by first listing a given purchase order and then adding what ever items you like to the order. Assuming you don't exceed the value for the purchase order. If the green screen and slow output is not for you, you can change the terminal type to Wizzy

```
ITerminal terminal = new VT100();  
  
ITerminal terminal = new WizzyTerminal();
```

The solution comprises of three projects, a primitive UI (PurchaseOrderTerminal), a Service layer that orchestrates domain logic (AcmeCorpServices) and finally an Entity Framework project that is responsible for data access and domain entities

(AcmeCorpEF). Currently the service layer is tightly coupled to the use of EF, your goal is to remove this dependency, whilst still leveraging the power of EF.

Open the AcmeCorp.edmx file and examine the model.

Create a new class library project called AcmeCorpEntities. This class library will be used to contain the POCOs.

To generate the POCO's you can either create them by hand or use POCO t4 template to generate the POCO's from the model. If you are feeling adventurous have a go at building them by hand, don't forget to generate anObjectContext too.

If you take the generate route, create a new solution folder inside the AcmeCorpEntities call it "Generated Code". Now add a new item to this folder of type "ADO .NET POCO Entity Generator", call it AcmeCorp.tt. You will see errors in the output window, don't panic this is because you haven't directed it to the edmx file yet. Modify the AcmeCorp.tt and the AcmeCorp.Context.tt files and ensure that the **inputFile** variable points to the edmx file and that the namespace for the pocos is set to **AcmeCorpEntities**

```
string inputFile = @"..\..\AcmeCorpEF\acmeCorp.edmx";
EdmItemCollection ItemCollection = loader.CreateEdmItemCollection(inputFile);
string namespaceName = "AcmeCorpEntities";
```

Now right click on each of the tt files at select run custom tool. Expand the .tt files in the solution explorer to see the generated POCO's and ObjectContext.

In addition to the database persisted properties the entities have some additional code.

Inside the AcmeCorpEF project you will see partial class definitions for **PurchaseOrder** and **PurchaseOrderLineItem** this code needs to be placed into AcmeCorpEntities too, to complete the definition of the entities, move these files across remembering to change the namespace to AcmeCorpEntities.

Currently the AcmeCorpEF project still builds the entities, open the model and then via the model's property sheet, set the Code Generation Strategy to **None**.

Recompile, this will break the service and ui code as they are expecting the entities to be in the AcmeCorpEF project, add a new reference to **AcmeCorpEntities** for both projects and modify the using statements appropriately. Recompile. Finally you will need to load the assembly that contains the entity framework model explicitly since we are not using any types inside the assembly before we attempt to use your poco's.

```
Assembly.Load("AcmeCorpEF");
```

Re-run the purchase order terminal, and confirm all is still well in the Acme Corp Purchasing department.

You now have an assembly that contains the entities, with no reference to Entity Framework. However you still have an ObjectContext that does reference Entity Framework, you will address this in part 2

Solution : After Part 1\AcmeCorp.sln

Part 2: Refactor to use a Repository and Unit Of Work

1. You will now modify the AcmeCorpEntities project further to completely decouple the application code from Entity Framework. To achieve this decoupling you will use the Repository pattern and Unit of work pattern. The Repository pattern to encapsulate

objectsets and Unit of work to encapsulate the **ObjectContext** and define an interface to represent access to a collection of **PurchaseOrders**.

2. Inside the AcmeCorpEntities project, create an interface called **IPurchaseOrderRepository** as below

```
public interface IPurchaseOrderRepository
{
    IQueryable<PurchaseOrder> Entities { get; }

    void Add(PurchaseOrder entity);
    void Remove(PurchaseOrder entity);
}
```

3. Create two interfaces to represent the Unit Of Work as below

```
public interface IUnitOfWork : IDisposable
{
    IPurchaseOrderRepository PurchaseOrders { get; }
    void Commit();
}

public interface IUnitOfWorkFactory
{
    IUnitOfWork Create();
}
```

4. Now refactor the PurchaseOrderService class to use the interfaces you have defined above, make the constructor of the PurchaseOrderService take an instance of a IUnitOfWorkFactory, store it in a field and make it the basis of your refactoring. Below shows the GetAllOpenPurchaseOrders after refactoring.

```
using (IUnitOfWork uw = uwFactory.Create() )
{
    return (from order in uw.PurchaseOrders.Entities
            select order).ToList();
}
```

