Creating a Test Driven DAL with Moq

(Data Access/Abstraction Layer)

A Helpful Guide Because Code Without Unit Tests Is No Code At All

Steps for Creating Your Repository (Data Access Layer)

1. ERD (entity relationship diagram)
2. Model classes
3. Your app’s context class
   a. Use DbSets for each of the models you’ll need in database tables
4. Entity Framework Migrations
   a. Enable migrations, add migrations, and update database as needed
5. Repository
   a. Use TDD to create the methods that use your context to hit the db.

Make Your ERD

Things to Consider

1. What does your application need to do?
2. What do you need to save in your database?
3. What are the relationships present in your data?
    a. One to One
    b. One to Many
    c. Many to Many

How Many Programmers does it take to change a lightbulb?

None. That's a Hardware Problem.

Make Your Models

Models

Classes That Represent Database Tables

Make a class for each of the entities in your ERD

The relationships between entities in your ERD are shown using the **virtual** keyword

Models - Data Annotations

```
using System.ComponentModel.DataAnnotations;

[Key] //table's primary key
public int Id { get; set; }

[Required] //table column will not allow null entries
public string Name { get; set; }
```

NOTE: Never use [Required] on properties with **virtual** keyword.

Models - Virtual Properties

Use **virtual** properties to represent the relationships between your models. For example, in a Cohort class you might see these virtual properties:

```
public virtual List<Student> Students {get;set;}
public virtual Instructor PrimaryInstructor {get;set;}
```

Setting these up allows Entity Framework to make the foreign key relationships in your database that exist in your ERD

Warning:

Setting up many-to-many relationships in your entities can cause a serialization error that gives the user of your app a 500 Internal Server Error for no reason.

If you are planning a many-to-many relationship in your Web API app (or need to JSON serialization of a many-to-many relationship in any app) add this to your Startup.cs:

```
GlobalConfiguration.Configuration.Formatters.JsonFormatter.
SerializerSettings.Re ferenceLoopHandling =
ReferenceLoopHandling.Ignore;
```

["HIP","HIP"]

(HIP HIP ARRAY!)

Create your application's DbContext

Application Context

Your application's context class should inherit from DbContext

Make a **public virtual DbSet<YourModel>** for each of your models that represent database tables. These are the classes mentioned in your ERD.

NOTE: Other types of classes like helper classes and services, controllers, and ViewModels are NOT a part of a DbSet!

Application Context

Think of your Context as the C# representation of your database. Once your database is created with migrations, the DbSets in your Context and your Tables in your Database will be very similar.

Some differences will occur as Entity Framework builds the Database structure needed for the data relationships between classes (your virtual properties).

While in your code, when you think Database, you need to think Context. When you think Tables, you need to think of the DbSet properties on your Context.

A man sends his Programmer wife to the grocery store for a loaf of bread. On Her way out He says "and while you're there, get a carton of eggs".

She never returned.

Create Your Database

Database Migrations - .NET 4.6.1

In your package manager console,

Enable-Migrations -ContextTypeName {YOUR_CONTEXT},

Add-Migration InitialCreate, and

Update-Database to apply the InitialCreate migration.

Database Migrations - .NET Core

In your package manager console,

```
    Add-Migration YourMigrationName
```

This will create a migration file that generates the code to build and tear down your database changes.

```
    Update-Database
```

This will apply the YourMigrationName migration.

Optimist: The Glass is Half Full

Pessimist: The Glass is Half Empty

Programmer: The Glass is Twice As Large as it Needs to Be

Make Your Repository

Repository

Make a class for your repository.

Consider the purpose of your repository in your application. What do you need to be able to do in with your database? Create, Read, Update, and Delete for all your models? Planning here will save you time! Sometimes you don’t need everything!

After you consider this - but before you write your actual implementation - it’s time to write your first unit test.

Unit Testing

Good Code is Testable Code

1. Add a Unit Test Project
2. Add NuGet Packages
   a. Moq
   b. Entity Framework
3. Add a Unit Test class

Unit Testing

The very first test is usually your ‘can I make an instance of this class’ test, because you need to be able to make an instance of your class in order to use it.

Don’t skip this one.

Seriously, don’t.

Let’s show you what it’s doing and how it’s important.

First things first, think about what would you like to be able to do with your repository?

Get or add something to the database?

Well, you can’t do that without your application’s dbcontext.

Your repo should *always* have access to an instance of Context.

Dependency Injection

In order for your repository to use your dbcontext, make the constructor of your repository class take an instance of your dbcontext class as a parameter, and assign the dbcontext being passed into a property on the repository class.

.NET 4.6.1 - without Dependency Injection Framework Only:

Then make another constructor that doesn’t take a Context instance as a parameter, but instead creates a new instance of Context for the property on your repository class.

