

Unit Testen Repository in ASP.NET Core 3.1

Start project:

<https://github.com/CSharpSyntraWest/ASPNETCoreUnitTest>

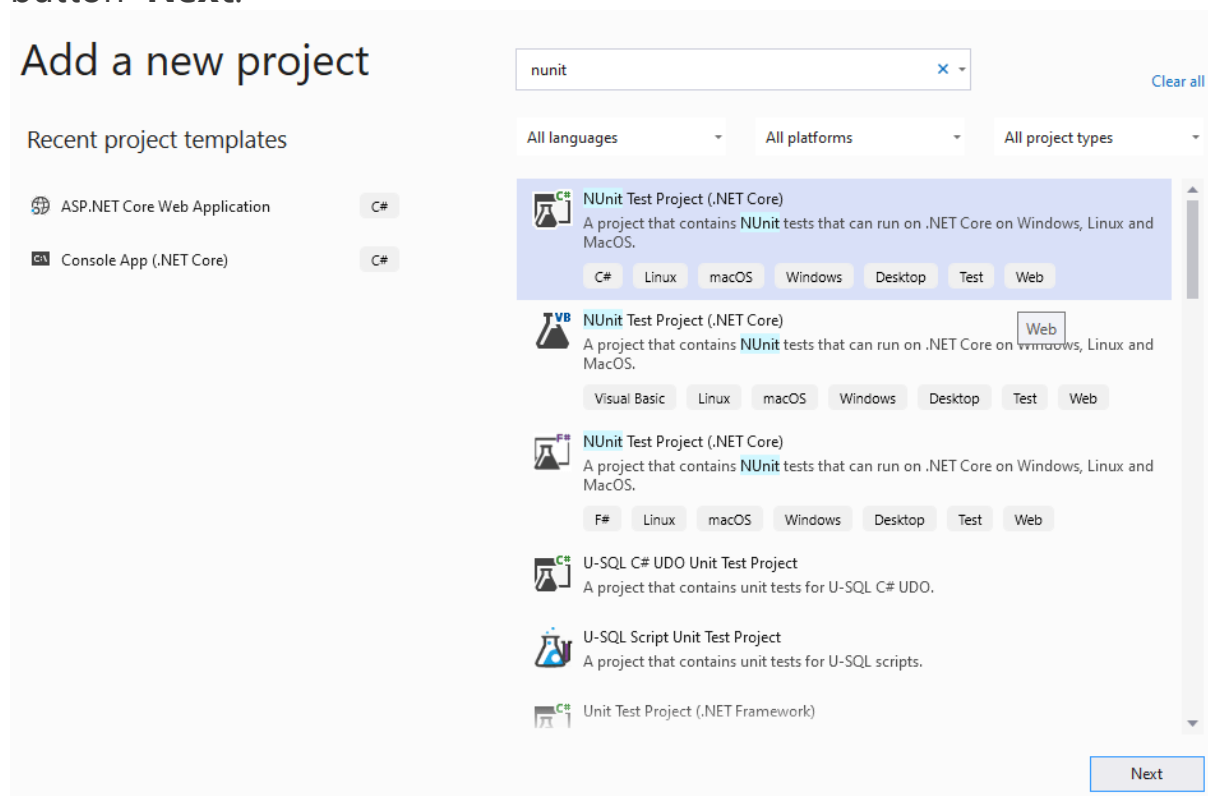
Dit document bevat een beschrijving hoe je NUnit test project toevoegt voor het testen van **ASP.NET Core 3.1** projecten, hoe je unit testen schrijft voor Repository classes en hoe je objecten kan mocken

Creëer een NUnit test project met Moq

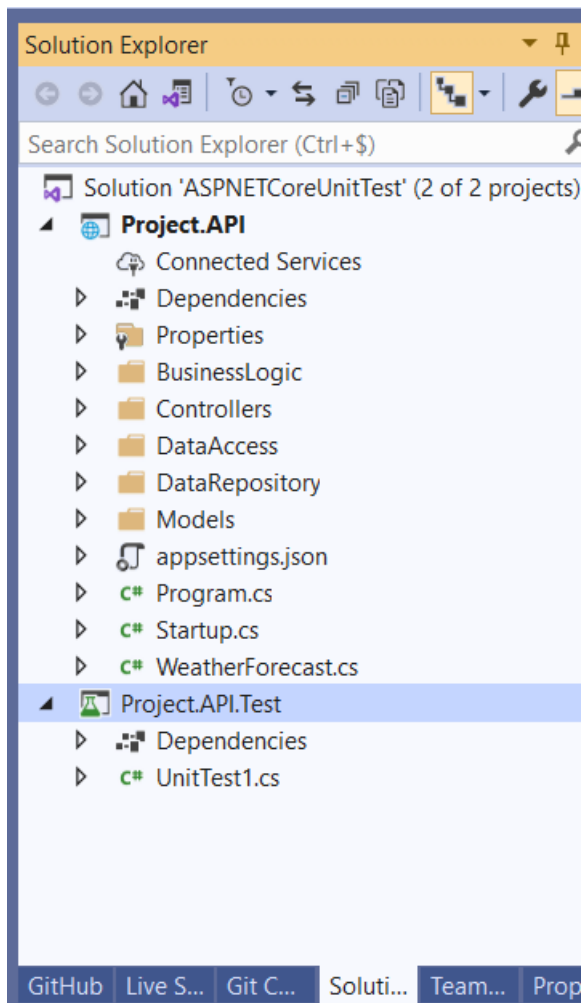
Step 1: Rechts-klik op de project solution en selecteer de optie

