

UI Test Automation Best Practices and Page Object Model in Cypress

8 best practices we should follow in UI Automation

1. Consider using a BDD framework
2. Always use test design patterns and principles
3. Avoid using static wait (cy.wait())
4. Proper naming convention to each test
5. Test should be simple and do not keep unnecessary comments
6. Every test must has one and only one validation
7. All tests should be independent
8. Use Data Driver approach for repeatable tests

Consider using a BDD framework

I always recommend BDD is one of the best practice we should follow while doing UI Automation. There are advantages in using BDD.

1. BDD is a methodology that helps teams understand each other, creating strong outside and inside team collaboration. By writing your tests with BDD you also create specifications that can help your team understand tests and requirements much better. This means that along with writing your tests, you are creating a clear tests documentation. This ensures you don't waste other team members' time (who might work on your tests later), as well as your own time, because you don't need to explain and help with such tests if they are unclear.
2. BDD helps the Business side (e.g. Test and Project managers) understand these tests. This brings additional value to testing because they can make recommendations based on business benefits.
3. BDD usually forces you to follow a strict code organization pattern, which helps avoid code duplication. This is done by having separate components called steps or actions that will be the building blocks for your tests.

Always use test design patterns and principles

A design pattern is a reusable solution for a common problem in software design. We can say that each pattern is a particular example of a specific solution for a specific problem, regardless of the programming language or paradigm. Here I will talk about **Page Object Model**, though we have other patterns like **Screenplay**.

Page Objects patterns purpose was to make UI automation tests consistent, to avoid code duplication, to improve readability and to organize code for web pages interaction. During web tests creation you always need to interact with web pages and web elements that are presented on these pages (buttons, input elements, images, etc.). The Page Objects pattern takes this requirement and applies object oriented programming principles on top of this, enforcing you to interact with all pages and elements as with objects.

Here page is anything you test (example Sign In page) and objects are User Name, Password text fields and SignIn button field.

Avoid using static wait (cy.wait())

Kindly refer why should avoid using cy.wait(). <https://docs.cypress.io/guides/references/best-practices.html>

Proper naming convention to each test

Test names should be very clear and provide a self-descriptive idea about which exact functionality is being tested by using this test. Why? First, you need to immediately understand what each test verifies even a year after you wrote the test. In addition to that, you should always help your team members and make all your tests clear for them. Moreover, if some of the tests failed during tests execution run, you should understand which functionality was broken just by having a quick look at the test name. You should not waste time on verifying what the test actually does.

Bad Example

```
it('Admin login', () => {})
```

Good Example

```
it('Login as the administrator', () => {})
```

Test naming convention ensures that:

1. The name of a test method describes a specific business or technical requirement.
2. The name of a test method describes expected input (or state) and the expected result for that input (state).

Test should be simple and do not keep unnecessary comments

Tests should always be clear and simple to read. If you have a feeling you need to leave a comment to understand what is done in this line, then you need to take a step back and think again about what are you doing wrong.

Recommendation to use Page Object pattern like below to avoid comments for each line what it does.

Example signin, verify_login_success, create_new_contact, global_search, validate_created_contact and delete_created_contact. These functions does not need additional comments to understand any layman.

```
describe('Contacts Test Cases', () => {  
  
  let created_contact  
  let testData  
  
  before('Load test data from fixtures', () => {  
    cy.fixture('example.json').then(($exampleData) => {  
      testData = $exampleData  
    })  
  })  
  
  // ...  
})
```

```

    })
  })

  beforeEach('Open Contacts screen', () => {
    cy.visit('/', {failOnStatusCode:false})
    cy.clearCookies()
    signin(testData.userName, testData.psw)
    verify_login_success('Praveen Narala')
  })

  it('Create new Contact', () => {
    cy.visit('/contacts')
    created_contact = create_new_contact()
    cy.log(created_contact)
    global_search(created_contact)
    validate_created_contact(created_contact + ' Reddy Narala')
  })

  it('Delete created Contact', () => {
    global_search(created_contact)
    delete_created_contact(created_contact + ' Reddy Narala')
    page_toolbar_actions('Delete')
    global_search(created_contact)
    validate_created_contact_got_deleted(created_contact + ' Reddy Narala')
  })
}

```

```

})

```

Every test must has one and only one validation

Ensure not to keep too many validation in any test. This will avoid the behaviour of test scenario. Highly recommend use simple tests which are independent to each other. Purpose of test should not void.

