**Unit Testing (xUnit & Moq)**

**Unit Test** is a type of software testing that involves rigorous testing of small independent **units** or **components**in **isolation**. Here **unit**refers to the smallest possible component of software that can be tested. Unit testing improves code quality and helps to identify all kinds of issues at a very early stage of development.

Here we will use **xUnit.net** a free, open-source, community-focused unit testing tool for the .NET framework. There are several testing tools for the .NET framework among them xUnit.net has gained a lot of popularity.

xUnit provides a set of attributes or decorators for testing methods and using those we can write testing code to test components and units. Available attributes are,

1. **[Fact]:** If we want to method to be part of unit testing and execute it during the test run it should be decorated with this attribute.
2. **[Theory]:** If we want to send some parameters to the test method then we need to use this attribute. It will execute the method and also provides parameters to the test data.
3. **[InlineData]:**  Used along with [Theory] attribute. This attribute is a way to provide parameters to the test method.

Any method that we want to be part of test runner must be **public**and the xUnit.net **attribute** should be used on the method. Test methods **naming**is as important for teams on long-term projects as any other code convention. The names should be self-describing there are several naming conventions. But here I would like to follow this,

**MethodName\_ExpectedBehavior\_StateUnderTest**

example: isVoter\_False\_AgetLessThan18

Create ASP.NET Core Application

For implementation purposes, we will create an ASP.NET Core Web API project.

**Add interface and concrete class**

Let's add an interface "ICalculatorService" for calculator operations and a concrete class "CalculatorService" in the "Services" folder.

**ICalculatorService.cs**

public interface ICalculatorService

{

double Add(double x1, double x2);

double Subtract(double x1, double x2);

double Multiply(double x1, double x2);

double Divide(double x1, double x2);

}

**CalculatorService.cs**

public class CalculatorService : ICalculatorService

{

public double Add(double x1, double x2)

{

return (x1 + x2);

}

public double Divide(double x1, double x2)

{

if (x2 == 0)

{

throw new DivideByZeroException("x2 cannot be zero");

}

return (x1 / x2);

}

public double Multiply(double x1, double x2)

{

return (x1 \* x2);

}

public double Subtract(double x1, double x2)

{

return (x1 - x2);

}

}

Now we will register our **ICalculatorService** in the startup.cs file inside the "**ConfigureServices()**" method.

public void ConfigureServices(IServiceCollection services)

{

//remaining code

services.AddTransient<ICalculatorService, CalculatorService>();

}

Now we will create a controller named "CalculatorController".

**CalculatorController.cs**

[Route("api/[controller]")]

[ApiController]

public class CalculatorController : ControllerBase

{

private ICalculatorService \_calculatorService = null;

public CalculatorController(ICalculatorService calculatorServices)

{

\_calculatorService = calculatorServices;

}

[HttpPost]

[Route("Add")]

public double Add(double x1, double x2)

{

return \_calculatorService.Add(x1, x2);

}

[HttpPost]

[Route("Divide")]

public double Divide(double x1, double x2)

{

return \_calculatorService.Divide(x1, x2);

}

[HttpPost]

[Route("Multiply")]

public double Multiply(double x1, double x2)

{

return \_calculatorService.Multiply(x1, x2);

}

[HttpPost]

[Route("Subtract")]

public double Subtract(double x1, double x2)

{

return \_calculatorService.Subtract(x1, x2);

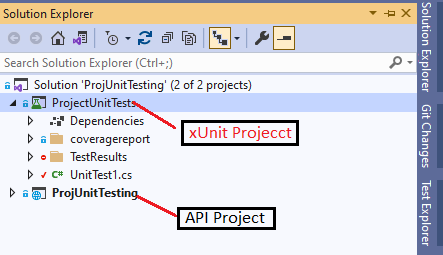
}

}

That's a simple setup for the API project now we will try to apply a unit test on this.

Create xUnit Project

Inside the same solution, let's create a xUnit Project. So now the project structure will look like this.



