Repository and Unit of Work

Abstracting away the data access layer

Objectives

- Introduce Object Relational Mappers (ORM)
- Issues with coupling application logic to Data Access Layer
- Introduce anti corruption layer
  - Repository pattern
  - Unit of Work pattern

Object Relational Mapper (ORM)

- Automate the process of moving relational data into and out of objects
- True ORM's work with your
  - Domain Objects
  - Database Tables,Views and SPROCS
- Domain Objects should be persistent ignorant
- ORM's on the .NET platform
  - Microsoft Entity Framework
  - NHibernate
  - Many more

Entity Framework

- Core Concepts
  - Modelling
    - Conceptual model
    - Storage model
    - Mapping
  - Context
    - Entity Sets
      - IQueryable<T>
    - Change Tracking
- Modelling can be represented either by
  - XML
  - Code

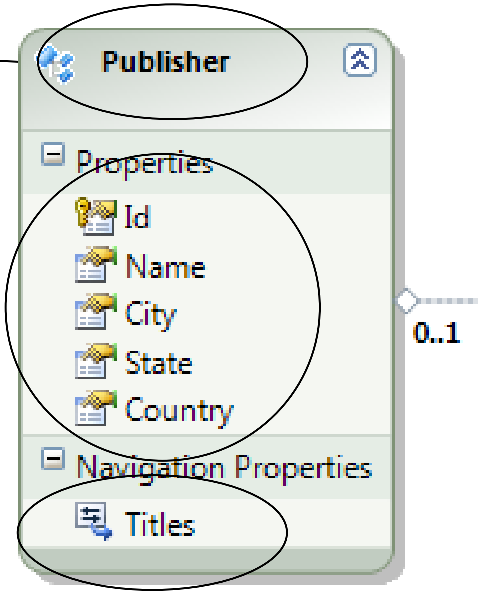
Plain Old CLR Object (POCO)

- True ORM’s don’t dictate object structure
- EF 4 supports POCOs
  - Entity types do not derive from EF base class
  - Mapping layer cares about shape not type

POCO, Publisher

- POCO class needs to look like conceptual model.
- Can be auto generated via T4 templates

```
 public class Publisher
{
  public Publisher()
  {
    Titles = new List<Title>();
  }

  public string Id { get; private set; }
  public string Name { get; set; }
  public string City { get; set; }
  public string State { get; set; }
  public string Country { get; set; }


  public List<Title> Titles {get; private set;}
}
```



Fetching entities

- EF ObjectContext class provides core services
  - Entity Sets
  - Change tracking
- Create instance of ObjectContext with entity connection string
- Use CreateObjectSet<T> to create the entity set
  - Uses class name to map to entity conceptual name

```
ObjectContext ctx = new ObjectContext(connectionString);

foreach( Publisher publisher in ctx.CreateObjectSet<Publisher>() )
{
   Console.WriteLine(publisher.Name);
}
```

POCO, Change tracking

- Two ways
  - Object Context snapshots fetched objects
    - Diffs current objects against snapshot
  - Properties are marked as virtual
- Computing differences is least preferred
- Virtual mechanism provides opportunity for property interception

Dynamic Proxy

- Properties to be tracked marked as virtual
- EF will create derived class at runtime to intercept property methods
- Derived class updates object context of entities changed state

```
public class Publisher
{
  public virtual string Name { get; set; }
  public virtual string Id { get; set; }
  public virtual string Country { get; set; }
  public virtual string State { get; set; }
  public virtual string City { get; set; }
}
```

Dynamic Proxy in action

- New trackable entity
  - Use CreateObject<T> not new

```
ObjectContext ctx = new ObjectContext(connectionString);

var publisher = ctx.CreateObject<Publisher>();

Console.WriteLine(publisher.GetType().Name);
Console.WriteLine(publisher.GetType().BaseType.Name);
```

```
Publisher_453BC767D7B3D7F27D01DC2794E7D8F4447CE2BF7D71DC230119D1AF9705C7E4
Publisher
```

Why not use an ORM directly

- Writing Application code against a given ORM can lead to
  - Unit test difficulty
  - High degree of coupling to a given provider
- Applications often need to evolve longer than their component parts
  - Would like to continually use best of breed
- POCO gets you some of the way
  - Not coupled to EF for Entities
- Need to replace direct use of ObjectContext
  - Add additional layer of abstraction

Introducing the Repository Pattern

- Provides a collection based view of entities
  - Entities can be fetched
  - New entities inserted
  - Entities can be removed
- Hides data access layer interactions
- Focus on objects
- Single repository often used to represent a graph of objects
  - Called an aggregate
- Repositories represent initial use case entry points
  - Don’t need to provide a repository for every domain object

Defining the repository

- Define interface, allowing implementation to vary
- Application coded against interface

```
 public interface IPublisherRepository {
   // Add and remove a publisher from the repository
   void AddPublisher(Publisher publisher);
   void DeletePublisher(Publisher publisher);

   // Return all the publishers
   IEnumerable<Publisher> All{ get; }

   // Return a given publisher
   Publisher FindByName( string name);

   // Save changes back to data store
   void SaveAll();
}
```

