



Cypress File

Efrem Yohanis

TABLE OF CONTENTS

1. CHAPTER 1: INTRODUCTION	1
1.1 INTRODUCTION CYPRESS.....	1
1.2 HOW TO SET UP ENVIRONMENT	1
1.3 MY FIRST CYPRESS CODE	5
2. CHAPTER 2: BASIC OF CYPRESS	9
2.1 TEST SENARIO VS TEST CASE	9
2.2 CYPRESS TEST CODE STRUCTURE.....	10
2.3 DESCRIBE BLOCK.....	10
2.4 IT BLOCK.....	10
2.5 SKIP.....	11
2.6 ONLY.....	11
2.7 TESTP STEPS.....	12
3. CHAPTER 3: STEP OF TEST CASE	14
2.1 VISIT WEB PAGE.....	14
2.2 CYPRESS COMMANDS FOR QUERRY ELEMENTS.....	15
2.3 LOCATE OR SELECT ELEMENTS.....	16
2.4 INTERACT WITH ELEMENT.....	18
2.9 ASSERTION.....	21
4. CHAPTER 4: WORKING WITH WEB ELEMENT.....	24
3.1 HANDLING TEXT VERIFICATION	24
3.4 HANDLING INPUT PAGE	24
3.5 HANDLING CHECKBOX.....	26
3.6 HANDLING RADIO.....	27
3.6 HANDLING FILE UPLOAD	28
3.7 HANDLING DROP DOWN	29
3.8 HANDLING AUTO COMPLETE.....	30
3.9 HANDLING ALERT	30
3.10 HANDLING WEB TABLE	32
3.11 HANDLING NEW TAB	33
3.12 HANDLING IFRAMES	34
3.13 HANDLING MOUSE EVENTS.....	35
5. CHAPTER 5: ADVANCED CONCEPT TESTING.....	37
4.2 CYPRESS HOOKS CONCEPT.....	37
4.3 FIXTURES IN CYPRESS	40
4.4 CUSTOM COMMANDS IN CYPRESS	42
4.5 PAGE OBJECT PATTERN IN CYPRESS	45
4.6 ENVIRONMENT VARIABLES.....	47
6. CHAPTER 6: RUNNING CYPRESS AND REPRTING	51
6.1 RUNNING CYPRESS FROM THE TERMINAL.....	51
6.2 RUNNING CYPRESS FROM TEST RUNNER.....	52
6.3 RUNNING CYPRESS FROM DASHBOARD.....	53
6.4 CAPTURE SCREENSHOTS, RECORD VIDEO.....	55
6.5 HTML REPORTING.....	56

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION CYPRESS

Cypress is an open-source end-to-end testing framework for web applications. It is designed to simplify the process of writing and executing tests, making it easier to ensure the quality and reliability of your web application.

Cypress is primarily used for testing the functionality of web applications by simulating user interactions and verifying the expected behavior of the application. It provides a rich set of built-in features and a powerful API that allows developers to write tests in a straightforward and expressive manner.

One of the key features of Cypress is its ability to run tests directly in the browser. This means that you can see the application being tested in real-time while the tests are running. It enables you to interact with the application and debug any issues that may arise during testing.

Cypress also offers an automatic waiting mechanism, which eliminates the need for explicit waits and timeouts in your tests. It intelligently waits for elements to appear and become interactive, ensuring that your tests are robust and reliable.

Furthermore, Cypress provides a comprehensive set of built-in tools for test debugging, including automatic screenshots and videos of test runs. This makes it easier to identify and troubleshoot issues that occur during testing.

Overall, Cypress offers a modern and developer-friendly approach to end-to-end testing. Its intuitive API, real-time browser testing, and powerful debugging tools make it a popular choice for testing web applications.

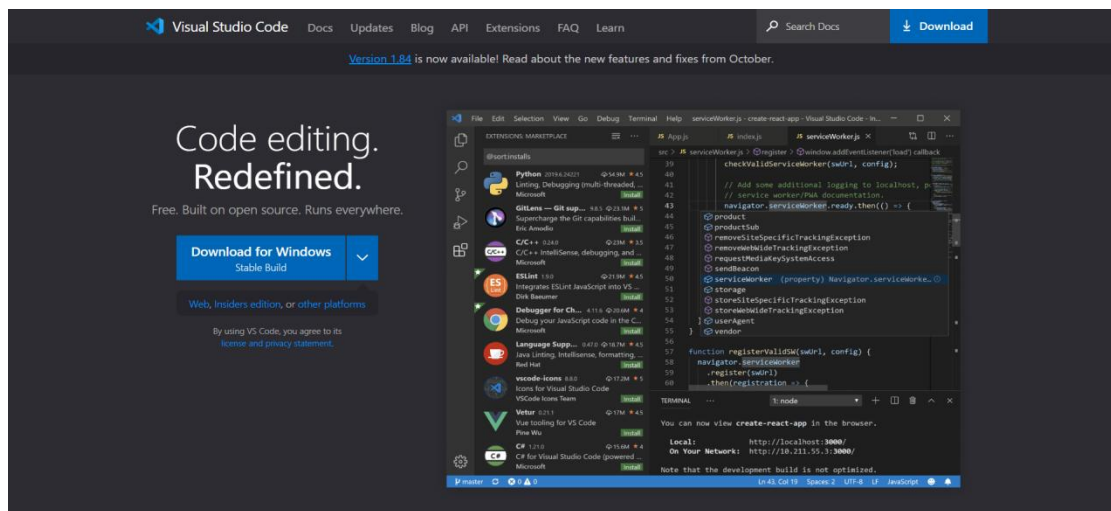
1.2 HOW TO SET UP ENVIRONMENT

To write and run Cypress test code, first, let's set up the environment. We need the following components:

1. Visual Studio Code (VS Code): a code editor.
2. Node.js: a JavaScript runtime environment.
3. Cypress: a JavaScript-based end-to-end testing framework.

1. install Visual Studio Code (VS Code), please follow these steps:

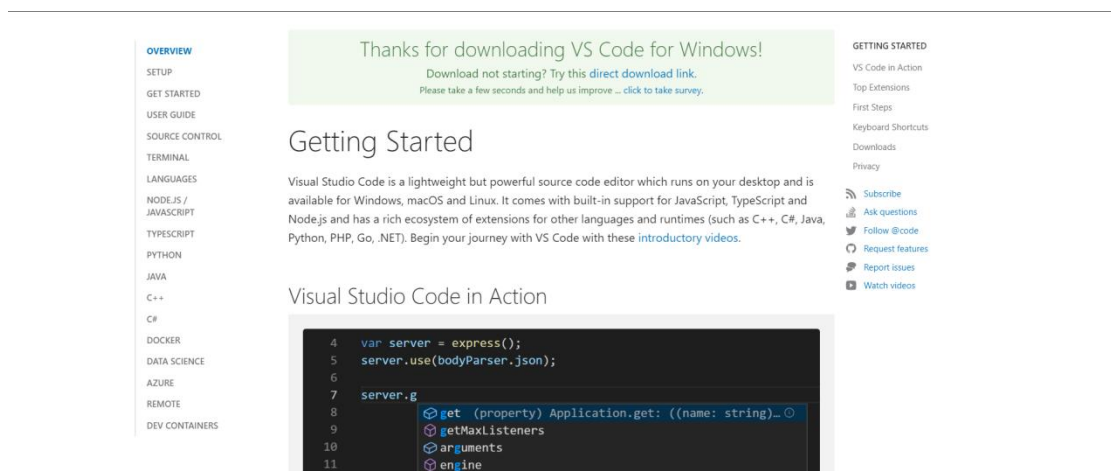
1. Visit the official Visual Studio Code website at <https://code.visualstudio.com/>.



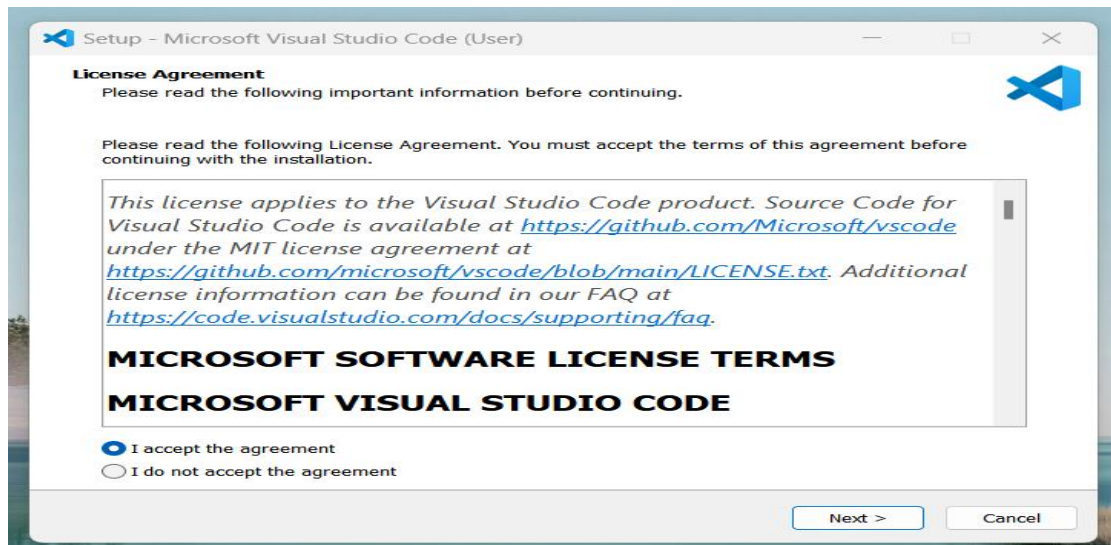
2. Click on the "Download" button for the operating system (Windows, macOS, or Linux) you are using.



