HANDS ON WITH BLAZOR

Principles of Testing Razor Components

Overview

Testing Razor Components

- Unit Testing is a important aspect of modern software development
- Unit Testing .NET applications and C# code is a well established discipline with an assortment of tools and unit testing libraries
- Visual Studio provides tooling such as Test Explorer
- Popular unit testing libraries include MSTest, NUnit and xUnit
- The last few years xUnit has been the favorite among C# developers
- So what about testing Razor Components used in Blazor? Is there a unit testing framework for that?

Overview

Testing Razor Components

- Well, not really, at least not an official library...
- Instead, a 3rd party library named **bUnit** will used
- Microsoft recommends bUnit in its Blazor documentation
  - https://docs.microsoft.com/en-us/aspnet/core/blazor/test?view=aspnetcore-6.0#test-components-with-bunit
- bUnit is only a unit testing library, it is not a unit test runner so it works in concert with xUnit, NUnit, or MS Test to run the tests
  - https://bunit.dev/index.html

Overview

Test Driven Development

- Not a new concept, originally described in an old computer programming book in 1968
- Test Driven Development (TDD) is a development process whereby the developer writes the unit's test before writing the unit's code
- The desired outcome of the unit is determined, the test is written, then code is written to pass the test
- As the code is written, the tests are re-executed to see if the code passes
- The re-execution can be manual or automatic (such as on file save)

