Guia Moq-Pruebas unitarias

En esta guía veamos de una prueba para cada capa de forma de mockear diferentes dependencias.

Pruebas WebApi

Primero hagamos pruebas de **WebApi**. Para crear un proyecto de prueba se tiene que correr el siguiente comando:

> *dotnet new mstest -n WebApi.Tests*

Una vez que se nos creo el proyecto de pruebas, hay que agregarle las siguientes referencias:

- **Moq**
- BusinessLogicInterface
- Microsoft.AspNetCore.App
- WebApi

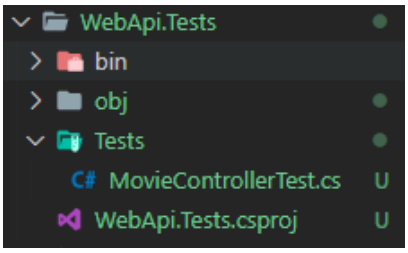
> dotnet sln add WebApi.Tests
> *cd WebApi.Tests*
> *dotnet add package Moq*
> *dotnet add package Microsoft.AspNetCore.App*
> *dotnet add reference ../WebApi*
> *dotnet add reference ../BusinessLogicInterface*
> *dotnet add reference ../Domain*

El paquete **Microsoft.AspNetCore.App** es un paquete necesario para poder usar el tipo **IActionResult** y algunos otros tipos los cuales sirven para los **códigos de error**.

Como pueden ver se esta agregando la referencia al paquete que se quiere probar y el de la dependencia para poder mockear. Tambien se agrego el de Domain porque es la entrada/salida de BusinessLogicInterface y en algun momento vamos a tener que crear objetos de esos tipos.

Para realizar pruebas sobre una clase, es necesario crear una clase de pruebas. Pasaremos a probar MovieController, entonces tenemos que crear una clase MovieControllerTest, la cual contendra todas nuestras pruebas de MovieController.

A continuacion se muestra un explorador de solucion:



Probando todos las peliculas

Creemos una prueba para probar que se este respondiendo bien en traer todas las peliculas que esten registradas en el sistema.

Para ello tendremos que escribir la clase MovieControllerTest de la siguiente manera:

```
namespace WebApi.Tests
{
  [TestClass]
  public class MovieControllerTest
  {
    [TestMethod]
    public void TestGetAllMoviesOk()
    {
    }
  }
}
```

Nuevamente estamos haciendo uso de los attributes para indicar un comportamiento especifico a la clase y al metodo. Esto nos va a permitir que abajo de esos attributes ejecutar/debuggear todas las pruebas de la clase o ejecutar/debuggear una prueba en especifico.

Avancemos una poco mas, una prueba unitaria se tiene que ver con claridad tres secciones, estas son:

- Arrange → construimos el objeto mock y se lo pasamos al sistema a probar
- Act → ejecutamos el sistema a probar

- Assert → verificamos la interaccion del **SUT** con el objeto mock

Arrange

En esta parte solo vamos a programar la parte del arrange. La parte del arrange para la prueba seria de la siguiente manera:

```
using System.Collections.Generic;
using BusinessLogicInterface;
using Domain;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;

//... CODE
[TestMethod]
public void TestGetAllOk()
{
  List<Movie> moviesToReturn = new List<Movie>()
  {
    new Movie()
    {
      Id = 1,
      Name = "Iron man 3",
      Description = "I'm Iron man",
      AgeAllowed = 16,
      Duration = 1.5
    },
    new Movie()
    {
      Id = 2,
      Name = "Iron man 2",
      Description = "I'm Iron man",
      AgeAllowed = 16,
      Duration = 1.5
    }
  };
  var mock = new Mock<IMovieLogic>(MockBehavior.Strict);
  mock.Setup(m => m.GetAll()).Returns(moviesToReturn);
  var controller = new MovieController(mock.Object);
}
```

El codigo de fondo amarillo es el codigo Arrange de la prueba. Veamos por partes que se fue haciendo.

1. Creo una lista de peliculas que usara el mock para retornar. Este retorno se usara para simular el comportamiento de la dependencia. Esta prueba lo que intenta probar es el caso de cuando en el sistema hay peliculas, pero deberian de estudiarse los casos de cuando no hay peliculas o cuando ocurre algun error en el sistema para ver como se comporta la WebApi.
2. Una vez que se tiene los objetos que retorna la dependencia, se crea el mock de la dependencia. Como IMovieLogic es la dependencia de MovieController, se crea un mock de este. Los mocks sirven para simular comportamiento de cualquier objeto, no necesariamente solo de interfaces. El parametro que se le pasa al constructor es un parametro de configuracion. El parametro .Strict hace que se tire una excepcion

cuando se llama un metodo que no fue simulado (no fue mockeado) y la otra configuracion que se tiene es la de .Loose que retorna un valor por defecto si se llama un metodo no simulado (no mockeado).