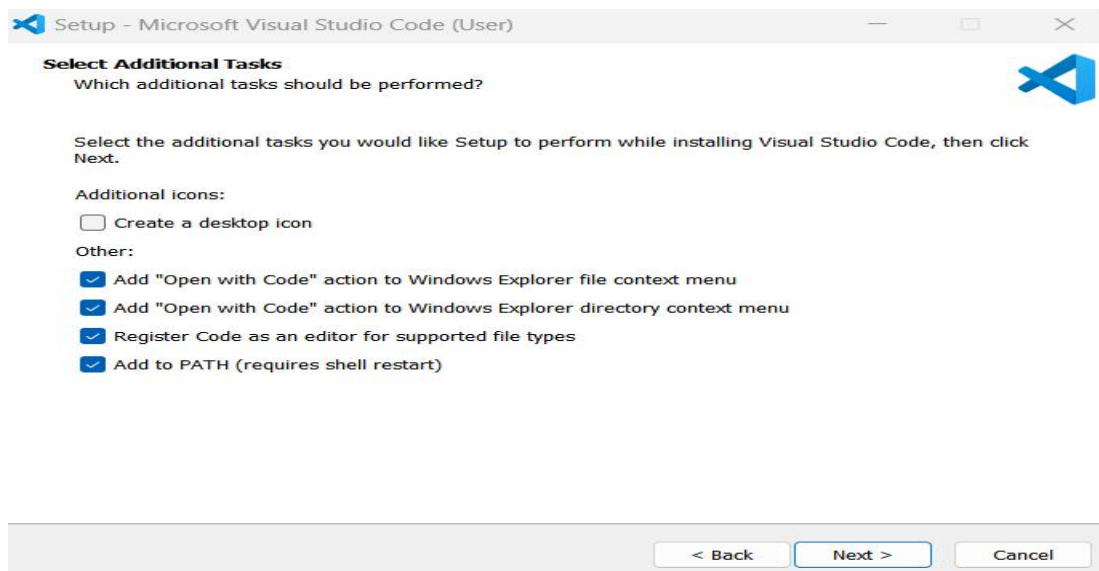
3. The executable file has been downloaded



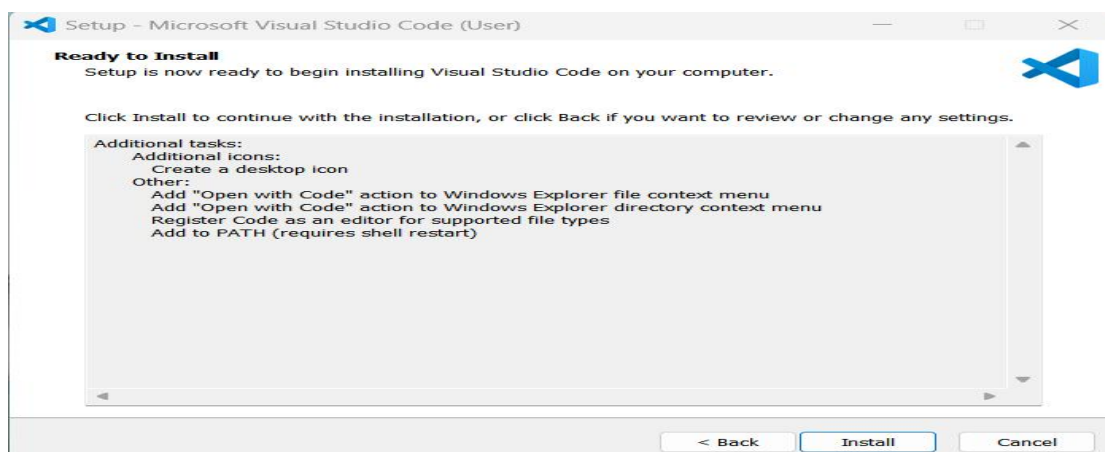
4. Once the download is complete, open the downloaded file from the folder and start the installation process by selecting "I accept the agreement"



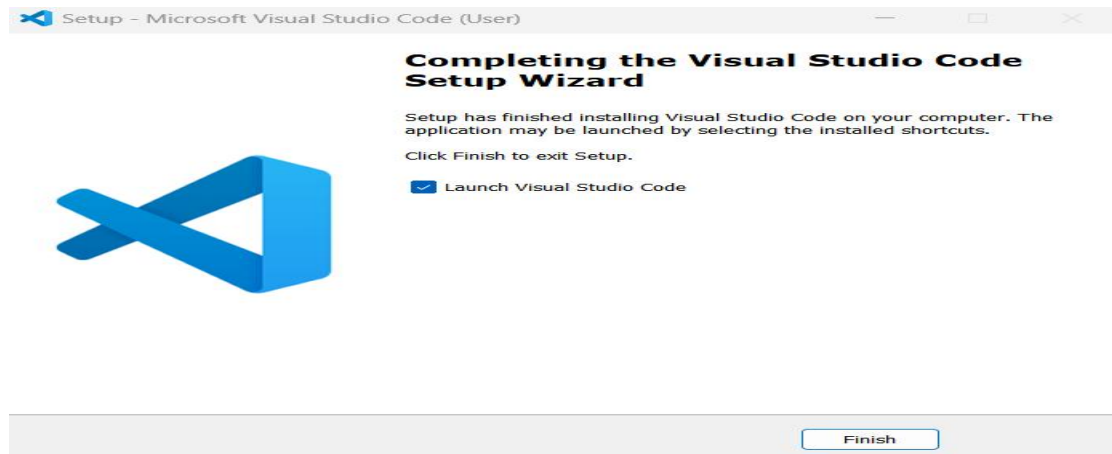
5. Check all option from listed option and click next.



6. Continue the installation process by pressing the "Install" option.



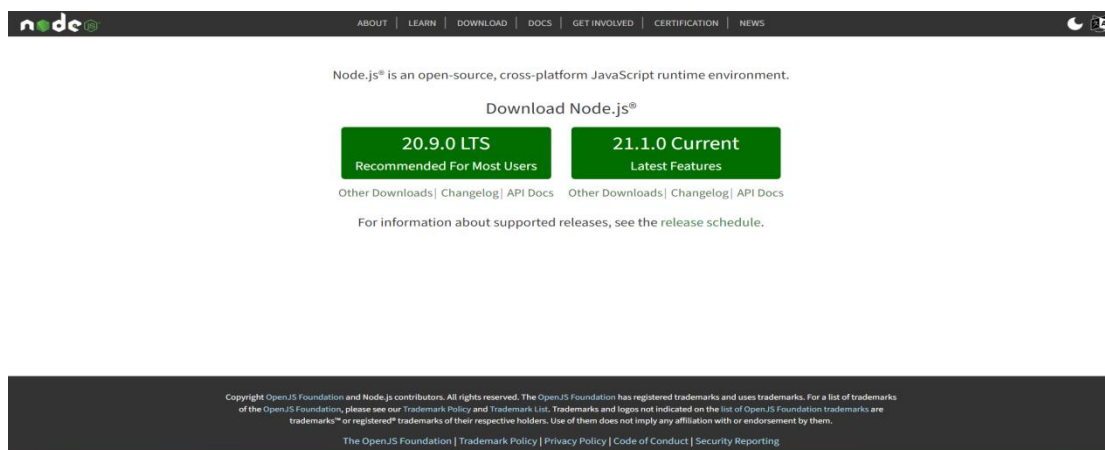
7. Once installation complete to launch the vscode check on **“launch visual studio code”** option and **click finish button**.



Congratulations! You have successfully viscode editor on your computer.

2. Install Node.js please follow these steps:

1. Visit the official Node.js website at <https://nodejs.org/>.



2. On the homepage, you will see two versions available for download: LTS (Long-Term Support) and Current. It is recommended to choose the LTS version for stability.

Select recommended for most user opetion.

3. Click on the LTS version to download the installer package for your operating system (Windows, macOS, or Linux).

4. Once the installer package is downloaded, run the installer by double-clicking on the downloaded file.

5. After the installation is complete, open the command prompt or terminal and type "node -v" to verify if Node.js is installed successfully. You should see the version number displayed.

```
Command Prompt
Microsoft Windows [Version 10.0.22621.2428]
(c) Microsoft Corporation. All rights reserved.

C:\Users\efrem>node -v
v16.20.1

C:\Users\efrem>
```

Congratulations! You have successfully installed Node.js on your computer.

3. To install Cypress please follow these steps.

Once you have Node.js installed, open a command prompt or terminal and navigate to the desired directory where you want to install Cypress.

Run the command to initialize a new npm project in the current directory.

```
npm init -i
```

This will create a `package.json` file.

Next, run the following command to install Cypress as a dev dependency in your project:

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

1.3 MY FIRST CYPRESS CODE

Now is the time to create your first Cypress code. To create a test file in the Cypress flow, follow the following steps.

Step 1 : open cypress. To open cypress there is two methods

1. On test Runner
2. On Terminal.

On Test runner:- to open cypress on Test Runner run this command on terminal.


```
npx cypress open
```

For the first time when open cypress we will see the follwing screen.

- ✧ This is the screen to select test type. We have two test type E2E aand Conmponet testing. Select E2E testing. To do end to end testing.

Welcome to Cypress!


[Review the differences between each testing type →](#)



E2E Testing

Build and test the entire experience of your application from end-to-end to ensure each flow matches your expectations.

● **Configured**



Component Testing

Build and test your components from your design system in isolation in order to ensure each state matches your expectations.

● **Not Configured**

- ✧ Quick Configuration and let's skip as it is and press "Continue" button.

Configuration files

We added the following files to your project:

✓	cypress.config.js The Cypress config file for E2E testing.	▼
✓	cypress\support\e2e.js The support file that is bundled and loaded before each E2E spec.	▼
✓	cypress\support\commands.js A support file that is useful for creating custom Cypress commands and overwriting existing ones.	▼
✓	cypress\fixtures\example.json Added an example fixtures file/folder	▼

Continue

- ✧ Here, all browsers that you have on your computer will be displayed. Select your preferred browser to browse the Test Runner. In my case, I have Chrome and Edge, but Electron has the default browser. After selecting the browser then press the "Start E2E Testing in Chrome" button.