Inside the xUnit Project lets create a new class named 'CalculatorControllerTest.cs". And here we will write our test method.

Each test method normally consists of 3 logical parts,

1. **Arrange -** prepare the data which will be required for testing. For example, data used for testing scenarios along with the expected result.
2. **Act -** call the method which is being tested making use of the data prepared in the Arrange. Here it will return us the actual result.
3. **Assert -** Compare expected results with actual results to decide if the test passed or failed.

Keeping this in mind let's write some tests.

**CalculatorControllerTest.cs**

public class CalculatorControllerTest

{

private CalculatorService \_unitTesting = null;

public CalculatorControllerTest()

{

if(\_unitTesting == null)

{

\_unitTesting = new CalculatorService();

}

}

[Fact]

public void Add()

{

//arrange

double a = 5;

double b = 3;

double expected = 8;

//act

var actual = \_unitTesting.Add(a, b);

//Assert

Assert.Equal(expected, actual, 0);

}

}

**Explanation**

**Line 14** - Declared a method "Add()" with **[Fact]** attribute to make it part of the test runner.

**Line 25** -**Assert.Equal** is used to compare the expected results with actual results to verify if the test succeeded or not.

So like this let's add methods for other operations,

[Fact]

public void Substract()

{

//arrange

double x1 = 10;

double x2 = 8;

double expected = 2;

//act

var actual = \_unitTesting.Subtract(x1, x2);

//assert

Assert.Equal(expected, actual, 0);

}

[Theory(DisplayName = "Maths- Divided with parameters")]

[InlineData(40, 8, 5)]

public void Divide(double value1, double value2, double value3)

{

//arrange

double x1 = value1;

double x2 = value2;

double expected = value3;

//act

var actual = \_unitTesting.Divide(x1, x2);

//assert

Assert.Equal(expected, actual, 0);

}

[Fact(Skip = "Do not run now")]

public void Multiply()

{

//arrange

double x1 = 5;

double x2 = 8;

double expected = 40;

//act

var actual = \_unitTesting.Multiply(x1, x2);

//assert

Assert.Equal(expected, actual, 0);

}

[Fact(DisplayName = "Maths - Divide by Zero Exception")]

public void DivideByZeroException()

{

//arrange

double a = 100;

double b = 0;

//act

Action act = () => \_unitTesting.Divide(a, b);

//assert

Assert.Throws<DivideByZeroException>(act);

}

C#

Copy

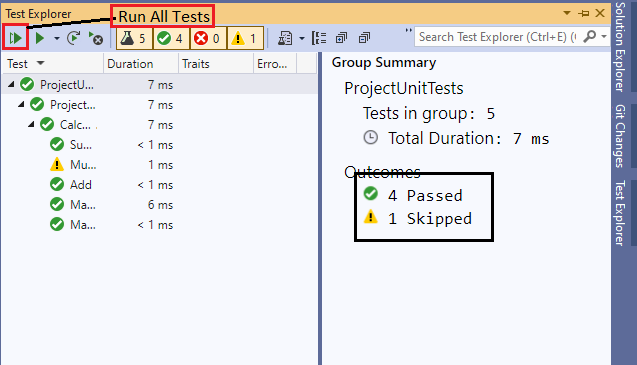
**Explanation**

**Line 17** - We created a method divide with **[Theory]** attribute as we want to pass parameters to this method and test it against the passed parameters. And with the **[InlineData]** parameter we passed the parameters to the method. We can have multiple**[InlineData]** attributes here. N.B. With **[Theory(DisplayName="<name>")]** we can change display name of this method in the test explorer.

**Line 31** - We created "Multiply()" with **[Fact(Skip="")]** attribute because we want to skip this test method from running. So the test runner will skip this method while running the tests.

*Run Tests*

Now let's go to the "Test Explorer" of the Visual Studio and run the tests.