3. Una vez creado el mock se pasa a querer simular cierto comportamiento de este, el que se va a utilizar en la prueba, no es necesario simularlo todo. Si se simula todo esta mal diseñada la proba y el mock. El metodo Setup recibe por parametro una funcion lamda que le pasan una IMovieLogic lo cual nos permite ver los comportamientos de este y seleccionar el correcto. Una vez seleccionado el metodo se tiene que simular su comportamiento ante la llamada, esto es simular el retorno si lo tiene o si lanza alguna excepcion si se esta probando ese caso.

   Como ahora solo se quiere simular el retorno, se llama ese metodo .Returns y se le pasa la lista de peliculas que dijimos que deberia de retornar. Basicamente lo que estamos haciendo es hardcodeando el retorno de la dependencia, hablando mal y pronto.

4. Luego de simular el comportamiento se crea la clase que se quiere probar, en este caso es MovieController y como este tiene un constructor con parametros donde se le inyecta la dependencia ahi es donde le pasamos el mock que acabamos de crear. Pero no se lo podemos pasar simplemente con el nombre de la variable porque la variable mock es de tipo Mock<IMovieLogic> y la dependencia es IMovieLogic, entonces el metodo .Object nos pasa el body del mock que es de tipo IMovieLogic

Seguramente cuando esten codificando esta parte tengan que hacer using de las referencias que crearon al principio.

Seguramente tengan que agregar la firma del metodo a la interfaz e implementarla en la clase concreta para que no haya errores de compilacion. De igual manera podemos dejar error en la capa BusinessLogic porque nuestras pruebas no van a compilar dicho codigo, pero de igual manera no cuesta nada poner la firma del metodo.

La interfaz IMovieLogic nos deberia de quedar de la siguiente manera:

```
using System.Collections.Generic;
using Domain;
namespace BusinessLogicInterface
{
  public interface IMovieLogic
  {
    IEnumerable<Movie> GetAll();
  }
}
```

Este comportamiento indica que se retornara una lista de todas las peliculas registradas en el sistema. El tipo es IEnumerable<T> porque este tipo es de una coleccion solamente para realizar iteracion, no nos intereza agregar o borrar peliculas, para realizar esas operaciones deberiamos de usar el tipo Logic<T>.

Si tienen problemas de referencias es porque les debe de faltar la referencia al proyecto Domain. Hasta esta altura ya saben como realizar dicha accion.

Act

Pasemos a ver solamente el codigo de act.

```
using System.Collections.Generic;
using BusinessLogicInterface;
using Domain;
using Microsoft.AspNetCore.Mvc;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;

//... CODE

[TestMethod]
public void TestGetAllOk()
{
  List<Movie> moviesToReturn = new List<Movie>()
  {
    new Movie()
    {
      Id = 1,
      Name = "Iron man 3",
      Description = "I'm Iron man",
      AgeAllowed = 16,
      Duration = 1.5
    },
    new Movie()
    {
      Id = 2,
      Name = "Iron man 2",
      Description = "I'm Iron man",
      AgeAllowed = 16,
      Duration = 1.5
    }
  };
  var mock = new Mock<IMovieLogic>(MockBehavior.Strict);
  mock.SetUp(m => m.GetAll()).Returns(moviesToReturn);
  var controller = new MovieController(mock.Object);

  var result = controller.Get();
  var okResult = result as OkObjectResult;
  var movies = okResult.Value as IEnumerable<Movie>;
}
```

La parte de act es la que se encuentra con fondo amarillo. Veamos por partes el codigo:

1. Primero se llamo el metodo a probar y se guardo el resultado en una variable.
2. Luego se casteo el resultado al tipo del objeto que retorna el cual es un OkObjectResult. Se casteo este a este tipo porque la respuesta contiene un body el cual es la lista de peliculas, pero si la respuesta solamente retornara el codigo de error se deberia de usar OkResult porque este es el tipo para indicar el codigo de error sin body.
3. Luego que se tiene una instancia del tipo correcto se paso a castear el valor del body al que se espera, el cual es una lista de peliculas. Mas adelante vamos a ver de cambiar este body por modelos lo cual era una buena practica mencionada en la guia de WebApi

Arrange

Pasemos a ver solamente el codigo de arrange.

```
using System.Collections.Generic;
using System.Linq;
using BusinessLogicInterface;
using Domain;
using Microsoft.AspNetCore.Mvc;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;

//... CODE

[TestMethod]
public void TestGetAllOk()
{
  List<Movie> moviesToReturn = new List<Movie>()
  {
    new Movie()
    {
      Id = 1,
      Name = "Iron man 3",
      Description = "I'm Iron man",
      AgeAllowed = 16,
      Duration = 1.5
    },
    new Movie()
    {
      Id = 2,
      Name = "Iron man 2",
      Description = "I'm Iron man",
      AgeAllowed = 16,
      Duration = 1.5
    }
  };
  var mock = new Mock<IMovieLogic>(MockBehavior.Strict);
  mock.SetUp(m => m.GetAll()).Returns(moviesToReturn);
  var controller = new MovieController(mock.Object);

  var result = controller.GetAll();
  var okResult = result as OkObjectResult;
  var movies = okResult.Value as IEnumerable<Movie>;

  mock.VerifyAll();
```

```
  Assert.IsTrue(moviesToReturn.SequenceEqual(movies));
}
```

