Playwright Demo

IMPORTANT: You are not expected to finish every scenario. Everyone has different levels of experience and can work at different speeds. What is important is that you learn something new. Work to your own pace to maximise your learning

IMPORTANT: At any time you need a hand just remember you and your team/table are in the same boat so ask others for help. If still stuck, raise your hand and the workshop host will endeavour to help as soon as possible.

Key Objectives

- Show how End-to-End tests can be shifted left into Social tests
- Understand cucumber and cucumber scenario outlines
- Run specific scenario using cucumber tags and understand command line output
- Show cucumber report success and failure with screenshots
- Ability to pause
- Use of playwright object when pausing
- Overview of API intercepting

Pre-requisites

We assume you have the following installed and have some familiarity with your Integrated Development Environment (IDE). If you do not then please install them or ask the host for assistance

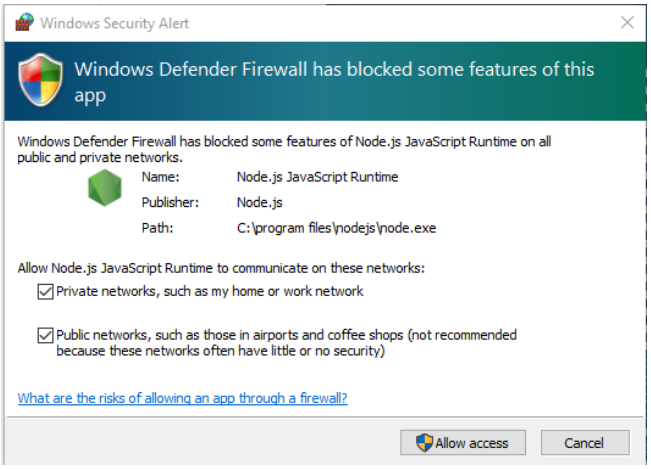
- Git
- Node LTS 18+
- Powershell
- Have run on powershell to avoid issues with Visa re-writing certificate authorities: *Set-Item -Path Env:NODE_TLS_REJECT_UNAUTHORIZED -Value "0"*
- Have one of the following IDEs installed
    - Visual Code - https://code.visualstudio.com/download
    - Intellij - https://www.jetbrains.com/idea/download/#section=linux
    - Webstorm - https://www.jetbrains.com/webstorm/download/#section=linux

Checkout the codebase

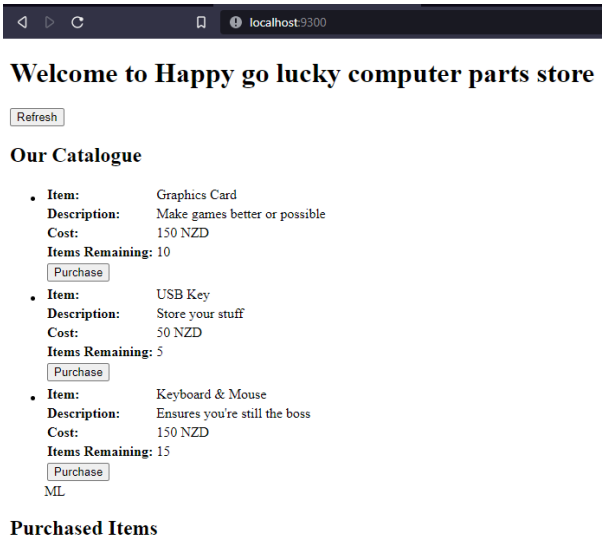
1. Open powershell
2. Go to the home directory. Run *cd ~*
3. Create a place for code to live. Run *mkdir -p Workspace*
4. Checkout code. Run *git clone https://github.com/davinryan/playwright.git*
5. Get into the repo. Run *cd playwright*

Run the application

1. Install 3rd party dependencies. Run *npm install*
2. Start application. Run *npm run start:standalone*
   If you see this prompt then select *Allow access*



