### What is Unit Testing?

Unit testing is a software testing methodology performed by developers to check the functionality of an individual code block or a set of programs. The main aim of unit testing is to identify and solve errors at the developers’ end.

Unit testing can reduce the time testers take to verify the functionality, compatibility, and security of .NET web applications. In addition, it will also help the developers to push streamlined and well-working code to the final product.

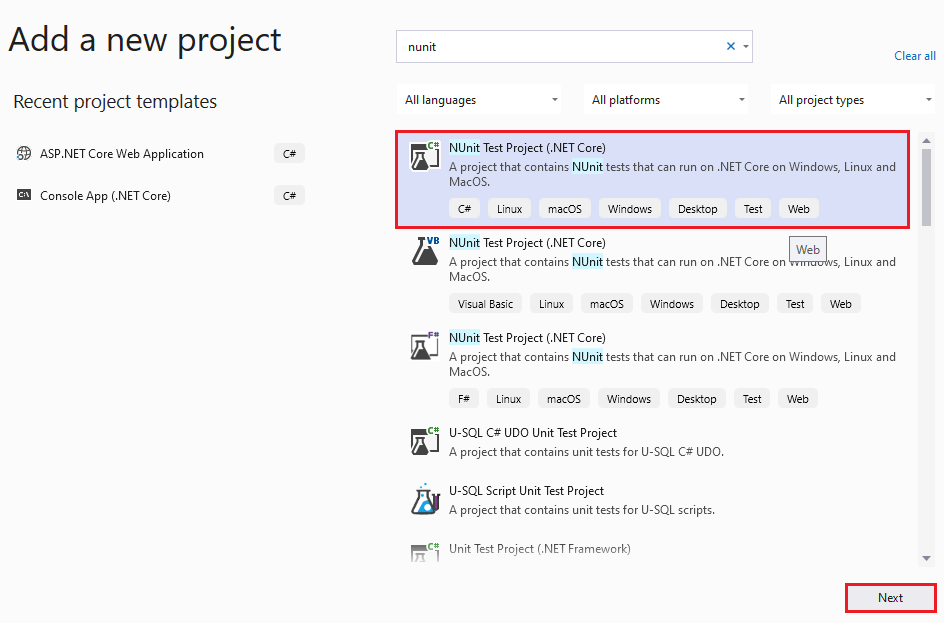
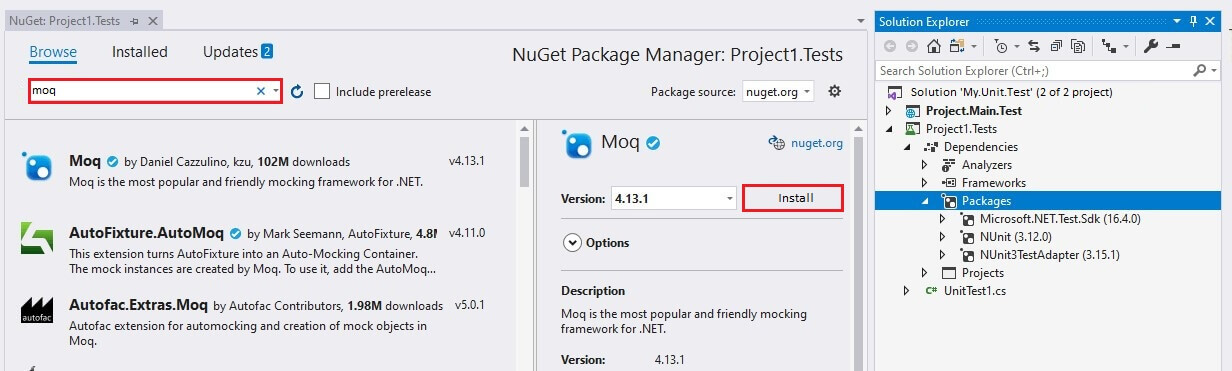
Unit tests are the smallest sort of tests that you’ll write. Ideally, they exercise a single method and should be trivial to write. In fact, many people suggest that if you find unit tests difficult to write then it is an indication that the code being tested is doing too much and should be split up. unit tests are an indicator that methods should be split up into multiple methods or even split into separate classes. These are the sorts of tests you should create during test-driven development.

### How to Use NUnit for Unit Testing in ASP.NET Core:

In .NET development, developers widely use NUnit, one of the most popular frameworks, for unit testing multiple programs or an individual code block. its primary functioning is to test the .NET web applications and the .NET Web API project.

If you are a .NET developer, then you must know this framework and how to integrate and use it with ASP.NET Core applications. It will help you offer the best-in-class dotnet development services to clients.

