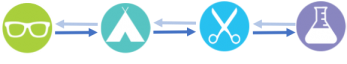
# Welke tooling en methodes kunnen er gebruikt worden van het schrijven van (automatische) tests?

## Doel

Het doel van dit onderzoek is het vinden van een geschikte manier voor het testen van onze eventuele toekomstige Blazor applicaties. Hierbij wordt er gekeken naar welke methodes en strategieën op de markt zijn gebracht voor automatiseren van tests. Hierbij wordt dus niet gekeken naar bijvoorbeeld end-to-end tests, acceptatietests of integratietests.

## Strategie

Ik ga dit onderzoek doen volgens ondertstaande onderzoeksmethodes

*F* *igure 1: Choose fitting pattern*

Dit betekend dat ik volgens de volgende volgorde ga werken:

1. Library: Literature study & community research

2. Field: Explore (user)requirements

3. Lab: Usability testing

4. Workshop: prototyping

# Uitwerking

## Welke technieken zijn beschikbaar?

Bij het onderzoek naar testmethodieken voor Blazor, ben ik steeds uitgekomen op twee belangrijke methodes: Unit testing en E2E testing (end-to-end). Twee totaal verschillende methodes die elkaar aan kunnen vullen en waarmee je een Blazor app op bijna elk vlak kan testen.

**Unit Tests**

Unit testing is de meeste bekende testmethode. Dit is een methode die, met enkele uitzonderingen, nagenoeg altijd automatisch wordt uitgevoerd. Toch is Unit testen in Blazor anders dan de gebruikelijk Unit Test. Over het algemeen wordt deze testmethodiek gebruik voor het testen van backends, toch kan deze ook gebruikt worden voor het testen van frontend, waar Blazor onder valt. Hiervoor wordt wel een speciale Unit Test library voor gebruikt. Deze libaries zorgen er voor dat de Blazor components geladen worden met de benodigde (Mock) data en de juiste state. Hierbij kunnen tests worden uitgevoerd op zowel de input, output, state, lifecycles, eventhandlers en meer. Dit is dan ook meteen het verschil met de conventionele unit tests, waar de conventionele unit tests alleen class-specifiek op een methode wordt uitgevoerd.

Er is eigenlijk maar een Unit Test library die genoemd wordt: bUnit.

**E2E**

Deze testmethodetest de workflow van een applicatie van begin tot eind en kan zowel manueel als automatisch worden gedraaid. De bedoeling van deze methode is het nabootsen van echte user input, zodat daadwerkelijke voorkomende scenario’s getest kunnen worden. Hierbij worden, in tegenstelling tot de Unit Tests, meerdere components getest.

Bij E2E tests zijn er meer mogelijkheden, zo worden bijvoorbeeld Cypress en Playwright for .NET erg vaak genoemd.

Microsoft heeft met onderstaande tabel in kaart proberen te brengen wat je voor kan stellen bij de specifieke tests.

| **Capability** | **Unit testing** | **E2E testing** |
| --- | --- | --- |
| Test scope | Razor component (Razor/C#) only | Razor 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 |

Het is niet de bedoeling om zomaar alles te testen met elke methode die er is. Zo is het verstandig om bij bepaalde scenario’s een keuze voor een bepaalde testaanpak te kiezen. Ook hier heeft Microsoft een interessante keuzetabel voor gemaakt.

| **Scenario** | **Suggested approach** | **Remarks** |
| --- | --- | --- |
| Component without JS interop logic | Unit testing | When there's no dependency on JS interop in a Razor 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](https://learn.microsoft.com/en-us/dotnet/api/microsoft.jsinterop.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 Razor 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 Razor 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 Razor components in a browser. |
| Component that depends on 3rd party class library with hard-to-mock dependencies | E2E testing | When a component's functionality is dependent on a 3rd party class library that has hard-to-mock dependencies, such as JS interop, E2E testing might be the only option to test the component. |