Add -> New Project

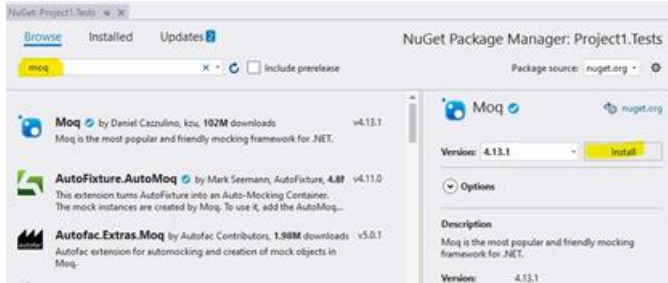
Step 2: Selecteer **NUnit test project (.NET Core)** en klik op de button **Next**:



Stap 3: Vul de naam van het project in met ***Project.API.Test*** en klik dan op de button **Create**. Je krijgt het volgende te zien:



Stap 4: Open **NuGet** manager voor **Project.API.Test** en zoek naar de package **Moq**. Selecteer **Moq** en installeer de laatste stabiele versie in het test project:



Stap 5: Controleer onder Dependencies/Packages dat de Moq NuGet package is geïnstalleerd

Maak je eerste unit test case:

We schrijven een test case voor een methode die de som van 2 gehele getallen teruggeeft.

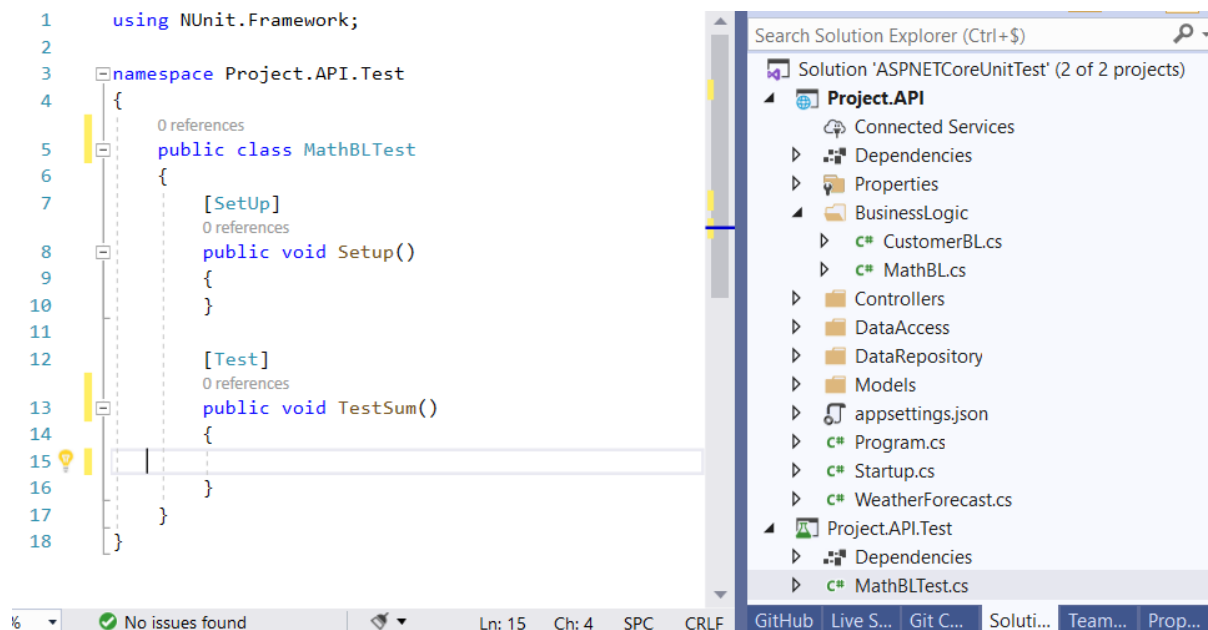
Stap 1: Er is reeds een class file met de naam **MathBL.cs** in de folder BusinessLogic. Deze heeft een methode **Sum()** in **Project.API**.

```
namespace Project.API.BusinessLogic
{
    0 references
    public class MathBL
    {
        0 references
        public int Sum(int num1, int num2)
        {
            int sum = num1 + num2;
            return sum;
        }
    }
}
```