Here we can see that 4 tests have been passed and 1 skipped as we intentionally skipped on the test.

Unit Testing with Moq

Let's go a little further by using Moq for unit testing.

Mock objects allow us to mimic the behavior of classes and interfaces, letting the code in the test interact with them like real. This isolates the code we are testing and ensures that it works on its own and no other code will make the tests fail. With mocks, we can set up objects, provide params and return values.

It's really very important for our real-life project architecture because in our project there are several layers and each layer is tightly or loosely coupled with each other. So when we plan to test any layer we need to provide the dependency for that layer. But thanks to dependency injection that ensures the loose coupling in our application and we can easily Mock the interface/class that any testing layer needed and provide the dependency in the runtime and complete our tests.

Here we will use **moq**as our Mocking library.

To implement this we will create two different .NET Core projects

1. A webapi project(ASP.NET Core Web API)
2. A Test project(xUnit)

In the web API project let's add a file "**EmployeeController.cs**" [Have attached the project here.]

[Route("api/[controller]")]

[ApiController]

public class EmployeeController : ControllerBase

{

private IGenericRepository<Employee> repo = null;

public EmployeeController(IGenericRepository<Employee> repo)

{

this.repo = repo;

}

[HttpGet]

[Route("GetEmployee")]

public ActionResult<IEnumerable<Employee>> GetEmployee()

{

var model = repo.GetAll();

return Ok(model);

}

[HttpGet("GetEmployeeById/{id}")]

public ActionResult<Employee> GetEmployeeById(long id)

{

Employee employee = repo.GetById(id);

if(employee == null)

{

return NotFound("The Employee record couldn't be found.");

}

return Ok(employee);

}

[HttpPost("CreateEmployee")]

public ActionResult<Employee> CreateEmployee(Employee employee)

{

if(employee == null)

{

return BadRequest("Employee is null");

}

repo.Insert(employee);

return CreatedAtRoute("GetEmployeeById", new { Id = employee.EmployeeId}, employee);

}

[HttpPut]

public IActionResult Put(Employee employee)

{

if(employee == null)

{

return BadRequest("Employee is null");

}

repo.Update(employee);

return NoContent();

}

public bool checkIfUserCanBeVoter(int age)

{

return (age >= 18) ? true : false;

}

}

We will also add an interface "IGenericRepository" and its concrete class "GenericRepository".

**IGenericRepository.cs**

public interface IGenericRepository<T> where T : class

{

IEnumerable<T> GetAll();

T GetById(object id);

void Insert(T obj);

void Update(T obj);

void Delete(object id);

void save();

}

**GenericRepository.cs**

public class GenericRepository<T> : IGenericRepository<T> where T : class

{

private readonly ApplicationContext \_context = null;

/// <summary>

/// as the class here is Generic so we can't access the model directly

/// so creating a DbSet depending on the T generic class

/// </summary>

private DbSet<T> table = null;

public GenericRepository(ApplicationContext context)

{

\_context = context;

table = \_context.Set<T>();

}

public void Delete(object id)

{

throw new NotImplementedException();

}

public IEnumerable<T> GetAll()

{

return table.ToList();

}

public T GetById(object id)

{

return table.Find(id);

}

public void Insert(T obj)

{

table.Add(obj);

\_context.SaveChanges();

}

public void save()

{

throw new NotImplementedException();

}

public void Update(T obj)

{

table.Attach(obj);

\_context.Entry(obj).State = EntityState.Modified;

\_context.SaveChanges();

}

}

and Employee Model class.