Choose a browser

Choose your preferred browser for E2E testing.

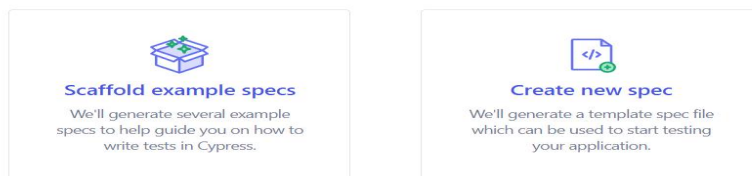


 Start E2E Testing in Chrome

← Switch testing type

Create your first spec

Since this project looks new, we recommend that you use the specs and tests that we've written for you to get started.




If you feel that you're seeing this screen in error, and there should be specs listed here, you likely need to update the spec pattern.

 [View spec pattern](#)

- ✧ Here, we have two options to create our first code. We can either use an example test code file or create a new test code file. Let's select "**Create new spec**" to create a new spec (test file).

- ✧ Our test file path is "**cypress/e2e/spec.cy.js**". The test file is located under the "cypress" folder and we have an "e2e" folder within that. The test file name is "spec.cy.js". Let's modify the test file name to "**First_Cypress_Test_code.cy.js**".

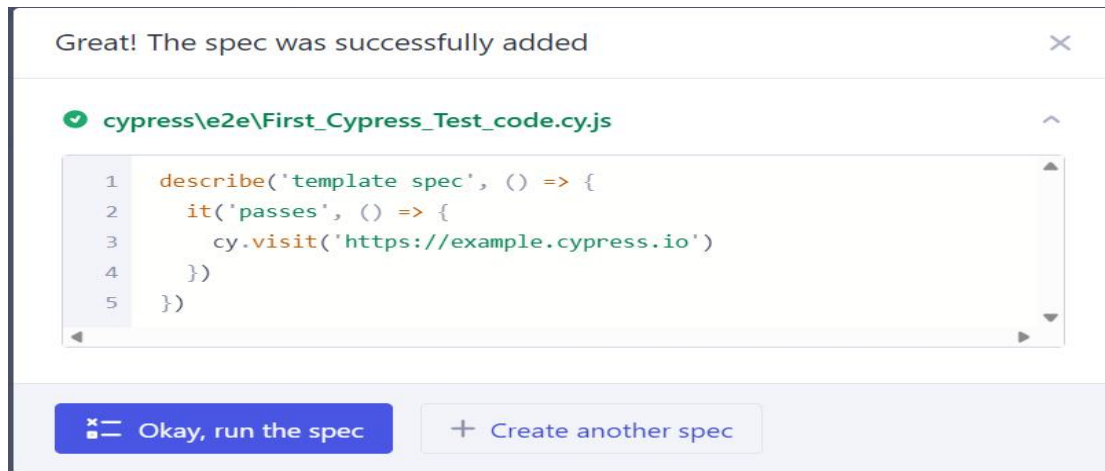
Enter the path for your new spec

 cypress\e2e\spec.cy.js

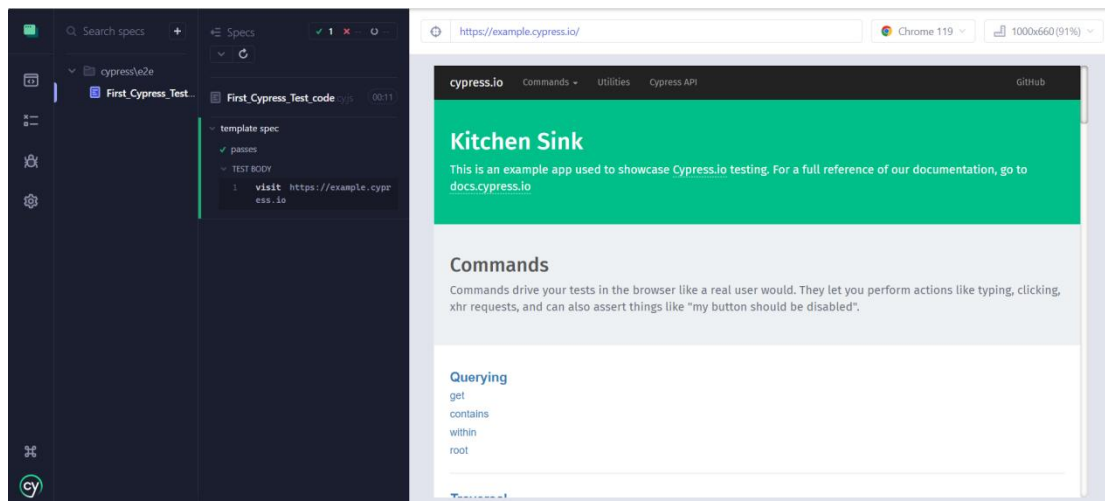
Create spec

Back

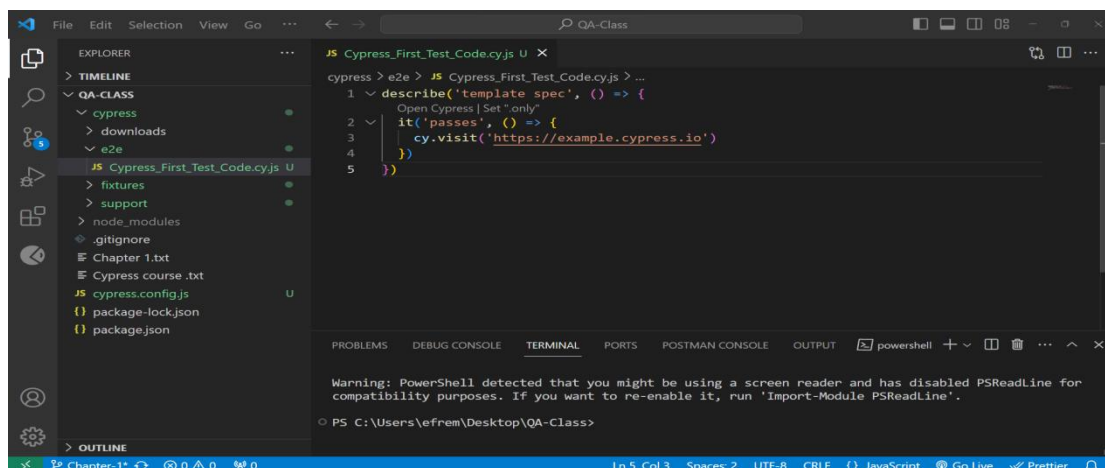
- ✧ This is our test file templet. Let's "**click Okay, run the spec**" button.



- ✧ Now our first test is opened with test runner. Let's go to VS Code to see what happens and modify our test code.



- ✧ Here, we can see that a "cypress" folder has been created. Inside the "cypress" folder, there are several subfolders including "download," "e2e," "fixtures," and "support." All our test code will be placed under the "e2e" folder, which is why we are seeing our first test file located under the "e2e" folder.



CHAPTER 2

BASIC OF CYPRESS

2.1 TEST SENARIO VS TEST CASE.

A test scenario and a test case are both important components of software testing, but they serve different purposes.

1. Test Scenario:- a test scenario is a high-level description or outline of a specific functionality or behavior that needs to be tested. It represents a real-life situation or event that the software should be able to handle correctly. Test scenarios are typically created during the early stages of test planning and are used to guide the creation of detailed test cases.

2. Test Case:- a test case is a detailed set of conditions or steps that need to be executed to verify whether a specific functionality of the software is working correctly. Each test case typically focuses on a specific aspect or feature of the software and provides detailed instructions on how to perform the test.

In Cypress, test scenarios and test cases can be implemented using the framework's syntax and features. Here's how you can define test scenarios and test cases in Cypress:

1. Test Scenarios:

In Cypress, you can define test scenarios by organizing your test cases into test suites or describe blocks. Each describe block represents a test scenario or a specific functionality to be tested.

2. Test Cases:

Test cases in Cypress are defined using `it` blocks. Each `it` block represents a specific test case with its steps and assertions.

For example:

```
describe('test suite', () => {  
  it('test case', () => {  
    // Test steps and assertions for this scenario  
  });  
})
```

2.2 CYPRESS TEST CODE STRUCTURE

Genraly test case have the following stracture and steps.

```
describe("Test Suite name", function () {  
  it("Test case Name", function () {  
    // Testp Steps  
    // 1. Visiting page  
    // 2. selecting (find) search box  
    // 3. intracting with element for type on search box  
    // 4. assertion.  
  });  
});
```

2.3 DESCRIBE BLOCK

- ✧ This keyword is used to create a test suite or a group of related test cases.
- ✧ It takes two arguments: a descriptive string that explains what is being tested and a callback function that contains the test cases that belong to this suite.
- ✧ The purpose of using `describe` is to organize the tests and provide a clear structure to the testing code.

2.4 IT BLOCK

- ✧ This keyword is used to define a single test case.
- ✧ It also takes two arguments: the name of test case and a callback function that contains the actual test logic (test steps).
- ✧ The purpose of using `it` is to clearly indicate what specific functionality is being tested.