3. Open your Chrome browser and *navigate to http://localhost:9300/*
   Confirm you can see this page



Run the playwright test suite - no tests

1. Open another powershell window
2. Go to the project directory. Run *cd ~/Workspace/playwright*

3. Checkout out the branch we are going to work from for the demo. Run *git checkout workshop*
4. Setup Social testing framework. Run *npm run test:social:init*
5. Run Social testing framework. Run *npm run test:social*
   You should get the output
   0 scenarios
   0 steps
   0m01.897s (executing steps: 0m00.000s)
6. Our job is to write some test scenarios for the purchase feature
    a. Can make a purchase (with mocked Catalogue API Edge)
    b. Can make a purchase (with mocked Catalogue API response from file)
    c. Can make a purchase (with mocked Catalogue API response using builder pattern)
    d. Show a useful error to customer when purchase function not available

Test Purchase feature

For the following scenarios you are going to need to open your favourite code editor and open the project at path ~/Workspace/playwright

Take time to explore the code base. Points of interest are

src/demo-reference-architecture.tsx	Micro-frontend mounting point
src/containers/catalogue.tsx	Catalogue of items users can buy (handes API call to get items
src/services/mock/catalogue/catalogue.server.ts	Mocked Edge for Catalogue micro-service which returns a list of items a user can purchase. Used for running and testing locally
src/services/mock/catalogue/catalogue.data.items.standard.json	Mocked list of standard items that the mocked catalogue micro-service will return
src/components/catalogueItems.tsx	Dumb component that just displays list of catalogue items passed to it
src/components/catalogueItem.tsx	Dumb component that displays a single catalogue item and its purchase function
src/components/purchaseSummary.tsx	Dumb component that displays list of messages indicating what catalogue items a user has purchased
src/components/purchaseItem.tsx	Dumb component that displays message for the purchase of a single catalogue item

Take time to explore the Cucumber and Playwright framework in src/socialTests/business

features/	Folder for all test scenarios and associated code implementations to simulate the user
hooks/	Location of tasks that happen before and after all or each scenario runs. E.g. handles getting screenshots for failures
reporters/	Contains code for reporters that will create a nice audit report or summary of results
sections/	Page objects (actually section objects) that encapsulate common user behaviour and actions for specific sections e.g. catalogueList
types/	Typescript types to support testing
util/	Important help functions to reduce duplication and make features implementations easier to read
world/	Cucumber constraint for common or shared functionality. E.g. This contains the code that will launch the browser and make playwright accessible to all scenarios

With mocked Catalogue API Edge

WARNING: if you are using VisualCode make sure you save after making changes as auto-save is not on by default

Purpose: Test we can fundamentally purchase something

1. Create the file *src/socialTests/business/features/purchase.feature*
2. Paste the following code

```
Business Need: Purchase from Catalogue

 Background: User is logged in and ready to purchase something

 @social
 Scenario Outline: PURCHASE01 - Purchase an item from the catalogue
   Given customer:1 is logged and on the catalogue screen
   When customer:1 purchases <item>
   Then the users is shown the purchase message <purchaseMessage>
   Examples:
     | item                 | purchaseMessage                          |
     | "Graphics Card"      | "You have purchased 'Graphics Card'"     |
     | "USB Key"            | "You have purchased 'USB Key'"           |
     | "Keyboard & Mouse"   | "You have purchased 'Keyboard & Mouse'" |
```

