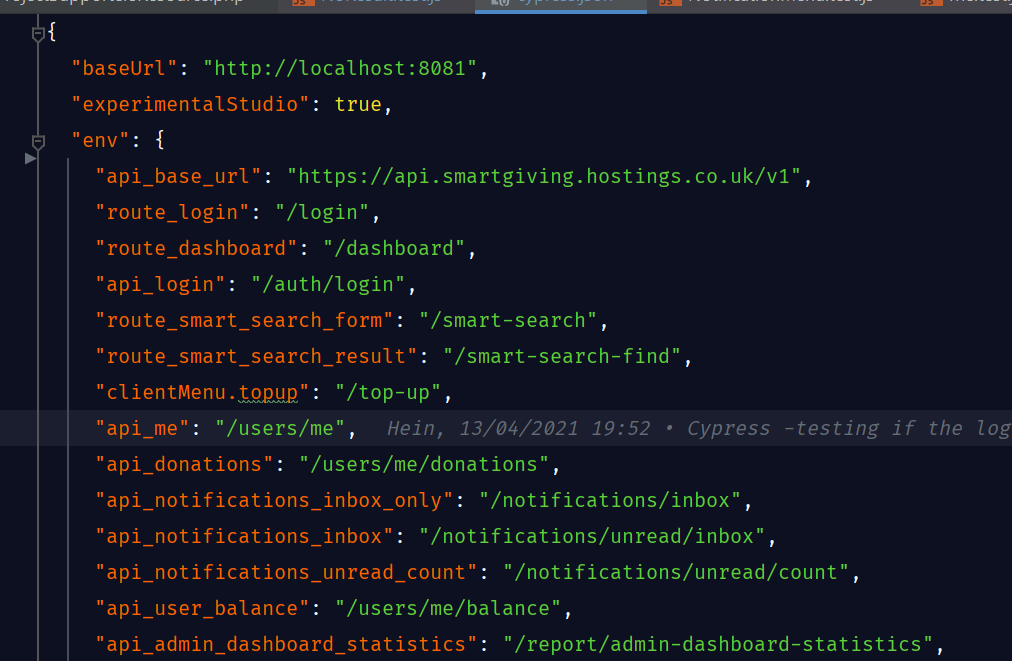
**Cypress Best Practices:**

**Storing the config values and constants**

We store them in the cypress.json file



**Storing the API endpoints in the cypress.json**

When we store the API endpoints in the cypress.json file, we should not include the GET parameters or Query String parameters because they are the variables and can be changed depending on the page or action or feature you are testing.

We should store them as follow:



The followings are not recommended because query string parameters are included in the endpoints.

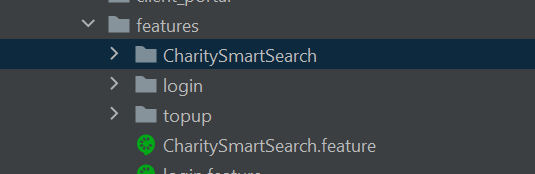


**File naming convention**

Cypress uses the underscore case instead of camel case unlike Jest for naming the files.

But if you are using Cypress with Cucumber, it will be different.

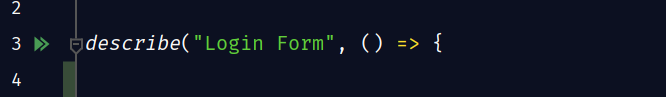
We will be using Camel Case for naming the files. Feature files and the folder to hose the test definition files will have the same name, otherwise, Cucumber will not work. Following is the example folder structure.



As you can see in the screenshot, there is a feature file called, CharitySmartSearch.feature whereas there is a folder called, CharitySmartSearch to store the test definition files.

**Naming the test suite (we do not need to follow this for Cypress with Cucumber)**

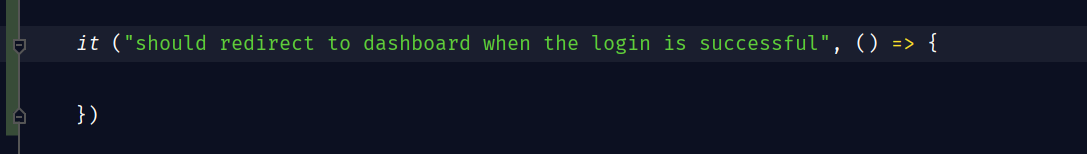
We use “describe” function to name the test suite. The name should be noun: the feature or page or action we are testing. For example, if we are testing login form, the name should be as follow.



Each word starts with a capital letter.

**Naming the test (we do not need to follow this for Cypress with Cucumber)**

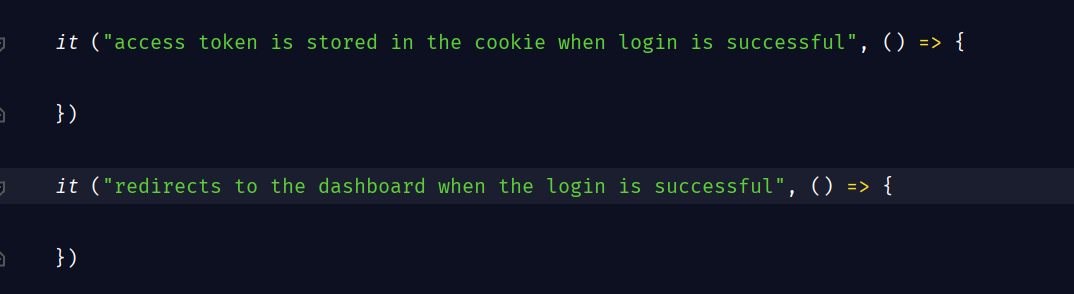
We use “it” function to declare tests. It should follow this pattern, {“expected behaviour/ additional condition”}.