All callback function written in arrow function on the above exaple, but we can use nameless function too.

```
describe('Test Suite name', function () {  
  it('Test case Name', function () {  
    // Testp Steps  
    // 1. Visiting page  
    // 2. selecting (find) search box  
    // 3. intracting with element for type on search box  
    // 4. assertion.  
  })  
})
```

Note:-

- ✧ In a test file, we can have multiple “**describe**” blocks, each containing multiple “**it**” blocks. The “**describe**” blocks help in organizing the tests into logical groups, while the “**it**” blocks define the individual test cases within each group.
- ✧ When running the test file, we have the option to selectively skip certain “**describe**” blocks or “**it**” blocks by using the `.skip` method. This allows us to exclude specific tests from running, which can be useful when debugging or temporarily disabling certain tests.
- ✧ To skip a `describe` block, we can add `.skip` after the `describe` keyword, like this:

2.5 SKIP

Example using skipping describe block.

```
Describe.skip('Test Suite name', function (){
  it('Test case Name', function (){
    // Testp Steps
    // 1. Visiting page
    // 2. selecting (find) search box
    // 3. intracting with element for type on search box
    // 4. assertion.
  })
})
```

To skip an “it” block, we can add “.skip” after the “it” keyword, like this:

Example using skipping it block.

```
Describe('Test Suite name', function (){
  It.skip('Test case Name', function (){
    // Testp Steps
    // 1. Visiting page
    // 2. selecting (find) search box
    // 3. intracting with element for type on search box
    // 4. assertion.
  })
})
```

2.6 ONLY

- ✧ In Cypress, the `.only`` keyword is a powerful tool that allows you to focus exclusively on running specific ``describe`` or ``it`` blocks, while ignoring all other tests in the file.
- ✧ When you add `.only`` after a ``describe`` block, only that specific ``describe`` block and its nested ``it`` blocks will be executed, excluding all other ``describe`` and ``it`` blocks.
- ✧ Similarly, when you use `.only`` after an ``it`` block, only that specific ``it`` block will be executed, regardless of its parent ``describe`` block.

Example using only in describe block.

```
Describe.only('Test Suite name', function (){
  it('Test case Name', function (){
    // Testp Steps
    // 1. Visiting page
    // 2. selecting (find) search box
    // 3. intracting with element for type on search box
    // 4. assertion.
  })
})
```

Example using only in it block.

```
Describe('Test Suite name', function () {
  It.only('Test case Name', function () {
    // Testp Steps
    // 1. Visiting page
    // 2. selecting (find) search box
    // 3. intracting with element for type on search box
    // 4. assertion.
  })
})
```

Example write automation test case code by using cypress for Verify search functionality for google home page.

Expected Result:- The search results page should be displayed with relevant results related to the entered query.

2.7 TESTP STEPS

Test steps:-

1. Open the Google homepage
2. Find search box
3. Enter a search query (text) in the search box
4. Find Googel search button
5. Press click on button.

For above example.

```
describe('Google home page testing', () => {
  it('verifying search functionality working', () => {
    // Testp Step
    // 1. Visiting page
    cy.visit('https://www.google.com')
    // 2. selecting (find) search box
    cy.get('') //
    // 3. intracting with element for type on search box.
    .type('cypress tesing tutorial') // to type in search box
    // 4. selecting (find) goole search button
    cy.get('')
    // 5. intracting with element (type, click)
    .click() // to click on button.
    // 4. assertion.
    cy.url().shuld('include','cypress tesing tutorial')
  })
})
```

Home work

1.Create floder with name “Qa_Cypress” on desktope and open it with viscode
install cypress.

2.Create cypress test file named “my_first_cypress_code_home_work”

1. Please write 2 “describe” block on test file under each “describe” block create 5 “it” blocks.

2. Using the skip keyword please skip the first “describe” block and using only keyword please execute the first two “it” block under the second “describe” block.

3. From the below link you can find the test case and perform the Manual test.

Link:-

Google home page for UI testing test case

https://docs.google.com/spreadsheets/d/1yp9MB_J3HE5_zF4x1L7Xn3Xg4cLR7EIM/edit?usp=sharing&ouid=111200036577268897674&rtpof=true&sd=true

Google home page for functional testing test case

<https://docs.google.com/spreadsheets/d/1YnxXWxbyE19torIp4KyzEp3pYzPMvqoo/edit?usp=sharing&ouid=111200036577268897674&rtpof=true&sd=true>

CHAPTER 3:

CYPRESS TEST CASE STRUCTURE

1. Visit page.
2. Query for element from page
3. Interact with element.
4. Assert about element.

1. VISIT PAGE.

For visit page we use `cy.visit(URL)`

Example

```
describe('Visit craft site, () => {  
  it('Should visit the target web page', () => {  
    cy.visit('https://craftknowledge.net/');  
  });  
});
```

In this example, we have a test case named "Visit craft site" within a describe block. The `cy.visit('https://craftknowledge.net/')` command is used to visit the specified URL, which in this case is

URL = "https://craftknowledge.net/".

To set a base URL in Cypress,

You can use the `baseUrl` configuration option in the `cypress.json` file. Here's how you can do it:

```
{  
  "baseUrl": "https://craftknowledge.net/"  
}
```

If the base URL is localhost

```
{  
  "baseUrl": "http://localhost:3000"  
}
```

Once the `baseUrl` is configured in `cypress.json` we can visit using only relative url. For example in craftknowledge site we have the following relative url.

https://craftknowledge.net/home	/home
https://craftknowledge.net/courses	/courses

<https://craftknowledge.net/contact-us> / contact-us
<https://craftknowledge.net/about-us> /about-us

Visiting using relative URL

```
describe('Visit craft site, () => {  
  it('Should visit the target web page', () => {  
    cy.visit('/home');  
  });  
});
```

Activity

1. How can you visit a specific page in Cypress?
2. What is the example code to visit the Craft Knowledge website?
3. How can you set a base URL in Cypress?
4. How can you visit a page using a relative URL in Cypress?
5. What are some examples of relative URLs for the Craft Knowledge website?
3. visit google page.

2. QUERY FOR ELEMENT FROM PAGE

In Cypress, there are several methods you can use to select or locate elements on a web page. Here are some commonly used methods:

1. `cy.get()`: This method is the primary way to select elements in Cypress. It accepts a CSS selector.

```
cy.get("Selector") // Selects all button elements
```

2. `cy.contains()`: This method selects elements based on their text content.

```
cy.contains('Submit') // Selects an element containing the text "Submit"
```

3. `.find()`: This method is used to find elements within a previously selected element.

```
cy.get('.parent-element').find('.child-element') // Selects child elements within a parent
```

4. `cy.eq()`: This method selects a specific element from a collection of elements based on its index. It accepts a zero-based index as a parameter.

```
cy.get('li').eq(2) // Selects the third li element
```

5. `cy.first()` and `cy.last()`: These methods select the first and last element, respectively, from a collection of elements. They do not accept any parameters.

```
cy.get('li').first() // Selects the first li element  
cy.get('li').last() // Selects the last li element
```

2.3 LOCATE OR SELECT ELEMENTS

CSS selectors.

1. **Element Selector** (selects elements by their tag name):

```
cy.get('button') // Selects all button elements
```

2. **Class Selector** (selects elements by their class name):

```
cy.get('.my-class') // Selects all elements with class "my-class"
```

3. **ID Selector** (selects an element by its ID):

```
Cy.get('#my-id') // Selects the element with ID "my-id"
```

4. **Attribute Selector** (selects elements based on their attributes):

```
cy.get('[data-cy="my-data"]') // Selects elements with attribute data-cy="my-data"
```

5. **Combining Selectors** (combines multiple selectors to make more specific selections):

```
cy.get('ul li') // Selects all li elements inside ul elements
```

6. **Child Selector** (selects elements that are direct children of another element):

```
cy.get('ul > li') // Selects li elements that are direct children of ul elements
```

XPath.

CSS selectors are generally recommended for their simplicity and performance, XPath selectors can be useful in certain situations where complex element traversal or attribute-based selection is required

To use XPath selectors in Cypress, you can install the "cypress-xpath" plugin. Here's how you can install it:

Navigate to your Cypress project's root directory.
Open a terminal or command prompt in that directory.
Run the following command to install the "cypress-xpath" plugin.

```
npm install -D cypress-xpath
```

To import the "cypress-xpath" plugin in the support file of your Cypress project, you need to follow these steps:

Open your Cypress project's support file, usually named support/index.js.
Add the following line at the top of the file to import the plugin:

```
import 'cypress-xpath';
```

Now you can use xpath to select element in the different way.

```
cy.xpath("XPath selectors") // Selects elements using an XPath selector
```

Activity

1. Which method is commonly used to select elements in Cypress?
a) cy.contains()
b) cy.eq()
c) cy.get()
d) cy.find()
2. How can you select an element based on its text content?
a) cy.get()
b) cy.contains()
c) cy.find()
d) cy.eq()
3. Which method is used to find elements within a previously selected element?
a) cy.contains()
b) cy.get()
c) cy.eq()
d) cy.find()
4. How can you select a specific element from a collection of elements based on its index?
a) cy.get()

- b) `cy.contains()`
 - c) `cy.eq()`
 - d) `cy.find()`
5. Which method selects the first element from a collection of elements?
- a) `cy.first()`
 - b) `cy.last()`
 - c) `cy.get()`
 - d) `cy.contains()`
6. Which selector is used to select elements by their tag name?
- a) Class Selector
 - b) Element Selector
 - c) ID Selector
 - d) Attribute Selector
7. How can you select elements with a specific class name?
- a) `cy.get()`
 - b) `cy.contains()`
 - c) `cy.eq()`
 - d) `cy.find()`
8. Which selector is used to select an element by its ID?
- a) Class Selector
 - b) Element Selector
 - c) ID Selector
 - d) Attribute Selector
9. Which method is used to combine multiple selectors to make more specific selections?
- a) `cy.get()`
 - b) `cy.contains()`
 - c) `cy.eq()`
 - d) `cy.find()`
10. Which selector is used to select elements that are direct children of another element?
- a) Child Selector
 - b) Element Selector
 - c) ID Selector
 - d) Attribute Selector
11. Locate or select all element in google page using css selectore and Xpath selectores.