Overview

Test Driven Development

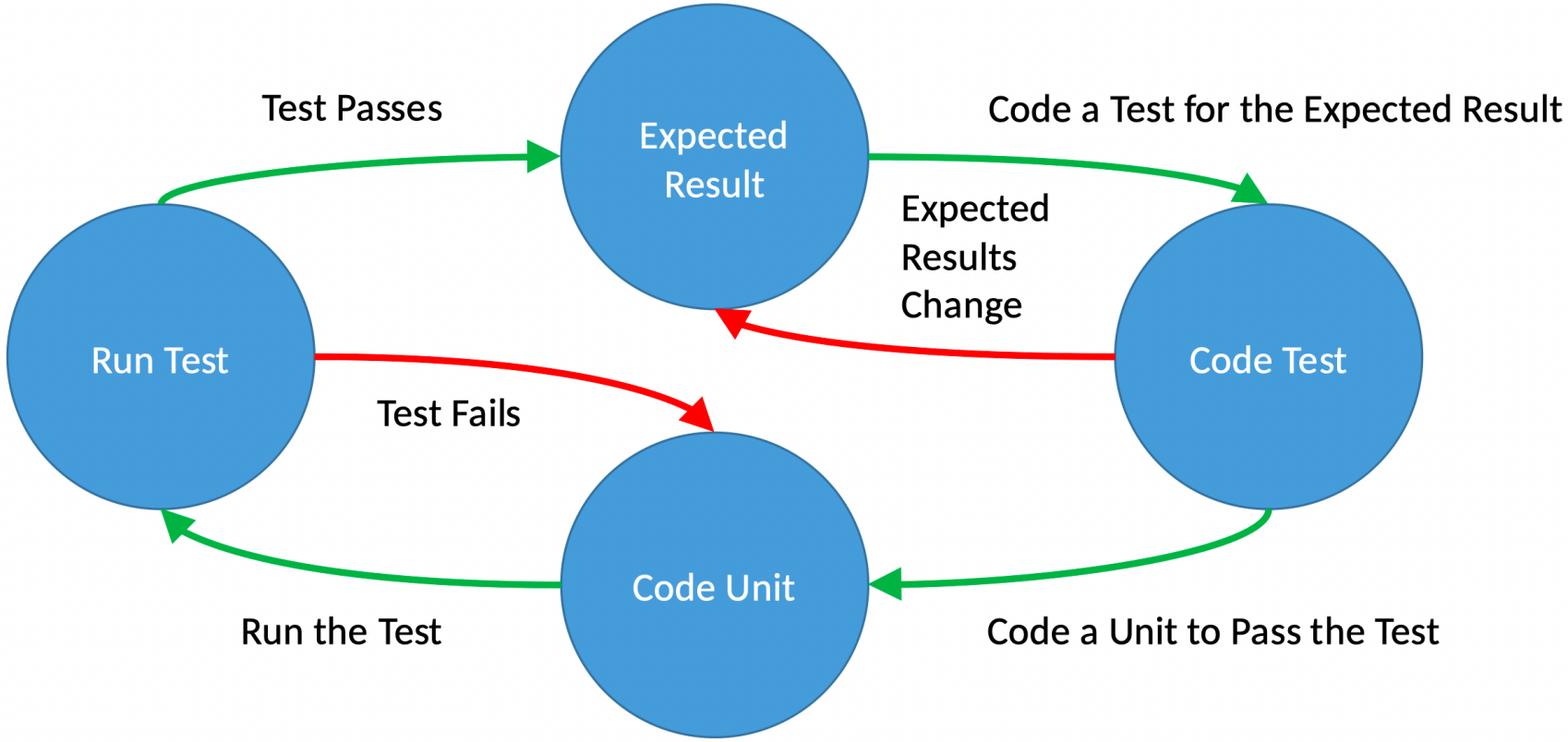

Diagram: Test-Driven Development

Overview

Test Driven Development

- True TDD development is really hard to do, and requires discipline and experience on the part of programmers, and support from the organization
- TDD development guarantees unit tests are written, and usually results in better unit code
- Unit code more likely to be focused on doing one thing (single responsibility principle), and will therefore typically have a lower cyclomatic complexity
- Allows quick regression testing when changes are made
- Very important to building high quality, maintainable applications
- Ensures the application's many units work as expected ("as expected" is key, unit tests do not ensure the unit works as needed by the larger system just that it works as understood by the developer)

Overview

Unit Testing – What It Measures

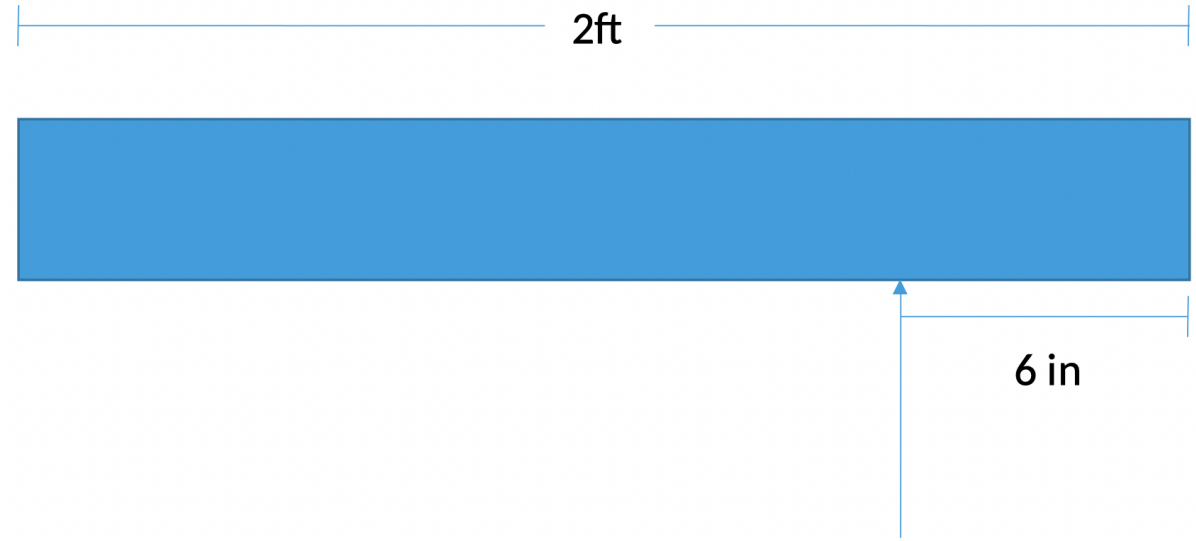

Diagram: What It Measures

- Writing code without a unit test is like cutting 6 inches off a 2 foot block of wood simply by measuring 6 inches over and cutting. The measure could be right, but as the old adage goes, "better to measure twice and cut once"