La parte de arrange es la que se encuentra con fondo amarillo. Veamos por partes el codigo:

1. Primero se verifico que se realizaron todas las llamadas que se dijeron que se iban a realizar. Es por eso que usamos el Mock.VerifyAll(). Este metodo revisa que todas las funciones mockeadas fueron llamadas.
2. Luego se tiene el assert el cual esta comparando la lista de peliculas esperada con la obtenida. Para realizar esta comparacion es necesario redefinir el equals en la clase Movie.

```
public class Movie
{
  //...MOVIE CODE

  public override bool Equals(object obj)
  {
    var result = false;

    if(obj is Movie movie)
    {
      result = this.Id == movie.Id && this.Name.Equals(movie.Name);
    }

    return result;
  }
}
```

Esta es una forma limpia de hacer override de Equals. Lo que se esta haciendo es preguntar si el tipo de obj es Movie, en caso de que sea se lo asigna a una variable de tipo Movie, la cual es movie. A partir de ahi puedo usar obj de forma como Movie entonces puedo acceder a sus diferentes estados para despues compararlo con this. En caso de que obj no fuese de tipo Movie, directamente no entra al if entonces retornaria false indicando que no son iguales.

Veamos como queda nuestro codigo en MovieController:

```
[ApiController]
[Route("api/movies")]
public class MovieController : ControllerBase
{
  private readonly IMovieLogic moviesLogic;

  public MovieController(IMovieLogic moviesLogic)
  {
    this.moviesLogic = moviesLogic;
  }

  [HttpGet]
  public IActionResult Get()
  {
```

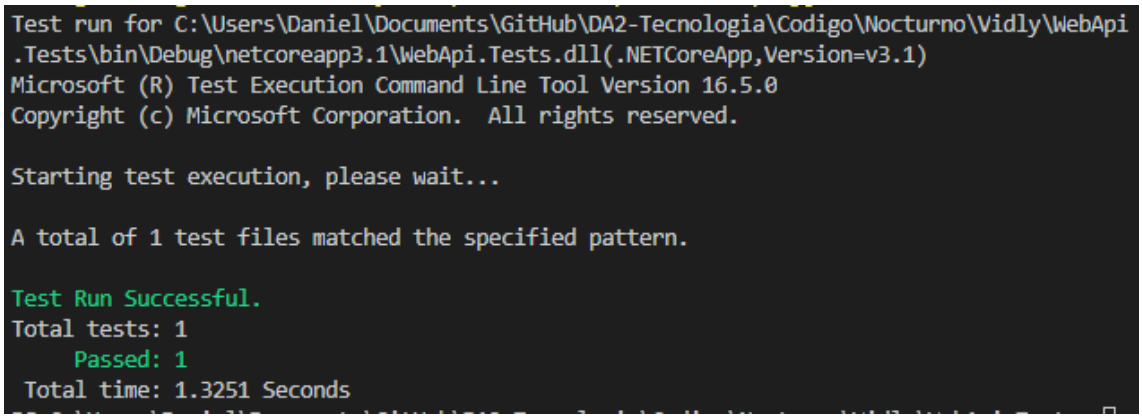
```
    return Ok(this.moviesLogic.GetAll());
  }

  //... REST WEB_API CODE
}
```

Una vez codificada esta prueba y nuestro codigo, para probarla corremos el comando:

> *dotnet test*

Esto correra todas las pruebas que se encuentren dentro de este directorio. Si tenemos mas clases de pruebas que contienen mas pruebas, se correran absolutamente todas y en caso de que haya algun error se notifica del mismo.

La salida esperada para esta prueba seria la siguiente:



Vemos que la prueba fue un exito 😎.

Refactor

Ahora pasemos hacer un refactor del codigo. Como se menciono mas arriba y en la guia de WebApi, esta capa no deberia de entender lo que son las entidades del negocio si no que deberia de manejar modelos, DTOs, etc (a partir de ahora se referiran a estos objetos como modelos).

Ahora la gran pregunta es, donde ubicamos estos modelos? y la respuesta es depende. Depende de lo que uno quiera lograr es donde deberian de estar ubicados.

- Si queremos respetar SRP, ser extensibles, reusables, remplazables, bajo acoplamimento, respetar information hiding, entre otros principios, lo mas correcto seria realizar una capa Model (un proyecto) donde WebApi dependenria de esta nueva capa y se encargue de transformar las entidades de salida de la BusinessLogicInterface en modelos de Model, sucederia lo mismo con las altas, de modelos de Model pasar a entidades de negocio del Domain, que son tipos que entiende la BusinessLogic.