3. INTERACT WITH ELEMENT.

Cypress provides a set of action commands that you can use to interact with web elements. These commands allow you to simulate user actions such as clicking, typing, hovering, and more. Here are some of the action commands provided by Cypress:

1. **`.click()`**: Simulates a click on the selected element.

```
cy.get('.my-button').click() // Clicks on an element with the class 'my-button'
```

2. **.type()**: Types text into an input field.

```
cy.get('.my-input').type('Hello, World!') // Enters 'Hello, World!' into an input field
```

3. **.clear()**: Clears the value of an input field.

```
cy.get('.my-input').clear() // Clears the value of an input field
```

4. **.check()** and **.unchecked()**: Checks or unchecks a checkbox or a radio button.

```
cy.get('.my-checkbox').check() // Checks a checkbox  
cy.get('.my-radio').unchecked() // Unchecks a radio button````
```

6. **.select()**: Selects an option from a dropdown menu.

```
cy.get('.my-dropdown').select('Option 1') // Selects 'Option 1' from a dropdown menu
```

7. **.trigger()**: Triggers a specific event on an element, such as a click, mouseover, or keypress event.

```
cy.get('.my-element').trigger('mouseover') // Triggers a mouseover event on the element
```

8. **.focus()** and **.blur()**: Sets focus on an element or removes focus from it.

```
cy.get('.my-input').focus() // Sets focus on an input field  
cy.get('.my-input').blur() // Removes focus from an input field````
```

9. **.dblclick()**: Simulates a double-click on an element.

```
cy.get('.my-element').dblclick() // Performs a double-click on the element
```

10. **.scrollIntoView()**: Scrolls the page so that the element becomes visible in the viewport.

```
cy.get('.my-element').scrollIntoView() // Scrolls to the element with the class 'my-element'````
```

Activity

10 multiple-choice questions related to interacting with elements using Cypress:

1. Which command is used to simulate a click on a selected element?
 - a) `.click()`
 - b) `.type()`
 - c) `.clear()`
 - d) `.check()`
2. How can you enter text into an input field using Cypress?
 - a) `.click()`
 - b) `.type()`
 - c) `.clear()`
 - d) `.check()`
3. Which command is used to clear the value of an input field?
 - a) `.click()`
 - b) `.type()`
 - c) `.clear()`
 - d) `.check()`
4. How can you check a checkbox using Cypress?
 - a) `.check()`
 - b) `.uncheck()`
 - c) `.click()`
 - d) `.type()`
5. How can you select an option from a dropdown menu using Cypress?
 - a) `.click()`
 - b) `.type()`
 - c) `.select()`
 - d) `.check()`
6. Which command is used to trigger a specific event on an element, such as a mouseover or click event?
 - a) `.trigger()`
 - b) `.focus()`
 - c) `.dblclick()`
 - d) `.scrollIntoView()`
7. How can you set focus on an input field using Cypress?
 - a) `.trigger()`
 - b) `.focus()`
 - c) `.dblclick()`
 - d) `.scrollIntoView()`
8. How can you perform a double-click on an element using Cypress?
 - a) `.trigger()`
 - b) `.focus()`
 - c) `.dblclick()`
 - d) `.scrollIntoView()`
9. Which command is used to scroll the page so that an element becomes visible in the viewport?
 - a) `.click()`
 - b) `.type()`
 - c) `.scrollIntoView()`
 - d) `.dblclick()`
10. How can you remove focus from an input field using Cypress?
 - a) `.trigger()`

- b) `.focus()`
- c) `.blur()`
- d) `.scrollIntoView()`

4. ASSERTIONS

Assertions are these validations in the test automation
Assertions classify into two segments based on the subject.

1. Implicit Assertions
2. Explicit Assertions

1. IMPLICIT ASSERTIONS

`.should()`:- This assertion is used for asserting properties or states of DOM elements.
`.and()`:- it is not a specific assertion method in Cypress. However, it can be used to chain multiple assertions together to perform multiple validations on an element or a value.

Implicit Assertions:

- 1 `.should('exist')`: Checks if an element exists in the DOM.
- 2 `.should('not.exist')`: Verifies that an element does not exist in the DOM.
- 3 `.should('be.visible')`: Ensures that an element is visible on the page.
- 4 `.should('not.be.visible')`: Verifies that an element is not visible on the page.
- 5 `.should('be.enabled')`: Checks if an element is enabled.
- 6 `.should('be.disabled')`: Verifies that an element is disabled.
- 7 `.should('have.text', 'expected-text')`: Checks if an element has the expected text.
- 8 `.should('have.value', 'expected-value')`: Verifies that an input element has the expected value.
- 9 `.should('have.attr', 'attribute-name', 'expected-value')`: Checks if an element has a specific attribute with the expected value.
- 10 `.should('have.class', 'expected-class')`: Verifies that an element has the expected CSS class.
- 11 `.should('have.css', 'property-name', 'expected-value')`: Checks if an element has a specific CSS property with the expected value.

2. EXPLICIT ASSERTIONS

`expect()`: This assertion allows you to assert various conditions on values.

`assert()`:

Explicit Assertions:

- 1 `expect(value).to.equal(expectedValue)`: Verifies that the value is equal to the expected value.
- 2 `expect(value).to.not.equal(unexpectedValue)`: Ensures that the value is not equal to the unexpected value.
- 3 `expect(value).to.be.true`: Checks if the value is true.
- 4 `expect(value).to.be.false`: Verifies that the value is false.
- 5 `expect(value).to.be.null`: Ensures that the value is null.

6. `expect(value).to.be.undefined`: Checks if the value is undefined.
7. `expect(value).to.be.empty`: Verifies that the value is empty (an empty string, array, or object).
8. `expect(array).to.include(value)`: Checks if an array includes the expected value.
9. `expect(string).to.contain('substring')`: Verifies if a string contains the specified substring.
10. `expect(value).to.have.length(expectedLength)`: Checks if a value (array, string) has the expected length.

Activity

10 multiple-choice questions related to assertion using Cypress:

1. Which of the following is an implicit assertion in Cypress?
 - a. `expect(value).to.equal(expectedValue)`
 - b. `assert(element).toBeVisible()`
 - c. `expect(array).toContain(value)`
 - d. `assert(response).toHaveStatus(200)`
2. The `"should('not.exist')"` assertion in Cypress is used to:
 - a. Verify that an element does not exist in the DOM
 - b. Check if an element has the expected text
 - c. Ensure that an element is visible on the page
 - d. Validate that an input element has the expected value
3. Which of the following is an explicit assertion in test automation?
 - a. `should('be.disabled')`
 - b. `expect(value).to.be.null`
 - c. `should('have.css', 'property-name', 'expected-value')`
 - d. `assert(element).toHaveAttribute('attribute-name', 'expected-value')`
4. The `"expect(value).to.contain('substring')"` assertion is used to:
 - a. Check if a value is true
 - b. Verify that a value is false
 - c. Ensure that a string contains the specified substring
 - d. Validate if an array includes the expected value
5. Implicit assertions in test automation are classified as:
 - a. `should()` and `and()`
 - b. `expect()` and `assert()`
 - c. `exist()` and `not.exist()`
 - d. `true()` and `false()`
6. The `"expect(value).to.have.length(expectedLength)"` assertion is used to check the length of:
 - a. A DOM element
 - b. A string
 - c. An array
 - d. An object
7. An example of an explicit assertion is:
 - a. `should('be.enabled')`
 - b. `expect(value).to.not.equal(unexpectedValue)`
 - c. `should('have.attr', 'attribute-name', 'expected-value')`
 - d. `assert(element).toBeVisible()`
8. Which assertion is used to validate that an element has a specific CSS class?

- a. `should('have.text', 'expected-text')`
 - b. `expect(value).to.be.true`
 - c. `should('have.class', 'expected-class')`
 - d. `assert(element).toHaveValue('expected-value')`
9. The `"expect(value).to.be.undefined"` assertion is used to check if a value is:
- a. Equal to the expected value
 - b. Not equal to the unexpected value
 - c. Null
 - d. Undefined

CHAPTER - 4

WORKING WITH WEB ELEMENT

3.1 HANDLING TEXT VERIFICATION

In Cypress, you can handle text verification using various assertions and commands provided by the Cypress API. Here's an example of how you can handle text verification in Cypress:

1. Find an element containing the text you want to verify:

```
cy.contains('h1', 'Welcome to Example');
```

This command searches for an h1 element that contains the text "Welcome to Example".

2. Assert the presence of the expected text:

```
cy.contains('h1', 'Welcome to Example').should('be.visible');
```

This assertion verifies that the element is visible on the page.

3. Alternatively, you can assert the exact text value:

```
cy.get('h1').invoke('text').should('eq', 'Welcome to Example');
```

Check example code from:- <https://html-element-for-testing.onrender.com/index.html>

```
describe('Testing html element', () => { it('Text verification', () => {  
  cy.visit("https://html-element-for-testing.onrender.com/index.html");  
  cy.get('#text').should('contain', 'This is h1 element');  
  });  
});
```

3.2 HANDLING INPUT TAG

To handle input tags in Cypress, you can use the `type()` command to simulate typing into an input field and the `should()` assertion to verify the input value. Here's an example:

```
cy.get('input[name="username"]').type('johnsmith');
```

This command simulates typing the text 'johnsmith' into the input field.

Verify the input value:

```
cy.get('input[name="username"]').should('have.value', 'johnsmith');
```

This assertion checks if the input field has the expected value 'johnsmith'.

