ITERA

Playwright introduction

Let's begin with the introduction now that you have successfully installed Playwright. If you have not, look in the installation folder first – we have created an installation guide for Playwright. This introduction will be using VS Code to demonstrate Playwright, but feel free to use whichever IDE you prefer. We only need IDE and a terminal 😊.

The folder structure should look something like this when you open the folder in your IDE:



Go to the 'playwright.config.ts' and navigate down to line 34:

```
33    /* Configure projects for major browsers */
34    projects: [
35      {
36        name: 'chromium',
37        use: { ...devices['Desktop Chrome'] },
38      },
39
40      {
41        name: 'firefox',
42        use: { ...devices['Desktop Firefox'] },
43      },
44
45      {
46        name: 'webkit',
47        use: { ...devices['Desktop Safari'] },
48      },
```

The project configuration dictates which browsers will be used when a test is ran. So if you want to only run one testcase, there will be three test-runs – one for each browser. **Comment out the ones that you don't want**. This introduction will only use chromium as its browser.

Additionally, **line 15** in the config file dictates whether Playwright will parallel run tests for you:

```ts
import { defineConfig, devices } from '@playwright/test';

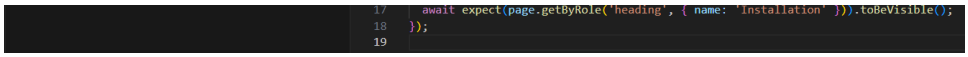
/**
 * Read environment variables from file.
 * https://github.com/motdotla/dotenv
 */
// require('dotenv').config();

/**
 * See https://playwright.dev/docs/test-configuration.
 */
export default defineConfig({
  testDir: './tests',
  /* Run tests in files in parallel */
  fullyParallel: true,
  /* Fail the build on CI if you accidentally left test.only in the sour
  forbidOnly: !!process.env.CI,
  /* Retry on CI only */
  retries: process.env.CI ? 2 : 0,
  /* Opt out of parallel tests on CI. */
  workers: process.env.CI ? 1 : undefined,
  /* Reporter to use. See https://playwright.dev/docs/test-reporters */
```

This means that if your testsuite includes 2-3 tests, Playwright will open up several browsers and execute them in the same time. Use this option at your preferance.

Now we are done with the config file (however, feel free to play around some more).

Navigate to the ‘tests’ / ‘test-examples’ folder, Playwright has provided some example files for us. Feel free to read through them before creating our own first test:



You can execute the tests in several ways, we recommend looking at the official Playwright documentation: Running and debugging tests | Playwright

Creating our very first test

In the ‘tests’ folder, create your own file that will include your first playwright test. The file **must end in .spec.ts** for Playwright to recognize it as a test-file. Copy the following line and paste it on the very top on your file:

```ts
import { test, expect, type Page } from '@playwright/test';
```

In Playwright, we use `test.describe` to organize our test-suites which groups several test cases that test the same area / functionality. **It serves no purpose other than visual grouping**. For example:

```
test.describe('testing the kitchen', async() => {
    test('makes sure that the oven works', async({ page }) =>{

    })

    test('makes sure that stove top works', async({ page })  => {

    })

    test('makes sure the faucet works', async({ page })  => {

    })
})
```

Notice that we're using arrow functions here, the is the 'default' way of writing Playwright tests and the most readable. We also have to insert the Page object as a parameter to the test-case, in order for the browser to perform our commands during execution (test.describe does not require this, since describe does not interact with anything)

For our first test-suite, we want to test a factorial calculator by visiting the URL https://qainterview.pythonanywhere.com/, enter some values, click a button, and assert the expected result against actual.

```
test.describe('testing factorial calculator', async() => {

})
```

How many test-cases you produce in this test-suite is up to you. But we should ensure that the functionality behaves as expected and handles any illegal inputs in a good way. We may produce two test-cases:

```
test.describe('testing factorial calculator', () => {
    test('tests with valid integer 4', async({ page }) => {

    })

    test(tests with illegal input',  async({ page })  => {

    })
})
```

To visit the URL, we will use [await page.goto()]. **However,** instead of writing the command in both test-cases:

```
test.describe('testing factorial calculator', async () => {
    test('tests with valid integer 4', async({page}) => {
        await page.goto('https://qainterview.pythonanywhere.com/');
    })

    test('tests with illegal input', async({page}) => {
        await page.goto('https://qainterview.pythonanywhere.com/');
    })
 })
```

We can use [hooks]. Hooks are helpful in setting the conditions prior to your tests

- *beforeEach()* runs before each test is executed
- *beforeAll()* runs once before all tests are executed
- *afterEach()* runs after each test is executed
- *afterAll()* runs once after all tests are executed