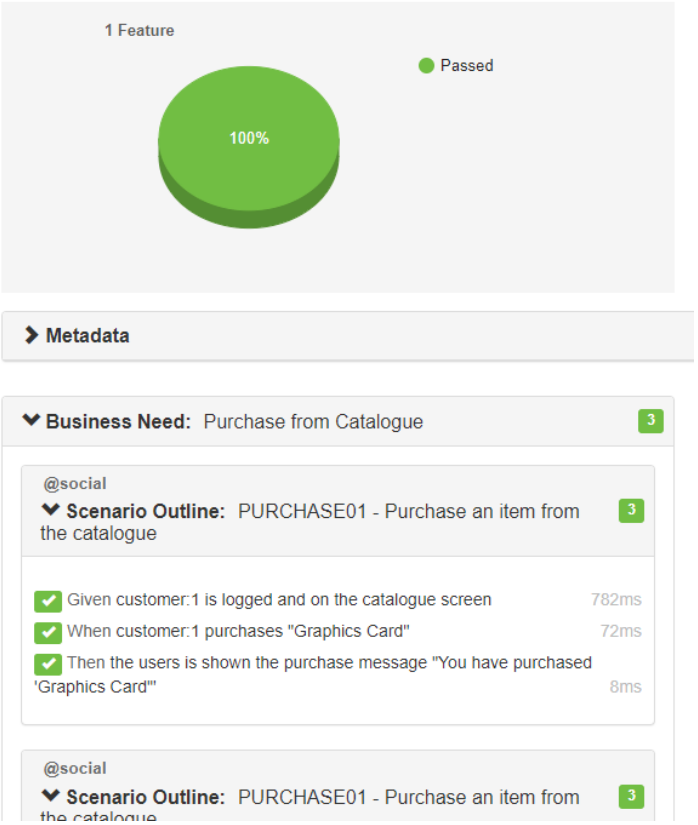
3. This is our simplest but most important scenario. It could be one you would use as our happy case End-to-End test but in this workshop we'll implement it as a social test in this code base. What is happening here?
    a. The first Given statement takes the user to the purchase page
    b. The When statement simulates the user purchasing an item
    c. The Then statement checks that the item was successfully purchased
4. Things to note
    a. Look at the Examples section. This scenario will run 3 times. Each time it will inject different values for item and purchaseMessage. The first time Item will equal "Graphics Card" and purchaseMessage will equal "You have purchased 'Graphics Card'"
5. Let's see if this scenario works. In the second powershell window run *npm run test:social*
6. You will notice that the Given, When and Then statements have not been implemented so you'll get a recommendation printed out in the console for what they might look like. We won't use these so please ignore them for now. Move onto the next step
7. Create the file *src/socialTests/business/features/purchase.steps.ts*
8. Paste the following code

```
import {Given, Then, When} from '@cucumber/cucumber'
import {PageUrls} from '../types/types.urls'
import {catalogueNameIdMap} from '../util/mappers'
import {expect} from '@playwright/test'

Given('customer:{int} is logged and on the catalogue screen', async function
(customerId: number) {
 await this.createBrowserSession(PageUrls.PURCHASE)
})

When('customer:{int} purchases {string}', async function (customerId: number,
itemName: string) {
 await
this.page.getByTestId(`catalogueItem_purchase_${catalogueNameIdMap[itemName.t
rim()]}`).click()
})

Then('the users is shown the purchase message {string}', async function
(purchaseMessage: string) {
 const actualMessage = this.page.getByText(purchaseMessage.trim())
 await expect(actualMessage).toContainText(purchaseMessage.trim())
})
```

9. What's going on
    a. The Given function is going to call the *this.createBrowserSession(PageUrls.PURCHASE)* which creates a playwright session and navigates to the Purchase page
    b. The When function is going to use *this.page.getByTestId* to find the catalogue item we want to purchase

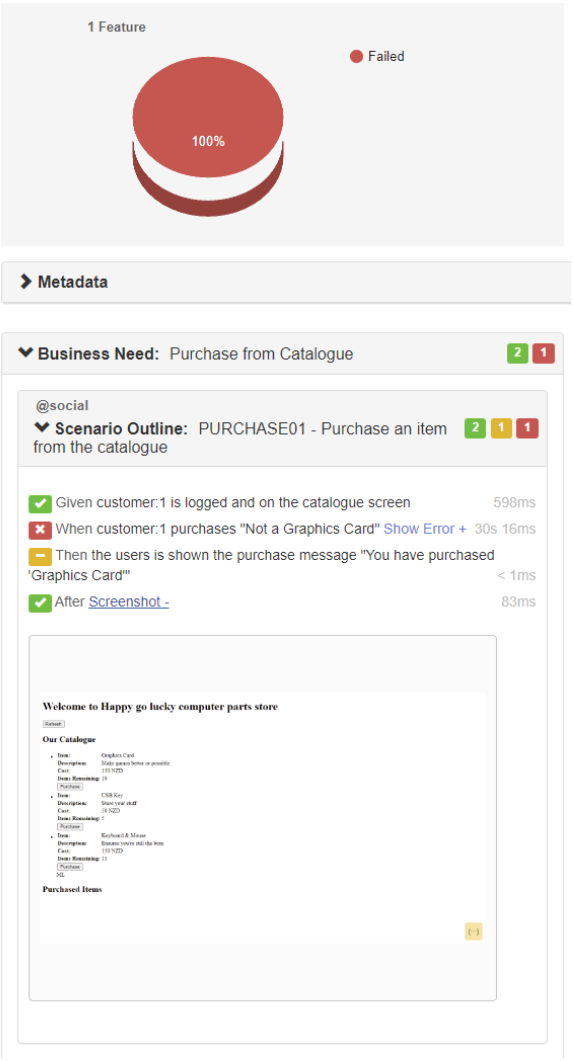
c. The Then statement is going to look up the purchase message with *this.page.getByText* to find the purchase message and make sure it contains *purchaseMessage*