Check example code from:- <https://html-element-for-testing.onrender.com/index.html>

Text	<input type="text"/>
Email	<input type="text"/>
Password	<input type="password"/>
Number	<input type="text"/>
Textarea	<input type="text"/>

```
describe('Text Box Tests', () => {  
  
  it('should enter text into a text input field', () => {  
    cy.visit("https://html-element-for-testing.onrender.com/Text_Box.html")  
    cy.get('.text').type('Hello, World!');  
    cy.get('.text').should('have.value', 'Hello, World!');  
  });  
  
  it('should enter an email into an email input field', () => {  
    cy.visit("https://html-element-for-testing.onrender.com/Text_Box.html")  
    cy.get('.email').type('example@example.com');  
    cy.get('.email').should('have.value', 'example@example.com');  
  });  
  
  it('should enter a password into a password input field', () => {  
    cy.get('.password').type('secretpassword')  
    cy.get('.password').should('have.value', 'secretpassword');  
  });  
  
  it('should enter a number into a number input field', () => {  
    cy.get('.number').type('42');  
    cy.get('.number').should('have.value', '42');  
  });  
  
  it('should enter text into a textarea', () => {  
    cy.get('.message').type('This is a textarea.');
```

3.5 HANDLING CHECKBOX

To handle checkboxes in Cypress, you can use the `check()` and `unchecked()` commands to select or deselect checkboxes, respectively. Here's an example:

Check the checkbox:

```
cy.get('input[type="checkbox"]').check();
```

This command checks the checkbox element(s).

Verify the checkbox is checked:

```
cy.get('input[type="checkbox"]').should('be.checked');
```

This assertion checks if the checkbox element(s) is checked.

2. Uncheck the checkbox:

```
cy.get('input[type="checkbox"]').unchecked();
```

This command unchecks the checkbox element(s).

```
cy.get('input[type="checkbox"]').should('not.be.checked');
```

This assertion checks if the checkbox element(s) is unchecked.

Check example code from:- https://html-element-for-testing.onrender.com/Check_Box.html

- ☐ checkbox 1
- ☒ checkbox 2
- ☒ checkbox 3
- ☒ checkbox 4

```
describe('Checkbox Tests', () => {  beforeEach(() => {  
  
    it('should check and uncheck a checkbox', () => {  
      cy.visit("https://html-element-for-testing.onrender.com/Check_Box.html");  
      cy.get('#gridCheck1').check();  
      cy.get('#gridCheck2').should('be.checked');  
      cy.get('#gridCheck3').unchecked();  
      cy.get('#gridCheck4').should('not.be.checked');  
    });  
  
    it('should check multiple checkboxes', () => {  
      cy.get('.form-check-input').check({ multiple: true });  
      cy.get('.form-check-input').each(($checkbox)=>{  
        cy.wrap($checkbox).should('be.checked');  
      });  
    });  
  });  
});
```

```
});  
});
```

3.5 HANDLING RADIO

To handle radio buttons in Cypress, you can use the `check()` command to select a radio button option. Here's an example:

Select a radio button option:

```
cy.get('input[type="radio"][name="option"]').check('option1');
```

This command selects the radio button option with the value attribute 'option1'. Replace 'option1' with the actual value attribute of the radio button option you want to select.

Verify the selected radio button option:

```
cy.get('input[type="radio"][name="option"]').should('be.checked');
```

This assertion checks if the radio button option is selected.

Check example code from:- https://html-element-for-testing.onrender.com/Radio_button.html

Radios Button

☒ First radio

☐ Second radio

☐ Third radio

```
describe('Radio Button Tests', () => {  
  it('should select a radio button', () => {  
    cy.visit('https://html-element-for-testing.onrender.com/Radio_button.html');  
    cy.get('.form-check-input').first().check();  
    cy.get('.form-check-input').first().should('be.checked');  
  });  
  
  it('should verify only one radio button can be selected at a time', () => {  
    cy.visit('https://html-element-for-testing.onrender.com/Radio_button.html');  
    cy.get('.form-check-input').eq(0).check();  
    cy.get('.form-check-input').eq(0).should('be.checked');  
    cy.get('.form-check-input').eq(1).check();  
    cy.get('.form-check-input').eq(0).should('not.be.checked');  
    cy.get('.form-check-input').eq(1).should('be.checked');  
    cy.get('.form-check-input').eq(2).check();  
    cy.get('.form-check-input').eq(1).should('not.be.checked');  
    cy.get('.form-check-input').eq(2).should('be.checked');  
  });  
});
```

```
});
```

3.6 HANDLING FILE UPLOAD

To handle file uploads in Cypress, you can use the `cypress-file-upload` plugin. This plugin extends Cypress with additional commands to facilitate file upload testing. Here's how you can install and use the `cypress-file-upload` plugin:

Install the plugin as a dev dependency using npm

```
npm install --save-dev cypress-file-upload
```

In your project's `cypress/support/index.js` file, import the plugin:

```
import 'cypress-file-upload';
```

Now you can use the additional commands provided by the plugin in your Cypress tests.

Handling file uploads in Cypress requires the use of the `fixture()` and `attachFile()` commands. Here's an example:

```
cy.get('input[type="file"]').attachFile('example.txt');
```

This command selects the file upload input element using a CSS selector and attaches the file specified by the fixture name `'example.txt'`.

Verify the file upload:

```
cy.get('input[type="file"]').should('have.attr', 'value', 'example.txt');
```

This assertion checks if the file upload input has the specified file name as its value.

Check example code from:- https://html-element-for-testing.onrender.com/Upload_File.html

File Upload

Choose File

No file chosen

Upload

Uploaded File Name:-

```
describe('File Upload Test', () => {
  it('should upload a file', () => {
    const file = 'example_file.txt'

    cy.visit('https://html-element-for-testing.onrender.com/Upload_File.html');
    cy.get('#formFile').attachFile(file);
    cy.get('#submit').click();
    cy.get('#FileName').should('have.text', 'example_file.txt');

  });
});
```

3.7 HANDLING DROP DOWN

Cypress provides a rich set of built-in commands that you can use to interact with menus on a web page.

Selecting a single option:

```
cy.get('select').select('option1');
```

In this example, the `cy.get('select')` command locates the `<select>` element on the page. The `.select('option1')` command selects the option with the value or text 'option1' within the drop-down menu.

To make an assertion on the selected option within a drop-down menu in Cypress, you can use the `should()` command. Here's an example:

```
cy.get('select').should('have.value', 'option1');
```

In this example, the `cy.get('select')` command locates the `<select>` element on the page. The `.should('have.value', 'option1')` command asserts that the selected option in the drop-down menu has the value 'option1'

Check example code from:- https://html-element-for-testing.onrender.com/Select_Menu.html

Open this select menu

Open this select menu

One
Two
Three

```
describe("Select Menu Verification", () => {  
  it('Selects option "Two"', () => {  
    cy.visit("https://html-element-for-testing.onrender.com/Select_Menu.html");  
    cy.get('#floatingSelect').select("Two");  
    cy.get('#floatingSelect').should("have.value", "2");  
  });  
});
```

3.7 HANDLING AUTO COMPLETE

To handle file auto selector in Cypress, first type in filed provided by selecting the filed and check if the desired option in list of drop down.

Check example code from:- https://html-element-for-testing.onrender.com/Select_Menu.html

a|

Apple

```
describe("Auto Complete Verification", () => {  
  it('Verifies auto complete functionality', () => {  
    cy.visit("https://html-element-for-testing.onrender.com/Auto_Complete.html");  
    cy.get('input#autocomplete-input').type("a");  
    cy.contains('li', 'Apple')  
      .should('contain.text', 'Apple')  
      .and('have.length', 1);  
  });  
});
```

3.8 HANDLING ALERT

To handle alerts in Cypress, you can use the `window:alert` event and the `on()` command to intercept and handle the alert dialog. Here's an example of how to handle alerts in Cypress:

1. To handle normal alert. The focus point is the message displayed on alert and the change after ok pressing button.



```
describe('template spec', () => {
  it('normalAlert', () => {
    // Visit the URL
    cy.visit('https://html-element-for-testing.onrender.com/Alert.html')

    // Click on the button with id "normalAlert"
    cy.get('#normalAlert').click()

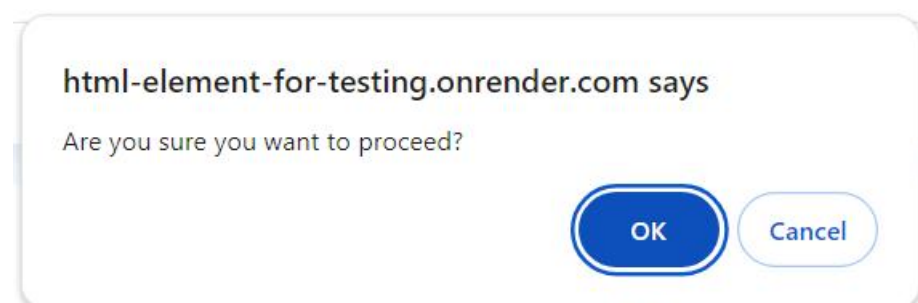
    // Intercept the window:alert event and assert the message
    cy.on('window:alert', (message) => {

      expect(message).to.contains('This is Normal alert')

    })

    // Assert the result element has the expected text
    cy.get('#result').should('have.text', 'Normal Alert')  })
  })
})
```

2. Confirmation Alert: The focus point is the message displayed on alert and the change after ok and cancel button.



```
describe('Alert handling', () => {

  it('confirmAlert', () => {

    // Visit the URL
    cy.visit('https://html-element-for-testing.onrender.com/Alert.html')

    // Click on the button with id "CofirmAlert"
    cy.get('#CofirmAlert').click()

    // Intercept the window:confirm event and assert the confirmation message
    cy.on('window:confirm', (message) => {

      // Assertion on message displayed on alert.
      expect(message).to.contains('Are you sure you want to proceed?')  })

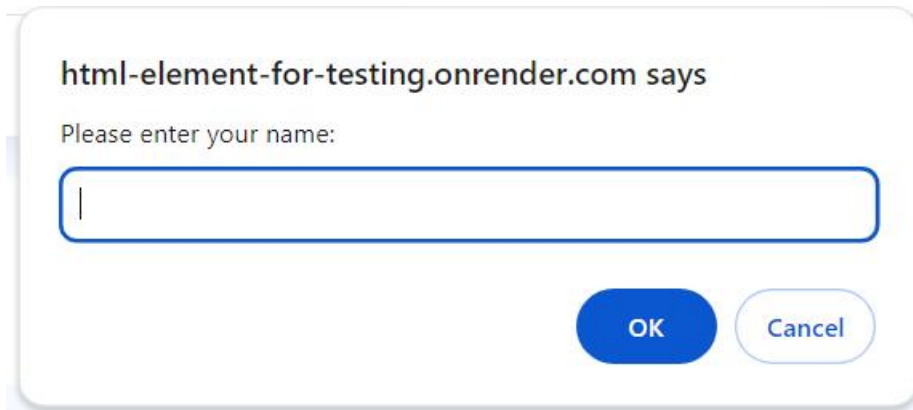
    // Assert the result element has the expected text after clicking "Ok"
    cy.get('#result').should('have.text', 'Ok Clicked')
  })
})
```

```
// To click cancel

cy.on('window:confirm', () => true)
cy.get('#result').should('have.text', 'Ok Clicked')

})
})
```

