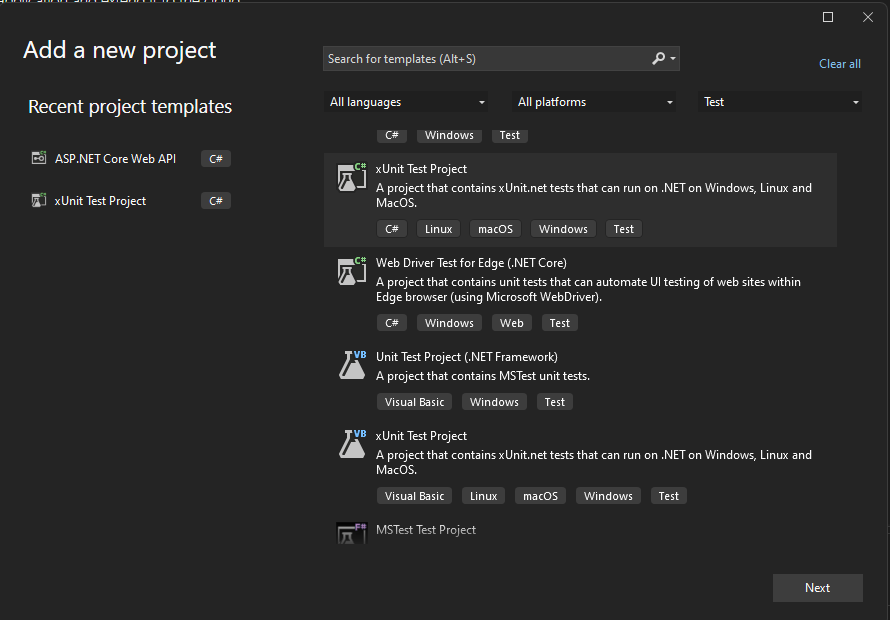
**Unit Testing**

**Prerequisite:**

* Create a test project on your solution



* Add reference with main API project.
* UserManager:

using Model.EntityModel;

namespace Manager

{

    public interface IUserManager

    {

        List<User> GetUsers();

        User GetUserById(int id);

    }

    public class UserManager : IUserManager

    {

        private List<User> \_users;

        public UserManager()

        {

            \_users = new List<User>

            {

                new User{ Id = 1, UserName = "user1", Email = "user1@mail.com" },

                new User{ Id = 2, UserName = "user2", Email = "user2@mail.com" },

                new User{ Id = 3, UserName = "user3", Email = "user3@mail.com" },

            };

        }

        public List<User> GetUsers()

        {

            return \_users;

        }

        public User GetUserById(int id)

        {

            var user = \_users.FirstOrDefault(x => x.Id == id);

            if (user is null)

            {

                throw new Exception("User not found");

            }

            return user;

        }

    }

}

In UserManager class has two method

1. GetUsers – return all users;
2. GetUserById

* Throw Exception if there is no user available for given id;
* Return specific user if user is available.
* Create a class in Test project named “UserManagerTests”
* Create test method named “GetUsers\_ShouldReturnUserList” for testing GetUsers method with attribute “[Fact]”

[Fact]

public void GetUsers\_ShouldReturnUserList()

{

    // Arrange

    var userManager = new UserManager();

    // Act

    var users = userManager.GetUsers();

    // Assert

    Assert.NotNull(users);

    Assert.NotEmpty(users);

}

* Create two methods for testing GetUserById method:

public void GetUserById\_ShouldThrowException\_IfUserNotFound()

{

    // Arrange

    var userManager = new UserManager();

    // Act

    var exception = Assert.Throws<Exception>(() => userManager.GetUserById(9));

    // Assert

    Assert.Equal("User not found", exception.Message);

}

[Fact]

public void GetUserById\_ShouldReturnUser\_IfUserFound()

{

    // Arrange

    var userManager = new UserManager();

    // Act

    var res = userManager.GetUserById(1);

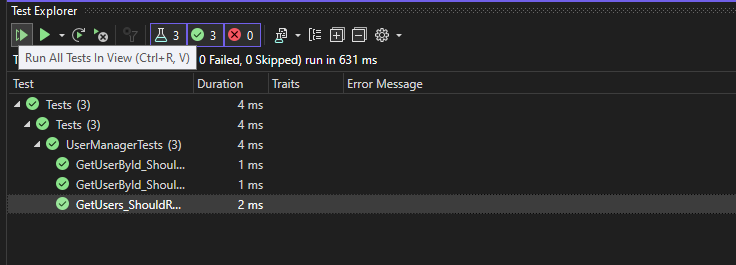
    // Assert

    Assert.NotNull(res);

    Assert.IsType<User>(res);

}

* Open Test Explorer from visual studio menu bar “Test > Test Explorer”



Click Run All Test button. If All tests passed it shows success button.