10. Things to note
    a. Injected values like ‘item’ and ‘purchaseMessage’ from purchase.feature’s Example section are passed in to each Given/When/Then function as parameters that we can use
    b. catalogueNameIdMap simply maps and catalogue plain text name to a backend unique identifier that is used to find the item on the screen
11. Let’s see if this scenario works. In the second powershell window run *npm run test:social*
12. You should have the following output
    3 scenarios (3 passed)
    9 steps (9 passed)
13. Let’s look at a report. Run *npm run test:social:report*
14. You should see something like this



15. Congratulations. Here are some important points to reflect on and bonus material
    a. You simulated the catalogue micro-service using the src/services/mock/catalogue/catalogue.server.ts. This is useful for getting things running locally but creates brittle tests when this mock data changes. When we do the next examples you’ll see how we can use playwright to mock in real time and keep the mocked data in the testing code not in the Mock
    b. You used cucumber as a test runner that runs the scenario three times with injected data that used playwright to simulate the user using the browser and then created a report

c. If you want to simulate a failure try changing the Examples section of purchase.feature to include items that do not exist and re-run the tests. You'll find that it will take 30 seconds (30000 milliseconds) to timeout and if you generate another report you should see a screenshot embedded in the report like this. You can change the timeout in hooks/hooks.common.ts but updating the value in milliseconds passed to the *setDefaultTimeout* function



With different Catalogue API response from file

Purpose: Test in a way where we can decide, using Playwright, what data our micro-services will respond with in order to test more interesting scenarios not covered by the Mock service and make our tests less brittle (they no longer rely on the standard and limited Mock Catalogue micro-service)

1. Add the following code to the file *src/socialTests/business/features/purchase.feature* after the previously added scenario PURCHASE01

```
@social
@thisOneOnly
Scenario Outline: PURCHASE02 - Purchase an item from the school holidays
catalogue
 Given customer:1 is logged and on the catalogue screen
 And available catalogue items are from the school holidays collection
 When customer:1 purchases <item>
 Then the users is shown the purchase message <purchaseMessage>
 Examples:
   | item                | purchaseMessage                          |
   | "iPhone 20"         | "You have purchased 'iPhone 20'"         |
   | "Dell XPS 7300"     | "Dell XPS 7300'"                         |
   | "External drive 1TB" | "You have purchased 'External drive 1TB'" |
```

2. What is new here compared to the last feature. We've added
    a. *And available catalogue items are from the school holidays collection* - this line will tell Playwright to look for calls to the Catalogue micro-service and replace them with a response that includes school holiday catalogue items
3. Let's implement that new step. Paste the following after the last import in purchase.steps.ts

```
import {join} from 'path'
import {setAPIMockResponseFromPath} from "../util/util.api";
```

4. Paste the following into purchase.steps.ts under the last Given step function

```
Given('available catalogue items are from the school holidays collection',
async function () {
 await setAPIMockResponseFromPath(this.page, '**/catalogue/items',
join(__dirname, 'purchase.data.summerCollection.json'))
 await this.page.getByTestId('catalogue_refresh').click()
})
```

