Blazor Server

Testing your Blazor Application

2023

Introduction

- Testing is a very important aspect of writing applications.
- When we have good tests in place, we can produce higher quality applications, faster—Blazor applications are no exception.
- When it comes to testing Blazor apps, the three most common programmatic testing options are:
  - Unit Testing
  - Integration Testing
  - End-to-End Testing

Unit Tests

- Unit testing is the lowest level of testing we can utilize.
- When writing these types of tests, we focus on testing the smallest piece of functionality we can—such as a single method in a class.
- The three most common are xUnit (https://xunit.net), NUnit (https://nunit.org), and MSTest (http://mng.bz/2nea)

Introduction

- Integration testing is a level up from unit testing.
- In these tests, we combine several components of the system and test them together, checking that they integrate with each other correctly.
- These types of tests tend to use the same frameworks as unit tests, but as they are testing more complex scenarios, they can take longer to run than unit tests.
- An example of integration testing is a test that checks when data is posted to an API endpoint and saves it into the database.

Introduction

- End-to-end testing (E2E testing) is yet another level higher.
- These types of tests aim to exercise the entire system end to end, hence their name.
- When writing these tests for web applications, special frameworks are used to operate a headless browser.
- A headless browser is essentially a regular browser running without its UI.
- It is controlled programmatically, usually via the command line.

Introducing bUnit

- bUnit is a testing library specifically designed for Blazor.
- It was created and is maintained by Egil Hansen—a Microsoft MVP.
- It’s also supported by the .NET Foundation.

Adding a bUnit test project

- To install the bUnit project template, we can run the following .NET CLI command:

```
dotnet new --install bunit.template
```

- Now that we have the bUnit project template installed, we can then create a new project with it

```
dotnet new bunit -o app.Tests
```

- We'll install a package called AutoFixture (https://github.com/AutoFixture/AutoFixture). AutoFixture is useful when writing tests, as it can generate fake test data:

```
dotnet add package AutoFixture --version 4.18.0
```

Adding a bUnit test project

- The next step is to add some using statements to the root _Imports.razor file.
- These are going to help us a little later when writing our tests.
- Open the file and add the following lines:

```
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.Extensions.DependencyInjection
@using AutoFixture
@using Bunit.JSInterop.InvocationHandlers
@using MediatR
@using BlazingTrails.Client.State
@using BlazingTrails.Client.Features.Shared
```

Adding a bUnit test project

- Go to Tests Project and add references from App:

```
dotnet add reference ../app/app.csproj
```

Testing rendered markup

```
@inherits TestContext
@code{
    [Fact]
    public void RendersSucessAlert(){
    // Arrange
    string expectedMessage = "Your card has been edited successfully!";
    // Act
    var cut = Render(@<SuccessAlert Message=@expectedMessage />);
    // Assert
    cut.MarkupMatches(
    @<div class="alert alert-success" role="alert">
        <svg xmlns="http://www.w3.org/2000/svg" width="18" height="18"
        fill="currentColor" class="bi bi-check-circle-fill" viewBox="0 0 16 16">
        <path fill-rule="evenodd" d="M16 8A8 8 0 1 1 0 8a8 8 0 0 1 16
        0zm-3.97-3.03a.75.75 0 0 0-1.08.022L7.477 9.417 5.384 7.323a.75.75 0 0
        0-1.06 1.06L6.97 11.03a.75.75 0 0 0 1.079-.02l3.992-4.99a.75.75 0 0
        0-.01-1.05z" />
        </svg>
        Your card has been edited successfully!
    </div>
      );
    }
}
```

https://gist.github.com/brunobmo/2e4763cb1142bdb0082e17b07dd609bd

Testing rendered markup

- We first inherit from the TestContext base class provided by bUnit. This base class provides all the bUnit functionality from creating and rendering components under test to verifying the markup they produce.
- Then we have the _fixture field. This stores an instance of the Fixture class, provided by AutoFixture. We use this to generate test data in our tests. We could new this up in each test, but by doing it once here, it helps keep our test code clean by avoiding repetition.

Testing rendered markup

- We first inherit from the TestContext base class provided by bUnit.
- This base class provides all the bUnit functionality from creating and rendering components under test to verifying the markup they produce.

Testing rendered markup

- The test is set up to use the arrange, act, assert (AAA) pattern.
- This is a widely used testing pattern across the industry. It arranges a test into three steps:
  - Arrange—Creates or configures any objects that are used in the execution of the test
  - Act—Invokes the component or method under test
  - Assert—Verifies that the action of the component or method being tested produces the desired result

Testing rendered markup

- bUnit's MarkupMatches method uses semantic markup verification to confirm if the component has rendered the expected markup.
- Here we're using the templated Razor delegate syntax to define what we "expect the final markup to be. If the expected markup matches what the component rendered, the test will pass.

Blazor Server

Testing your Blazor Application

2023