- Otra opcion es en un paquete interno a WebApi que se llamaria WebApi.Model que se encontraria dentro del proyecto WebApi. Esta solucion es menos flexible porque si se quiere usar otra tecnologia que no fuese WebApi como puede ser WebPage o WinForm perderiamos los modelos y los tendriamos que crear de nuevo. Otra desventaja es que el proyecto WebApi ya esta haciendo muchas responsabilidades. Su resonsabilidad mayor es procesar las request, agregnado este paquete aca estamos sumandole la responsabilidad de que sepa como transformar modelos a entidades y viceversa. De igual manera sigue siendo una opcion valida.

Si se les ocurre alguna otra idea de donde ubicar los modelos, podemos discutir sos pros y cons en clase.

En esta guia lo vamos a realizar en una capa separada a WebApi. Para ello hay que crear un proyecto classlib como cualquier otro y llamarlo Model.

> *dotnet new classlib -n Model*

Una vez que se creo el proyecto pasemos agegarlo a la solucion y a referenciarlo en WebApi.

> *dotnet sln add Model*
> *dotnet add WebApi reference Model*

Una vez realizado esto pasemos a crear nuestro primer modelo de lectura.

```
namespace Model.Out
{
  public class MovieBasicInfoModel
  {
    public int Id { get; set; }
    public string Name { get; set; }
    public string Image { get; set; }
    public int Rank { get; set; }
  }
}
```

Este modelo, es un modelo de lectura, es una buena practica tener diferentes modelos para diferentes acciones. La idea es que no quede un mapeo de uno a uno con las entidades del negocio porque si no, no lograriamos el desacoplamiento de los clientes a la estructura de las entidades. Simplemente estariamos agregando una capa mas de complejidad con cero beneficios.

Este modelo como se dijo, se utilizara solamente para lectura, seria el body de la response de la WebApi. Como pueden ver la informacion que se mostrara a los clientes no es la misma a la que se encuentra en la entidad de negocio, Movie, si no que es mas

acotada. Este modelo lo vamos a ver cuando queramos ver un listado de todas las peliculas, si lo pensamos en una web o una app, lo que veriamos de las peliculas es la imagen de cartelera, el nombre y como un extra el rank, el id lo mandamos para que la api entienda de que pelicula la app quiere realizar diferentes operaciones.

De igual manera falta algo, donde se hacer la conversion, en este caso, de entidad de negocio a modelo? Nuevamente depende de lo que se quiera lograr. Algunas de las opciones que se pueden manejar son:

- En otra capa que se maneje esta conversion. Con esto lograriamos:
  - OCP
  - SRP
  - Estensibilidad
  - Desacoplamiento entre WebApi y Domain
  - Entre otras
- En la capa de modelo. Con esto lograriamos:
  - Desacoplamiento entre WebApi y Domain
  - Patron experto de GRASP
- En la capa Domain como un inner package
- En la logica de negocio que se encargue de ralizar esta conversion.

> 💻 Como desafio los invito a pensar cual seria el mejor lugar para realizar la conversion teniendo en cuenta buenas practicas.

Para limitar la complejidad lo haremos dentro de los modelos. Mas adelante en otra guia se hara un refactor de este codigo, pero lo normal seria que lo vayan haciendo ustedes a medida que van avanzando con las pruebas.

Nuestro modelo quedaria de la siguiente manera:

```
using Domain;

public class MovieBasicInfoModel
{
  public int Id { get; private set; }
  public string Name { get; private set; }
  public string Image { get; private set; }
  public int Rank { get; private set; }

  pulic MovieBasicInfoModel(Movie movie)
  {
    this.Id = movie.Id;
    this.Name = movie.Name;
    this.Image = movie.Image;
```

```
    this.Rank = movie.Rank;
  }
}
```

Bastante sensillo la conversion de entidad a modelo. Es un buen lugar ya que el experto poblar el estado de MovieBasicInfoModel es el mismo.

Seguramente al realizar este modelo impacto en Movie porque no tenian el estado Image ni Rank.

Veamos como quedaria WebApi:

De pronto si estuvieron muy atentos, notaron que el metodo set de los property pasaron de ser public a private. Felicitaciones aquellas personas que lo notaron 🎉, pero ahora se deben de estar preguntando, porque ese cambio? Ese cambio fue para aplicar el patron Information Hiding, no queremos exponer comportamiento de mas para aquellos sistemas que lo usen. Como la conversion se esta haciendo en el constructor y no por fuera, no tiene sentido exponer el comportamiento set de las properties a externos, si tiene sentido que el get si sea accesible porque es el comportamiento que utiliza ASP.NET Core para deserializar el objeto y pasarlo a un json que es lo que entienden los diferentes clientes 😱 😎.

```
using System.Collections.Generic;
using Model.Out;
using System.Linq;

// CODE
[HttpGet]
public IActionResult Get()
{
  return Ok(this.moviesLogic.GetAll().Select(m => new MovieBasicInfoModel(m));
}
```

El codigo nuevo aparece en amarillo. Lo que se esta haciendo es aplicar una funcion a todas las peliculas que retorna la BusinessLogicInterface. Se esta realizando parte de la conversion aca en WebApi. Lo podrian ver como que se usa la funcion map de javascript pero en C# y con un nombre distinto, pero la idea es la misma.