Example Below test job/behavior to create new contact. Hence, the validation of this test is to get created contact and match with expected one. If you have delete, please keep that test independent from here.

```

it('Create new Contact', () => {
  cy.visit('/contacts')
  created_contact = create_new_contact()
  cy.log(created_contact)
  global_search(created_contact)
  validate_created_contact(created_contact + ' Reddy Narala')
})

```

In BDD stype...

Scenario: Create new contact

Given user navigate to "contacts" screen

When user create new contact:

FirstName	LastName	MiddleName	Category	Status	Description
Praveen	Narala	Reddy	Affiliate	New	Sample

|

Praveen	Narala	Reddy	Affiliate	New	Sample
---------	--------	-------	-----------	-----	--------

Description! |

Then user should be able to validate created contact

Scenario: Deleted created contact

And user search for created contact

And user delete created contact

Then user should not be able to see deleted contact

Page Object pattern

In typical Page Object Model, each web page has its own class and where you maintain all page element locators & its actions. Using class objects, we can expose pages in test classes. What if, the application has 100+ pages and it is still growing and ended up in creating too many page classes.

Unlike the above pattern, we still follow the Page Object model without using Object-Oriented programming. JavaScript is a Light Oriented programming language and Cypress took this advantage is developing E2E tests without any hassle. Hence, instead of creating page classes, should use page files, utils or Cypress common.js.

Page files (.js or .ts). are independent and maintains all page level locators and its actions at one place. Instead class, we use simple JavaScript functions (if not better because the type check step can understand individual function signatures) and export them.

login_page.js

```
import { expect } from "chai";

//Page locators
const emailField = '[name="email"]';
const passField = '[name="password"]';
const loginBtn = '.ui.fluid.large.blue.submit.button';
const userNameLabel = '.user-display';

//Simple JavaScript function to SignIn
export const signin = (username, password) => {
  cy.setValue(emailField, username)
  cy.setValue(passField, password)
  cy.clickElement(loginBtn)
}

//Function to validate SignIn success
export const verify_login_success = (expected_username) => {
```

```

        cy.getText('CSS', userNameLabel, 8000).then((text) => {
            expect(text).to.eq(expected_username)
        })
    }
}

```

Cucumber step definition file

```

import { Before, Given } from "cypress-cucumber-preprocessor/steps";
import { signin, verify_login_success } from
"../../framework/pageObjects/openCRM_login_page";

let testData

Before(() => {
    cy.fixture('example.json').then(($exampleData) => {
        testData = $exampleData
    })
})

Given('user login to applications', () => {
    cy.visit('/', {failOnStatusCode:false})
    cy.clearCookies()
    signin(testData.userName, testData.psw)
    verify_login_success('Praveen Narala')
})

```

Cucumber Feature

Feature: Open CRM Contacts

Background: Login to application

Given user login to application //Login behaviour

Scenario: Create new contact

Given user navigate to "contacts" screen

When user create new contact:

FirstName	LastName	MiddleName	Category	Status	Description
Praveen	Narala	Reddy	Affiliate	New	Sample Description!

Then user should be able to validate created contact

Otherway, calling Sign In page in Cypress Specification file

```

import { signin, verify_login_success } from
"../../framework/pageObjects/openCRM_login_page"; //Page Object reference

describe('Contacts Test Cases', () => {

  let testData

  before('Load test data from fixtures', () => {
    cy.fixture('example.json').then(($exampleData) => {
      testData = $exampleData
    })
  })

  it('Open Contacts screen', () => {
    cy.visit('/', {failOnStatusCode:false}) //This will avoid uncertain UI page
load failures
    cy.clearCookies()
    signin(testData.userName, testData.psw) // Page Object
    verify_login_success('Praveen Narala')
  })
})

```

Best ways of maintaining Reusable Functions

Using Cypress **commands.js** file to maintain all application level reusable functions. This will reduce the dependency and maintenance. It acts like an one stop shot. Example button clicks, typing text, selecting value from dropdown and handling list controls etc...

```

//Create element
Cypress.Commands.add('getElement', (locatorType, weblocator,
timeout=10) => {
  if(locatorType === 'CSS'){
    cy.get(weblocator, {timeout: timeout}).then(($ele) => {
      return cy.wrap($ele)
    })
  }else{
    cy.xpath(weblocator).then(($ele) => {
      cy.wrap($ele)
    })
  }
})

//Click element
Cypress.Commands.add('clickElement', (element_locator, timeout=10) =>
{
  try {
    cy.get(element_locator, {timeout:
timeout}).should('be.visible').click()

