Introduction to Cypress

Cypress is an end-to-end testing framework designed to simplify the process of testing web applications. Unlike Selenium, which runs outside of the browser, Cypress executes within the browser to offer a more consistent testing environment. Cypress is widely used for its robust set of features like real-time browser preview, time-travel capabilities, automatic waiting, and network request handling.

Sections we cover

1. How to select element
2. How to add inputs
3. How to assert
4. How to wait

Element Selection in Cypress

Selecting elements accurately is a crucial aspect of writing effective end-to-end tests. While you can select elements by tag name, class name, or ID, using data attributes like data-testid is often the most stable way to avoid brittle tests. Below are some of the commonly used methods to select elements in Cypress.

By Tag Name

Selecting elements by their HTML tag name is the most basic form of element selection. It is often used for generic actions applied to all elements of the same type.

```
// Selects all <button> elements and clicks them
cy.get('button').click()
```

By Class Name

Class-based selection targets elements that share the same CSS class. This method is useful when a group of elements have similar styles or behaviors.

```
// Selects all elements with the class 'btn-submit' and clicks the first one
cy.get('.btn-submit').first().click()
```

By ID

ID-based selection is straightforward and ideal when the element has a unique ID. The ID should be unique per page.

```
// Selects the element with the ID 'username' and types into it
cy.get('#username').type('admin')
```

By Attribute

Attribute-based selection can be useful for targeting elements with specific HTML attributes, which may or may not have a unique value.

```
// Selects input elements of type 'password' and types into the first one
cy.get('[type="password"]').first().type('secret')
```

By data-testid

This method is often the most resilient to design changes in your application. By adding a data-testid attribute to your HTML, you can isolate your tests from specific UI implementations.

```
// Targets the element with the 'data-testid' attribute set to 'submit-button' and clicks it
cy.get('[data-testid=submit-button]').click()
```

Interacting with Elements

Inputs and Buttons

The interaction with webpage elements like inputs and buttons is a crucial part of end-to-end testing. This section aims to provide an overview of how you can simulate these interactions using Cypress.

Providing Inputs

Cypress offers a .type() command that allows you to simulate typing text into an input element. This works with various types of inputs, including text fields, password fields, and even text areas.

```
// Type 'username' into an input field with data-testid 'username'
cy.get('[data-testid=username]').type('username')
```

Here, .type('username') simulates typing the word "username" into the selected element.

Clicking Buttons

To simulate mouse clicks, Cypress provides a .click() command. This can be used on any clickable element, like a button or a link.

```
// Click a button with data-testid 'submit-button'
cy.get('[data-testid=submit-button]').click()
```

By chaining .click() after a cy.get(), you're telling Cypress to click on the first element that matches the selector.

Advanced Clicking: Options

You can also pass options to the .click() command to specify the position where the click should be made, among other things.

```
// Click at a specific position within the button
cy.get('[data-testid=submit-button]').click(10, 10)
```

Clearing Inputs

Sometimes, you may want to clear an input field before providing new input. Cypress provides a .clear() command for this.

```
// Clear an input field before typing into it
cy.get('[data-testid=username]').clear().type('new_username')
```

In this example, any existing text in the input will be cleared before the new text "new_username" is typed in.

Assertions in Cypress: Validating Behavior

Assertions in Cypress allow you to verify the state of your application at various points in time. This helps ensure that your app is not only functional but also reliable. Cypress uses Chai assertions, and they can be used with a .should() command or an .and() command for chaining.

Basic Assertions

Basic assertions generally involve using .should() to assert the state of a particular element.

```
// Assert that an element with data-testid 'error-message' contains the text 'Invalid Credentials'
cy.get('[data-testid=error-message]').should('contain', 'Invalid Credentials')
```

Existence Assertions

To check if an element exists within the DOM, you can use:

```
// Asserts that an element with data-testid 'submit-button' exists
cy.get('[data-testid=submit-button]').should('exist')
```

Count Assertions

You can assert the count of matching DOM elements.

```
// Assert that there are exactly 4 elements with class 'list-item'
cy.get('.list-item').should('have.length', 4)
```

Exercise

Installation

Before you start, ensure that Node.js and npm are installed. Open your terminal and navigate to your React project directory.

Install Cypress using npm:

```
npm install cypress --save-dev
```

After installation, add a script to your package.json to run Cypress:

```
"scripts": {
  "start": "react-scripts start",
  "cypress:open": "cypress open"
}
```

Run Cypress to open its dashboard:

```
npm run cypress:open
```

Writing Your First Test with Cypress

Now that you have a grasp of element selection methods, let's write a simple test case. We'll use a hypothetical login page for this example.

Test File Setup

1. Navigate to your project directory and find the cypress/integration folder.
2. Create a new file named login_spec.js.

Example Test Code for a Login Page

Here's how you might write Cypress tests for a login page:

```
// cypress/integration/login_spec.js

describe('Login Page Tests', () => {

  beforeEach(() => {
    // Navigate to the login page before each test
    cy.visit('http://localhost:3000/login');
  });

  it('displays an error for invalid credentials', () => {
    // Use data-testid for stable element targeting
    cy.get('[data-testid=username]').type('wrong_username');
    cy.get('[data-testid=password]').type('wrong_password');
    cy.get('[data-testid=login-button]').click();

    // Validate error message
    cy.get('[data-testid=error-message]').should('contain', 'Invalid Credentials');
  });

  it('redirects to the dashboard for valid credentials', () => {
    // Use data-testid for stable element targeting
    cy.get('[data-testid=username]').type('correct_username');
    cy.get('[data-testid=password]').type('correct_password');
    cy.get('[data-testid=login-button]').click();

    // Validate navigation to dashboard
    cy.url().should('eq', 'http://localhost:3000/dashboard');
    cy.get('[data-testid=welcome-message]').should('contain', 'Welcome, User!');
  });

});
```

- describe is a Mocha function that allows you to organize your tests into test suites.
- beforeEach is a hook that runs before each test in the suite. It's useful for setting up the test environment.
- `it` is another Mocha function for defining individual test cases.
- cy.visit navigates to a given URL.
- cy.get selects elements, and .type inputs text into them.
- cy.click simulates a click event.
- .should is used for assertions to validate if conditions are met.

Running Your Tests

To run your tests:

1. Open your terminal and navigate to your project directory.
2. Run the following command to open the Cypress dashboard:

```
npm run cypress:open
```

3. In the Cypress dashboard, click on login_spec.js to run the test suite.