Overview

Unit Testing – What It Measures

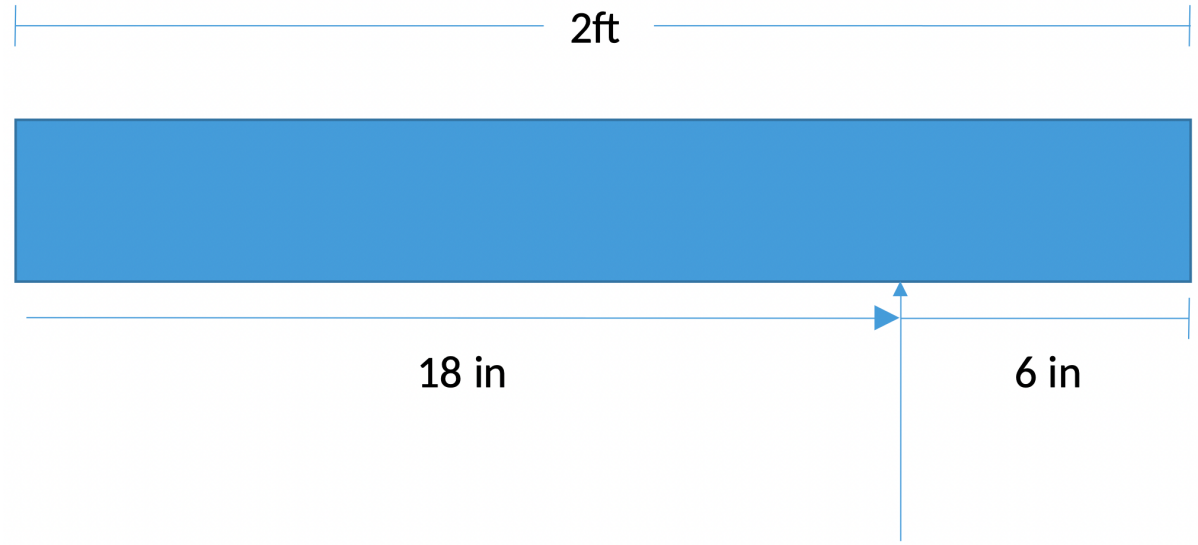

Diagram: What It Measures

- Writing code with a unit test is like cutting 6 inches off a 2 foot block of wood by measuring 6 inches over, then measuring 18 inches back and then cutting. Measuring back is a verification that the first measurement was 6 inches

Overview

Unit Testing – What It Measures

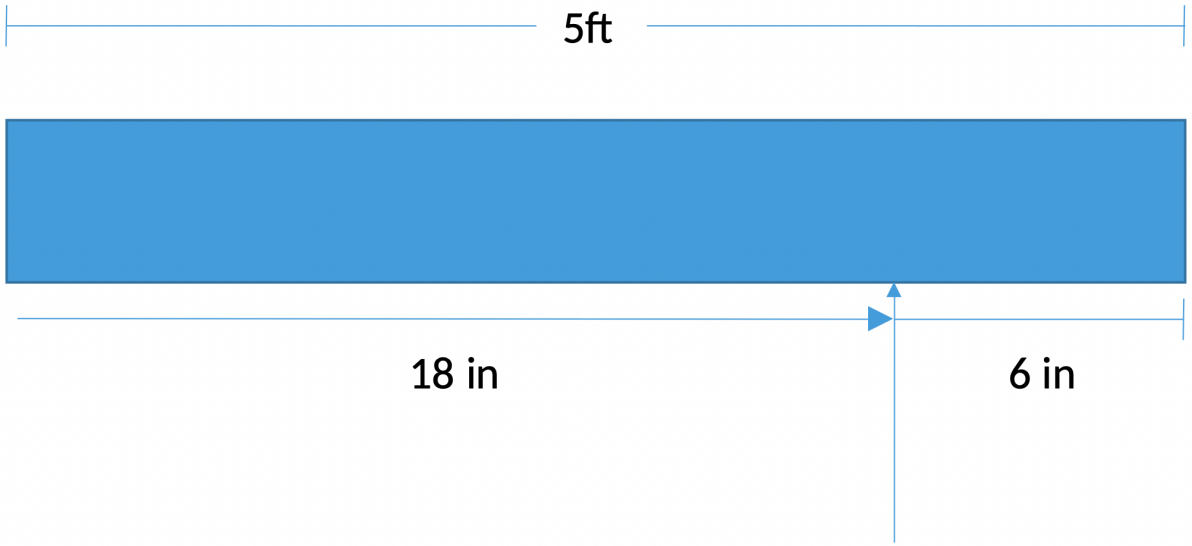

Diagram: What It Measures

- However, unit testing assumes that the board is in fact 2 feet long. If the board is not the size the developer expect it to be then the unit's code and its test are both likely to be wrong even if it appears to pass

Overview

What is a Unit?

