Nunit with Ado .net

Favorite	☐
Tag	C# Programming

This guide explains how to implement and test a gRPC service that retrieves data from a SQL database using dependency injection, wrapping the `SqlDataReader` class for testability, and creating NUnit test cases.

Prerequisite understanding

1. Mock(moq).
2. Wrapper class.

Steps

1. Create a new folder in GrpcService Project and anything eg.Data.
2. Create an Interface for wrapping `DataReader`.

   We wrap
   `SqlDataReader` to make it mockable for unit testing and to abstract its behavior

```
namespace GrpcService1.Data
{
    public interface IDataReader : IDisposable
    {
        Task<bool> ReadAsync();
        string GetString(string columnName);
    }

}
```

3. write Implementantion of that interface

```
using Microsoft.Data.SqlClient;

namespace GrpcService1.Data
{
    public class DataReaderWrapper : IDataReader
    {
        private readonly SqlDataReader _reader;

        public DataReaderWrapper(SqlDataReader reader)
        {
            _reader = reader;
        }

        public async Task<bool> ReadAsync() //we wrapped this method
        {
            return await _reader.ReadAsync();
        }

        public string GetString(string columnName) //to get the value of that column
        {
            return _reader[columnName]?.ToString();
        }

        public void Dispose()
        {
            _reader?.Dispose();
        }
    }
}
```

4. Create an another interface for the database services (which will perform the action of fetching/writing data from database)(ISqlService in my case)

```
using Microsoft.Data.SqlClient;

namespace GrpcService1.Data
{
    public interface ISqlService
    {
        Task<IDataReader> ExecuteTheReader(string Procedure_name, SqlParameter[] parameters);
    }
}
```

5. Create implementation for for that interface

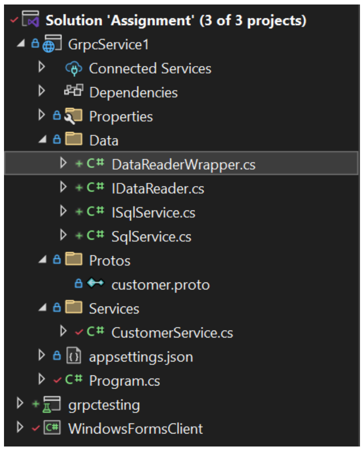
```
using Microsoft.Data.SqlClient;

namespace GrpcService1.Data
```

```
{
    public class SqlService : ISqlService
    {
        public async Task<IDataReader> ExecuteTheReader(string Procedure_name, SqlParameter[] parameters)
        {
            SqlConnection conn = new SqlConnection("Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=Customer;Integrated Security=True;");
            await conn.OpenAsync();
            using SqlCommand cmd = new SqlCommand(Procedure_name, conn);
            cmd.CommandType= System.Data.CommandType.StoredProcedure;
            if (parameters != null)
            {
                cmd.Parameters.AddRange(parameters);
            }
            var reader = await cmd.ExecuteReaderAsync();
            return new DataReaderWrapper(reader);



        }
    }
}
```



6. Add dependent injection in your program.cs file(below builder.services.Addgrpc())

```
builder.Services.AddTransient<ISqlService,SqlService>();
```