**Employee.cs**

public class Employee

{

[Key]

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public long EmployeeId { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public DateTime DateOfBirth { get; set; }

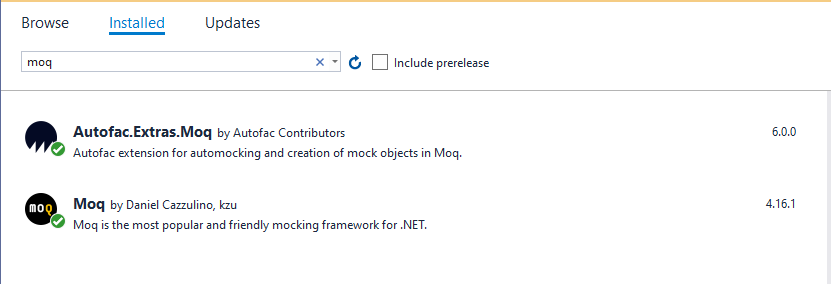
public string PhoneNumber { get; set; }

public string Email { get; set; }

}

**Creating tests on the test project**

Now let's write some tests for our EmployeeController class. For this first install moq library in our xUnit project using the Nuget packager Manager.



Add a new class in the xUnit project named "EmployeeControllerTest.cs".

public class EmployeeControllerTests

{

private readonly Mock<IGenericRepository<Employee>> service;

public EmployeeControllerTests()

{

service = new Mock<IGenericRepository<Employee>>();

}

[Fact]

//naming convention MethodName\_expectedBehavior\_StateUnderTest

public void GetEmployee\_ListOfEmployee\_EmployeeExistsInRepo()

{

//arrange

var employee = GetSampleEmployee();

service.Setup(x => x.GetAll())

.Returns(GetSampleEmployee);

var controller = new EmployeeController(service.Object);

//act

var actionResult = controller.GetEmployee();

var result = actionResult.Result as OkObjectResult;

var actual = result.Value as IEnumerable<Employee>;

//assert

Assert.IsType<OkObjectResult>(result);

Assert.Equal(GetSampleEmployee().Count(), actual.Count());

}

private List<Employee> GetSampleEmployee()

{

List<Employee> output = new List<Employee>

{

new Employee

{

FirstName = "Jhon",

LastName = "Doe",

PhoneNumber = "01682616789",

DateOfBirth = DateTime.Now,

Email = "[jhon@gmail.com](mailto:jhon@gmail.com)",

EmployeeId = 1

},

new Employee

{

FirstName = "Jhon1",

LastName = "Doe1",

PhoneNumber = "01682616787",

DateOfBirth = DateTime.Now,

Email = "[jhon@gmail.com](mailto:jhon@gmail.com)",

EmployeeId = 4

},

new Employee

{

FirstName = "Jhon2",

LastName = "Doe2",

PhoneNumber = "01682616787",

DateOfBirth = DateTime.Now,

Email = "[jhon2@gmail.com](mailto:jhon2@gmail.com)",

EmployeeId = 5

}

};

return output;

}

}

**Explanation**

**Line 6** - If you look into our EmployeeController you can see that this Controller has a dependency in the "IGenericRepository<Employee>" so in the constructor we tried to **Mock**this **IGenericRepository<Employee>** class so that we can test the controller in isolation. So we created a mocking object by "**Mock<IGenericRepository<Employee>>**" and make it global to use for testing through the test class.

**Line 10** - we are trying to test our "**GetEmployee()**" method and decorate it with the [Fact] attribute to make it a part of the test runner

**Line 13** - Created some sample Employees.

**Line 14** - Instructing a mock object that if a certain method is called then it can answer with a certain response. Herewith "**Setup()**" which method can be called and by the "**Returns()**" we define certain responses for that specific call. For example: we use "service.Setup(x => x.GetAll()).Returns(GetSampleEmployee);" we define if GetAll() method it called then provide them with the "**GetSampleEmployee()**" which basically returns a list of dummy employee.

**Line 16** - Creating an instance of the Controller class providing its a dependency to the **IGenericRepository<Employee>**.