**Unit Test with MOQ**

* Add “Moq” library in Test project.
* Manager:

using Model.EntityModel;

using Repository;

namespace Manager

{

    public interface IWithRepoUserManager

    {

        List<User> GetUsers();

        User GetUserById(int id);

    }

    public class WithRepoUserManager : IWithRepoUserManager

    {

        private readonly IUserRepository \_userRepository;

        public WithRepoUserManager(IUserRepository userRepository)

        {

            \_userRepository = userRepository;

        }

        public List<User> GetUsers()

        {

            return \_userRepository.GetUsers();

        }

        public User GetUserById(int id)

        {

            var user = \_userRepository.GetUserById(id);

            if (user is null)

            {

                throw new Exception("User not found");

            }

            return user;

        }

    }

}

* WithRepoUserManagerTests:

using Manager;

using Model.EntityModel;

using Moq;

using Repository;

namespace Tests

{

    public class WithRepoUserManagerTests

    {

        private readonly WithRepoUserManager \_sut;

        private readonly Mock<IUserRepository> \_userRepositoryMock = new Mock<IUserRepository>();

        public WithRepoUserManagerTests()

        {

            \_sut = new WithRepoUserManager(\_userRepositoryMock.Object);

        }

        [Fact]

        public void GetUsers\_ShouldReturnUserList\_IfFound()

        {

            // Arrange

            var users = new List<User>

            {

                new User{ Id = 1, UserName = "user1", Email = "user1@mail.com" },

                new User{ Id = 2, UserName = "user2", Email = "user2@mail.com" },

                new User{ Id = 3, UserName = "user3", Email = "user3@mail.com" },

            };

            \_userRepositoryMock.Setup(x => x.GetUsers()).Returns(users);

            // Act

            var res = \_sut.GetUsers();

            // Assert

            Assert.NotNull(res);

            Assert.NotEmpty(res);

        }

        [Fact]

        public void GetUserById\_ShouldThrowException\_IfUserNotFound()

        {

            // Arrange

            \_userRepositoryMock.Setup(x => x.GetUserById(It.IsAny<int>())).Returns<User>(null);

            // Act

            var exception = Assert.Throws<Exception>(() => \_sut.GetUserById(9));

            // Assert

            Assert.Equal("User not found", exception.Message);

        }

        [Fact]

        public void GetUserById\_ShouldReturnUser\_IfUserFound()

        {

            // Arrange

            var user = new User { Id = 1, UserName = "user1", Email = "user1@mail.com" };

            \_userRepositoryMock.Setup(x => x.GetUserById(It.IsAny<int>())).Returns(user);

            // Act

            var res = \_sut.GetUserById(1);

            // Assert

            Assert.NotNull(res);

            Assert.IsType<User>(res);

        }

    }

}

* Create mock variable for every dependency injection.
* Initialize Manager in contractor with dependency injection.
* Mock data which comes from repository.
* Run tests.

**Integration Tests**

* Add a fixture class

using Xunit.Abstractions;

namespace Tests

{

    public class Fixture<TFixture> : IDisposable, IClassFixture<TFixture>, IAsyncDisposable

    where TFixture : class

    {

        protected readonly ITestOutputHelper \_testOutputHelper;

        protected readonly TFixture \_fixture;

        private bool \_disposedValue;

        private bool \_disposedAsync;

        public Fixture(ITestOutputHelper testOutputHelper, TFixture fixture)

            => (\_testOutputHelper, \_fixture) = (testOutputHelper, fixture);

        protected virtual void Dispose(bool disposing)

        {

            if (!\_disposedValue)

            {

                if (disposing)

                {

                    // TODO: dispose managed state (managed objects)

                    Clear();

                }

                // TODO: free unmanaged resources (unmanaged objects) and override finalizer

                // TODO: set large fields to null

                \_disposedValue = true;

            }

        }

        public void Dispose()

        {

            // Do not change this code. Put cleanup code in 'Dispose(bool disposing)' method

            Dispose(disposing: true);

            GC.SuppressFinalize(this);

        }

        public async ValueTask DisposeAsync()

        {

            if (!\_disposedAsync)

            {

                await DisposeAsyncCore();

                GC.SuppressFinalize(this);

                \_disposedAsync = true;

            }

        }

        protected virtual void Clear() { }

        protected virtual ValueTask DisposeAsyncCore() => new();

    }

}

* Create ‘appsettings.json’ file like:

{

  "ConnectionStrings": {

    "conn-sql": "Server=.;Initial Catalog=Examples;Integrated Security=False;Persist Security Info=False;User ID=sa;Password=sa;encrypt=false;"

  }

}

* Add startup class:

using Microsoft.AspNetCore.Http;

using Microsoft.Data.SqlClient;