- A unit is like beauty, it's in the eye of the holder
- A unit can be almost anything you want it to be
- The key to unit testing is honoring the sanctity of whatever the unit is determined to be
- Units do not traverse system boundaries such as communicating over networks, with file systems, interacting with databases, etc...
- When unit testing, only the unit is real, the rest of YOUR code is mocked
- It is safe to assume that third-party libraries, frameworks, and such are unit tested on their own and can be used "as is" so long as the boundaries mentioned above are not crossed

Overview

What is a Unit?

- All units have a certain number of internal code paths (aka cyclomatic complexity)
- Most units have some kind of parameterized inputs
- Most units have some kind of output
- Most units need to communicate with other units
- Many units (but hopefully very few) require external dependencies which are not parameterized (think global variables)

Overview

What is a Unit?

- All inputs to the unit should be mocked data (fake data which is pre-determined)
- All outputs from the unit (direct & indirect) should be asserted to be equal to an expected value
- There should be one unit test per linear code path
  - Each test should pass in mock data appropriate for the tested linear path
  - And each test should have at least one assertion per output
- Developers should aim for 100% code coverage in spirit

Overview

Testing a Unit

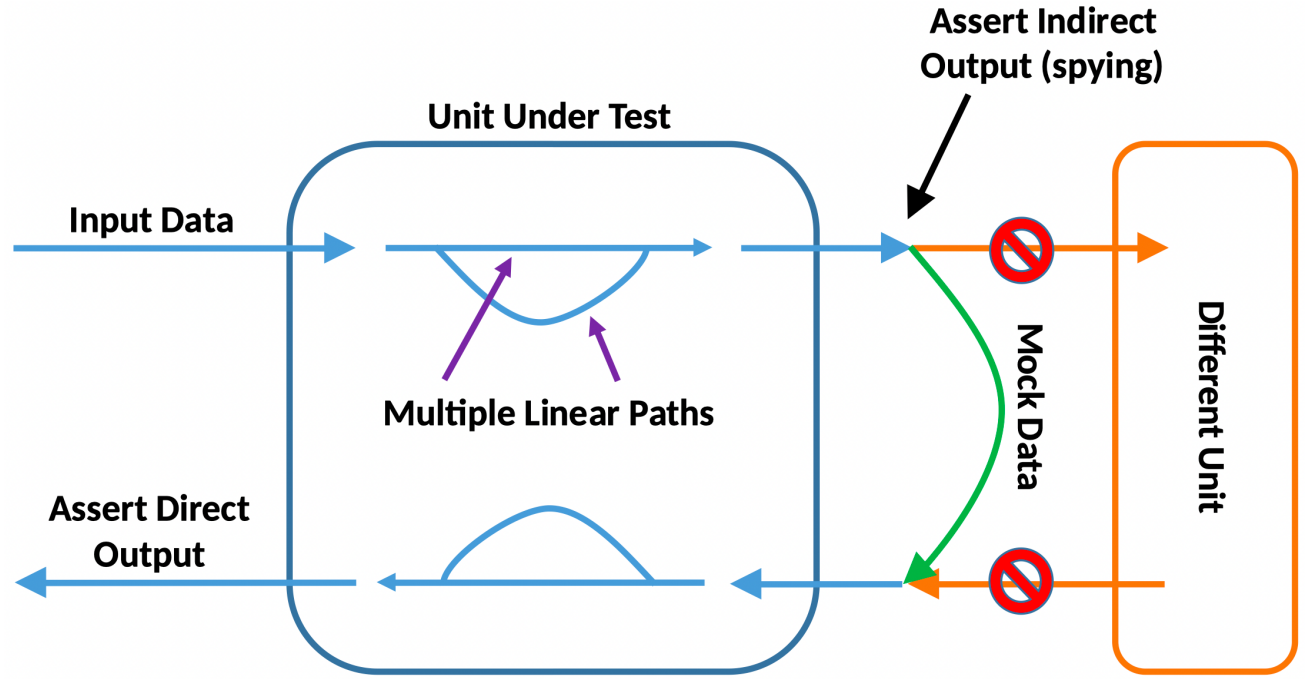

Diagram: Testing a Unit

Overview

Unit Testing Razor Components

- Let's explore how these universal principles of unit testing apply to Razor Components
- In focus will be the following
  - Setting up an xUnit Testing project with bUnit
  - Understanding a Razor Component as a unit
  - Exploring the two flavors of unit testing with bUnit
  - Learn how to apply test various outputs of a Razor Component
  - Demonstrate how to use mocks in Razor Component testing

