Entity Framework and Repository Pattern

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Agenda

- **Decouple Application code from EF**
  - Repository pattern
  - Unit of Work pattern

Entity Framework

- **Application code making direct calls to**
  - DBContext and DBSet<T>
- **Your application is now coupled to EF**
- **What if**
  - You want to unit test without a database ?
  - MS drops EF ?
  - You want to drop EF ?
- **Solution**
  - Provide a new layer above DAL non EF specific
    - Anti corruption layer

Replacing DbContext and DbSet<T>

- **Application code**
  - using DbContext will always hit database
  - using DbSet<T> allows it to build custom queries
- **Replace direct use of DbContext**
  - Allow injection of test data
- **Replace direct use of DbSet<T>**
  - Encapsulate all queries
  - Less restrictions on backend, greater flexibility of persistance

```
using (PubsContext ctx = new PubsContext())
{
    var usPublishers = from publisher in ctx.Publishers
                       where publisher.country == "USA"
                       select publisher;
}
```

Introducing the Repository Pattern

- **Provides a collection based view of entities**
  - Entities can be fetched
  - New entities inserted
  - Entities can be removed
- **Hides database interactions**
- **Focus on objects**
- **Single repository often used to represent a graph of objects**
  - Called an aggregate

Defining the repository

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

```
public interface IPublisherRepository {
   // Create a version of new "Proxied" Publisher
   Publisher Create();

   // Add and remove a publisher from the repository
   void AddPublisher(Publisher publisher);
   void DeletePublisher(Publisher publisher);

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

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

Entity Framework Repository Implementation

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

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

  public Publisher Create()
        {return publishers.CreateObject<Publisher>();}
  public IEnumerable<Publisher> Publishers { get{return publishers;} }
  public Publisher FindByName(string name )  {
        return publishers.Where( p=>p.Name == name).Single();
  }
  public void AddPublisher(Publisher publisher)
     { publishers.Add(publisher); }
  public void DeletePublisher(Publisher publisher)
     { publishers.Remove(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

- **Queries defined by FindXXX style methods**
  - Advantages
    - Encapsulate query mechanics
  - Cons
    - Application logic can't utilise LINQ directly
- **Allow adhoc queries**
  - Exposing IQueryable<Publisher>

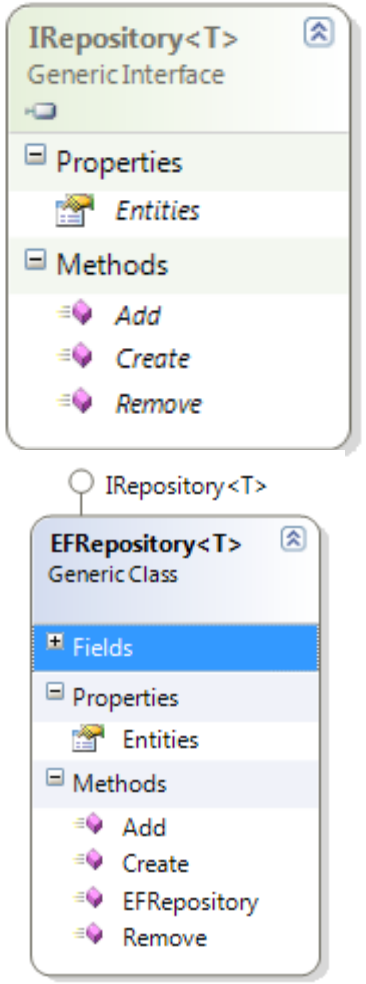
```
IPublisherRepository repository = new EFRepository();

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

foreach(Publisher publisher in withManyTitles )
{
   Console.WriteLine(publisher.Name);
}
```

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



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 DbContext ctx;
 private IPublisherRepository publishers;

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

   publishers = new EFPublisherRepository(ctx);
 }

 public IPublisherRepository Publishers {
   get { return publishers; }
 }

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

Object Context created shared across repositories

IxxRepository vs IDbSet<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 querys

Testing

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

Summary

- Entity Framework 4, provides true ORM through the use of POCO's
- Use POCO's to truly separate persistence from application logic
- Repository Pattern and Unit of Work standard patterns for separating application logic from data access technology