



xUnit Testing

.NET

xUnit.net is a free, open source, community-focused unit testing tool for the .NET Framework. xUnit.net is part of the .NET Foundation.

[HTTPS://XUNIT.NET/](https://xunit.net/)

Arrange, Act, Assert

<https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2019>

The **Arrange-Act-Assert** pattern is a common way of writing unit tests.

- The **Arrange** section of a unit test method initializes objects and sets the value of the data that is passed to the method under test.
- The **Act** section invokes the method under test with the arranged parameters.
- The **Assert** section verifies that the action of the method under test behaves as expected.

```
[TestMethod]
public void Withdraw_ValidAmount_ChangesBalance()
{
    // arrange
    double currentBalance = 10.0;
    double withdrawal = 1.0;
    double expected = 9.0;
    var account = new CheckingAccount("JohnDoe", currentBalance);

    // act
    account.Withdraw(withdrawal);

    // assert
    Assert.AreEqual(expected, account.Balance);
}
```

xUnit Testing Step-By-Step in Visual Studio

<https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-dotnet-test#create-a-test>
<https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-dotnet-test>

1. Open a Solution in Visual Studio.
2. Right-Click the Solution.
3. Add >> new project...
4. Type “xunit” in the template search box.
5. Select xUnit Test Project(.NET Core).
6. Name the project whatever you want (VS inserts ‘_’ for spaces).
7. Right-Click ‘Dependencies’ in the test project.
8. Click ‘Add Reference’
9. In the left pane of the Add References window, click ‘Projects’
10. In the center pane, click to check the Projects containing methods you want to test.
11. Click ‘OK’
12. Add your tests to the Test project.

xUnit Testing in VS Code - Step-by-Step

<https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-dotnet-test>

<https://stackoverflow.com/questions/45127849/xunit-namespace-could-not-be-found-in-visual-studio-code>

1. Create a **.sln** and a directory of the same name holding that .sln to which you will add the various projects with

- `dotnet new sln -o [.sln name] -`

2. Enter the new directory with:

- `cd [newDirectoryName]`

4. Create the App project to be tested. Skip this step if you already have an app project created.

- `dotnet new console -o [projectName]`

5. Add the App project to the .sln with:

- `dotnet sln add <relativePathToProject>.csproj`

6. Add code to the project (i.e. methods to test).

7. Create the testing project with:

- `dotnet new xunit -o [testingProjectName].Tests`

8. Add the testing project to the solution with

- `dotnet sln add`
`./[directoryOfTestingProject]/[NameOfTestingProject].csproj`

9. Add the project as a dependency to the testing project with

- `dotnet add`
`<relativePathToXunitProj>.csproj reference <relativePathToAppProj>.csproj`

10. Make sure your project classes are public.

11. Reference the App project inside the Xunit project with

- `using [App project namespace];`

12. Create tests in the testing project

13. Enter the testing project folder with

- `cd TestingProjectFolderName`

13. From the testing project directory, run **dotnet test** to run the tests.

InMemory DB - Step-by-Step

<https://docs.microsoft.com/en-us/ef/core/miscellaneous/testing/in-memory>

<https://www.thereformedprogrammer.net/using-in-memory-databases-for-unit-testing-ef-core-applications/>

EF Core database providers do not have to be relational databases. *InMemory* is designed to be a general-purpose database for testing. It is not designed to mimic a relational database.

There are a few steps to setting up a *InMemory* DB.

1. Set up the constructor in your DB Context class to accept a DB configuration parameter called *DbContextOptions*.

```
public class BloggingContext : DbContext
{
    public BloggingContext()
    { }

    public BloggingContext(DbContextOptions<BloggingContext> options)
        : base(options)
    { }
}
```