Development Environment

Getting Started

- Blazor requires .NET 6 (.NET 5 and .NET Core 3.1 can be used as well)
  - .NET 6 runs on Windows, macOS, and Linux
  - https://dotnet.microsoft.com/download
- Blazor web apps can be coded with any text editor
- Editors such as Visual Studio Code, Visual Studio and Visual Studio for Mac are commonly used
- Blazor development can be accomplish via the .NET CLI and the traditional Visual Studio tooling

Razor Component Library

Organize Razor Components

- Razor Components can be shared among Blazor projects, both Server Model and WebAssembly Model
- To share Razor Components, they should be placed in a Razor Component Library project
- Razor Component Library projects help organize Razor Components independent of a particular Blazor project
- Razor Component Libaries are not required, a Razor Component in the main project can be tested, but organizing Blazor projects well is part of building code that is testing friendly
- The project should be added to to the blank solution created earlier
    - Note: When creating the project, please do NOT check the box for "Support pages and views"
- After creating the project, delete the `Component1.razor` and `ExampleJsInterop.cs` files
- Also, delete the files `background.png` and `exampleJsInterop.js` files in the `wwwroot` folder, do not delete the `wwwroot` folder itself

Razor Component Unit Test Library

Organize Razor Component Unit Tests

- To program the unit tests, a new unit test project needs to be created
- To test Razor Components any of the built-in unit testing project types can be selected: MS Test, NUnit, and xUnit
- For this course, an xUnit project will be created
- The project should be added to to the blank solution created earlier
- After creating the project, delete the `UnitTest1.cs` file

xUnit and bUnit Testing

Installing Packages and Configure Unit Testing Project

- Add the `bunit` NuGet package to the Unit Testing probject
- The `bunit` NuGet package can be installed with the .NET CLI, NuGet Package CLI, or the Visual Studio Package Manager

```
dotnet add package bunit
```

- You can specify a specific version if you do not want the latest version

```
dotnet add package bunit --version 1.9.8
```

- For a listing of all versions and example install instructions: https://www.nuget.org/packages/bunit
- Edit the project file for the Unit Testing project and update the project SDK to `Microsoft.NET.Sdk.Razor`
- Add a project reference from the Razor Component library project to the Unit Testing project

xUnit and bUnit Testing

Add Imports Razor

- Using the Razor Component template, create a file named `_Imports.razor` in the root folder of the unit testing project
- Add the following using statements to the `_Imports.razor` file

```
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.JSInterop
@using Microsoft.Extensions.DependencyInjection
@using AngleSharp.Dom
@using Xunit
@using Bunit
@using Bunit.TestDoubles
```

- Also, you can add a using statement for the desired namespaces of the components project
- Build the solution, everything should build successfully

Unit Tests in Razor Files

Testing in Razor and C# Files

- bUnit supports two syntax flavors for writing Razor Component unit tests
- bUnit supports programming unit tests in Razor files and C# files
- The benefit of Razor files is that the declarative HTML-like syntax can be used to call the component
- This declarative syntax is how the component is used within an application
- Alternatively, unit tests can be coded in C# files as well
- In this section, coding a unit test in a Razor file will be explored

Unit Tests in Razor Files

Test-Driven Development

- Building upon Test-Driven Development as explained earlier, let's employ that approach for writing the first Razor Component unit test with bUnit
- To write the first unit test, a Test-Driven Development (TDD) approach will be taken
  - For more information on TDD: https://en.wikipedia.org/wiki/Test-driven_development
- Please remember:
  - TDD forces the developer to think of requirements first, then the code (usually the opposite is the case)
  - TDD helps to produce the smallest amount of focused code that passes the test, code can be refactored later to make it more elegant with the test ensure the new, more elegant code still passes the test
  - TDD ensures there are actually unit tests

Unit Tests in Razor Files

Test-Driven Development

- The following sequence is from the Wikipedia documentation: https://en.wikipedia.org/wiki/Test-driven_development
- The basic flow is as follows:
  - Add a test
  - Run all tests. The new test should fail for expected reasons.
  - Write the simplest code that passes the new test.
  - All tests should now pass.
  - Refector as needed, using tests after each refactor to ensure the functionality is preserved
  - Repeat