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.DependencyInjection.Extensions;

using Microsoft.Extensions.Logging;

using Repository;

using System.Data;

using Xunit.Abstractions;

namespace Tests

{

    public class TestStartup : IDisposable, IAsyncDisposable

    {

        private readonly IServiceCollection \_services;

        private IServiceProvider? \_serviceProvider;

        private bool \_disposedValue;

        private bool \_disposedAsync;

        public TestStartup()

        {

            \_services = new ServiceCollection();

            ConfigurationBuilder = new ConfigurationBuilder().SetBasePath(Directory.GetCurrentDirectory());

            Configuration = GetConfigurationRoot();

            AddServices(\_services, Configuration);

        }

        public IConfigurationRoot? Configuration { get; private set; }

        public IConfigurationBuilder ConfigurationBuilder { get; private set; }

        public IServiceProvider GetServiceProvider(ITestOutputHelper testOutputHelper)

        {

            if (\_serviceProvider != default)

            {

                return \_serviceProvider;

            }

            return \_serviceProvider = \_services.BuildServiceProvider();

        }

        public T? GetScopedService<T>(ITestOutputHelper testOutputHelper)

        {

            var serviceProvider = GetServiceProvider(testOutputHelper);

            using var scope = serviceProvider.CreateScope();

            return scope.ServiceProvider.GetService<T>();

        }

        public T? GetService<T>(ITestOutputHelper testOutputHelper)

            => GetServiceProvider(testOutputHelper).GetService<T>();

        protected void AddServices(IServiceCollection services, IConfiguration? configuration)

        {

            var connectionstring = configuration?.GetConnectionString("conn-sql");

            services.AddDbContext<DbEntity>(options => options.UseSqlServer(connectionstring));

            services.AddTransient<IDbConnection>(db => new SqlConnection(connectionstring));

            services.AddDbContext<DbEntity>(ServiceLifetime.Transient);

            services.AddTransient<IUserRepository, UserRepository>();

            services.TryAddTransient<IHttpContextAccessor, HttpContextAccessor>();

        }

        protected IEnumerable<TestAppSettings> GetTestAppSettings()

        {

            yield return new() { Filename = "appsettings.json", IsOptional = false };

        }

        protected virtual ILoggingBuilder AddLoggingProvider(ILoggingBuilder loggingBuilder, ILoggerProvider loggerProvider)

            => loggingBuilder.AddProvider(loggerProvider);

        private IConfigurationRoot? GetConfigurationRoot()

        {

            var testAppSettings = GetTestAppSettings();

            return

                testAppSettings.All(setting => !string.IsNullOrEmpty(setting.Filename))

                ? GetConfigurationRoot(testAppSettings)

                : default;

        }

        private IConfigurationRoot GetConfigurationRoot(IEnumerable<TestAppSettings> configurationFiles)

        {

            foreach (var configurationFile in configurationFiles)

            {

                ConfigurationBuilder.AddJsonFile(configurationFile.Filename, optional: configurationFile.IsOptional);

            }

            ConfigurationBuilder.AddEnvironmentVariables();

            return ConfigurationBuilder.Build();

        }

        protected virtual void Dispose(bool disposing)

        {

            if (!\_disposedValue)

            {

                if (disposing)

                {

                    // TODO: dispose managed state (managed objects)

                    if (\_serviceProvider is not null)

                    {

                        ((ServiceProvider)\_serviceProvider).Dispose();

                    }

                    \_services.Clear();

                }

                // TODO: free unmanaged resources (unmanaged objects) and override finalizer

                // TODO: set large fields to null

                \_disposedValue = true;

            }

        }

        public void Dispose()

        {

            // Do not change this code. Put cleanup code in 'Dispose(bool disposing)' method

            Dispose(disposing: true);

            GC.SuppressFinalize(this);

        }

        public async ValueTask DisposeAsync()

        {

            if (!\_disposedAsync)

            {

                await DisposeAsyncCore();

                GC.SuppressFinalize(this);

                \_disposedAsync = true;

            }

        }

        protected ValueTask DisposeAsyncCore() => new();

    }

}

* Create test class and inherit Fixture<TestStartup>:

using Repository;

using Xunit.Abstractions;

namespace Tests.UnitTests

{

    public class UserManagerTests : Fixture<TestStartup>

    {

        public UserManagerTests(ITestOutputHelper testOutputHelper, TestStartup fixture) : base(testOutputHelper, fixture)

        {

        }

        [Fact]

        public void Gets\_ReturnUserList\_IfFound()

        {

            // Arrange

            var userRepository = \_fixture.GetService<IUserRepository>(\_testOutputHelper);

            // Act

            var res = userRepository?.Gets();

            // Assert

            Assert.NotNull(res);

        }

    }

}