3. we can made several assertion on prompt alert. On our assertion we haave to consider the text deisplyed on page and input and the result after ok and cancel button.



```
describe('Alert handling', () => {
  it('promptAlert', () => {

    // Visit the URL
    cy.visit('https://html-element-for-testing.onrender.com/Alert.html')

    // Stub the prompt function to return a specific value
    cy.window().then((win) => {

      cy.stub(win, 'prompt').returns('Craft');

    })

    // Click on the button with id "promptAlert"
    cy.get('#promptAlert').click()

    // Assert the prompt alert message
    cy.on('window:alert', (text) => {
      expect(text).to.contains('Please enter your name:')
    })

    // Assert the result element has the expected text
    cy.get('#result').should('have.text', 'Welcome, Craft!');
  })
})
```

3.9 HANDLING WEB TABLE

To handle web tables in Cypress, you can use various Cypress commands and assertions to interact with table elements and perform actions such as reading data, searching for

specific values, and validating the table's content. Here's an example of how you can handle a web table in Cypress:

Check example code from:- <https://html-element-for-testing.onrender.com/Alert.html>

```
describe('Table Test', () => {

  // Test case: Check the existence of the table on the page
  it('should check the existence of the table on the page', () => {
    cy.visit("https://html-element-for-testing.onrender.com/Table.html")
    cy.get('#table').should('exist').and('be.visible')

    })

  // Test case: Check the length of columns
  it('should check the length of columns', () => {
    cy.visit("https://html-element-for-testing.onrender.com/Table.html")
    cy.get('#table-head th').should('have.length', 4)

    })

  // Test case: Check the length of rows
  it('should check the length of rows', () => {
    cy.visit("https://html-element-for-testing.onrender.com/Table.html")
    cy.get('#table-body tr').should('have.length', 5)

    })

  // Test case: Check the name of each column
  it('should check the name of each column', () => {
    cy.visit("https://html-element-for-testing.onrender.com/Table.html")
    cy.get('#table-head th').eq(0).should('have.text', '#')
    // Assert the text of the first column
    cy.get('#table-head th').eq(1).should('have.text', 'Name')
    // Assert the text of the second column
    cy.get('#table-head th').eq(2).should('have.text', 'Position')
    // Assert the text of the third column
    cy.get('#table-head th').eq(3).should('have.text', 'Age')
    // Assert the text of the fourth column

    })

  // Test case: Should display the correct data in the table
  it('should display the correct data in the table', () => {
    cy.visit("https://html-element-for-testing.onrender.com/Table.html")
    cy.get('#table-body tr').eq(0).find('td').eq(0).should('have.text', '1')
    // Assert the data in the first row, first column
    cy.get('#table-body tr').eq(0).find('td').eq(1).should('have.text', 'Brandon Jacob')
    // Assert the data in the first row, second column
    cy.get('#table-body tr').eq(0).find('td').eq(2).should('have.text', 'Designer')
    // Assert the data in the first row, third column
    cy.get('#table-body tr').eq(0).find('td').eq(3).should('have.text', '28')
    // Assert the data in the first row, fourth column

    })
})
```

3.10 HANDLING NEW TAB

Cypress does not have a specific command to work with tabs. In the html code, a link or button opens to a new tab, because of the attribute target. If the target attribute has value blank, it opens to a new tab. The removeAttr deletes the attribute, which is passed as one

Once the target=blank is removed, then a link opens in the parent window. back to the parent URL with the go command.

Check example code from:- https://html-element-for-testing.onrender.com/New_Tab.html

```
describe('New tab handling', () => {
  it('should handle new tab by removing target attribute', () => {
    cy.visit('https://html-element-for-testing.onrender.com/New_Tab.html')
    cy.get('#newTab').invoke('removeAttr', 'target').click()

    // Remove the target attribute and click
    cy.url().should('eq', 'https://html-element-for-testing.onrender.com/sample.html')
    cy.get('h3').should('exist').and('contain', 'This is in side frame')
    cy.go('back')
    // Go back to the previous page
  })
})
```

3.11 HANDLING IFRAMES

To handle iframes in Cypress, you can use the `cy.iframe()` command provided by the `cypress-iframe` plugin. Here's an example of how to handle iframes in Cypress:
First, install the `cypress-iframe` plugin by running the following command in your Cypress project directory:

```
npm install -D cypress-iframe
```

Then, add the following line to your project's `commands.js` file or `support/index.js` file to import and add the plugin:

```
import 'cypress-iframe';
```

Now you can use the `cy.iframe()` command to interact with iframes in your tests. Here's an example:

Check example code from:- <https://html-element-for-testing.onrender.com/Frames.html>

```
import 'cypress-iframe'

describe('Tutorialspoint Test', function () {

  it.only('Test Case6', function () {
    cy.visit('https://html-element-for-testing.onrender.com/Frames.html');
    cy.frameLoaded('#frame1')
    cy.iframe().find('input[type="text"]').type("Efrem").should("have.value", "Efrem")
    cy.iframe().find('input[type="email"]').type("efrem@gail.com").should("have.value", "efrem@gail.com")
    cy.iframe().find('input[type="password"]').type("123dew3").should("have.value", "123dew3")
    cy.iframe().find('input[type="number"]').type("24").should("have.value", "24")
    cy.iframe().find('textarea').type("this is address").should("have.value", "this is address")

  })
})
```

3.11 HANDLING MOUSE EVENTS

In Cypress, handling mouse events is straightforward and can be done using the `.trigger()` command.

1. Normal Button:- In Cypress, to interact with a normal button, you can use the `.click()` command to simulate a button click event. Here's an example of how to click a normal button in Cypress:

```
cy.get('button').click();
```

2. In Cypress, you can simulate a double click event on an element using the `.dblclick()` command. This command allows you to trigger a double click action on a specified element. Here's an example of how to perform a double click in Cypress:

```
cy.get('.my-element').dblclick();
```

3. In Cypress, you can simulate a mouseover event on an element using the `.trigger()` command with passing 'mouseover' as parameter. Here's an example of how to perform a mouseover event in Cypress:

```
cy.get('.my-element').trigger('mouseover');
```

4. In Cypress, you can simulate a right-click (contextmenu) event on an element using the `.trigger()` command.

```
cy.get('.my-element').trigger('contextmenu');
```

Check example code from:- https://html-element-for-testing.onrender.com/Mouse_Event.html

Click Me

Double Click Me

Mouseover on Me

Right Click Me

```

describe("Mouse Event testing", () => {

  it("should perform a normal click", () => {
    cy.visit("https://html-element-for-testing.onrender.com/Mouse_Event.html");
    cy.get('#normalClick').click();
    cy.get("#result").should('have.text', 'Normal Clicked');

  });

  it("should perform a double click", () => {
    cy.visit("https://html-element-for-testing.onrender.com/Mouse_Event.html");
    cy.get('#doubleClick').dblclick();
    cy.get("#result").should('have.text', 'dblclick Clicked');

  });

  it("should perform a mouse over", () => {
    cy.visit("https://html-element-for-testing.onrender.com/Mouse_Event.html");
    cy.get('#mouseOver').trigger('mouseover');
    cy.get("#result").should('have.text', 'mouseover');

  });

  it("should perform a right click", () => {
    cy.visit("https://html-element-for-testing.onrender.com/Mouse_Event.html");
    cy.get('#rightClick').trigger('contextmenu');
    cy.get("#result").should('have.text', 'Right Click');

  });

});

```

CHAPTER - 5

ADVANCED CONCEPT TESTING

5.1 CYPRESS HOOKS CONCEPT

In cypress there is way to Handling the pre and post conditions for a test case or test suite.before/after either each test case in the test suite.before/after all the test cases in the test suite.