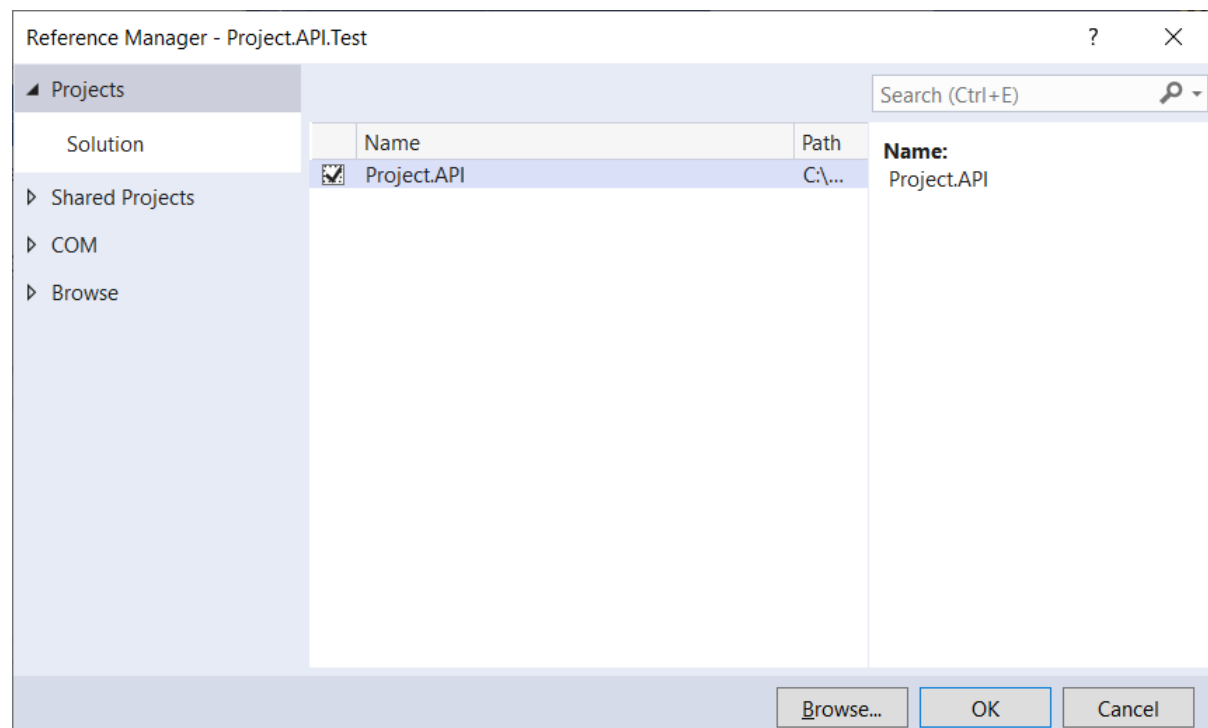
We maken een Test class en test methode aan voor MathBL:

Stap 2: Selecteer het **Project.API.Test** project en voeg de class file **MathBLTest.cs** toe:

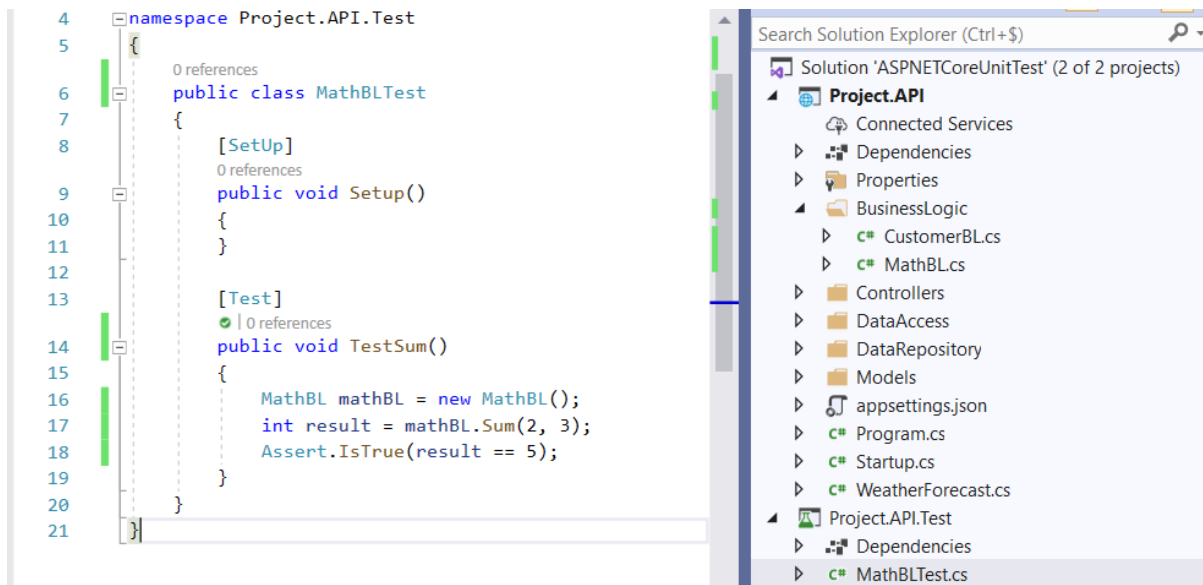
Stap 3: Voeg de methoden **Setup()** en **TestSum()** toe aan de MathBLTest.cs class:



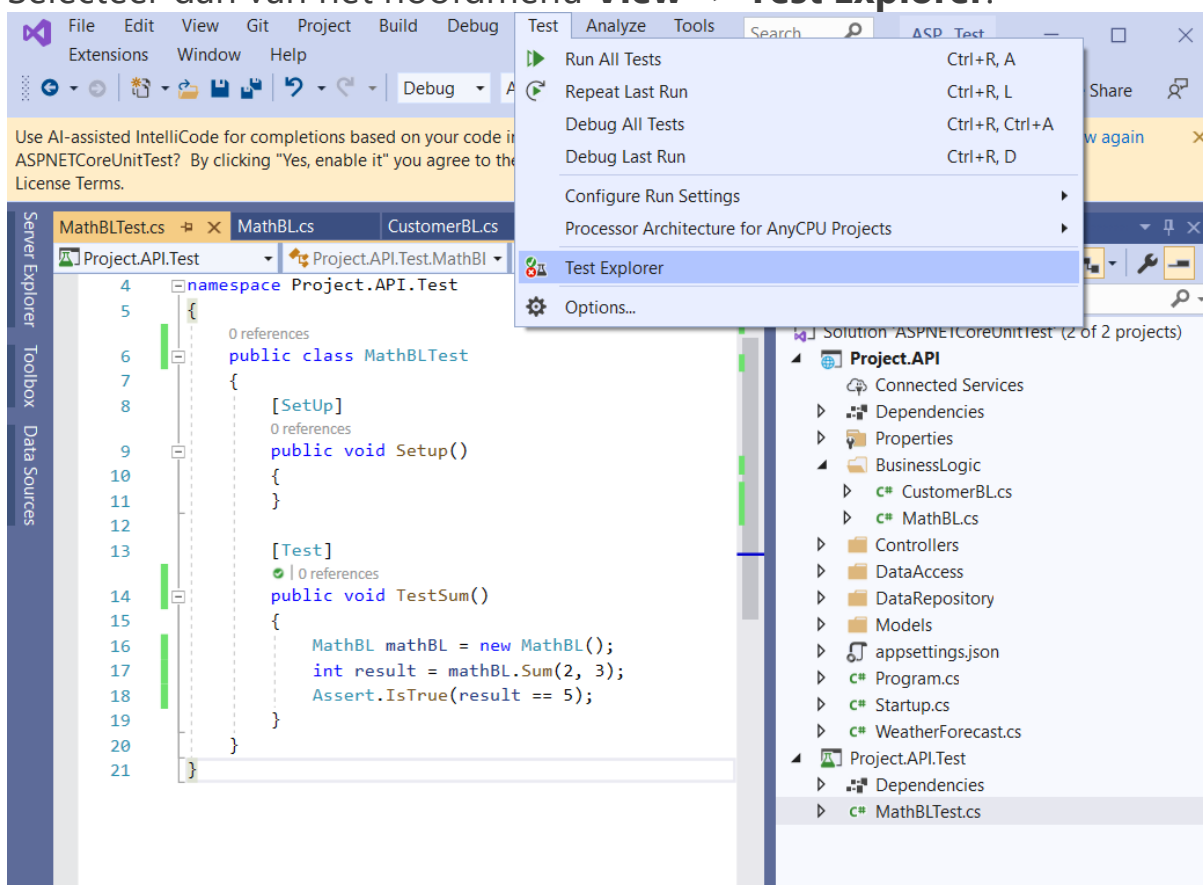
Stap 4: Rechtsklik op de **Project.API.Test** dependencies en voeg een project referentie toe naar **Project.API**:



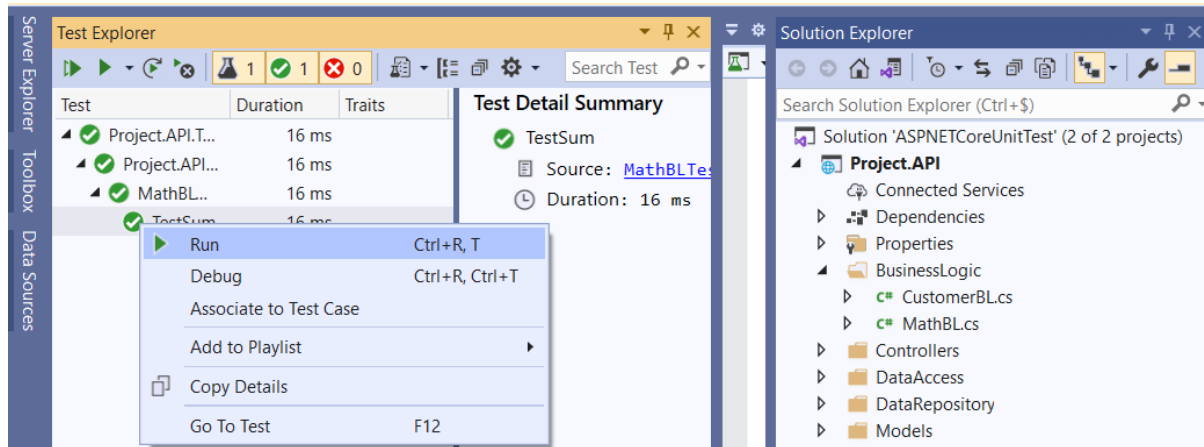
Stap 5: schrijf de volgende code om de **Sum()** methode aan te roepen :



Stap 6: Build het project en controleer of de build succesvol is. Selecteer dan van het hoofdmenu **View -> Test Explorer**.



Stap 7: Je krijgt de lijst van Tests te zien. Rechtsklik op **MathTest.BL** en klik op **Run**:



Als de test groen kleurt, is de test succesvol verlopen.