Al realizar esto si ejecutamos nuestra prueba nos deberia de aparecer la siguiente salida:

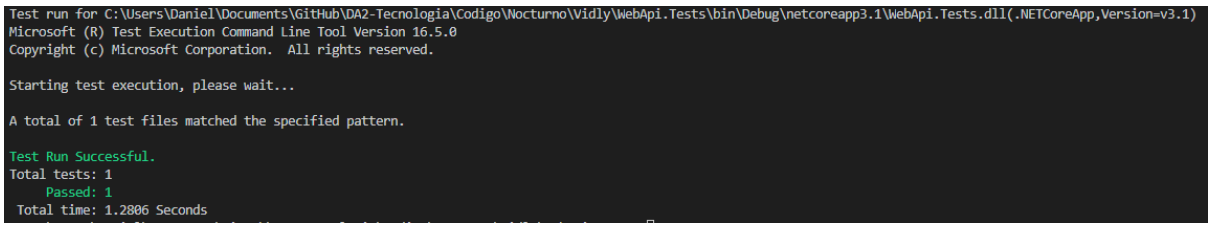
Como se puede ver, falla la prueba. El origen del fallo es porque el resultado de la conversion del body de la response a IEnumerale<Movie> es null porque ese body ahora es de tipo IEnumerable<MovieBasicInfoModel>.

Para arrelgar este error hay que tener la prueba de la siguiente manera:

```
using Model.out;

//... SOME CODE
[TestMethod]
public void TestGetAllMoviesOk()
{
  //REST TEST CODE
  var movies = okResult.Value as IEnumerable<MovieBasicInfoModel>;

  mock.VerifyAll();
  Assert.IsTrue(moviesToReturn.Select(m => new MovieBasicInfoModel(m)).SequenceEqual(movies));
}
```

Para que esto compila necesitamos agregar la referencia a Model y pasar el equals que se encontraba en Movie a MovieBasicInfoModel. Tambien se tiene que convertir las peliculas que se esperan que se retorne a modelos para poder comparar el resultado correctamente.

Ahora si corremos nuestra prueba nos deberia de salir el siguiente output:



Como era de esperarse la prueba es exitosa 😎.

DataAccess

Ahora pasaremos a probar DataAccess. Nuevamente vamos a tener que crear un proyecto de prueba pero con el nombre DataAccess.Tests.

> *dotnet new mstest -n DataAccess.Tests*
> *dotnet sln add DataAccess.Tests*
> *cd DataAccess.Tests*
> *dotnet add package Microsoft.EntityFrameworkCore*
> *dotnet add package Microsoft.EntityFrameworkCore.InMemory*
> *dotnet add reference ../DataAccess*
> *dotnet add reference ../Domain*

Antes de continuar analisemos que tipo de pruebas queremos hacer.

La dependencia de nuestros repositorios es de DbContext, una clase que no esta a nuestra alcance. Pero muchas veces cuando pensamos probar esta capa queremos ver un impacto en una base de datos de prueba o en una base de datos en memoria.

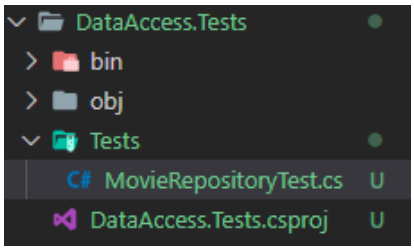
Ese tipo de pruebas que visualizan un impacto en algo externo son pruebas de integracion, porque deberiamos de crear una instancia del contexto, configurarle el proveedor de base de datos y pasarselo a nuestro repositorio, en ningun momento mockeamos nada ahi, nunca se hizo un mock de una dependencia.

Ahora si solamente queremos probar el codigo que esta dentro del repositorio que es el que esta a nuestro alcance, ahi estamos haciendo pruebas unitarias y se necesita hacer un mock de DbContext para simular el comportamiento de este. Estariamos simulando que DbContext realizo una query a la base ya sea para traer o agregar un nuevo elemento.

Dicho esto vamos a ver igualmente los dos tipos de pruebas como serian.

Por ahora creamos el proyecto, lo agregamos a la solucion y le agregamos la referencia de proyectos que va a necesitar. Si estuvieron atentos falta agregar la referencia a Moq, esto lo haremos mas adelante cuando hagamos pruebas unitarias.