Repository Constructors (.NET 4.6.1 w/o Dependency Injection Framework)

```
public KatesAwesomeRepository(KatesAwesomeContext context)
{
    Context = context;
}

public KatesAwesomeRepository()
{
    Context = new KatesAwesomeContext();
}

private KatesAwesomeContext Context {get;}
```

Repository Constructor (.NET Core or 4.6.1 with Dependency Injection Framework)

```
public class KatesAwesomeRepository: IKatesAwesomeRepository
{
    public KatesAwesomeRepository(KatesAwesomeContext context)
    {
        Context = context;
    }
    private KatesAwesomeContext Context {get;}

    ... //Insert your repo functionality here.
}
```

Dependency injection SetUp in StartUp.CS (.NET Core built in)

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    services.AddDbContext<KatesAwesomeContext>(options =>
         options.UseInMemoryDatabase()
     );
    services.AddTransient<IKatesAwesomeRepository,
    KatesAwesomeRepository>();
    ...
}
```

Why is this important?

Injecting dependencies like your dbcontext into your class in the constructor allows you to use a Mock in your unit tests. Later on, you might also want to use a different context for a QA database vs. a Production database. By using the constructor you can decide every time you make a new instance of the class what dbcontext you want to use.

Understanding the usage of a dependency injection framework (whether built in .NET Core or an external package in .NET 4.6.1) is an important part of building a large .NET Application in a maintainable way.

Dependency Injection

A software design pattern that implements inversion of control for resolving dependencies.

A dependency is an object that can be used (your dbcontext). An injection is the passing of a dependency to a dependent object (your repository) that would use it. **Your repository only needs to know how to USE your dbcontext – not how to CREATE one.**

Dependency injection makes the creation of objects on which a class depends another class's problem and keeps your classes concise and cohesive.

"It should be noted that no ethically-trained software engineer would ever consent to write a DestroyBaghdad procedure. Basic professional ethics would instead require him to write a DestroyCity procedure, to which Baghdad could be given as a parameter."
- Nathaniel Borenstein

Mocking With Moq

You use your application’s dbcontext class to interact with the database using methods like Add, Remove, and SaveChanges. Using your dbcontext in your repository makes sense, but you don’t want your unit tests testing or touching your actual database.

This is where Moq and Mocking comes in.

Your First Repo Unit Test

Your first test should see if your Repo can be instantiated. Because you know that you will *always* need a context, that's what your test should verify:

```
public void RepositoryHasContextWhenInstantiated()

{

    KatesAwesomeRepository repo = new KatesAwesomeRepository();

    Assert.IsNotNull(repo.Context);

}
```

But like we talked about, you don’t want to use the real DbContext in your unit tests. So you can set up your Mock dbcontext now. Here’s how.

Add a property or data member to your unit test class for your Mock dbcontext.

```
private Mock<KatesAwesomeContext> context = new Mock<KatesAwesomeContext>();
```

When you are using your repo after your first test, we are going to pass the new instance of your repo this mock context. This is the syntax for that:

```
KatesAwesomeRepository repo = new KatesAwesomeRepository(context.Object);
```

Your Second Repo Unit Test

Your second unit test should see if your Repo can be instantiated with an instance of your Context passed in the constructor. Because you will *always* need a context, that’s what your test should verify this time around as well:

```
public void RepositoryHasAccessToPassedInContext()

{

    KatesAwesomeRepository repo = new KatesAwesomeRepository(context.Object);

    Assert.IsNotNull(repo.Context); //What does this tell us?

    Assert.AreEqual(context.Object, repo.Context); //This tells us more.

}
```

A QA Engineer Walks Into A Bar And Asks For

- 1 Beer
- 2 Beers
- 1.3 Beers
- -1 Beers
- Null Beers
- A Lizard
- ;eoaighsergudsl2hi23434h

Mocking Your DbSets With Moq

Once your first tests pass, think about what is the next thing you’d like to be able to do with your repository. Think about your basic CRUD actions.

After you decide what you need to do with your repository first, you need to set up the mocks you’ll need to interact with your DbSets (the DbSets represent your database tables, remember?)

For every DbSet on your Context that your repo will interact with, make a Mock DbSet and a List<> of that model as a property on your unit test class.

```
    private Mock<DbSet<TShirt>> Shirts { get; set; }

    private Mock<DbSet<Logo>> Logos { get; set; }

    private List<TShirt> ShirtList { get; set; }

    private List<Logo> LogoList { get; set; }
```

We are going to use that List to set up your Mock DbSet.

Make a [Test Initialize] SetUp method.

This method will run at the beginning of every unit test you write. You’re going to use this method to set up your Mocks so you don’t clutter your actual unit tests with stuff you are going to have to repeat.

```
[TestInitialize]
public void SetUp()
{
    //you will set up your Mocks in here.
}
```

Test Initialize Method

You need to set each of your properties to a new instance in your test initialize method. I like to add test data to my lists at this point.