As you can see in the screenshot, the test name starts with “should redirect to dashboard” which is the expected behaviour which then followed by “when the login is successful” which is the additional condition.

**Covering every part of the feature**

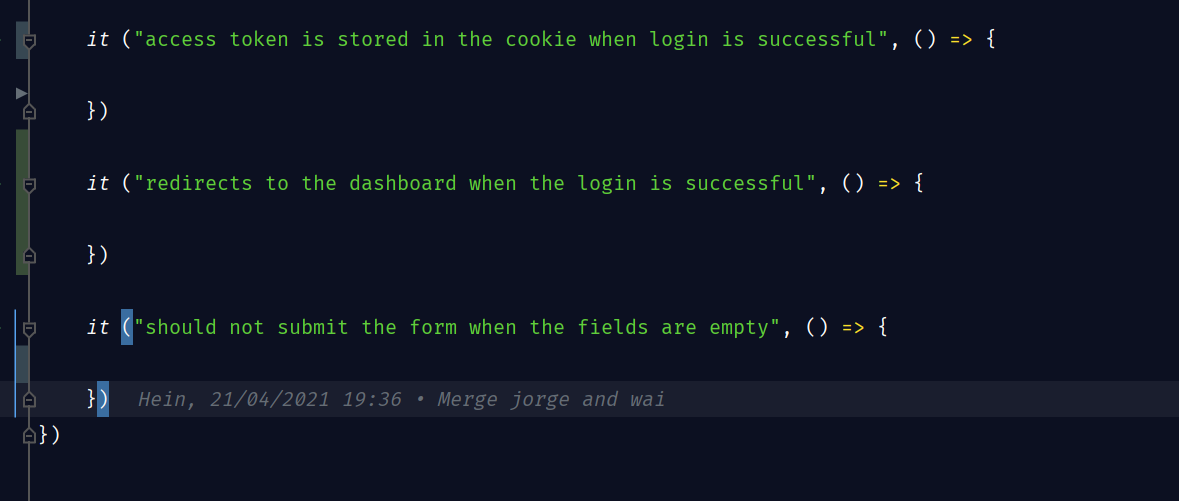
When we are testing a feature we should cover every part of the feature if possible.



As you can see in the code example, we are testing if the access token is stored in the cookie when the login is successful. We are also testing if it redirects the user to the dashboard when the login is successful. Basically, we are trying to cover every part of the feature.

**NOTE:** instead of testing both of them in the same test, it is a recommended practice to split the test into two different tests.

**Covering both happy path and unhappy path**



As you can see in the code, we are not only testing if the login is successful (happy path), we are also testing the scenario where the login fails when the form validation fails (unhappy path). We are covering both happy path and unhappy path. We should also try to cover the edge-case scenarios if there is any.

**Use data-cy as selector if possible**

For selector, Cypress recommend to use custom attribute “data-cy” as selector rather than using HTML tag name (button, input, p, etc etc), or class name or id.

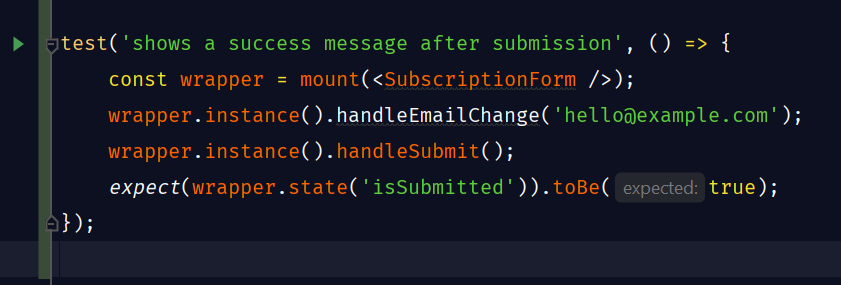
**Jest Unit Testing best practice:**

**File naming convention**

Jest is using Camel case unlike Cypress. For example, DonorSelect.test.js.

**Avoid Internal Testing**

Imaging we are writing the test for a form that has a email field and a button on it. When the button is clicked, it will submit the form. When the form is submitted, it will update the state and display a message.



As you can see in the test, it is testing if the isSubmitted variable is updated to be true when the form is submitted. It will potentially give us two problem when we refactor our code.

1. “False Negative”: imagine you rename the “isSubmitted” variable to something else. But the behaviour of the form stays the same which means it is still displaying the message when the form is submitted. The test will give us the “False Negative” result because even though the behaviour of the form stays the same after refactoring, the test is failing because the variable name was changed.
2. “False Positive”: after refactoring, the message might not be displayed but the test is still passing because the variable value is updated. The variable value is updated does not mean that the message is displayed. The test will give us the “False” positive result because it is passing but the message is not displayed.

Solution: we should test that the message is displayed instead of testing that the variable state to track the form state is changed.

**Avoid Unnecessary Expectation**



As you can see in the code, it is testing if the “pizza” variable is defined as well as its value. If we can test the value of the variable, we do not need to test if it is defined because that is unnecessary.

Also, in the second example, if we can test that the message in the model is visible, we can say that the modal is visible too. So that writing the test if the modal is visible in this case won’t be necessary.

**Putting the snapshot in the unit test**



As you can see in the code above, we are writing unt test for the NoResult component. As you can see in the code, we create a variable for the component outside of the test methods. We also write a snapshot test using that variable. In this way we do not need to another test file for the snapshot testing. As a rule of thumb, we can always add a snapshot test in that way whenever we write a unit test for a component.

**Create a function to create the test props**



Sometimes, we might need to pass the “props” to the component we are testing with the test values. In this case it is a good practice to create a dedicated function to create the test props and the function also allows us to override the “props” too.