```

```

        } catch (error) {
            cy.log('Unable to click element')
        }
    })

    // -- This will select an item from list control
    Cypress.Commands.add('selectlistitem', (weblocator, option) => {
        cy.get(weblocator).each(($ele, index, $list) => {
            if($ele.text() === option){
                cy.wrap($ele).click()
            }
        })
    })
})

```

Expose all reusable functions in Cypress tests/page object files using **Cy** command

Create Chainable interface for all custom commands in index.d.ts file in cypress/support folder.

```

/// <reference types="cypress" />

declare namespace Cypress {
    interface Chainable<Subject> {
        /**
         * This will select an item from list control
         * @param weblocator
         * @param option
         */
        selectlistitem(weblocator:string, option:string):
Chainable<any>

        /**
         * Get text on an element
         * @param locatorType
         * @param weblocator
         * @param timeout
         */
        getText(locatorType:string, weblocator:string, timeout?:10):
Chainable<any>

        /**
         * Set value in Text Field
         * @param element_locator
         * @param value
         */
        setValue(element_locator:string, value:string): Chainable<any>
    }
}

```

IMPORTANT NOTE (Please find below steps to see how to configure Intellisense to expose all custom commands)

Setup

System Requirements:

1. Windows 7 or above
2. macOS 10.9 and above (64-bit only)
3. Linux Ubuntu 12.04 and above, Fedora 21 and Debian 8 (64-bit only)

Node.js:

1. Node.js 10 or 12 and above

Visual Studio Code:

1. Install Visual Studio Code

Install Cypress via npm:

1. `cd /your/project/path`
 2. `npm init`
 3. `npm install cypress --save-dev`
- (OR)
1. `cd /your/project/path`
 2. run `'code .'` and enter. It will launch Visual Studio Code IDE
 3. In Terminal -> run `npm install cypress --save-dev`

Setting baseUrl:

```
cypress.json -> add "baseUrl":"https://www.google.com/"
```

Enabling Cypress Intellisense

Triple slash directive

The simplest way to see IntelliSense when typing a Cypress command or assertion is to add a triple-slash directive to the head of your JavaScript or TypeScript testing file. This will turn the IntelliSense on a per file basis. Copy the comment line below and paste it into your spec file.

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


If you write custom commands and provide TypeScript definitions for them, you can use the triple slash directives to show IntelliSense, even if your project uses only JavaScript. For example, if your custom commands are written in `cypress/support/commands.js` and you describe them in `cypress/support/index.d.ts` use.

```
// type definitions for Cypress object "cy"
/// <reference types="cypress" />

// type definitions for custom commands like "createDefaultTodos"
/// <reference types="../support" />
```

Highly recommended way of working with Intellisense. Stop adding triple slash directives to each JavaScript spec file.

Reference type declarations via **tsconfig.json / jsconfig.json**

```
{
  "compilerOptions": {
    "lib": ["es2015", "dom"],
    "allowJs": true,
    "noEmit": true,
    "types": [
      "cypress"
    ]
  },
  "include": [
    "./node_modules/cypress",
    "cypress/**/*.js",
    "cypress/**/*.d.ts" //index.d.ts -> This will enable intellisense
in all specs for custom commands
  ]
}
```

Important/Note: Please remove `tsconfig.json` file from project root folder. Otherwise `tsconfig.json` file does not work.

Run Tests in different browsers:

1. Run cmd -> `cypress run --browser chrome/firefox/electron` or
2. We can set in `package.json` // `"cy:run:chrome": "cypress run --browser chrome"`, // `"cy:run:firefox": "cypress run --browser firefox"`

Run in Headless mode (cmd):

```
$ cy:run --headless --browser chrome
```

Run in parallel:

```
$ cypress run --record --parallel
```

Run tests specifying a single test file to run instead of all tests:

```
$ cypress run --spec "cypress/integration/examples/actions.spec.js"
.\node_modules\.bin\cypress run --browser chrome --record --key e2ccadb8-bf34-47bb-852b-9fe78e387d57 --headless
```

Run tests specifying multiple test files to run:

```
$ cypress run --spec
"cypress/integration/examples/actions.spec.js,cypress/integration/examples/files.s
pec.js"
```

Enable XPATH: (GITHUB path -> <https://github.com/cypress-io/cypress-xpath>)

1. Install npm install -D cypress-xpath
2. Then include in your project's cypress/support/index.js -> require('cypress-xpath')

Enable Cucumber:

Configuration:

1. Install npm install --save-dev cypress-cucumber-preprocessor
2. To enable usage of Cucumber in the Cypress automation framework, we need to make some configurations in below 3 files..