InMemory DB - Step-by-Step

<https://docs.microsoft.com/en-us/ef/core/miscellaneous/testing/in-memory>

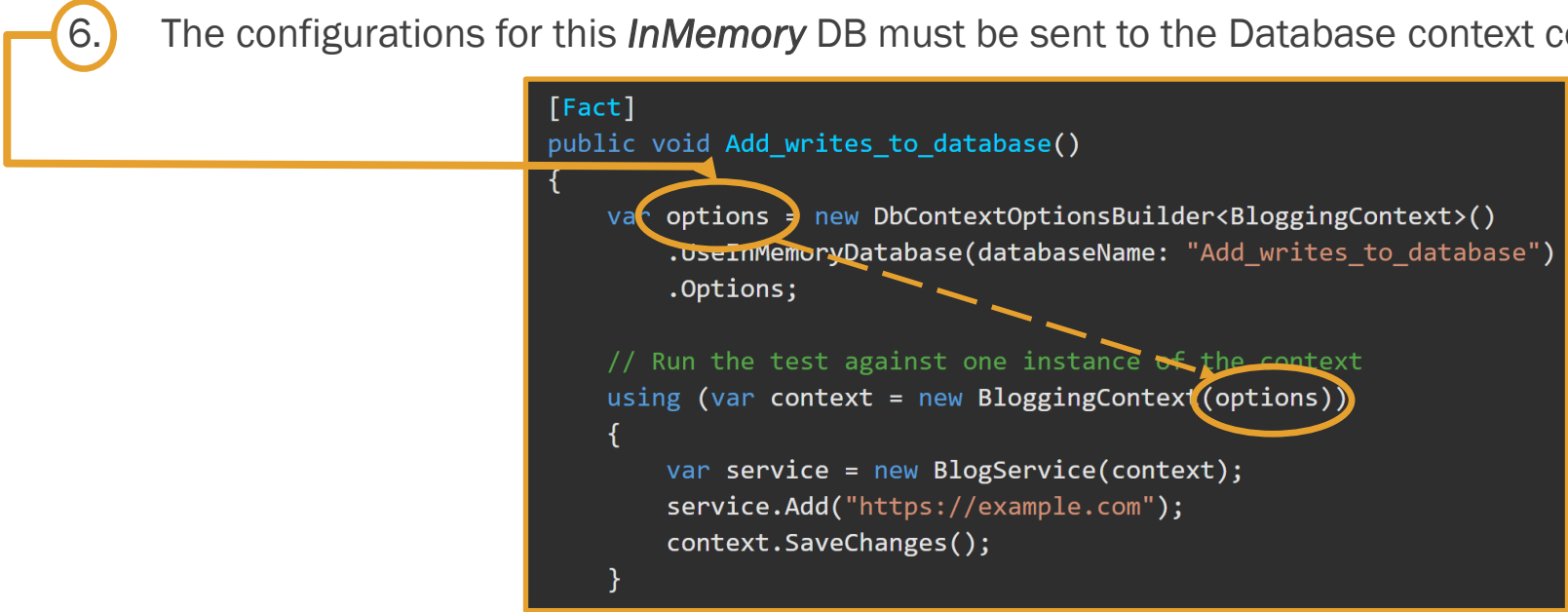
2. Alter your `DbContext.OnConfiguring()` to check for an already configured DB and to not use your production DB if there's already a DB configured.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseSqlServer(@"Server=(localdb)\mssqllocaldb;Database=EF
    }
}
```


InMemory DB - Step-by-Step

<https://docs.microsoft.com/en-us/ef/core/miscellaneous/testing/in-memory>

3. Right-click your test project to download the NuGet Package *Microsoft.EntityFrameworkCore.InMemory*.
4. In the test project, configure a new, clean context for every test.
5. Instead of creating a new Database for each test you can create the InMemory Db above all tests, then, in each test, call:
 3. `context.Database.EnsureDeleted();` // delete any Db from a previous test
 4. `context.Database.EnsureCreated();` // create anew the Db. You will need to seed the Db again.
6. The configurations for this *InMemory* DB must be sent to the Database context constructor on instantiation.



```
[Fact]
public void Add_writes_to_database()
{
    var options = new DbContextOptionsBuilder<BlogggingContext>()
        .UseInMemoryDatabase(databaseName: "Add_writes_to_database")
        .Options;

    // Run the test against one instance of the context
    using (var context = new BlogggingContext(options))
    {
        var service = new BlogService(context);
        service.Add("https://example.com");
        context.SaveChanges();
    }
}
```


InMemory DB - Step-by-Step

<https://docs.microsoft.com/en-us/ef/core/miscellaneous/testing/in-memory>

Here is a
sample test
for
comparison.

Arrange

Act

Assert

```
[Fact]
public void Find_searches_url()
{
    var options = new DbContextOptionsBuilder<BloggingContext>()
        .UseInMemoryDatabase(databaseName: "Find_searches_url")
        .Options;

    // Insert seed data into the database using one instance of the context
    using (var context = new BloggingContext(options))
    {
        context.Blogs.Add(new Blog { Url = "https://example.com/cats" });
        context.Blogs.Add(new Blog { Url = "https://example.com/catfish" });
        context.Blogs.Add(new Blog { Url = "https://example.com/dogs" });
        context.SaveChanges();
    }

    // Use a clean instance of the context to run the test
    using (var context = new BloggingContext(options))
    {
        var service = new BlogService(context);
        var result = service.Find("cat");
        Assert.Equal(2, result.Count());
    }
}
```

Assert()

`Assert.Equal(<expected>,<result>)`

`Assert.Equal(<expected>,<result>, <roundedToHowManyDecimals>)`

`Assert.Equal(<expected>,<result>, true)` => Set the ignoreCase property to true to be case insensitive.

`Assert.Contains(<substring>, <result>);` => check if some string is present

`Assert.Contains(<substring>, <result>, StringComparison.InvariantCultureIgnoreCase);` => to be case insensitive

`Assert.StartsWith(<substring>,<result>);` => check if the result begins with some substring

`Assert.EndsWith(<substring>,<result>);` => check if the result ends with some substring

`Assert.Matches(<regex>, <result>);` => check if the result matches a regular expression.

Assert()

`Assert.Null(<nullResult>);` Passes if the value is true

`Assert.NotNull(<result>);` Passes if the value is not null

`Assert.True(<trueResult>);`

`Assert.False(<falseResult>);`