Unit test case voor mock database en data access service aanroep

Implementeer business logica en database context

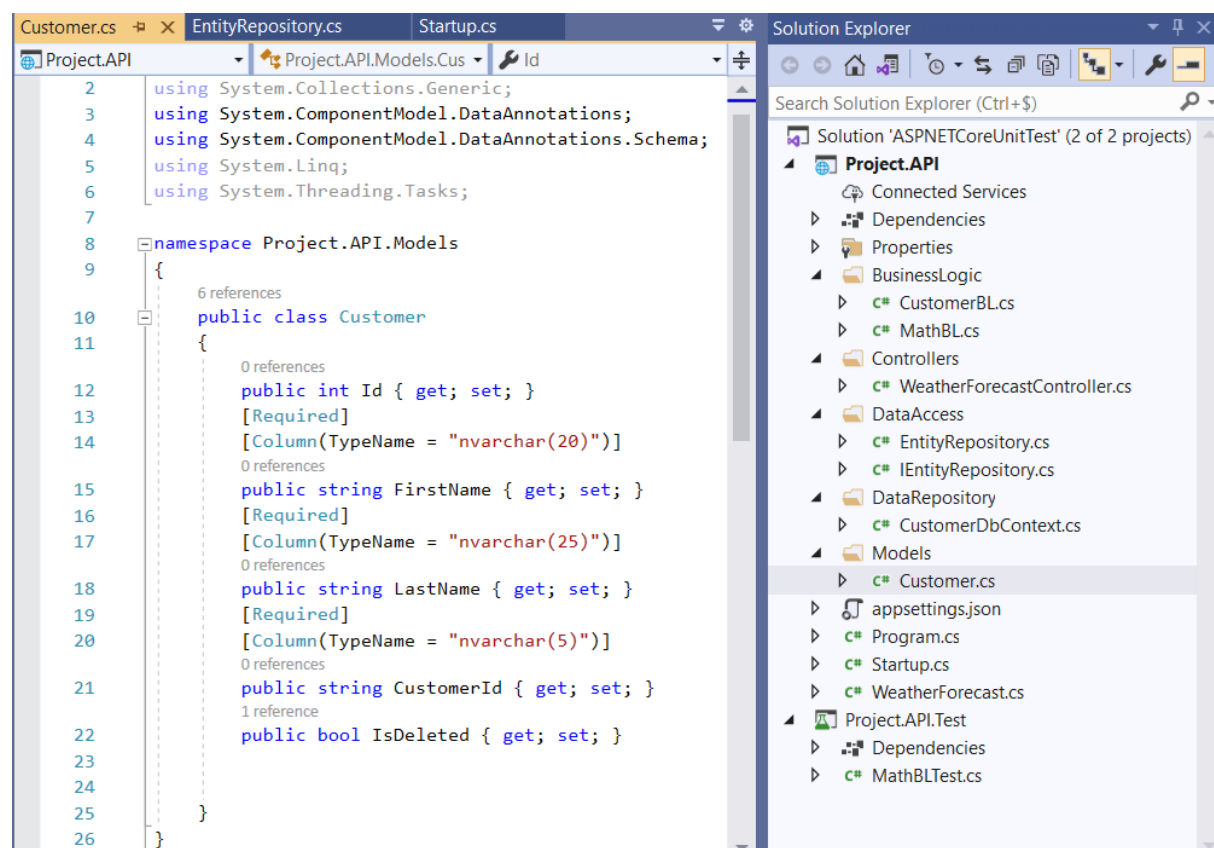
Neem het volgende project met code-first aanpak.

Genereer eerst de database via de volgende instructie in de Package Manager Console:

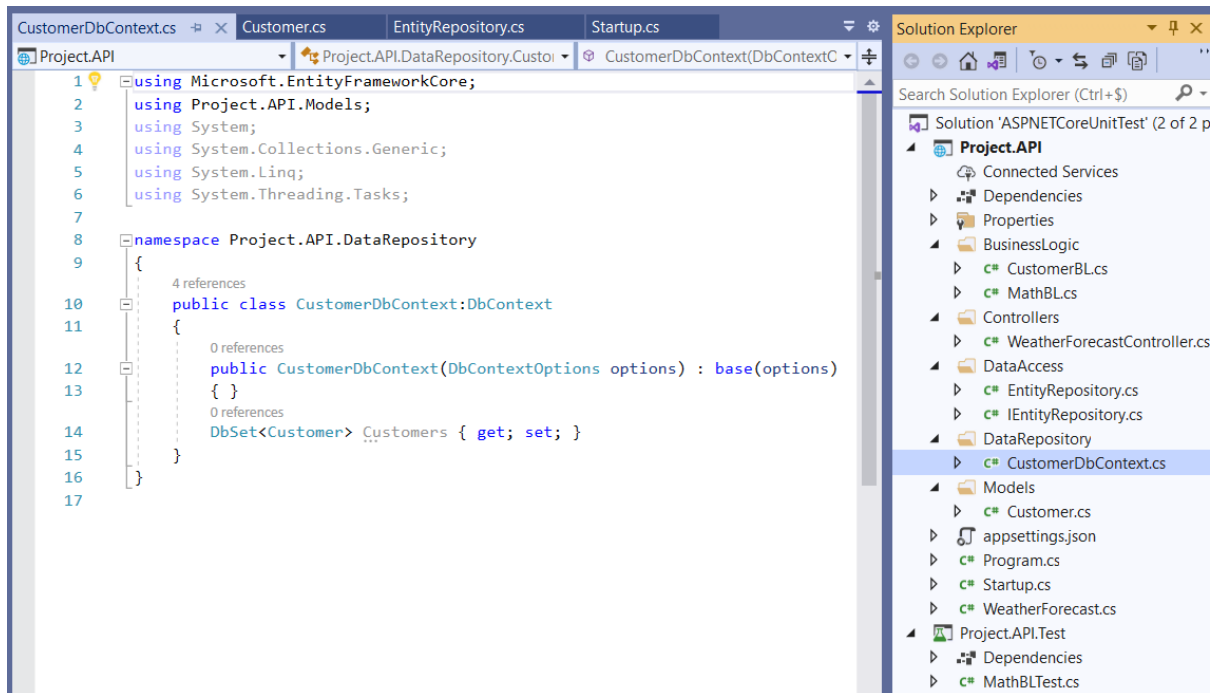
```
PM> update-database
```

We bekijken eerst de code in het te testen project:

Modal class Customer:



database context class CustomerDbContext:



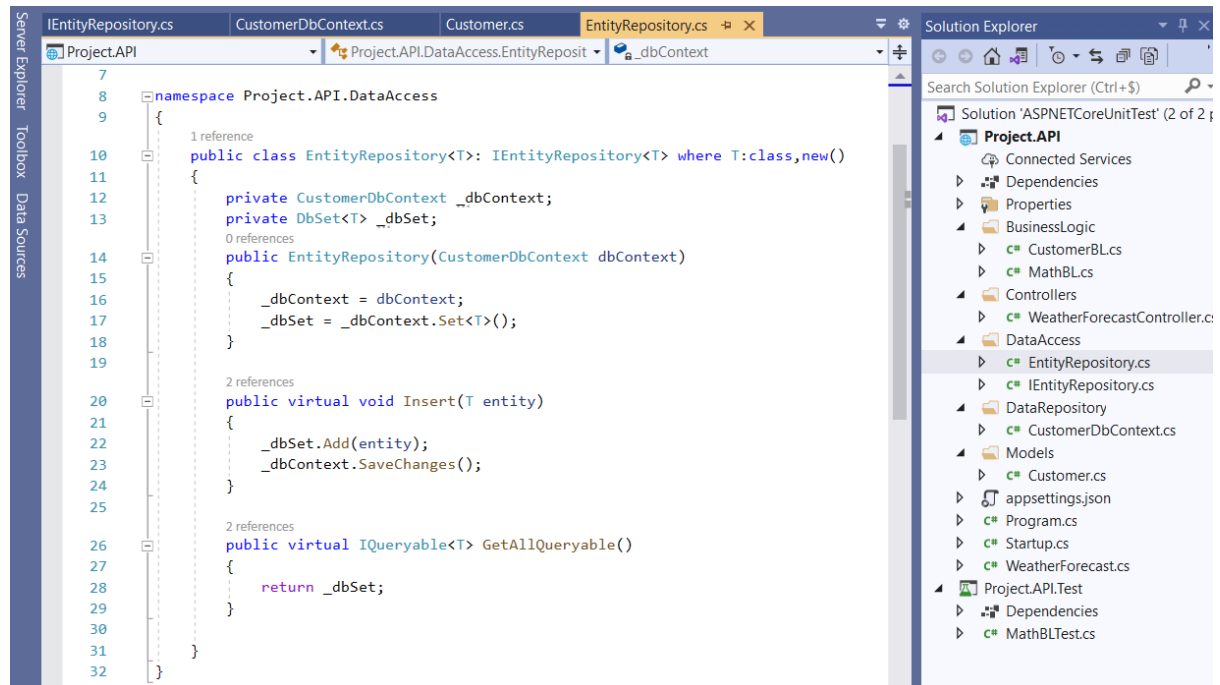
The screenshot displays the Visual Studio IDE with the `CustomerDbContext.cs` file open. The code defines a `DbContext` class within the `Project.API.DataRepository` namespace. The class has a constructor that takes `DbContextOptions` and a `DbSet<Customer>` property named `Customers`.

```
1 using Microsoft.EntityFrameworkCore;
2 using Project.API.Models;
3 using System;
4 using System.Collections.Generic;
5 using System.Linq;
6 using System.Threading.Tasks;
7
8 namespace Project.API.DataRepository
9 {
10     public class CustomerDbContext : DbContext
11     {
12         public CustomerDbContext(DbContextOptions options) : base(options)
13         { }
14         DbSet<Customer> Customers { get; set; }
15     }
16 }
17
```

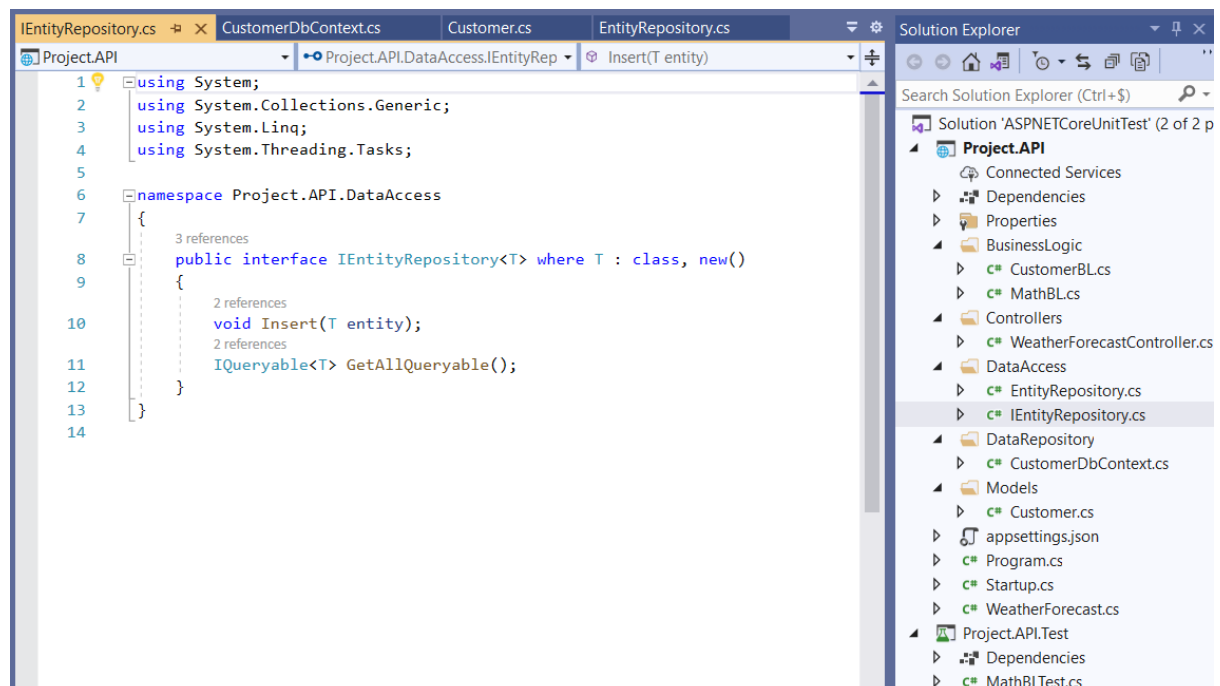
The Solution Explorer on the right shows the project structure for `Project.API`, including folders for `Connected Services`, `Dependencies`, `Properties`, `BusinessLogic`, `Controllers`, `DataAccess`, `DataRepository`, `Models`, and `Project.API.Test`. The `CustomerDbContext.cs` file is highlighted under the `DataRepository` folder.