```
[Test Initialize]
public void Setup()
{
    Shirts = new Mock<DbSet<TShirt>>();
    ShirtList = new List<TShirt> { new TShirt {Color = “red”,
… } };
    ...
}
```

Make another method in your class. We’ll call it SetUpMocksAsQueryable. Below where you set up your mocks and lists in your TestInitialize method, call SetUpMocksAsQueryable. In your method, cast the List you created for your model as an IQueryable.

```
public void SetUpMocksAsQueryable ()
{
    var shirtQueryable = ShirtList.AsQueryable();
    var logoQueryable = LogoList.AsQueryable();
}
```

Note: Casting is changing one type to another. Think about it like .ToString(). This changes as List<> to an IQueryable<>.

You’re going to use that IQueryable to set up your Mock DbSet. You need “borrow” the parts of the IQueryable you need for the Mock DbSet to work.

You need to set up your Mock DbSet with the IQueryable’s query provider, expression tree, type, and its ability to enumerate.

All of these properties and methods of an IQueryable will allow you to treat your Mock DbSet like a real DbSet.

Set your Mock DbSets up!

```
Shirts.As<IQueryable<TShirt>>().Setup(x => x.Provider).Returns(shirtQueryable.Provider);

Shirts.As<IQueryable<TShirt>>().Setup(x => x.Expression).Returns(shirtQueryable.Expression);

Shirts.As<IQueryable<TShirt>>().Setup(x => x.ElementType).Returns(shirtQueryable.ElementType);

Shirts.As<IQueryable<TShirt>>()
     .Setup(x => x.GetEnumerator()).Returns(() => shirtQueryable.GetEnumerator());
```

The setup here says that whenever you use a method on your Shirts Mock DbSet, the application should use the methods that are on your shirtQueryable IQueryable instead.

Finally, in the last line of your TestInitialize method set up your Mock context to return your Mock DbSets.

```
context.SetUp(x => x.TShirts).Returns(Shirts.Object);
```

```
context.SetUp(x => x.Logos).Returns(Logos.Object);
```

As you write your unit tests you will add more code to your [TestInitialize] and SetUpMocksAsQueryable methods to make your Mocks respond the way you want them to.

Write your next unit test. This one can test whether your repository can get all the shirts in your database.

```
[Test Method]
public void CanGetAllTShirts()
{
    //Arrange
    var repo = new KatesAwesomeRepo (context.Object);//my Mock context
    //Act
    var actualShirts = repo.GetAll();
    //Assert
    Assert.AreEqual(actualShirts.Count, 1); //I have one in my dummy data.
}
```

Cool, now implement this .GetAll() in your repository class to make this test pass. Looking good!

```
public List<TShirt> GetAll()
{
    return context.TShirts.ToList();
}
```

A SQL STATEMENT WALKS INTO A BAR AND SEES TWO TABLES. IT APPROACHES, AND ASKS "MAY I JOIN YOU?"

A WEB DEVELOPER WALKS INTO A BAR. HE IMMEDIATELY LEAVES IN DISGUST AS THE BAR IS LAID OUT IN TABLES.

Now write another unit test. This one can test whether your repository can add an item to your database.

```
[Test Method]
public void CanAddTShirt()
{
     //Arrange
     var repo = new KatesAwesomeRepo (context.Object);//my Mock context
     TShirt testShirt = new TShirt {color= “green”, size= “large”};

     //Act
     repo.Add(testShirt);
}
```

Well, this won’t work. Your DbSet doesn’t know what to do when you try to add something to it. So let’s tell it, so we’ll have a way of testing it.

In your SetUp method, tell your Mock DbSet what to do when .Add is called on it.

```
Shirts.Setup(t => t.Add(It.IsAny<TShirt>())).Callback((TShirt shirt) => shirtList.Add(a));
```

This method says that whenever the .Add is called on the Mock DbSet, it should add it to your List. This is important, because then you’ll be able to verify your .Add is working.

Back in your unit test, finish your Assert section

```
[Test Method]
public void CanAddTShirt()
{
     //Arrange
     var repo = new KatesAwesomeRepo (context);//my Mock context
     TShirt testShirt = new TShirt {color= “green”, size= “large”};

     //Act
     repo.Add(testShirt);

     //Assert
     mockShirts.Verify(x => x.Add(testShirt), Times.Once); //Moq Testing Methods
     mockContext.Verify(x => x.SaveChanges(), Times.Once); //Moq Testing Methods
     Assert.IsTrue(shirtList.Contains(testShirt));
}
```

Implement this method in your repository in order to make your unit test pass.

```
public void Add(TShirt tshirt)
{
    context.TShirts.Add(tshirt);
    context.SaveChanges();
}
```

Repeat the unit test creation and implementation for each method you need in your repository.

Debugging = Removing the needles from the haystack.

THE END