Creemos una clase que se llama MovieRepositoryTest y veamos el explorador de solucion:



Veamos tambien DataAccess.Tests.csproj:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>

    <IsPackable>false</IsPackable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="3.1.8" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.InMemory" Version="3.1.8" />
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.5.0" />
    <PackageReference Include="MSTest.TestAdapter" Version="2.1.0" />
    <PackageReference Include="MSTest.TestFramework" Version="2.1.0" />
    <PackageReference Include="coverlet.collector" Version="1.2.0" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\DataAccess\DataAccess.csproj" />
    <ProjectReference Include="..\Domain\Domain.csproj" />
  </ItemGroup>

</Project>
```

Seguramente se esten preguntando que es la libreria Microsoft.EntityFrameworkCore.InMemory. Esta libreria es la que se utiliza para usar un proveedor de base de datos en memoria. Usaremos este proveedor de base de datos para las pruebas. Ustedes si quieren pueden instalarle el de SQLServer que es Microsoft.EntityFrameworkCore.SqlServer. Acuerdense de usar una base de datos especialmente para las pruebas, para no estar impactando sobre datos reales.

Nuestra clase MovieRepositoryTest quedaria asi inicialmente:

```
namespace DataAccess.Tests
{
  [TestClass]
  public class MovieRepositoryTest
  {
    [TestMethod]
    public void TestGetAllMoviesOk()
    {
    }
  }
}
```

Veamos el Arrange de la prueba TestGetAllMoviesOk, que prueba traer todas las peliculas del sistema.

Prueba de integracion (sin mock)

Arrange

Veamos solamente el codigo de Arrange.

```
using System.Linq;
using Domain;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;

/*...SOME_CODE */

[TestMethod]
public void TestGetAllMoviesOk()
{
  List<Movie> moviesToReturn = new List<Movie>()
  {
      new Movie()
      {
          Id = 1,
          Name = "Iron man 3",
          Description = "I'm Iron man",
          AgeAllowed = 16,
          Duration = 1.5
      },
      new Movie()
      {
          Id = 2,
          Name = "Iron man 2",
          Description = "I'm Iron man",
          AgeAllowed = 16,
          Duration = 1.5
      }
  };
  var options = new DbContextOptionsBuilder<VidlyContext>()
                    .UseInMemoryDatabase(databaseName: "VidlyDb").Options;
  var context = new VidlyContext(options);
  moviesToReturn.ForEach(m => context.Add(m));
  context.SaveChanges();
  var repository = new MovieRepository(context);
}
```

Analicemos este parte del codigo:

1. Primero creamos la lista de peliculas que se van a obtener desde la base de datos en memoria.
2. Luego se tiene que crear la configuracion de nuestro proveedor de base de datos para nuestro contexto.
3. Creamos nuestro contexto y le pasamos la configuracion recien creada.
4. Una vez que tenemos nuestra configuracion necesitamos agregarle las peliculas para que luego las retorne el contexto.
5. Una vez que se agregaron las peliculas hay que guardar esos cambios.
6. Una vez que tenemos el contexto se lo tenemos que pasar a nuestro repositorio que es la dependencia que necesita y es lo que queremos probar.

Act

Veamos solamente el codigo de Act.

```
[TestMethod]
public void TestGetAllMoviesOk()
{
  List<Movie> moviesToReturn = new List<Movie>()
  {
      new Movie()
      {
          Id = 1,
          Name = "Iron man 3",
          Description = "I'm Iron man",
          AgeAllowed = 16,
          Duration = 1.5
      },
      new Movie()
      {
          Id = 2,
          Name = "Iron man 2",
          Description = "I'm Iron man",
          AgeAllowed = 16,
          Duration = 1.5
      }
  };
  var options = new DbContextOptionsBuilder<VidlyContext>()
                  .UseInMemoryDatabase(databaseName: "VidlyDb").Options;
  var context = new VidlyContext(options);
  moviesToReturn.ForEach(m => context.Add(m));
  context.SaveChanges();
  var repository = new MovieRepository(context);

  var result = repository.GetAll();
}
```

La parte de act no tiene mucha ciencia, es igual al resto.

Assert

Veamos solamente el codigo de Assert.

```
[TestMethod]
public void TestGetAllMoviesOk()
{
  List<Movie> moviesToReturn = new List<Movie>()
  {
      new Movie()
      {
          Id = 1,
          Name = "Iron man 3",
          Description = "I'm Iron man",
          AgeAllowed = 16,
          Duration = 1.5
      },
      new Movie()
      {
          Id = 2,
          Name = "Iron man 2",
          Description = "I'm Iron man",
          AgeAllowed = 16,
```

```
            Duration = 1.5
        }
    };
    var options = new DbContextOptionsBuilder<VidlyContext>()
                      .UseInMemoryDatabase(databaseName: "VidlyDb").Options;
    var context = new VidlyContext(options);
    moviesToReturn.ForEach(m => context.Add(m));
    var repository = new MovieRepository(context);

    var result = repository.GetAll();

