Why the Repository Pattern

By **Jamie Munro**, 30 May 2013

Introduction

The following article will explore a few different reasons that I believe in why the Repository pattern is an extremely useful paradigm that should be used more often - especially when it comes to ORM frameworks like Entity Framework.

After the incredible reaction to a recent blog post, Entity Framework Beginner's Guide Done Right, I feel like before writing some more code to further the basic example, I'll take a step back and explain my beliefs in the Repository pattern.

I was really overwhelmed with the reaction; what started with a simple 30 minute blogging effort has turned into something absolutely incredible. The original post was really a starting point about not placing direct querying and saving to the database mingled with the core code.

Please note, these are my thoughts on the pattern based on current and previous pain points that I've/am experiencing and I'd love to hear others input in making things better.

Before diving in, you can find the full source code up on GitHub: https://github.com/endyourif/RepoTest/

Making my code more readable

At its most basic level, the goal of a repository layer is to remove querying of content from my code. I personally find reading a function like this is much easier to read:

Collapse | Copy Code

```
// Display all Blogs from the database
var query = repo.GetAll();
Console.WriteLine("All blogs in the database:");
foreach (var item in query)
{
    Console.WriteLine(item.Name);
}
```

Than a function like this:

Collapse | Copy Code

```
// Display all Blogs from the database
var query = from b in db.Blogs
orderby b.Name
select b;
Console.WriteLine("All blogs in the database:");
foreach (var item in query)
{
    Console.WriteLine(item.Name);
}
```

In the first code example, I'm not boggled down by reading a LINQ query; I just need to deal with the results of the query from the function `GetAll`. This of course is probably a bad name; it might better to update the function name to be `GetAllBlogsOrderedByName`. It's of course very long, but incredibly descriptive. It leaves nothing to the imagination about what is happening in that function.

Segregating Responsibility

I work in a reasonably sized team. In this team, some people are great at writing highly optimized LINQ or SQL queries; while others on the team are great at executing the business logic pieces; while others are just juniors and need to earn my trust!

By adding a repository layer, I can segregate that responsibility. You're great at writing LINQ – perfect – you own the repository layer. If someone requires data access, they go through you. This will help prevent bad queries, multiple functions that achieve similar results (because somebody didn't know about XYZ function), etc.

Layering my code

I love the MVC design pattern – Model View Controller. Where the Controller talks to the Models to get some data and passes this through to the View for output.

So many times I have seen people mashing everything into the Controller – because it's the middle man. I actually wrote a blog post in 2009 that at the time was directed towards CakePHP: Keeping your CakePHP Controllers Clean. This of course applies to more than just CakePHP and even more than just controllers.

I like to start with a nice clean function that splits off into multiple small functions that perform the necessary work for me. E.g. error checking, business logic, and data access. When I have to debug this code, it makes it extremely easy to pin point the potential lines of code causing the issue; typically inside a single 10 line function; opposed to inside a large 100+ line function.

Adding a caching layer

This is not always an obvious feature, the average website or application might not require caching. But in the world I deal with daily, it's a must. By creating a repo layer, this is really simple.

Let's look at the previously discussed function `GetAllBlogsOrderedByName` with caching implemented:

⊟ Collapse | Copy Code

```
private static IOrderedQueryable<Blog> _blogs;
public IOrderedQueryable<Blog> GetAllBlogsOrderedByName()
{
    return _blogs ?? (_blogs = from b in _bloggingContext.Blogs
        orderby b.Name
        select b);
}
```

Knowing that my blogs only change when I save something, I can leverage C#'s ability of a static variable being in memory for the lifetime of the application running. If the `_blogs` variable is `null` I then and only then need to query for it.

To ensure this variable gets reset when I change something, I can easily update either my `AddBlog` or `SaveChanges` function to clear the results of that variable:

⊟ Collapse | Copy Code

```
public int SaveChanges()
{
    int changes = _bloggingContext.SaveChanges();
    if (changes > 0)
    {
        _blogs = null;
    }
    return changes;
}
```

If I was really courageous, I could even re-populate the `_blogs` variable instead of simply clearing it.

This example of course is only a starting point where I'm leveraging static variables. If I had a centralized caching service, e.g., Redis or Memcache, I could add very similar functionality where I check the variable in the centralized cache instead.

Unit Testing

I briefly mentioned this in the previous post as well, by separating my functionality it makes unit testing my code much easier. I can test smaller pieces of my code, add several different test scenarios to some of the critical paths and less on the non-critical paths. For example, if I'm leveraging caching, it's very important to test this feature thoroughly and ensure my data is being refreshed appropriately as well as not being constantly queried against unnecessarily.

A few changes are required to the original code to accomplish this. An interface is required for `BlogRepo`. The constructor of `BlogRepo` requires subtle changes as well. It should be updated to accept an interface so I can fake the `DbContext` so my unit tests don't connect to a real database.