## Wat is de criteria van deze techniek?

De gekozen methodiek moet in elk geval zorgen voor een applicatie, waarvan de kwaliteit gegarandeerd is. Hiervoor wil ik gebruik gaan maken van zowel Unit tests als automatisch E2E tests. De keuze voor automatische E2E tests is voornamelijk de tijdsbesparing die gemoeid gaat met het laten uitvoeren van een constante test. Op dit moment doen wij veel E2E tests zelf handmatig, soms worden er dan bepaalde dingen over het hoofd gezien, iets wat moeilijker gaat bij een geautomatiseerde test. Het opzetten van de tests duurt natuurlijk langer, maar die tijd wordt snel ingehaald. Automatische E2E tests duren enkele seconden, waar handmatige E2E tests soms vele minuten kunnen duren.

## Uitwerking gekozen methodieken

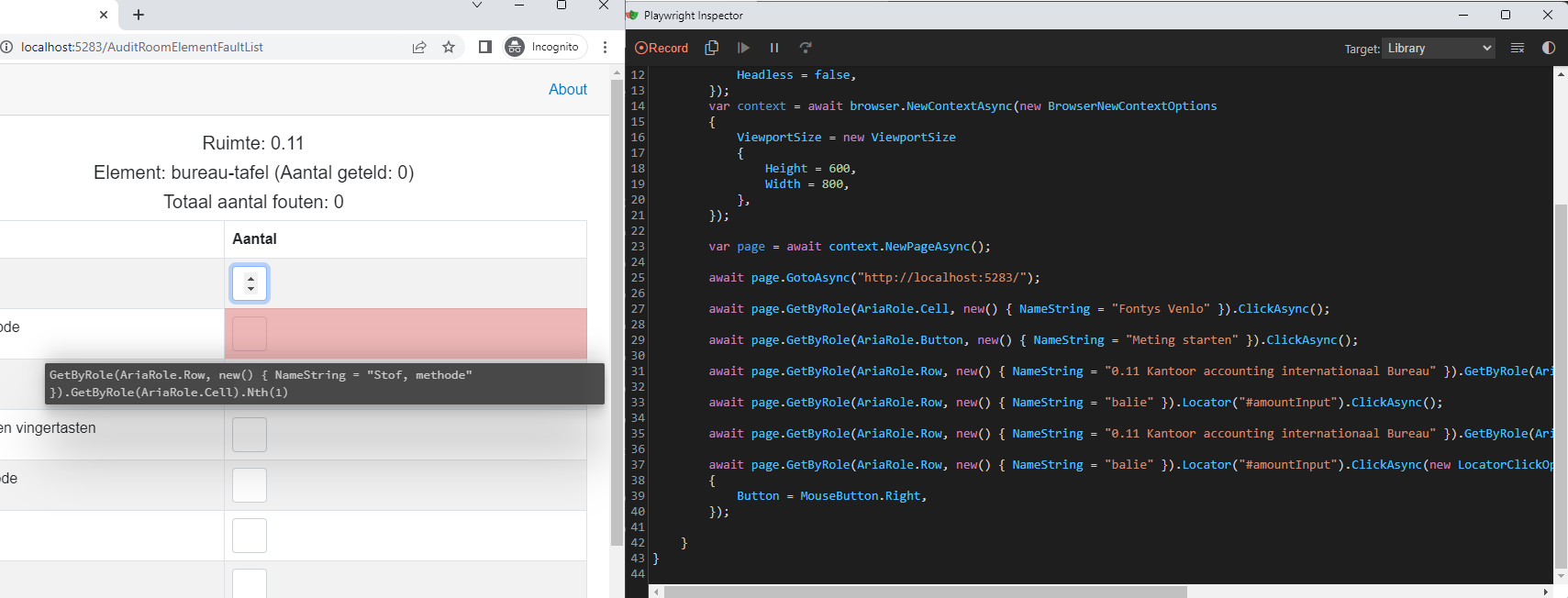
**bUnit**

De keuze voor het uitvoeren van de Unit tests is gevallen op bUnit. Wanneer je zoekt op Unit Tests voor Blazor leidt elke pagina naar bUnit. Ook is bUnit gesponsord door de .NET foundation (een door Microsoft opgerichte non-profit organisatie) en verwijst Microsoft zelf in de officiële Blazor guide ook naar bUnit. Dit samen met het feit dat Microsoft ook nog eens de bedenker is van Blazor zelf, lijkt mij dit een goed onderbouwde keuze.