    Assert.IsTrue(moviesToReturn.SequenceEqual(result));
}
```

La parte de assert tampoco tiene mucha magia. Para que esta parte funcione hay que redefinir el equals en la clase movie. Anteriormente lo habiamos hecho para la prueba de WebApi pero lo migramos para MovieBasicInfoModel, ahora lo tendremos que hacer nuevamente. Quedándonos asi:

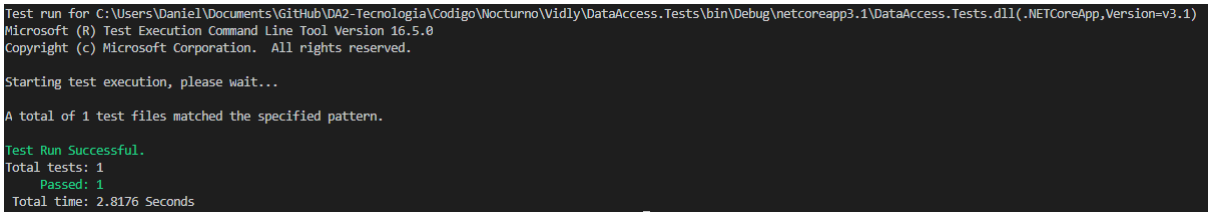
```
public class Movie
{
    //...SOME CODE
    public override bool Equal(Object obj)
    {
        var result = false;

        if(result is Movie movie)
        {
            result = this.Id == movie.Id && this.Name.Equals(movie.Name);
        }

        return result;
    }
}
```

Veamos como codificar nuestro codigo, el cual se esta probando.

```
public class MovieRepository : IMovieRepository
{
    //...SOME CODE
    public IEnumerable<Movie> GetAll()
    {
        return this.movies;
    }
}
```

Podemos realizar el retorno de esta forma porque si entramos a la metadata de DbSet<T> nos vamos a encontrar con que implementa varias interfaces una de ellas es IEnumerable<T> que es el tipo de retorno de nuestro metodo. Basicamente se esta aplicando el principio de Liskov ya que se espera el padre pero yo paso una clase hija que la trato como el padre.

Si corremos nuestra prueba en este paquete, tendriamos la siguiente salida:



Prueba unitaria (con mock)

Ahora veamos como podemos realizar una prueba unitaria al mismo codigo. Para esto vamos a necesitar una referencia a Moq en nuestro proyecto DataAccess.Tests.

> *cd DataAccess.Tests*
> *dotnet add package Moq*

Ahora que tenemos la referencia, pasemos a crear la misma prueba pero con mock.

Assert

```
/*...SOME CODE*/
[TestMethod]
public void TestGetAllMoviesMockOk()
{
  List<Movie> moviesToReturn = new List<Movie>()
  {
      new Movie()
      {
          Id = 1,
          Name = "Iron man 3",
          Description = "I'm Iron man",
          AgeAllowed = 16,
          Duration = 1.5
      },
      new Movie()
      {
          Id = 2,
          Name = "Iron man 2",
          Description = "I'm Iron man",
          AgeAllowed = 16,
          Duration = 1.5
      }
  };
  var mockSet = new VidlyDbSet<Movie>();
  var mockDbContext = new Mock<DbContext>(MockBehavior.Strict);
  mockDbContext.Setup(d => d.Set<Movie>())
  .Returns(mockSet.GetMockDbSet(moviesToReturn).Object);
  var repository = new MovieRepository(mockDbContext.Object);
}
```

Expliquemos el siguiente codigo:

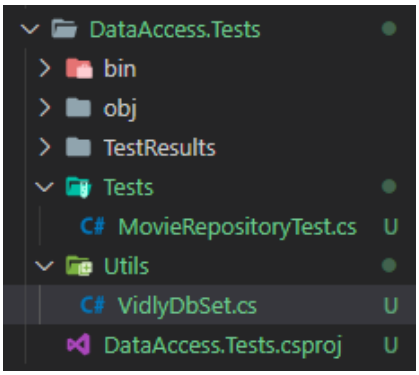
1. El primer paso es igual que siempre, se esta creando una lista de peliculas que supuestamente estan en la base de datos y es la que se espera que se retorne.

2. Como nuestra clase MovieRepository tiene una dependencia a DbContext y este hace uso del metodo Set<T> que retorna un DbSet<T>, estamos dependiendo de DbSet<T> tambien. El problema es que esta clase es una clase abstracta que implementa un monton de interfaces de coleccion. Para poder realizar un mock de DbSet<T> se tiene que crear la clase VidlyDbSet<T> que tiene un unico metodo internal para obtener el mock. El codigo de esta clase es el siguient:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Moq;

namespace DataAccess.Tests.Utils
{
    public class VidlyDbSet<T> where T : class
    {
        internal  Mock<DbSet<T>> GetMockDbSet(ICollection<T> entities)
        {
            var mockSet = new Mock<DbSet<T>>();
            mockSet.As<IQueryable<T>>().Setup(m => m.Provider)
                   .Returns(entities.AsQueryable().Provider);
            mockSet.As<IQueryable<T>>().Setup(m => m.Expression)
                   .Returns(entities.AsQueryable().Expression);
            mockSet.As<IQueryable<T>>().Setup(m => m.ElementType)
                   .Returns(entities.AsQueryable().ElementType);
            mockSet.As<IQueryable<T>>().Setup(m => m.GetEnumerator())
                   .Returns(entities.AsQueryable().GetEnumerator());
            mockSet.Setup(m => m.Add(It.IsAny<T>())).Callback<T>(entities.Add);

            return mockSet;
        }
    }
}
```

Basicamente lo que se esta haciendo en este codigo es pasar de una ICollection<T> a un Mock<DbSet<T>>, que es lo que necesitamos.