Data access logica

De **CustomerDbContext** class wordt aangesproken via een Repository class **EntityRepository** die de business logica bevat:



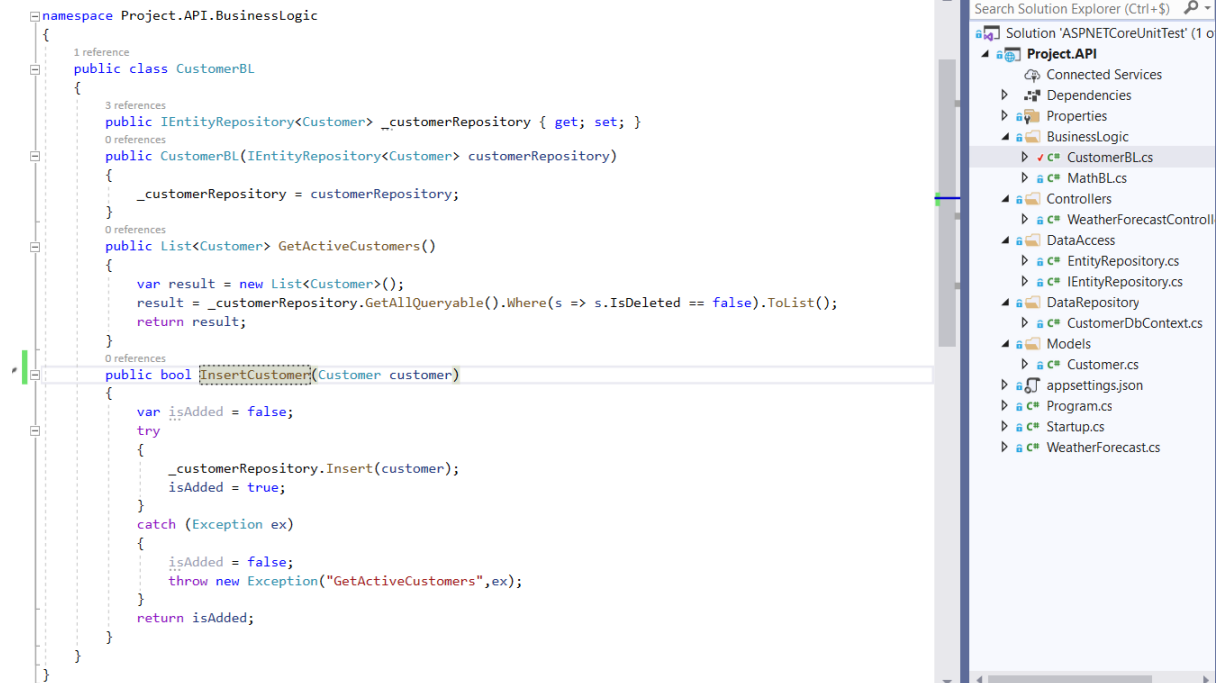
Deze implementeert een interface **IEntityRepository**:



Business logica

De class **CustomerBL** bevat de volgende methoden:

- ***GetActiveCustomer()***
- ***SaveCustomer()***



CustomerBL.cs

Unit test voor class met business logica

Maak een class **CustomerBLTest** in het NUnit test project en implementeer de test methode **GetActiveCustomersTest()**

```
using Moq;
using Project.API.DataAccess;
using Project.API.Models;
using System;
using System.Collections.Generic;
using System.Text;
using NUnit.Framework;
using System.Linq;
using Project.API.BusinessLogic;

namespace Project.API.Tests.BusinessLogicTests
{
    public class CustomerTestBL
    {
        private Mock<IEntityRepository<Customer>> _customerRepository;
        private List<Customer> _customers;
        [SetUp]
        public void Setup()
        {
            //Set up de mock
            _customerRepository = new Mock<IEntityRepository<Customer>>();
            _customers = new List<Customer>();
            _customers.Add(new Customer() { Id = 1, FirstName = "Cus1", CustomerId = "C001", IsDeleted = false });
            _customers.Add(new Customer() { Id = 2, FirstName = "Cus2", CustomerId = "C002", IsDeleted = true });
            _customers.Add(new Customer() { Id = 3, FirstName = "Cus3", CustomerId = "C003", IsDeleted = false });
        }

        [Test]
        public void GetActiveRecordsTest()
        {
            //Act
            _customerRepository.Setup(a =>
a.GetAllQueryable()).Returns(_customers.AsQueryable());
            //Arrange
            var customerBL = new CustomerBL(_customerRepository.Object);
            var customerList = customerBL.GetActiveCustomers();
            //Assert
            Assert.IsTrue(customerList.Count == 2);
            Assert.IsTrue(customerList.All(s => s.IsDeleted == false));
        }
    }
}
```

*Arrange/Act/Assert wordt het **AAA** of 3A design pattern genoemd*

Uitleg over Mock en test asserts

1. **Arrange**: De nodige voorbereidingen (initialisaties en mock training gebeuren voor de dependency-objecten): Een Mock instantie `customerRepository` wordt aangemaakt voor **`IEntityRepository<Customer>`** in de Setup method en statische test data wordt gedefinieerd voor `customers List`.