**Stages:**

1. Creation of Unit Test Project:
2. open the NuGet package manager, and find the Moq. Match the result with the following snippet and integrate the correct package:
3. Creation of Unit Test Case:

Go to Solutions Explorer and under the tested project add a MathBL.cs file containing the Sum() method as below:

1. navigate to TestProject to add MathTestBL.cs class file

internal class MathBLTest

{

[Test]

public void TestSum()

{

MathBL bl = new MathBL();

int result = bl.Sum(2, 3);

Assert.IsTrue(result == 5);

}

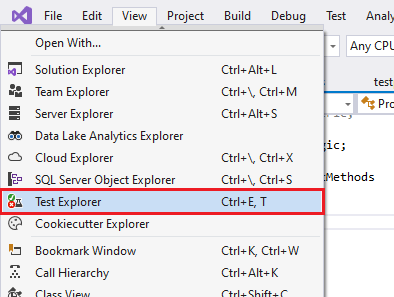
[SetUp]

public void SetUp()

{

}

1. Go to View and choose Test Explorer under it. It will help you build the project and check the code.



1. The Test Explorer will display all the methods. You have to use right-click on ***MathTestBL*** and choose the ***Run*** option. The following snippet demonstrates that the IDE will test the class file and provide results.
2. Testing the Data Access Service Call and Mock Database:

* Create product model with id,name description and price
* Create interface and repository for products:

public class ProductsRepository : IProductsRepository

{

private AppDbContext \_context;

public ProductsRepository(AppDbContext context)

{

\_context = context;

}

public async Task<Product> AddNewProduct(Product p)

{

\_context.Products.Add(s);

await \_context.SaveChangesAsync();

return p;

}

public async Task<bool> Delete(int id)

{

var productToDelete = \_context.Products.ToList().Where(p => p.Id == id).FirstOrDefault();

if (productToDelete!= null)

{

\_context.Products.Remove(productToDelete);

await \_context.SaveChangesAsync();

return true;

}

return false;

}

public Task<List<Product>> GetAllProducts()

{

return Task.FromResult(\_context.Products.ToList());

}

public Task<Product> GetProductById(int id)

{

return Task.FromResult(\_context.Products.ToList().Where(p => p.Id == id).FirstOrDefault());

}

public Task Update(Product s)

{

\_context.Products.Update(s);

\_context.SaveChangesAsync();

return Task.CompletedTask;

}

public class AppDbContext : DbContext

{

public DbSet<Product> Products { get; set; }

protected readonly DbContextOptions Configuration;

public AppDbContext(DbContextOptions configuration) : base(configuration)

{

Configuration = configuration;

}

}

public interface IProductService

{

Task<List<Product>> GetAllProducts();

Task<Product> GetProductById(int id);

Task<Product> AddNewProduct(Product p);

Task<bool> Delete(int id);

Task Update(Product p);

}

In program.cs:

builder.Services.AddDbContext<AppDbContext>(options => options.UseSqlServer(builder.Configuration.GetConnectionString("DbConn")));

builder.Services.AddScoped(typeof(IProductsRepository), typeof(ProductsRepository));

In appsetting:

"ConnectionStrings": {

"DbConn": "data source=TallyPC\\MSSQLSERVER01;initial catalog=University;trusted\_connection=true;Encrypt=False"

},

In BL Folder create ProductsBL:

public class ProductsBL

{

private readonly IProductsRepository \_repository;

public ProductsBL(IProductsRepository repository)

{

\_repository = repository;

}

public async Task<List<Product>> GetAllProducts()

{

return await \_repository.GetAllProducts();

}

}

In testing Project add productsTestBL:

public class ProductsBLTest

{

private Mock<IProductsRepository> productsRepositoryMock;

private List<Product> products;

[SetUp]

public void Setup()

{

productsRepositoryMock = new Mock<IProductsRepository>();

products = new List<Product>();

products.Add(new Product { Id = 1, Name = "Bread", Price = 30, Description = "1" });

products.Add(new Product { Id = 2, Name = "Milk", Price = 10, Description = "2" });

products.Add(new Product { Id = 3, Name = "Cream", Price = 20, Description = "3" });

products.Add(new Product { Id = 4, Name = "Tomato", Price = 40, Description = "4" });

}

[Test]

public void Test1()

{

//act

productsRepositoryMock.Setup(a => a.GetAllProducts()).Returns(Task.FromResult(products));

//arrange

var bl = new ProductsBL(productsRepositoryMock.Object);

var productsList = bl.GetAllProducts().Result;

//assert

Assert.IsTrue(productsList.Count == 4);