Como se puede observar esta en otro paquete porque es solamente una clase de ayuda. Silo vemos en el file system seria:



Cabe destacar que se esta creando un Mock<DbSet<T>>(), esto indica que su comportamiento es el Default el cual es el Loose. Esto indica que si se intento simular

algun metodo y no se llamo no va a lanzar una excepcion. Esta informacion la podemos visualizar en la metadata de mock.

```
// Summary:
//     Provides a mock implementation of T.
//
// Type parameters:
//   T:
//     Type to mock, which can be an interface, a class, or a delegate.
//
// Remarks:
//     Any interface type can be used for mocking, but for classes, only abstract and
//     virtual members can be mocked.
//     The behavior of the mock with regards to the setups and the actual calls is determined
//     by the optional Moq.MockBehavior that can be passed to theMoq.Mock`1.#ctor(Moq.MockBehavior)
//     constructor.
public class Mock<T> : Mock, IMock<T> where T : class
{
    //
    // Summary:
    //     Initializes an instance of the mock with Moq.MockBehavior.Default behavior.
    public Mock();
```

Y podemos ver los diferentes tipos de comportamiento del mock:

```
// Summary:
//     Options to customize the behavior of the mock.
public enum MockBehavior
{
    //
    // Summary:
    //     Causes the mock to always throw an exception for invocations that don't have
    //     a corresponding setup.
    Strict = 0,
    //
    // Summary:
    //     Will never throw exceptions, returning default values when necessary (null for
    //     reference types, zero for value types or empty enumerables and arrays).
    Loose = 1,
    //
    // Summary:
    //     Default mock behavior, which equals Moq.MockBehavior.Loose.
    Default = 1
}
```

3. Ahora se esta creando el mock para DbContext el cual es la dependencia directa de MovieRepository porque el espera que se lo inyecten en el constructor.

4. Una vez creado el mock de DbContext se tiene que configurar el comportamiento. Como en el constructor de MovieRepository el unico comportamiento que usa es el Set<T> entonces ese es el que hay que simular haciendo uso del metodo Setup.

5. La parte del Returns del Setup es crucial, ahi estamos viendo como el mock de DbContext configura el comportamiento Set<Movie> de tal forma que cuando se llame

tenga que retornar el mock de DbSet que se creo con la clase extra.

6. Una vez que se tiene configurado ahora se tiene que crear la clase a probar, MovieRepository, e inyectarle las dependencias que se necesita, en este caso es el mock de DbContext. Lo que estamos haciendo es crear una instancia de MovieRepository y configurarlo para que pueda funcionar.

Act

```
/*...SOME CODE*/
[TestMethod]
public void TestGetAllMoviesMockOk()
{
  List<Movie> moviesToReturn = new List<Movie>()
  {
      new Movie()
      {
          Id = 1,
          Name = "Iron man 3",
          Description = "I'm Iron man",
          AgeAllowed = 16,
          Duration = 1.5
      },
      new Movie()
      {
          Id = 2,
          Name = "Iron man 2",
          Description = "I'm Iron man",
          AgeAllowed = 16,
          Duration = 1.5
      }
  };
  var mockSet = new VidlyDbSet<Movie>();
  var mockDbContext = new Mock<DbContext>(MockBehavior.Strict);
  mockDbContext.Setup(d => d.Set<Movie>())
  .Returns(mockSet.GetMockDbSet(moviesToReturn).Object);
  var repository = new MovieRepository(mockDbContext.Object);

  var result = repository.GetAll();
}
```

Todo muy normal hasta ahora 🙂.

Assert

```
/*...SOME CODE*/
[TestMethod]
public void TestGetAllMoviesMockOk()
{
  List<Movie> moviesToReturn = new List<Movie>()
  {
      new Movie()
      {
          Id = 1,
          Name = "Iron man 3",
```

```
            Description = "I'm Iron man",
            AgeAllowed = 16,
            Duration = 1.5
        },
        new Movie()
        {
            Id = 2,
            Name = "Iron man 2",
            Description = "I'm Iron man",
            AgeAllowed = 16,
            Duration = 1.5
        }
    };
    var mockSet = new VidlyDbSet<Movie>();
    var mockDbContext = new Mock<DbContext>(MockBehavior.Strict);
    mockDbContext.Setup(d => d.Set<Movie>())
    .Returns(mockSet.GetMockDbSet(moviesToReturn).Object);
    var repository = new MovieRepository(mockDbContext.Object);

    var result = repository.GetAll();

    Assert.IsTrue(moviesToReturn.SequenceEqual(result));
}
```

Nada nuevo en esta seccion 😄.

Estas son las dos formas diferentes de probar DataAccess, con pruebas de integracion y pruebas unitarias.