We can thus simplify our test-cases:

```
test.describe('testing factorial calculator', async () => {
    test.beforeEach(async ({ page }) => {
        await page.goto('https://qainterview.pythonanywhere.com/');
    })

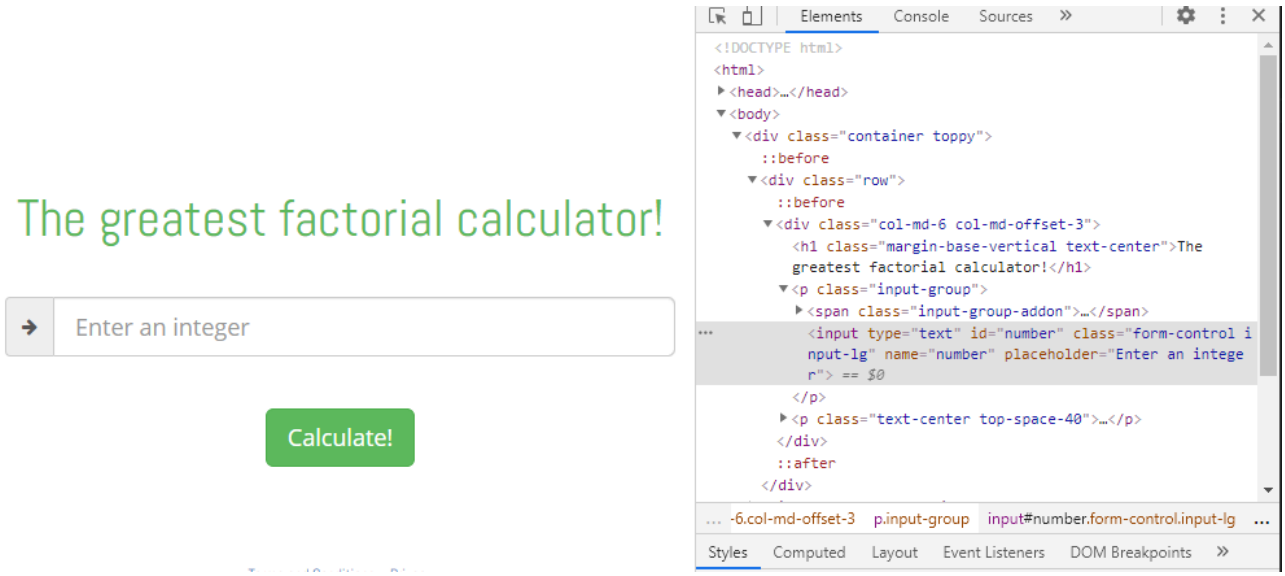
    test('tests with valid integer 4', async({page}) => {

    })

    test('tests with illegal input', async({page}) => {

    })
 })
```

Now we need make Playwright select the input field before we actually send a value, we'll use [locators] along with a selector. The simplest way is to right click on the input field and inspect its HTML elements:



There are several ways we can make Playwright select the input field:

- Through the id
- Through the class (although this is not recommended, as classes are not unique per element, and elements can change class during development)

- Through the input type
- Etc.

We then chain .fill() to inject a value into the targeted selector:

```
test('tests with valid integer 4', async({page}) => {
    //Using one of these is enough:

    //Select by id:
    await page.locator('#number').fill('4');

    //Select by class:
    await page.locator('.form-control').fill('4');

    //Select by element:
    await page.locator('input').fill('4');
})
```

It is worth noting that **every command is inherently an assertion**. **.goto()** will fail the test if the input URL isn't responding after some certain time, **.locator()** will fail if no elements are found.

So how do we know if the output value the website gives is correct? Playwright offers a lot of assertion methods. We can use an **Expect** and **toContainText** to assert the outcome:

```
test('tests with valid integer 4', async({page}) => {
    await page.locator('#number').fill('4');
```

```
        await page.locator('.btn').click();

        await expect(page.locator('#resultDiv')).toContainText('The factorial of 4
is: 24');
    })
```

Playwright will get the div containing the result text, then assert it against the .toContainText that you have put in.

Since Playwright is a JavaScript/Typescript testing framework, it is still very much a programming language, meaning you can create variables, functions, and check assertions with string interpolations etc.

```
    test('tests with valid integer 4', async({page}) => {
        let input = '4';
        let expectedAnswer = '24';

        await page.locator('#number').fill(input);
        await page.locator('.btn').click();

        await expect(page.locator('#resultDiv'))
            .toContainText(`The factorial of ${input} is: ${expectedAnswer}`);
    })
```

Now that you've got the basics, it's time for you to practice, whether it be expanding the test-suite with negative tests or find other ways of testing the same functionality. We encourage you to use the official Playwright documentation as much as possible.