

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; }
```

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

```
["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`!

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

In your package manager console,

Enable-Migrations -ContextTypeName {YOUR_CONTEXT},

Add-Migration InitialCreate, and

Update-Database to apply the InitialCreate 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.

After your first test which seems pretty simple, think about what is the next thing you'd 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

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.

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 in to a property on the repository class.

(Yes, this is a refactor of test 1, not test 2.)

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.

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

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>();
```

Use this mock context when refactoring your first test:

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

NOTE: A mock's .Object is the part of the mock that will have the same type as the class it's mocking.

Once you finish making your “can create an instance of your repository” test, you should go to your repository and write the implementation code. Here’s an example of how your constructor should look with dependency injection.

```
public class KatesAwesomeRepository
{
    private KatesAwesomeContext context { get; }

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

A QA ENGINEER WALKS INTO A BAR AND ASKS FOR

- 1 BEER
- 2 BEERS
- 1.3 BEERS
- -1 BEERS
- NULL BEERS
- A LIZARD
- ;EOAIGHSERGUDS12HI23434H

MOCKING YOUR DBSETS WITH MOQ

Once your first test passes, 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?)

Make private properties or data members in your unit test class for the Mock<> of your dbsets.

a. `Mock<DbSet<TShirt>> Shirts = new Mock<DbSet<TShirt>>();`

For every Mock DbSet, make a List of that model. I like to add some dummy data for my tests.

a. `List<TShirt> shirtList = new List<TShirt>`
`{`
 `new TShirt { Color = "Blue", Size = "Large"`
`};`

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.
}
```

In your `SetUp` method, cast the `List` you created for your model as an `IQueryable`

```
var shirtQueryable = shirtList.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, set up your Mock context to return your Mock DbSets.

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

As you write your unit tests you will add more code to your [TestInitialize] Setup method 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