}

}

Test Controller:

using day4.Contracts;

using day4.Controllers;

using day4.Models;

using Microsoft.AspNetCore.Mvc;

using Moq;

namespace AppTesting;

public class ProductsControllerTest

{

private Mock<IProductsRepository> productsRepositoryMock;

private List<Product> products;

private ProductsController \_controller;

[SetUp]

public void Setup()

{

productsRepositoryMock = new Mock<IProductsRepository>();

\_controller = new ProductsController(productsRepositoryMock.Object, null);

products = new List<Product>();

products.Add(new Product { Id = 1, Name = "Bread", Price = 30, Description = "1" });

products.Add(new Product { Id = 2, Name = "Milk", Price = 10, Description = "2" });

products.Add(new Product { Id = 3, Name = "Cream", Price = 20, Description = "3" });

products.Add(new Product { Id = 4, Name = "Tomato", Price = 40, Description = "4" });

}

[Test]

public async Task TestGetAll()

{

productsRepositoryMock.Setup(a => a.GetAllProducts()).ReturnsAsync(products);

// Act

var result = await \_controller.Get();

// Assert

Assert.That(result.Result, Is.TypeOf<OkObjectResult>());

var okResult = result.Result as OkObjectResult;

Assert.That(okResult.Value, Is.EqualTo(products));

}

[Test]

public async Task GetItemAsync\_ReturnsOk\_WhenItemExists()

{

int itemId = 1;

productsRepositoryMock.Setup(a => a.GetById(itemId)).ReturnsAsync(products.FirstOrDefault(p=>p.Id == itemId));

// Act

var result = await \_controller.Get(itemId);

// Assert

Assert.That(result, Is.TypeOf<OkObjectResult>());

var okResult = result as OkObjectResult;

Assert.That(((Product)okResult.Value).Id, Is.EqualTo(itemId));

}

[Test]

public async Task GetItemAsync\_ReturnsNotFound\_WhenItemDoesNotExist()

{

int itemId = 1;

productsRepositoryMock.Setup(service => service.GetById(itemId))

.ReturnsAsync((Product)null);

// Act

var result = await \_controller.Get(itemId);

// Assert

Assert.That(result.Result, Is.TypeOf<NotFoundResult>());

}

[Test]

public async Task DeleteItemAsync\_ReturnsOk\_WhenItemExists()

{

int itemId = 1;

productsRepositoryMock.Setup(a => a.Delete(itemId)).ReturnsAsync(true);

// Act

var result = await \_controller.Delete(itemId);

// Assert

Assert.That(result, Is.TypeOf<OkResult>());

}

[Test]

public async Task DeleteItemAsync\_ReturnsNotFound\_WhenItemDoesNotExist()

{

int itemId = 1;

productsRepositoryMock.Setup(service => service.Delete(itemId))

.ReturnsAsync(false);

// Act

var result = await \_controller.Delete(itemId);

// Assert

Assert.That(result, Is.TypeOf<NotFoundResult>());

}

}

1. Avoid functional complexity

Focus on one functionality or feature at a time – keep it simple! Deliberate on it with the whole team to ensure the test case covers the desired functionality in every way possible. As the test case is the driver, it should be reviewed for correctness and completeness.

2. Focus on what you need to achieve

Ensure you understand where the code needs to be called and frame the test suite accordingly.  Ensure test cases follow standard naming conventions and depict what needs to be achieved by the end of the development process. This is crucial as functionality keeps getting added with iterations. Future developers should be able to look at the test and easily deduce the intended functionality.

3. Maintain code Strictness

Ensure your code has just enough meat to satisfy your test case. This is a basic tenet of TDD. This minimizes the chances of defects and simplifies review and testing processes. However, do ensure the code is understandable and allows future enhancements.

4. Test repeatedly

Test before coding and after coding. Then, test once again after code refactoring. This reinforces that no code is broken in any of the steps. During refactoring, ensure the new code is maintainable and adheres to the standards. The rule of thumb here is to repeat testing whenever a code change, code move, or code merger occurs.

5. Maintain code Guidelines

Use version control tools to check out and check in code. This is important, specifically when more than one developer works on the code. Using continuous integration tools like Jenkins can avoid code merger issues.

6. Application knowledge

In TDD, coding must be limited but effective in achieving its purpose without breaking anything else. Also, the new code should ideally pass the test case in the first run. Maintaining adequate system documentation, including a repository of test cases and engaging team members with good application knowledge, can ensure a smooth and successful project execution.

7. Know when to use TDD

Lastly, TDD works best in specific scenarios like any other development concept. Use TDD for developments that can be quickly tested. Any testing that is prolonged or complex defeats the purpose of TDD.