5. Create the file *src/socialTests/business/features/purchase.data.summerCollection.json* and past the following into it

```
{
 "total": 3,
 "page": 1,
 "limit": 200,
 "data": [
   {
     "id": "4",
```

```
      "name": "iPhone 20",
      "price": {
        "value": 1200,
        "currency": "NZD"
      },
      "description": "Great for being connected... and anti-social",
      "quantity": 10,
      "attributes": [
        {
          "memory": "1GB",
          "model": "20"
        }
      ]
    },
    {
      "id": "5",
      "name": "Dell XPS 7300",
      "price": {
        "value": 3500,
        "currency": "NZD"
      },
      "description": "Participate in society",
      "quantity": 5,
      "attributes": [
        {
          "memory": "32GB",
          "cpu": "12 Gen Intel Core i7"
        }
      ]
    },
    {
      "id": "6",
      "name": "External drive 1TB",
      "price": {
        "value": 150,
        "currency": "NZD"
      },
      "description": "Store your stuff",
      "quantity": 15,
      "attributes": [
        {
          "size": "1TB",
          "writeSpeed": "1 MB/s",
          "readSpeed": "1 MB/s"
        }
      ]
    }
  ]
}
```


6. Let's see if this scenario works. In the second powershell window run *npm run test:social -- --tags "@thisOneOnly"*
7. You should have got the following output

3 scenarios (3 passed)
12 steps (12 passed)

8. Congratulations! Things to note
    a. We used --tags “@thisOneOnly” when we ran the tests. This allows us to only run scenarios that have this tag
9. Now there’s a problem here. Do we really want to maintain lots and lots of JSON files with mocked data? How can we improve on this by centralising it so we make changes in one place. This is what the next scenario will answer…

With different Catalogue API response using builder pattern

Purpose: Same as before but let's test in a way that reduces the maintenance of mocked data

1. In purchase.feature please remove any references to the tag @thisOneOnly as we want to use this to run our next scenario only
2. Add the following code to the file *src/socialTests/business/features/purchase.feature* after the previously added scenario PURCHASE02

```
@social
@thisOneOnly
Scenario Outline: PURCHASE03 - Purchase an item from the school holidays
catalogue
 Given customer:1 is logged and on the catalogue screen
 And available catalogue items are from the school holidays collection using
builder pattern
 When customer:1 purchases <item>
 Then the users is shown the purchase message <purchaseMessage>
 Examples:
   | item                 | purchaseMessage                          |
   | "iPhone 20"          | "You have purchased 'iPhone 20'"         |
   | "Dell XPS 7300"      | "Dell XPS 7300'"                         |
   | "External drive 1TB" | "You have purchased 'External drive 1TB'" |
```

3. What is new here compared to the last feature. We've added
    a. *And available catalogue items are from the school holidays collection using builder pattern* - (you wouldn't have the phrase *using builder pattern* in your steps but we are using this for this workshop to make a point) this line will tell Playwright to look for calls to the Catalogue micro-service and replace them with a response that includes school holiday catalogue items but with a builder (no need to use a JSON file)
4. Let's implement that new step. Paste the following after the last import in purchase.steps.ts.

```
import {CatalogueBuilder} from "../util/CatalogueBuilder";
import {setAPIMockResponse, setAPIMockResponseFromPath} from
"../util/util.api";
```

5. Delete this line in purchase.steps.ts

```
import {setAPIMockResponseFromPath} from "../util/util.api";
```

6. Paste the following into purchase.steps.ts under the last Given step function

```
Given('available catalogue items are from the school holidays collection
using builder pattern', async function () {
 await setAPIMockResponse(this.page, '**/catalogue/items',
CatalogueBuilder.make()
   .addItem('4', 'iPhone 20', 1200, 'NZD', 'Great for being connected... and
anti-social', 10)
   .addItem('5', 'Dell XPS 7300', 3500, 'NZD', 'Participate in society', 5)
   .addItem('6', 'External drive 1TB', 150, 'NZD', 'Store your stuff', 15)
```

```
    .build())
  await this.page.getByTestId('catalogue_refresh').click()
})
```

7. Let's see if this scenario works. In the second powershell window run *npm run test:social -- --tags "@thisOneOnly"*
8. You should have got the following output
   3 scenarios (3 passed)
   12 steps (12 passed)
9. Congratulations! Notice how we used the builder to create the mocked data we want to return instead of getting data from a JSON file

