Entity Framework and Repository Pattern

Estimated time for completion: 45 minutes

Overview:

In this lab you will take advantage of Entity Framework 5 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.
- Construct unit tests for application logic that doesn't require a database.

Lab Notes:

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 four projects, a primitive UI (PurchaseOrderTerminal), a Service layer that orchestrates domain logic (AcmeCorpServices), an Entity Framework project that is responsible for data access (AcmeCorpEF) and finally a DomaiObjects assembly containing all domain entities (AcmeCorpDomainObjects). Currently the service layer is tightly coupled to use EF, your goal is to remove this dependency, whilst still leveraging the power of EF.
Open the AcmeCorp.edmx file and examine the model.

Part 1: Refactor to use a Repository and Unit Of Work

1. You will now modify the AcmeCorpEntities project 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 **DbSets** and Unit of work to encapsulate the **DbContext**
2. Inside the AcmeCorpDomainObjects 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. Repeat this refactoring process for all the methods inside the PurchaseOrderService
6. Modify program.cs inside the PurchaseOrderTerminal project to create an instance of PurchaseOrderService with a null IUnitOfWorkFactory, recompile. Obviously it won't work.
7. Inside the AcmeCorpEF project create a new class called EFPurchaseOrderRepository, make it implement the interface IPurchaseOrderRepository. The constructor will take a **DbContext** as a parameter, use the object context to create an instance of a **DbSet<PurchaseOrder>** store it in a field, to be used later by the interface methods.

```
public EFPurchaseOrderRepository(DbContext ctx)
{
   this.objectSet = ctx.Set<PurchaseOrder>();
```

```
}
```

8. 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);
}
```

9. 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 DbContext(connectionString);
  this.purchaseOrderRepository = new EFPurchaseOrderRepository(ctx);
}
```

10. Finally implement **Commit** and **Dispose** to call the **SaveChanges** and **Dispose** on the context.
11. 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("AcmeCorpEntities");
  }
}
```

12. Finally modify program.cs to provide an instance of **EFUnitOfWorkFactory** instead of the null parameter to PurchaseOrderService. Recompile and re-run the application.
13. 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"); }
}
```

14. You now have application logic that is completely decoupled from the persistence layer implementation.
15. After confirming the application runs as expected, try re-factoring to create a generic IRepository<T> base interface, and an occompanying class called EFRepository<T>. Remove all the methods from IPurchaseOrderRepository, and simply make it extend IRepository<PurchaseOrder>.

Solution: After Part 1\AcmeCorp.sln

Part 2: 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 2\AcmeCorp.sln