**Line 19** - calling the GetEmployee() method.

**Line 24** - Checking whether the method returns us the **Status 200 OK.**

**Line 25** - Comparing the number of employee **count**between the actual and the expected list of employees.

Let's create some other tests for this EmployeeController,

public class EmployeeControllerTests

{

private readonly Mock<IGenericRepository<Employee>> service;

public EmployeeControllerTests()

{

service = new Mock<IGenericRepository<Employee>>();

}

[Fact]

//naming convention MethodName\_expectedBehavior\_StateUnderTest

public void GetEmployee\_ListOfEmployee\_EmployeeExistsInRepo()

{

//arrange

var employee = GetSampleEmployee();

service.Setup(x => x.GetAll())

.Returns(GetSampleEmployee);

var controller = new EmployeeController(service.Object);

//act

var actionResult = controller.GetEmployee();

var result = actionResult.Result as OkObjectResult;

var actual = result.Value as IEnumerable<Employee>;

//assert

Assert.IsType<OkObjectResult>(result);

Assert.Equal(GetSampleEmployee().Count(), actual.Count());

}

[Fact]

public void GetEmployeeById\_EmployeeObject\_EmployeewithSpecificeIdExists()

{

//arrange

var employees = GetSampleEmployee();

var firstEmployee = employees[0];

service.Setup(x => x.GetById((long)1))

.Returns(firstEmployee);

var controller = new EmployeeController(service.Object);

//act

var actionResult = controller.GetEmployeeById((long)1);

var result = actionResult.Result as OkObjectResult;

//Assert

Assert.IsType<OkObjectResult>(result);

result.Value.Should().BeEquivalentTo (firstEmployee);

}

[Fact]

public void GetEmployeeById\_shouldReturnBadRequest\_EmployeeWithIDNotExists()

{

//arrange

var employees = GetSampleEmployee();

var firstEmployee = employees[0];

service.Setup(x => x.GetById((long)1))

.Returns(firstEmployee);

var controller = new EmployeeController(service.Object);

//act

var actionResult = controller.GetEmployeeById((long)2);

//assert

var result = actionResult.Result;

Assert.IsType<NotFoundObjectResult>(result);

}

[Theory]

[InlineData(18)]

[InlineData(20)]

public void checkIfUserCanBeVoter\_true\_ageGreaterThan18(int age)

{

//arrange

var controller = new EmployeeController(null);

//act

var actual = controller.checkIfUserCanBeVoter(age);

//Assert

Assert.True(actual);

}

[Theory]

[InlineData(17)]

[InlineData(15)]

public void checkIfUserCanBeVoter\_true\_ageLessThan18(int age)

{

//arrange

var controller = new EmployeeController(null);

//act

var actual = controller.checkIfUserCanBeVoter(age);

//Assert

Assert.False(actual);

}

[Fact]

public void CreateEmployee\_CreatedStatus\_PassingEmployeeObjectToCreate()

{

var employees = GetSampleEmployee();

var newEmployee = employees[0];

var controller = new EmployeeController(service.Object);

var actionResult = controller.CreateEmployee(newEmployee);

var result = actionResult.Result;

Assert.IsType<CreatedAtRouteResult>(result);

}

private List<Employee> GetSampleEmployee()

{

List<Employee> output = new List<Employee>

{

new Employee

{

FirstName = "Jhon",

LastName = "Doe",

PhoneNumber = "01682616789",

DateOfBirth = DateTime.Now,

Email = "[jhon@gmail.com](mailto:jhon@gmail.com)",

EmployeeId = 1

},

new Employee

{

FirstName = "Jhon1",

LastName = "Doe1",

PhoneNumber = "01682616787",

DateOfBirth = DateTime.Now,

Email = "[jhon@gmail.com](mailto:jhon@gmail.com)",

EmployeeId = 4

},

new Employee

{

FirstName = "Jhon2",

LastName = "Doe2",

PhoneNumber = "01682616787",

DateOfBirth = DateTime.Now,

Email = "[jhon2@gmail.com](mailto:jhon2@gmail.com)",

EmployeeId = 5

}

};

return output;

}

}

C#

Copy

Now let's go to the "Test Explorer" and run our tests.

