Developing Cross-Platform Web Apps With Blazor

Wael Kdouh - @waelkdouh

Senior Customer Engineer

v1.0

Conditions and Terms of Use

Microsoft Confidential

This training package is proprietary and confidential, and is intended only for uses described in the training materials. Content and software is provided to you under a Non-Disclosure Agreement and cannot be distributed. Copying or disclosing all or any portion of the content and/or software included in such packages is strictly prohibited.

The contents of this package are for informational and training purposes only and are provided "as is" without warranty of any kind, whether express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Training package content, including URLs and other Internet Web site references, is subject to change without notice. Because Microsoft must respond to changing market conditions, the content should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Copyright and Trademarks

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

For more information, see Use of Microsoft Copyrighted Content at

http://www.microsoft.com/about/legal/permissions/

How to View This Presentation

- To switch to **Notes Page** view:
  - On the ribbon, click the **View** tab, and then click **Notes Page**

- To navigate through notes, use the Page Up and Page Down keys
  - Zoom in or zoom out, if required

- In the **Notes Page** view, you can:
  - Read any supporting text—now or after the delivery
  - Add notes to your copy of the presentation, if required

- Take the presentation files home with you

Module 11: Testing

Module Overview

Module 11: Testing

Section 1: Test Approaches

Lesson: Overview

How to Test a Blazor Component?

- To test a Blazor component, the Component Under Test (CUT) is:
  - Rendered with relevant input for the test
  - Depending on the type of test performed, possibly subject to interaction or modification. For example, event handlers can be triggered, such as an onclick event for a button
  - Inspected for expected values

Test Approaches

- Two common approaches for testing Blazor components are end-to-end (E2E) testing and unit testing:
  - **Unit testing**: Unit tests are written with a unit testing library that provides:
    - Component rendering
    - Inspection of component output and state
    - Triggering of event handlers and life cycle methods
    - Assertions that component behavior is correct

  bUnit is an example of a library that enables Razor component unit testing

  - **E2E testing**: A test runner runs a Blazor app containing the CUT and automates a browser instance. The testing tool inspects and interacts with the CUT through the browser. Selenium is an example of an E2E testing framework that can be used with Blazor apps

Test Approaches

- In unit testing, only the Blazor component (Razor/C#) is involved. External dependencies, such as services and JS interop, must be mocked

- In E2E testing, the Blazor component and all its auxiliary infrastructure are part of the test, including CSS, JS, and the DOM and browser APIs

Test Scope

- Test scope describes how extensive the tests are. Test scope typically has an influence on the speed of the tests

- Unit tests run on a subset of the app's subsystems and usually execute in milliseconds

- E2E tests, which test a broad group of the app's subsystems, can take several seconds to complete

Test Scope

- Unit testing also provides access to the instance of the CUT, allowing for inspection and verification of the component's internal state. This normally isn't possible in E2E testing

- With regard to the component's environment, E2E tests must make sure that the expected environmental state has been reached before verification starts. Otherwise, the result is unpredictable. In unit testing, the rendering of the CUT and the life cycle of the test are more integrated, which improves test stability

- E2E testing involves launching multiple processes, network and disk I/O, and other subsystem activity that often lead to poor test reliability. Unit tests are typically insulated from these sorts of issues

Test Scope

- The following table summarizes the difference between the two testing approaches.

Capability	Unit testing	E2E testing
Test scope	Blazor component (Razor/C#) only	Blazor component (Razor/C#) with CSS/JS
Test execution time	Milliseconds	Seconds
Access to the component instance	Yes	No
Sensitive to the environment	No	Yes
Reliability	More reliable	Less reliable

Choosing the Appropriate Test Approach

- Consider the scenario when choosing the type of testing to perform. Some considerations are described in the following table

Scenario	Suggested approach	Remarks
Component without JS interop logic	Unit testing	When there's no dependency on JS interop in a Blazor component, the component can be tested without access to JS or the DOM API. In this scenario, there are no disadvantages to choosing unit testing.
Component with simple JS interop logic	Unit testing	It's common for components to query the DOM or trigger animations through JS interop. Unit testing is usually preferred in this scenario, since it's straightforward to mock the JS interaction through the IJSRuntime interface.
Component that depends on complex JS code	Unit testing and separate JS testing	If a component uses JS interop to call a large or complex JS library but the interaction between the Blazor component and JS library is simple, then the best approach is likely to treat the component and JS library or code as two separate parts and test each individually. Test the Blazor component with a unit testing library, and test the JS with a JS testing library.
Component with logic that depends on JS manipulation of the browser DOM	E2E testing	When a component's functionality is dependent on JS and its manipulation of the DOM, verify both the JS and Blazor code together in an E2E test. This is the approach that the Blazor framework developers have taken with Blazor's browser rendering logic, which has tightly-coupled C# and JS code. The C# and JS code must work together to correctly render Blazor components in a browser.
Component that depends on 3rd party component library with hard-to-mock dependencies	E2E testing	When a component's functionality is dependent on a 3rd party component library that has hard-to-mock dependencies, such as JS interop, E2E testing might be the only option to test the component.

Module 11: Testing

Section 2: Unit Testing

Lesson: bUnit

Unit Testing Blazor Components

- There's no official Microsoft testing framework for Blazor, but the community-driven project bUnit provides a convenient way to unit test Blazor components

bUnit

- bUnit works with general-purpose testing frameworks, such as MSTest, NUnit, and xUnit
- These testing frameworks make bUnit tests look and feel like regular unit tests
- bUnit tests integrated with a general-purpose testing framework are ordinarily executed with:
  - Visual Studio's Test Explorer
  - dotnet test CLI command in a command shell
  - An automated DevOps testing pipeline

bUnit

- The following demonstrates the structure of a bUnit test on the Counter component in an app based on a Blazor project template. The Counter component displays and increments a counter based on the user selecting a button in the page:

razor

```razor
@page "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

bUnit

- The following actions take place at each step of the test:
  - **Arrange**: The Counter component is rendered using bUnit's TestContext. The CUT's paragraph element (<p>) is found and assigned to paraElm
  - **Act**: The button's element (<button>) is located and then selected by calling Click, which should increment the counter and update the content of the paragraph tag (<p>). The paragraph element text content is obtained by calling TextContent
  - **Assert**: MarkupMatches is called on the text content to verify that it matches the expected string, which is Current count: 1

C#

```
[Fact]
public void CounterShouldIncrementWhenSelected()
{
    // Arrange
    using var ctx = new TestContext();
    var cut = ctx.RenderComponent<Counter>();
    var paraElm = cut.Find("p");

    // Act
    cut.Find("button").Click();
    var paraElmText = paraElm.TextContent;

    // Assert
    paraElmText.MarkupMatches("Current count: 1");
}
```

ⓘ Note

The `MarkupMatches` assert method differs from a regular string comparison assertion (for example, `Assert.Equal("Current count: 1", paraElmText);`) `MarkupMatches` performs a semantic comparison of the input and expected HTML markup. A semantic comparison is aware of HTML semantics, meaning things like insignificant whitespace is ignored. This results in more stable tests. For more information, see **Customizing the Semantic HTML Comparison**.

Demo: BUnit

Module Summary

- In this module, you learned about:
  - Unit Testing with BUnit

References

- https://docs.microsoft.com/en-us/aspnet/core/blazor/test?view=aspnetcore-5.0


Microsoft