7. Modify our GrpcService File to use that function of fetching the data

```
using Grpc.Core;
using GrpcService1;
using GrpcService1.Data;
using Microsoft.Data.SqlClient;

namespace GrpcService1.Services
{
    public class CustomerDataService : CustomerData.CustomerDataBase
    {
        public readonly ISqlService _sqlService;
        public CustomerDataService(ISqlService sqlservice) //inject the service througn the constructor
        {
            _sqlService = sqlservice;
        }

        private readonly string connectionstring = "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=Customer;Integrated Security=True;"; // connection str


        //service defination for  rpc function GetCustomers
        public override async Task<CustomerList> GetCustomers(Empty request, ServerCallContext context)
        {
            CustomerList customerList = new(); //Initialing return list
            try
            {


                using IDataReader reader=await _sqlService.ExecuteTheReader("get", null); //using the service of fetching data

                    while (await reader.ReadAsync())
                    {
                        var customer = new Customer()
                        {
                            FirstName = reader.GetString("First_Name"),
                            LastName = reader.GetString("Last_Name"),
                            Dateofbirth = reader.GetString("Date_of_Birth"),
                            Id = reader.GetString("Id"),
                        };
                        customerList.Custometrs.Add(customer);  // maping customer data customer object and adding it to list

                    }
                //}
                return customerList;
            }
            catch (Exception e)
            {

                customerList.Isfailed = true; // if some error occurs
                customerList.Errortxt = e.Message;
                return customerList;

            }

        }.
        .
        .
        .
        .
        .
        .
```

8. For nunit create new project in solution of type nunit3 and add refrence to GrpcService project.
9. install `moq` library from packet manager
10. and write the testcases by mocking the
    a. `SqlService` (service which having function of fetching/writing data to database) and mocking.
    b. `DataReader()` : which we have wrapped