When server returns a HTTP 500 error

Purpose: How to use Playwright to simulate a server error coming back from the Catalogue micro-service

1. In purchase.feature please remove any references to the tag @thisOneOnly as we want to use this to run our next scenario only
2. Add the following code to the file *src/socialTests/business/features/purchase.feature* after the previously added scenario PURCHASE03

```
@social
@thisOneOnly
Scenario Outline: PURCHASE04 - Fail to refresh catalogue
 Given customer:1 is logged and on the catalogue screen
 When no catalogue items are available due to a server outage
 Then the user is presented with the outage error <outageError>
 Examples:
   | outageError                                                    |
   | "There is currently an outage. Please try again in 10 minutes." |
```

3. What is new here compared to the last feature. We've added
    a. *When no catalogue items are available due to a server outage* - this line will tell Playwright to look for calls to the Catalogue micro-service and replace them with a failure response with HTTP code 500
4. Let's implement that new step. Paste the following into purchase.steps.ts under the last When step function

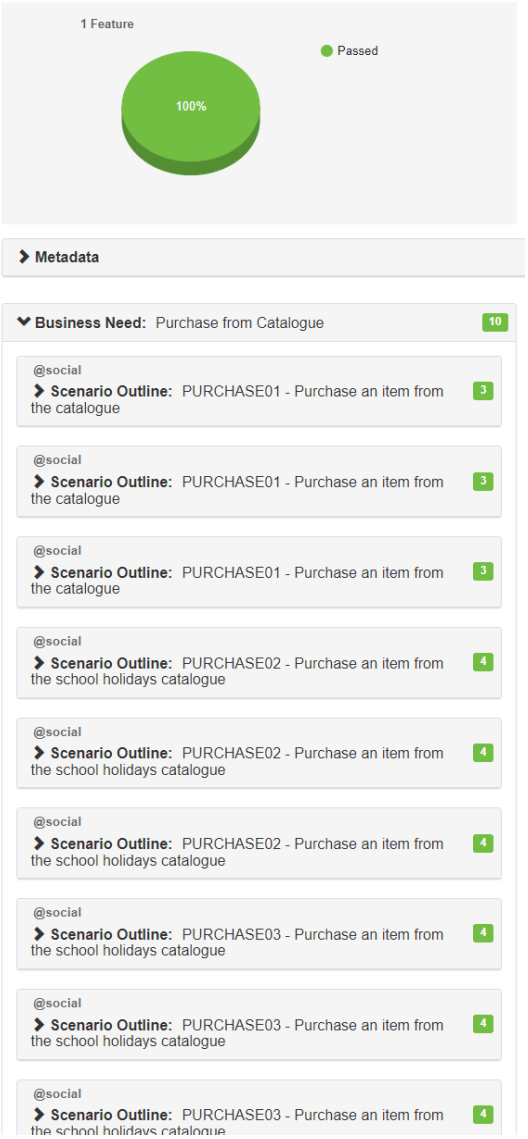
```
When('no catalogue items are available due to a server outage', async
function () {
 await setAPIMockResponse(this.page, '**/catalogue/items',
CatalogueBuilder.make()
   .hasServer500Error()
   .build(), 500)
 await this.page.getByTestId('catalogue_refresh').click()
});
```

5. Paste the following into purchase.steps.ts under the last Then step function

```
Then('the user is presented with the outage error {string}', async function
(outageError: string) {
 const errorElement = this.page.getByTestId('catalogueItems_outageError')
 await expect(errorElement).toHaveText(outageError.trim())
});
```

6. Let's see if this scenario works. In the second powershell window run *npm run test:social -- --tags "@thisOneOnly"*
7. You should have got the following output
   1 scenario (1 passed)
   3 steps (3 passed)
8. Congratulations! Notice how we used *setAPIMockResponse* to send a server error with an HTTP status of 500

9. For a bonus try removing the tag @thisOneOnly and run all the tests with *npm run test:social*
10. You Should get the output
    10 scenarios (10 passed)
    36 steps (36 passed)
11. Run the report again by running *npm run test:social:report*



12. This is the end of the workshop