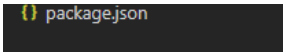
ITERA

Cypress introduction

Now that you have successfully installed Cypress and opened Cypress using npx, let's begin with the introduction. Open the Cypress folder in your favorite IDE, it should look something like this →



The sample tests you see when Cypress is running, are the same files in the folder e2e→ 1-getting-started and 2-advanced-examples. All these files contain useful commands you can use (especially actions.cy.js!)

Let's make our very first test, by making a new file under the 1-getting-started-folder, call it whatever you like, as long as it is a JavaScript file. Remember to add

```
/// <reference types="Cypress" />
```

On the top of your file in order to activate the Intelligent Code Completion.

In Cypress, we use `describe` in order to describe a test-suite which groups several test-cases that tests the same 'element'. Each test-case is described with `it` that resides in a test-suite. (Alternatively, you can use `context` and `specify` instead. They are identical to the prior two, the only difference being readability).

For example:

```
describe('testing the kitchen', () => {
    it('makes sure that the oven works', () =>{

    })

    it('makes sure that stove top works', () => {

    })

    it('makes sure the faucet works', () => {
```

```
    })
})
```

Notice that we're using arrow functions here, the is the 'default' way of writing Cypress tests and the most readable.

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.

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

})
```

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:

```
describe('testing factorial calculator', () => {
    it('tests with valid integer 4', () => {

    })

    it('tests with illegal input', () => {

    })
})
```

In order to visit the URL, we will use Cy.visit(). **However,** instead of writing the command in both test-cases:

```
describe('testing factorial calculator', () => {
    it('tests with valid integer 4', () => {
        cy.visit('https://qainterview.pythonanywhere.com/');
    })

    it('tests with illegal input', () => {
        cy.visit('https://qainterview.pythonanywhere.com/');
    })
})
```

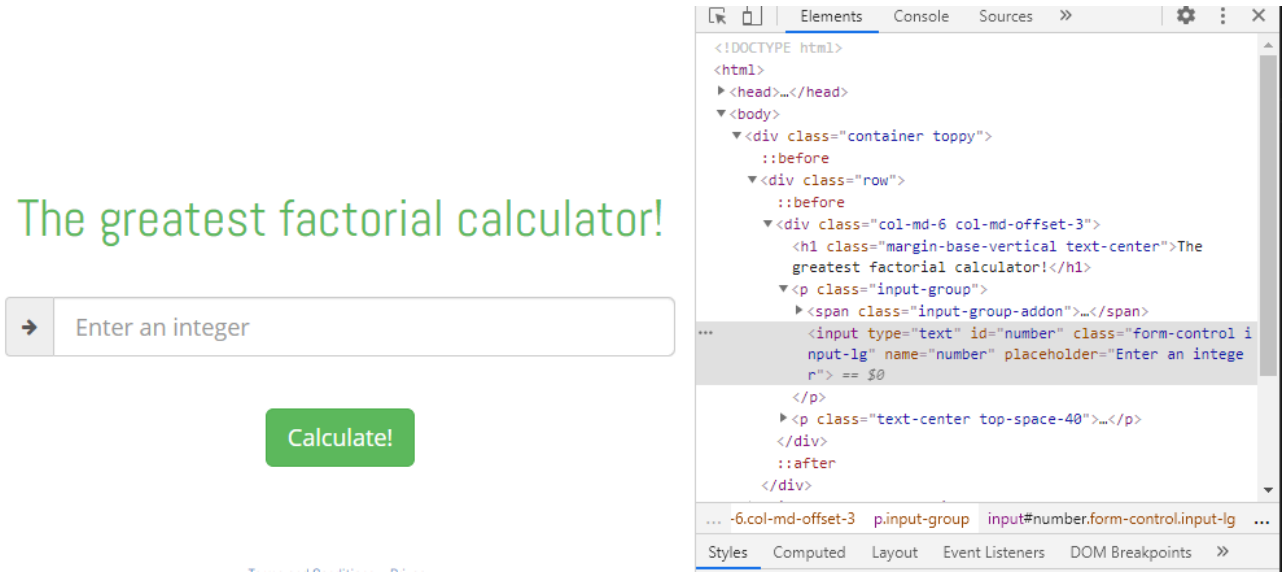
We can use Hooks. Hooks are helpful in setting the conditions prior to your tests

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

We can thus simplify our test-cases:

```
describe('testing factorial calculator', () => {
    before(() => {
        cy.visit('https://qainterview.pythonanywhere.com/');
    })

    it('tests with valid integer 4', () => {

    })

    it('tests with illegal input', () => {

    })
})
```

Now we need make Cypress select the input field before we actually send a value, we'll use Cy.get() 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 Cypress 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 use Cy.type() to inject a value into the targeted selector:

```
it('tests with valid integer 4', () => {
    // Using one of these is enough

    // HTML id are targeted using #
    cy.get('#number').type(4);

    // HTML classes are targeted using .
    cy.get('.form-control').type(4);

    // This one shouldn't be used if there are multiple input fields on webapge
    cy.get('input').type(4);
})
```

We are chaining .type() onto the subject returned from the .get() command, read more about the chains of commands here

Another useful command is Cy.contains(), which gets the DOM element containing the text. Try getting the button using .contains() or .get(), and then chain Cy.click() to simulate a press on the button.

It is worth to note that **every command is inherently an assertion**. **.visit()** will fail the test if the inputted URL isn't responding after some certain time, Both **.get()** and **.contains()** will fail if no elements are found.

So how do we know if the outputted value the website gives is correct? A quite simple solution is using .**contains()**:

```
    it('tests with valid integer 4', () => {
        cy.get('#number').type(4);
        cy.get('.btn').click();

        //One of the following alternatives:
        cy.contains('The factorial of 4 is: 24')
        cy.get('#resultDiv').should('contain', 'The factorial of 4 is: 24')
    })
```

Cy.should() creates an assertion, and is usually used for more advanced assertions (such as when we want to find out if a certain element is equal to another element, or if an element has a certain HTML/CSS attribute) we recommend reading through the assertion section written by Cypress and try for yourself using the list of available assertions.

- It is also important to remember that there are so-called 'getters' that does nothing but to improve readability and allow you to write clear, English sentences. For example:

```
    it('tests with valid integer 4', () => {
        cy.get('#number').type(4);
        cy.get('.btn').click();

        //The word 'be' doesn't do anything
        cy.get('#resultDiv').should('be.visible')
```

```
    })
```

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

```
    it('tests with valid integer 4', () => {
        let input = 4;
        let answer = 24;
        cy.get('#number').type(input);
        cy.get('.btn').click();

        cy.contains(`The factorial of ${input} is: ${answer}`);
    })
```

If you find that there are multiple test-cases that utilize the same steps, consider putting them in a *beforeEach()* hook, or create a function which you can call from each test-case.

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 Cypress documentation as much as possible.