Bij bUnit kan er gekozen worden uit het schrijven van tests in de .cs of de .blazor file. Bij het schrijven van tests in de .blazor file is het makkelijker om HTML markup te gebruiken en is hier om ook de meest gekozen methode.

**Playwright for .NET**

Voor het uitvoeren van de E2E tests is de keuze gevallen op Playwright. Playwright is een door Microsoft ontwikkelde cross-browser automatische E2E testing library. Voor Playwright bestaan handige tools, zoals de Codegen, waarbij een handmatige test door middel van een Record knop omgezet kan worden in een uitvoerbare test in het test framework naar keuze. Daarbij voorzien ze in een erg goed gedocumenteerde website, waarbij erg veel tutorials en documentatie te vinden is. Het enige nadeel zou zijn, dat in tegenstelling tot de Unittests, de tests per stuk enkele seconden duren. De browser wordt namelijk virtueel opgestart, waarbij de tests door de emulator worden uitgevoerd.



Playwright Codegen tool

## Conclusie

**bUnit**

Ik heb het idee dat bUnit, wanneer er ook gebruik gemaakt wordt van een automatische E2E test, weinig toevoeging heeft. Het enige voordeel wat ik kan bedenken is dat de te gebruiken data makkelijk te managen en te mocken is, waar er bij Playwright gebruik wordt gemaakt van een draaiende instantie van de app. Het grootste nadeel is, wat Microsoft zelf ook zegt, ***“When a component's functionality is dependent on a 3rd party class library that has hard-to-mock dependencies, such as JS interop, E2E testing might be the only option to test the component.”.*** En aangezien mijn applicatie veel gebruik maak van JSInterop, zal het lastig worden bUnit volledig te kunnen benutten.

**Playwright for .NET**

Binnen Factos maken wij bij front-end alleen gebruik van handmatige front-end tests. Hierbij is vooral Playwright een hele goede en tijdbesparende aanvulling. Wel moet ik er bij zeggen dat de leercurve erg hoog is, en het veel tijd kost om de vele mogleijkheden door te krijgen en volledig te kunnen benutten. Ik denk dat het een onderzoek op zich zal zijn om alle mogelijkheden uit te kunnen proberen ten te snappen. Het nadeel van Playwroght is dat de data lastiger te mocken is ddan met bUnit, waar er bij deze methode een draaiende instantie van de app moet draaien, waar Playwright door middel van een emulator zijn tests op gaat uitvoeren. Wel merk ik dat wanneer het eenmaal staat het een erg mooi testmiddel is.

Een andere conclusie is dat ik gemerkt heb, dat wanneer je deze methode gebruikt, het een groot voordeel is dat je werkt volgens een test-driven ontwikkelmethode. Wanneer de app eenmaal staat, en je de tests nog moet inbouwen, merk je dat je veel zaken (zoals id’s, extra classes etc), nog moet toevoegen succesvol te kunnen testen. Ook merk je dat sommige functionaliteiten beter zouden werken als je test-driven ontwikkeld. Deze conslusie is zeker iets om over na te denken bij eventueel maken van Blazor apps in de toekomst.

## Bronnen

<https://code-maze.com/test-blazor-webassembly-bunit/>

<https://learn.microsoft.com/en-us/aspnet/core/blazor/test?view=aspnetcore-7.0>

<https://www.browserstack.com/guide/end-to-end-testing#:~:text=End%20to%20end%20testing%20(E2E,for%20integration%20and%20data%20integrity>.

<https://bunit.dev/docs/getting-started/writing-tests.html?tabs=xunit>

<https://medium.com/younited-tech-blog/end-to-end-test-a-blazor-app-with-playwright-part-2-3980b573e92a>