⊟ Collapse | Copy Code

```
public interface IBlogRepo : IDisposable
{
    void AddBlog(Blog blog);
    int SaveChanges();
    IOrderedQueryable<Blog> GetAllBlogsOrderedByName();
}
```

Interface for the `DbContext` and minor updates to `BloggingContext`:

⊟ Collapse | Copy Code

```
public interface IBloggingContext : IDisposable
{
    IDbSet<Blog> Blogs { get; set; }
    IDbSet<Post> Posts { get; set; }
    int SaveChanges();
}
```

```
public class BloggingContext : DbContext, IBloggingContext
{
    public IDbSet<Blog> Blogs { get; set; }
    public IDbSet<Post> Posts { get; set; }
}
```

Now the constructor in BlogRepo can be updated to expect an IBloggingContext. We'll also perform another enhancement by creating an empty constructor that will create a new BloggingContext:

⊟ Collapse | Copy Code

```
public class BlogRepo : IBlogRepo
{
    private readonly IBloggingContext _bloggingContext;
    public BlogRepo() : this(new BloggingContext())
    {
    }
    public BlogRepo(IBloggingContext bloggingContext)
    {
        _bloggingContext = bloggingContext;
    }
    public void Dispose()
    {
        _bloggingContext.Dispose();
    }
    …
}
```

As some of the comments alluded to from the original post, I was still creating the DbContext in the main *Program.cs*. By altering the BlogRepo and interface to implement IDisposable, I can either automatically create a new BloggingContext or pass in an already created context that will get disposed automatically at the end of execution.

Now our Main function can remove the using statement that creates a new BloggingContext:

⊟ Collapse | Copy Code

```
static void Main(string[] args)
{
    // Create new repo class
    BlogRepo repo = new BlogRepo();
    …
}
```

Now that I have improved the core code, I can create and add a new test project to my solution. Inside the test project, I need to create a new class called FakeDbSet and FakeDbContext. These classes will be used in my testing to mock my BloggingContext:

⊟ Collapse | Copy Code

```
public class FakeDbSet<T> : IDbSet<T> where T : class
{
    ObservableCollection<T> _data;
    IQueryable _query;
    public FakeDbSet()
    {
        _data = new ObservableCollection<T>();
        _query = _data.AsQueryable();
    }
    public T Find(params object[] keyValues)
    {
        throw new NotSupportedException("FakeDbSet does not support the find operation");
    }
    public T Add(T item)
    {
        _data.Add(item);
        return item;
    }
    public T Remove(T item)
    {
        _data.Remove(item);
        return item;
    }
    public T Attach(T item)
    {
        _data.Add(item);
        return item;
    }
    public T Detach(T item)
    {
        _data.Remove(item);
        returnreturn item;
    }

}
```

And `FakeDbContext` which implements `IBloggingContext` interface:

Collapse | Copy Code

```
class FakeDbContext : IBloggingContext
{
    public FakeDbContext()
    {
        Blogs = new FakeDbSet<Blog>();
        Posts = new FakeDbSet<Post>();
    }
    public IDbSet<Blog> Blogs { get; set; }
    public IDbSet<Post> Posts { get; set; }
    public int SaveChanges()
    {
        return 0;
    }
    public void Dispose()
    {
        throw new NotImplementedException();
    }
}
```

And finally a `BlogRepoTest` class that tests the `AddBlog` and `GetAllBlogsOrderedByName` functions:

Collapse | Copy Code

```
/// <summary>
///This is a test class for BlogRepoTest and is intended
///to contain all BlogRepoTest Unit Tests
///</summary>
[TestFixture]
public class BlogRepoTest
{
    private BlogRepo _target;
    private FakeDbContext _context;
    #region Setup / Teardown
    [SetUp]
    public void Setup()
    {
        _context = new FakeDbContext();
        _target = new BlogRepo(_context);
    }
    #endregion Setup / Teardown
    /// <summary>
    ///A test for AddBlog
    ///</summary>
    [Test]
    public void AddBlogTest()
    {
        Blog expected = new Blog {Name = "Hello"};
        _target.AddBlog(expected);
        Blog actual = _context.Blogs.First();
        Assert.AreEqual(expected, actual);
    }
    /// <summary>
    ///A test for GetAllBlogsOrderedByName
    ///</summary>
    [Test]
    public void GetAllBlogsOrderedByNameTest()
    {
        FakeDbSet<Blog> blogs = new FakeDbSet<Blog>();
        IOrderedQueryable<Blog> expected = blogs.OrderBy(b => b.Name);
        IOrderedQueryable<Blog> actual = _target.GetAllBlogsOrderedByName();
        Assert.AreEqual(expected, actual);
    }
}
```

Interchangeable Data Access Points

In the original post, I mentioned about swapping out ORMs, this of course is not a regular scenario but could happen one day.

The other more likely scenario is multiple or different end points for the repository layer. There might be a reason for a repository to want to connect both to a database and some static files.

By creating the repository layer, the program itself that uses the data doesn't need to care or concern itself with where the day is coming from, just that it comes back as a LINQ object for further use.

Summary

This probably might just be a tipping point for many of the great reasons to use a repository pattern. Hopefully this posts helps fill in some of the gaps from the original post on Entity Framework Beginner's Guide Done Right. Once again, the full source is available on GitHub: https://github.com/endyourif/RepoTest/.