Unit Tests in Razor Files

Unit Test Principles

- Within a test, the common pattern of Arrange, Act, and Assert will be followed
- When writing unit tests the goal is to test the code we write, the goal is NOT to test the frameworks and libraries we are using, they should already be tested by their creators
- The code unit under test must be tested independently of all other code that we have written
- All code paths within the code need to be tested
- No calls to networks, disks, databases, etc... can be used within a unit test, all data must be hard coded into the test

Unit Tests in Razor Files

Create the Razor Test

- In the unit test project create a new Razor Component, name the file `HelloWorldRazorTest.razor`
- Remove the HTML code from the top of the file, and add an empty unit test function to the code block
- The `FactAttribute` marks the function as a unit test
- In the three commented blocks, the parts of the unit test will be written

```
@code {
  [Fact]
  public void HelloWorldComponentRendersCorrectly()
  {
    // Arrange

    // Act

    // Assert
  }
}
```

Unit Tests in Razor Files

Create the Razor Test

- To test a Razor Component, a `TestContext` is needed to render the component

```
// Arrange
using var ctx = new TestContext();
```

Unit Tests in Razor Files

Create the Razor Test

- To render the component, the `Render` method on the test context is used
- Using inline Razor syntax, the component under test is called using the HTML-like syntax with the result being passed into the `Render` method
- The `Render` method returns a rendered fragment

```
// Act
var cut = ctx.Render(@<HelloWorld />);
```

Unit Tests in Razor Files

Create the Razor Test

- Using the rendered fragment, the results of the component rendering can be compared to the expected output
- The expected output is coded up with inline Razor syntax

```
// Assert
cut.MarkupMatches(@<header>
  <h1>Hello, World!</h1>
</header>);
```

- Rebuild the solution, it should still build even though there is no `<HelloWorld />` component
- Under the View menu, click "Test Explorer", and click the double green arrow to run all of the tests
- The one test, `HelloWorldComponentRendersCorrectly` should fail

Unit Tests in Razor Files

Create the Razor Component

- With the unit test written, let's program the `HelloWorld` component to pass the test
- Create a new Razor Component named `HelloWorld` in the Razor Component library project
- Replace the initial code in the file with this code:

```
<header>
  <h1>Hello, World!</h1>
</header>
```

- Using Test Explorer, run the unit test again, it should pass
- Congrats! You just did TDD!

Unit Tests in C# Files

Create C# Unit Test for a Razor Component

- Create a new C# Class file named `HelloWorldCSharpTest.cs`
- Update the class to be `public`
- Add the following code to the top of the class file

```
using Xunit;
using Bunit;
```

Unit Tests in C# Files

Create C# Unit Test for a Razor Component

- Add the following unit test method to the `HelloWorldCSharpTest` class

```
[Fact]
public void HelloWorldComponentRendersCorrectly()
{
  // Arrange

  // Act

  // Assert
}
```

Unit Tests in C# Files

Create C# Unit Test for a Razor Component

- Create the context as was done in the Razor Component test

```
// Arrange
using var ctx = new TestContext();
```

Unit Tests in C# Files

Create C# Unit Test for a Razor Component

- To render the component, the `RenderComponent` generic method is called, the generic type is the type of the `HelloWorld` Razor Component

```
// Act
var cut = ctx.RenderComponent<HelloWorld>();
```

Unit Tests in C# Files

Create C# Unit Test for a Razor Component

- The assertion is almost the same as the Razor Component test
- Instead of inline Razor syntax, the output is compared to a string

```
// Assert
cut.MarkupMatches("<header><h1>Hello, World!</h1></header>");
```

- With Test Explorer, run the tests again, they should both pass

Conclusion

What we've learned...

- Microsoft provides no official way to test Razor Components
- However, there is a 3rd party library named bUnit that works with existing unit testing frameworks (MS Test, NUnit, and xUnit) to test Razor Components
- Components can be tested using Razor syntax or plain C# code
- bUnit supports testing the HTML output from components
- Also, output from event handlers can be tested as well
- bUnit provides support for triggering events, triggering re-renders, etc...
- Using techniques and tools such as bUnit, TDD, and Test Explorer it is possible to have robust testing platform for Razor Components