Sample testcase file

```
using Grpc.Core;
using GrpcService1;
using GrpcService1.Data;
using GrpcService1.Services;
using Microsoft.Data.SqlClient;
using Moq;
using NUnit.Framework.Internal;

namespace grpctesting
{
    [TestFixture]
    public class Test
    {
        private Mock<ISqlService> _sqlServiceMock;
        private CustomerDataService _customerService;


        [SetUp]
        public void Setup()
        {
            _sqlServiceMock = new Mock<ISqlService>();
            _customerService = new CustomerDataService(_sqlServiceMock.Object);
        }

        [Test]
        public async Task GetCustomers_ReturnsCustomerList_WhenDataExists()
        {
            // Arrange
            var mockDataReader = new Mock<IDataReader>();
            var sequence = new MockSequence();

            mockDataReader.InSequence(sequence).Setup(r => r.ReadAsync()).ReturnsAsync(true);
            mockDataReader.Setup(r => r.GetString("First_Name")).Returns("John");
            mockDataReader.Setup(r => r.GetString("Last_Name")).Returns("Doe");
            mockDataReader.Setup(r => r.GetString("Date_of_Birth")).Returns("2000-01-01");
            mockDataReader.Setup(r => r.GetString("Id")).Returns("123");

            mockDataReader.InSequence(sequence).Setup(r => r.ReadAsync()).ReturnsAsync(false);

            _sqlServiceMock.Setup(s => s.ExecuteTheReader("get", null)).ReturnsAsync(mockDataReader.Object);

            // Act
            var result = await _customerService.GetCustomers(new Empty(), It.IsAny<ServerCallContext>());

            // Assert
            Assert.IsFalse(result.Isfailed);
            Assert.AreEqual(1, result.Custometrs.Count);
            Assert.AreEqual("John", result.Custometrs[0].FirstName);
            Assert.AreEqual("Doe", result.Custometrs[0].LastName);
        }

        [Test]
        public async Task GetCustomers_ReturnsEmptyCustomerList_WhenDataDoesNotExists()
        {
```

```
        // Arrange
        var mockReader = new Mock<IDataReader>();

        mockReader.SetupSequence(r => r.ReadAsync())
                  .ReturnsAsync(false); // No rows to read

        var mockSqlService = new Mock<ISqlService>();
        mockSqlService.Setup(s => s.ExecuteTheReader("get", null))
                      .ReturnsAsync(mockReader.Object); // Return the mock reader

        var service = new CustomerDataService(mockSqlService.Object);

        // Act
        var result = await service.GetCustomers(new Empty(), null);

        // Assert
        Assert.IsNotNull(result);
        Assert.IsFalse(result.Custometrs.Any()); // Ensure the customer list is empty
        Assert.IsFalse(result.Isfailed); // Ensure no failure flag is set
        Assert.AreEqual(0, result.Custometrs.Count());
    }


    [Test]
    public async Task GetCustomers_ReturnsError_WhenDatabaseConnectionFails()
    {
        // Arrange
        var mockSqlService = new Mock<ISqlService>();
        mockSqlService.Setup(s => s.ExecuteTheReader("get", null))
                      .ThrowsAsync(new Exception("Database connection failed"));

        var service = new CustomerDataService(mockSqlService.Object);

        // Act
        var result = await service.GetCustomers(new Empty(), null);

        // Assert
        Assert.IsNotNull(result);
        Assert.IsTrue(result.Isfailed); // Ensure failure flag is set
        Assert.AreEqual("Database connection failed", result.Errortxt); // Check error message
        Assert.IsEmpty(result.Custometrs); // Ensure no customers are returned
    }


    [Test]
    public async Task GetCustomers_ReturnsPartialCustomerList_WhenSomeRowsHaveInvalidData()
    {
        // Arrange
        var mockReader = new Mock<IDataReader>();
        mockReader.SetupSequence(r => r.ReadAsync())
                  .ReturnsAsync(true) // Row 1
                  .ReturnsAsync(true) // Row 2 (invalid)
                  .ReturnsAsync(false); // End of data

        // First row is valid
        mockReader.Setup(r => r.GetString("First_Name")).Returns("John");
        mockReader.Setup(r => r.GetString("Last_Name")).Returns("Doe");
        mockReader.Setup(r => r.GetString("Date_of_Birth")).Returns("11/10/2002");
        mockReader.Setup(r => r.GetString("Id")).Returns("1");

        // Second row is invalid (throws exception for some fields)
        mockReader.Setup(r => r.GetString("First_Name"))
                  .Throws(new Exception("Invalid field data"));

        var mockSqlService = new Mock<ISqlService>();
        mockSqlService.Setup(s => s.ExecuteTheReader("get", null))
                      .ReturnsAsync(mockReader.Object);

        var service = new CustomerDataService(mockSqlService.Object);

        // Act
        var result = await service.GetCustomers(new Empty(), null);

        // Assert
        Assert.IsNotNull(result);
        Assert.IsTrue(result.Isfailed); // Ensure failure flag is set
        Assert.AreEqual("Invalid field data", result.Errortxt); // Check error message
        Assert.AreEqual(1, result.Custometrs.Count); // Ensure only valid rows are processed
    }




    [Test]
    public async Task GetCustomers_ReturnsCorrectlyMappedData_WhenDatabaseContainsDifferentDataTypes()
    {
        // Arrange
        var mockReader = new Mock<IDataReader>();
        mockReader.SetupSequence(r => r.ReadAsync())
```

```
                        .ReturnsAsync(true)
                        .ReturnsAsync(false); // Single row

            mockReader.Setup(r => r.GetString("First_Name")).Returns("Jane");
            mockReader.Setup(r => r.GetString("Last_Name")).Returns("Smith");
            mockReader.Setup(r => r.GetString("Date_of_Birth")).Returns("1990-05-10");
            mockReader.Setup(r => r.GetString("Id")).Returns("123");

            var mockSqlService = new Mock<ISqlService>();
            mockSqlService.Setup(s => s.ExecuteTheReader("get", null))
                          .ReturnsAsync(mockReader.Object);

            var service = new CustomerDataService(mockSqlService.Object);

            // Act
            var result = await service.GetCustomers(new Empty(), null);

            // Assert
            Assert.IsNotNull(result);
            Assert.IsFalse(result.Isfailed); // Ensure no failure flag is set
            Assert.AreEqual(1, result.Custometrs.Count); // Ensure correct number of customers
            Assert.AreEqual("Jane", result.Custometrs[0].FirstName);
            Assert.AreEqual("Smith", result.Custometrs[0].LastName);
            Assert.AreEqual("1990-05-10", result.Custometrs[0].Dateofbirth);
            Assert.AreEqual("123", result.Custometrs[0].Id);
        }




    }

}
```