5. Modify program.cs inside the PurchaseOrderTerminal project to create an instance of PurchaseOrderService with a null IUnitOfWorkFactory, recompile. Obviously it won't work.
6. Inside the AcmeCorpEF project create a new class called EFPurchaseOrderRepository, make it implement the interface IPurchaseOrderRepository. The constructor will take an **ObjectContext** as a parameter, use the object context to create an instance of a **ObjectSet<PurchaseOrder>** store it in a field, to be used later by the interface methods.

```
public EFPurchaseOrderRepository(ObjectContext ctx)
{
    this.objectSet = ctx.CreateObjectSet<PurchaseOrder>();
}
```

7. Implement the interface methods, as follows

```
public virtual IQueryable<PurchaseOrder> Entities
{
    get { return objectSet; }
}
```

```

}
public virtual void Add(PurchaseOrder entity)
{
    objectSet.AddObject(entity);
}
public virtual void Remove(PurchaseOrder entity)
{
    objectSet.DeleteObject(entity);
}

```

8. Now provide an implementation of **IUnitOfWork**, called **EFUnitOfWork**. Add a constructor that takes a connection string as a parameter. Inside the constructor create an instance of an **ObjectContext** and save it in a field. Create a private field for **IPurchaseOrderRepository**, and initialize it with an instance of **EFPurchaseOrderRepository**.

```

public EFUnitOfWork(string connectionString)
{
    this.ctx = new ObjectContext(connectionString);
    this.purchaseOrderRepository = new EFPurchaseOrderRepository(ctx);
}

```

9. Finally implement **Commit** and **Dispose** to call the **SaveChanges** and **Dispose** on the context.
10. Now implement **IUnitOfWorkFactory** called **EFUnitOfWorkFactory**, and make the create method return an instance of **EFUnitOfWork**.

```

public class EFUnitOfWorkFactory : IUnitOfWorkFactory
{
    public IUnitOfWork Create()
    {
        return new EFUnitOfWork("name=AcmeCorpContainer");
    }
}

```

11. Finally modify program.cs to provide an instance of **EFUnitOfWorkFactory** instead of the null parameter to **PurchaseOrderService**. Recompile and re-run the application.
12. The application will crash, examine the exception. Do you know why ?
The reason is that we are only loading the **PurchaseOrder** objects and not any related objects. The **PurchaseOrderRepository** should load the full object graph for a purchase order (aka. Aggregate), modify **EFPurchaseOrderRepository** to not return just entities but to also include the **Supplier** and **LineItems**.

```

public IQueryable<PurchaseOrder> Entities
{
    get { return objectSet.Include("Supplier").Include("LineItems"); }
}

```

13. You now have application logic that is completely decoupled from the persistence layer implementation.

14. After confirming the application runs as expected, try re-factoring to create a generic `IRepository<T>` base interface, and an accompanying class called `EFRepository<T>`. Remove all the methods from `IPurchaseOrderRepository`, and simply make it extend `IRepository<PurchaseOrder>`.

Solution: After Part 2\AcmeCorp.sln

Part 3: Testing the Service Layer

In this part of the lab you will reap the benefits of allowing the implementation of the repository to vary by building a unit test against an in memory repository.

1. Create a new unit test project, call it `TestAcmeCorpServices`, create a unit test called `TestPurchaseOrderService`.
2. Now define a test that will verify that the method **`IPurchaseOrderService.GetAllOpenPurchaseOrders`** only returns open purchase orders. Open purchase orders are defined as `PurchaseOrders` with a `Status` of “Open”.
3. You will need to provide stubbed implementations of `IUnitOfWorkFactory`, `IUnitOfWork` and `IPurchaseOrderRepository`. Your fake repository will need to return its contents as `IQueryable<PurchaseOrder>`, use `AsQueryable<T>` extension method on `IEnumerable<T>` to convert to a query.
4. Your test code will find a bug, fix it and validate that the test now passes.

Solution: After Part 3\AcmeCorp.sln