In de `GetActiveRecordsTest` method wordt de Mock wordt getraind dmv van de Setup methode van de Mock `customerRepository`. Wanneer de methode `GetAllQueryable()` wordt aangeroepen, zal de statische test data gedefinieerd voor `customers` worden teruggegeven door de Mock.

2. **Act**: De te testen methode wordt hier effectief aangeroepen. Hier is dit **`GetActiveCustomers();`** deze wordt aangeroepen en het resultaat wordt in de variabele `customerList` gezet.
3. **Assert**: via Assert statements worden de verwachte resultaten vergeleken met de teruggegeven waarden van de te testen methode. In dit voorbeeld is het te verwachten resultaat dat er 2 elementen in de `customerList` staan en enkel Customer elementen waarvan `IsDeleted==false` worden teruggegeven door de methode `GetActiveCustomers()`.

```

public class CustomerTestBL
{
    private Mock<IEntityRepository<Customer>> _customerRepository;
    private List<Customer> _customers;
    [SetUp]
    0 references
    public void Setup()
    {
        //Set up de mock
        _customerRepository = new Mock<IEntityRepository<Customer>>();
        _customers = new List<Customer>();
        _customers.Add(new Customer() { Id = 1, FirstName = "Cus1", CustomerId = "C001", IsDeleted = false });
        _customers.Add(new Customer() { Id = 2, FirstName = "Cus2", CustomerId = "C002", IsDeleted = true });
        _customers.Add(new Customer() { Id = 3, FirstName = "Cus3", CustomerId = "C003", IsDeleted = false });
    }

    [Test]
    0 references
    public void GetActiveRecordsTest()
    {
        //Arrange
        _customerRepository.Setup(a => a.GetAllQueryable()).Returns(_customers.AsQueryable());
        //Act
        var customerBL = new CustomerBL(_customerRepository.Object);
        var customerList = customerBL.GetActiveCustomers();
        //Assert
        Assert.IsTrue(customerList.Count == 2);
        Assert.IsTrue(customerList.All(s => s.IsDeleted == false));
    }
}

```

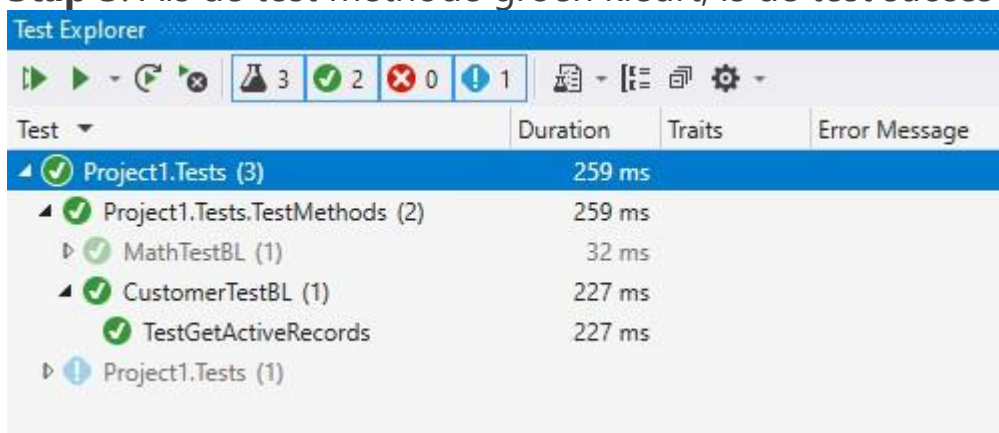
Run test

Stap 1: Selecteer **View -> Test Explorer**. Deze geeft de lijst van test case methoden.

Stap 2: Rechtsklik op de test methode **TestGetActiveCustomer()** en klik op **Run**.



Stap 3: Als de test methode groen kleurt, is de test succesvol.



Stap 4: Voeg nog een 2de test toe om de InsertCustomer methode van de CustomerBL class te testen. Hier maken we een interactie-test, we gaan namelijk verifiëren of de methode Insert van de mocked repository exact 1 maal is aangeroepen:

```
[Test]
public void InsertTest()
{
    //Arrange
    Customer testCustomer = new Customer() { Id = 4, FirstName = "Jos",
CustomerId = "C004", IsDeleted = false };
    //Act
    var customerBL = new CustomerBL(_customerRepository.Object);
    var customerList = customerBL.InsertCustomer(testCustomer);
    //Assert
    _customerRepository.Verify(c => c.Insert(testCustomer), Times.Once);
}
```

Run deze test en controleer of deze groen kleurt.

Referenties

<https://www.syncfusion.com/blogs/post/how-to-integrate-unit-testing-with-asp-net-core-3-1.aspx>

[Moq voorbeeld](#)