- a. plugins/index.js
- b. cypress.json
- c. package.json

index.js: 1. The first file (shown by marker 1) is the "index.js" file under the plugins folder. We need to make the following changes in the "index.js" file, which exports Cucumber as a module and make it accessible in other Cypress files.

```
const cucumber = require('cypress-cucumber-preprocessor').default

module.exports = (on, config) => {
  on('file:preprocessor', cucumber())
}
```

cypress.json:

```
Add below lines consider only .feature extension files
"testFiles": "**/*.feature",
"ignoreTestFiles": "**/*.js,ts"
```

package.json:

```
Add "cypress-cucumber-preprocessor": {
  "nonGlobalStepDefinitions": true
} //Here we need to specify the configuration that non-global step
definitions are allowed, which means that step definitions can exist in sub-
folders as well.
```

Recommended Reference: <https://wanago.io/2020/01/13/javascript-testing-cypress-cucumber/>

Creating Feature and step definition files:

Feature File:

1. Always feature files should be created in 'integration' folder **Example:** integration\event.feature, integration\contact.feature

Step Definition Files:

1. Always create folder based on feature file name in 'integration' folder **Example:** integration\event\event_steps.js, integration\contact\contact_steps.js **Note:** Folder structure should not change

Common/Global/Repeated Test Step Files:

1. Create common folder inside 'integration' folder
Example: integration\common\common_steps.js

Cucumber Hooks:

1. Cucumber has two main Hooks a. Before b. After

Before: This will execute before all scenarios. We can also create tagged Hooks.

```
Before(() => {

})
Example: Before({ tags: "@foo" }, () => {

})
```

After: This will execute after all scenarios execution. We can also create tagged hooks. Tagged hooks will work based on Scenario tags.

```
Example: After(() => {

})
Example: After({ tags: "@foo" }, () => {

})
```

Note: We can also use Cypress Hooks as well in cucumber

Parameterization in Cucumber Scenarios:

1. Data Table ->

Example Scenario:

```
Scenario: Create new contact
Given user navigate to "contacts" screen
When user create new contact:
    | FirstName | LastName | MiddleName | Category | Status |
Description |
    | Praveen   | Narala   | Reddy      | Affiliate | New     | Sample
Description! |
Then user should be able to validate created contact
```

Example Steps:

```

    let created_contact //Global variable
    When('user create new contact:', (dataTable) => {
        cy.log(dataTable.hashes())
        const userDetails = dataTable.hashes()
        for(const row of userDetails){
            created_contact = create_new_contact(row.FirstName,
            row.LastName, row.MiddleName, row.Category, row.Status, row.Description)
        }
    })

```

2. Example Scenario Outline:

Scenario Outline: Update Manage Calendar
 When user update manages calendar "<CalendarName>"
 Then user should be able to see updated manage calendar "
 <CalendarName>"

Examples:

CalendarName	
Praveen Narala <praveenreddy.narala@gmail.com>	1

3. Example Steps:

```

When('user update manages calendar {string}', (eventName) => {
    manage_calendar(eventName)
})

```

4. Background:

Example:

Background: Login to application
 Given user login to applications
 And user navigate to "calendar" screen

Enabel Cucumber JSON Report:

The cypress-cucumber-preprocessor can generate a cucumber.json file output as it runs the features files. This is separate from, and in addition to, any Mocha reporter configured in Cypress.

Output, by default, is written to the folder cypress/cucumber-json, and one file is generated per feature. Add the following to the cypress-cucumber-preprocessor section in package.json to turn it off or change the defaults:

```
"cypress-cucumber-preprocessor": {  
  "cucumberJson": {  
    "generate": true,  
    "outputFolder": "cypress/cucumber-json",  
    "filePrefix": "",  
    "fileSuffix": ".cucumber"  
  }  
}
```