Entity Framework Repository Implementation

```
public class EFPublisherRepository : IPublisherRepository {
 private ObjectContext ctx = new ObjectContext("...");
 private ObjectSet<Publisher> publishers;

 public EFRepository() {
   publishers = ctx.CreateObjectSet<Publisher>();
 }

public IEnumerable<Publisher> All { get{return publishers;} }

public Publisher FindByName(string name )  {
        return publishers.Where( p=>p.Name == name).Single();
}
 public void AddPublisher(Publisher publisher)
     { publishers.AddObject(publisher); }
 public void DeletePublisher(Publisher publisher)
     { publishers.DeleteObject(publisher); }

public void SaveAll() { ctx.SubmitAllChanges(); }
}
```

Using the repository

- Application logic coded against IPublisherRepostiory
  - Repository implementation can vary

```
 IPublisherRepository repository = new EFRepository();

 foreach(Publisher publisher in repository.Publishers)
 {
    Console.WriteLine(publisher.Name);
 }

Publisher publisher = repository.FindByName(“Rich”);
```

Queries, IEnumerable<T>

- Queries defined by FindXXX style methods
  - Advantages
    - Encapsulate query mechanics
    - Tune one place not many
    - Replace expensive Linq queries with SPROCs
    - Keeps testing simpler
  - Cons
    - Application logic can't utilise LINQ directly

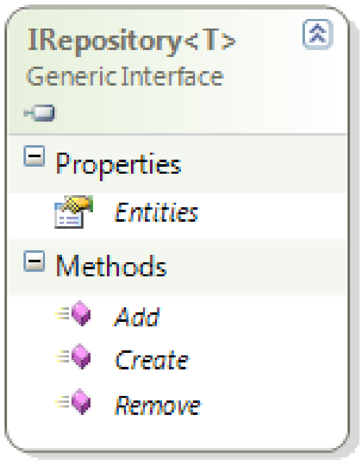
Queries, IQueryable<T>

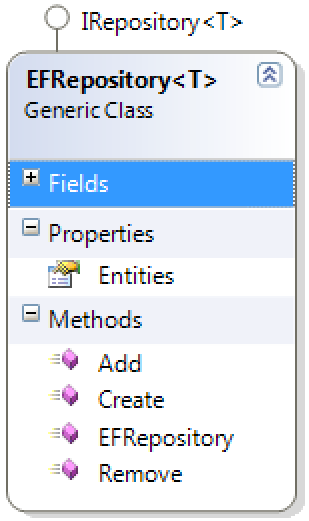
- Repository provides simple properties to access sets
  - Exposing IQueryable<Publisher> All
  - Related objects exposed via AllWithXXXX
- Advantages
  - Application code refines sets through Linq
- Cons
  - Hard to enforce good db query plans
  - Can cause issues with unit testing

```
public interface IPublisherRepository {
        IQueryable<Publisher> All{ get; }
        IQueryable<Publisher> AllWithTitles {get;}
}

IPublisherRepository repository = new EFRepository();

 var withManyTitles = from publisher in repository.Publishers
                      where publisher.Titles.Count > 0
                      select publisher;
```

Generic Repository

- Repository interface good candidate for generics
  - IRepository<T>
  - Build generic implementation
- IRepository<T> defines
  - CRUD operations
- Still consider building specific repository interfaces
  - Allows opportunity
    - Encapsulate complex queries
    - Encapsulate calls to stored procs





Standalone Repositories

- Issues
  - Business transactions will often need to touch many repositories
    - All updates should work or none should work
  - Repositories should really just represent a collection of objects
    - Repository currently has the notion of "Save"

Unit of Work

- Application transaction
  - May require the use of many repositories
  - All repositories should update or none update
  - Known as a Unit of Work
- Transaction behaviour needs to be moved out of repository
  - Remove Save method from repository
  - Create new interface to represent Unit of Work
- Unit of Work provides
  - Abstract factory for creating Repositories
  - Commit method

Unit of Work in action

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

```
public interface IUnitOfWork : IDisposable
{
   IPublisherRepository Publishers { get; }
   ITitlesRepository Titles { get; }

   void Commit();
}
```

```
using (IUnitOfWork uw = uwFactory.Create())
{
  IPublisherRepository publishers = uw.Publishers;

  // . . .
  uw.Commit()
}
```

Entity Framework Unit of Work

```
public class EFUnitOfWork : IUnitOfWork{
 private ObjectContext ctx;
 private IPublisherRepository publishers;

 public EFUnitOfWork(string connectionString)
 {
   ctx = new ObjectContext(connectionString);

   publishers = new EFPublisherRepository(ctx);
 }

 public IPublisherRepository Publishers {
   get { return publishers; }
 }

 public void Save(){
   ctx.SaveChanges();
 }
}
```

Object Context created shared across repositories

IxxRepository vs IObjectSet<T>

- IRepository
  - "anti-corruption" layer no reference to implementation types
  - Can contain additional queries that meet exact business needs
    - FindProductsOnSale()
  - Adding explicit queries gives greater control on how the queries are executed
  - Consider returning IEnumerable rather than IQueryable to take complete control of queries

Testing

- Unit testing application logic
  - Stub repository to behave as required
  - Build general purpose In Memory repository

Summary

- Repository Pattern used to provide persistent ignorant abstraction for collections of domain objects
  - Consider using IEnumerable over IQueryable to take greater control
- Unit of work, provides change tracking across many repositories
- Own Repository and Unit Of Work Interfaces offer greater flexibility
  - Vendor neutral
  - Designed with testing in mind