Cypress supports several types of hooks:

1. before and after: These hooks run once, before and after all test cases in a test suite, respectively. They are typically used for global setup and teardown tasks that need to be performed before and after the entire test suite.
2. beforeEach and afterEach: These hooks run before and after each individual test case. They are commonly used for setting up and cleaning up test data or performing actions that are required for each test case.

```
describe('Test Suite', () => {  
  before(() => {  
    // Runs once before all test cases in the test suite  
    // Global setup code  
  });  
  after(() => {  
    // Runs once after all test cases in the test suite  
    // Global teardown code  
  });  
  beforeEach(() => {  
    // Runs before each test case  
    // Test case setup code  
  });  
  afterEach(() => {  
    // Runs after each test case  
    // Test case teardown code  
  });  
  it('Test Case 1', () => {  
    // Test case 1 code  
  });  
  it('Test Case 2', () => {  
    // Test case 2 code  
  });  
});
```

Example. This is the login functionality testing code. And we need code optimization. As you see in the code, there is duplication and hardcoding. Let's apply hooks.

```
describe("Testing login functionality ", () => {

  it("Verify that a user can successfully log in with valid credentials.", () => {
    // visit page
    cy.visit("https://end-to-end-v1.onrender.com/");

    // select and interact with element
    cy.xpath("//input[@id='username']").type("test");
    cy.xpath("//input[@id='password']").type("test");
    cy.xpath("//button[@type='submit']").click();

    // assertion about URL https://end-to-end-v1.onrender.com/home/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/home/");

    // assertion about element on home page
    cy.get("h2").should("exist");
    cy.get("h2").should("be.visible");
    cy.get("h2").should("have.text", "Registered Users");
    cy.get("h2")
      .should("exist")
      .and("be.visible")
      .and("have.text", "Registered Users");
  });

  it("Verify that the system behavior with an invalid username and valid password.", () => {

    // visit page
    cy.visit("https://end-to-end-v1.onrender.com/");

    // select and interact with element
    cy.xpath("//input[@id='username']").type("invalidusername");
    cy.xpath("//input[@id='password']").type("test");
    cy.xpath("//button[@type='submit']").click();
    //assertion about error message stating "Invalid credentials"
    cy.xpath("//li[@class='error']").should("have.text", "Invalid credentials")

    //assertion about URL https://end-to-end-v1.onrender.com/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");
  });

  it("Verify that the system with valid username an invalid password.", () => {

    // visit page
    cy.visit("https://end-to-end-v1.onrender.com/");

    // select and interact with element
    cy.xpath("//input[@id='username']").type("test");
    cy.xpath("//input[@id='password']").type("invalidpassword");
    cy.xpath("//button[@type='submit']").click();
    //assertion about error message stating "Invalid credentials"
    cy.xpath("//li[@class='error']").should("have.text", "Invalid credentials")

    //assertion about URL https://end-to-end-v1.onrender.com/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");
  });
});
```



```

});

it("Verify that the system handles invalid login credentials with both an invalid username
and an invalid password.", () => {

    // visit page
    cy.visit("https://end-to-end-v1.onrender.com/");

    // select and interact with element
    cy.xpath("//input[@id='username']").type("invalidusername");
    cy.xpath("//input[@id='password']").type("invalidpassword");
    cy.xpath("//button[@type='submit']").click();
    //assertion about error message stating "Invalid credentials"
    cy.xpath("//li[@class='error']").should("have.text", "Invalid credentials")

    //assertion about URL https://end-to-end-v1.onrender.com/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");

});
});

```

On the above code we have four test case and each test case there is visit command to visit the page before going to assertion. So that this can be handled by beforeEach hook.

After applying hook on the above code this will be.

```

describe("Testing login functionality ", () => {

    beforeEach('visity webpage', ()=>{
        // visit command
        cy.visit("https://end-to-end-v1.onrender.com/");
    })

    it("Verify that a user can successfully log in with valid credentials.", () => {

        // select and interact with element
        cy.xpath("//input[@id='username']").type("test");
        cy.xpath("//input[@id='password']").type("test");
        cy.xpath("//button[@type='submit']").click();

        // assertion about URL https://end-to-end-v1.onrender.com/home/.
        cy.url().should("eq", "https://end-to-end-v1.onrender.com/home/");

        // assertion about element on home page
        cy.get("h2").should("exist");
        cy.get("h2").should("be.visible");
        cy.get("h2").should("have.text", "Registered Users");
        cy.get("h2")
            .should("exist")
            .and("be.visible")
            .and("have.text", "Registered Users");
    });
});

```

```

it("Verify that the system behavior with an invalid username and valid password.", () => {

    // select and interact with element
    cy.xpath("//input[@id='username']").type("invalidusername");
    cy.xpath("//input[@id='password']").type("test");
    cy.xpath("//button[@type='submit']").click();
    //assertion about error message stating "Invalid credentials"
    cy.xpath("//li[@class='error']").should("have.text", "Invalid credentials")

    //assertion about URL https://end-to-end-v1.onrender.com/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");
});

it("Verify system handles invalid login credentials with an invalid password.", () => {

    // select and interact with element
    cy.xpath("//input[@id='username']").type("test");
    cy.xpath("//input[@id='password']").type("invalidpassword");
    cy.xpath("//button[@type='submit']").click();

    //assertion about error message stating "Invalid credentials"
    cy.xpath("//li[@class='error']").should("have.text", "Invalid credentials")

    //assertion about URL https://end-to-end-v1.onrender.com/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");

});

it("Verify system handles invalid login credentials", () => {

    // select and interact with element
    cy.xpath("//input[@id='username']").type("invalidusername");
    cy.xpath("//input[@id='password']").type("invalidpassword");
    cy.xpath("//button[@type='submit']").click();

    //assertion about error message stating "Invalid credentials"
    cy.xpath("//li[@class='error']").should("have.text", "Invalid credentials")

    //assertion about URL https://end-to-end-v1.onrender.com/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");

});
});

```

5.2 FIXTURES IN CYPRESS

We hardcoded the value of the credential in the code above; hence, we would need to modify our test case if those values must changed. The code's maintainability suffers as a result. Cypress gives us a method of handling dynamic data from fixtures to address this problem.

fixtures are a way to define and load external data into your Cypress tests.

To use test data from fixture first define it under fixtures folder. Then you can consume it. The test data can be any file. That mean it can be .txt, .json, .doc etc. For our case we use .json file.

A JSON (JavaScript Object Notation) file is a lightweight data interchange format commonly used to store and exchange data.

Login_Test_Data.json

```
{
  "TestCase1": {
    "username": "test",
    "password": "test"
  },
  "TestCase2": {
    "username": "username",
    "password": "test"
  },
  "TestCase3": {
    "username": "test",
    "password": "password"
  },
  "TestCase4": {
    "username": "username",
    "password": "password"
  }
}
```

Consume this test data from fixture. To consume test data from using `cy.fixture(file_name)`. after applying fixture.

```
describe("Testing login functinality ", () => {

  beforeEach("visity webpage", () => {
    // visit command
    cy.visit("https://end-to-end-v1.onrender.com/");
  });

  it("Verify that a user can successfully log in with valid credentials.", () => {

    // using fixuter
    cy.fixture('examplelog').then((data)=>{

      const logindata = data.TestCase1
      cy.xpath("//input[@id='username']").type(logindata.username);
      cy.xpath("//input[@id='password']").type(logindata.password);
      cy.xpath("//button[@type='submit']").click();

    })

    // assertion about URL https://end-to-end-v1.onrender.com/home/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");

    // assertion about element in on home page
```

```

    cy.get("h2").should("exist");
    cy.get("h2").should("be.visible");
    cy.get("h2").should("have.text", "Registered Users");
    cy.get("h2")
      .should("exist")
      .and("be.visible")
      .and("have.text", "Registered Users");
  })

  it("Verify that the system behavior with an invalid username and valid password.", () => {

    // using fixuter
    cy.fixture('examplelog').then((data)=>{

      const logindata = data.TestCase2
      cy.xpath("//input[@id='username']").type(logindata.username);
      cy.xpath("//input[@id='password']").type(logindata.password);
      cy.xpath("//button[@type='submit']").click();

    })

    //assertion about error message stating "Invalid credentials"
    cy.xpath("//li[@class='error']").should("have.text", "Invalid credentials");

    //assertion about URL https://end-to-end-v1.onrender.com/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");

  });

  it("Verify system handles invalid login credentials with an invalid password.", () => {
    // Using Fixuter
    cy.fixture('examplelog').then((data)=>{

      const logindata = data.TestCase4
      cy.xpath("//input[@id='username']").type(logindata.username);
      cy.xpath("//input[@id='password']").type(logindata.password);
      cy.xpath("//button[@type='submit']").click();

    })

    //assertion about error message stating "Invalid credentials"
    cy.xpath("//li[@class='error']").should("have.text", "Invalid credentials");

    //assertion about URL https://end-to-end-v1.onrender.com/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");

  });

});

```

5.3 CUSTOM COMMANDS IN CYPRESS

Cypress, custom commands allow you to define your own reusable commands that extend the functionality of Cypress. Custom commands can simplify your test code, promote reusability, and improve the readability of your tests.

In the above code we this step depulcated in all test case and its porpuse to ligin in to system.

```
cy.xpath("//input[@id='username']").type(logindata.username);
cy.xpath("//input[@id='password']").type(logindata.password);
cy.xpath("//button[@type='submit']").click();
```

Define your custom commands: In the "commands.js" file, you can define your custom commands using the `Cypress.Commands.add()` method.

For example, let's create a custom command named "login" that performs the login functionality for the above code.

```
Cypress.Commands.add('login', (username, password) => {
  cy.xpath("//input[@id='username']").type(username);
  cy.xpath("//input[@id='password']").type(password);
  cy.xpath("//button[@type='submit']").click();
})
```

Let's use this consume (use) our command in our test code.

```
describe("Testing login functinalty ", () => {

  beforeEach("visity webpage", () => {
    // visit command
    cy.visit("https://end-to-end-v1.onrender.com/");
  });

  it("Verify that a user can successfully log in with valid credentials.", () => {

    // using fixuter
    cy.fixture('examplelog').then((data) => {

      const logindata = data.TestCase1

      // using custom command
      cy.login(logindata.username, logindata.password);

    })

    // assertion about URL https://end-to-end-v1.onrender.com/home/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");

    // assertion about element in on home page
    cy.get("h2").should("exist");
    cy.get("h2").should("be.visible");
    cy.get("h2").should("have.text", "Registered Users");
    cy.get("h2")
      .should("exist")
      .and("be.visible")
      .and("have.text", "Registered Users");
  })

  it("Verify that the system behaber with an invalid username and valid password.", () => {
```

```

// using fixuter
cy.fixture('examplelog').then((data)=>{

    const logindata = data.TestCase2

    // using custom command
    cy.login(logindata.username,logindata.password);

})

//assertion about error message stating "Invalid credentials"
cy.xpath("//li[@class='error']").should("have.text", "Invalid credentials");

//assertion about URL https://end-to-end-v1.onrender.com/.
cy.url().should("eq", "https://end-to-end-v1.onrender.com/");

});

it("Verify that the system behavior with an invalid username and valid password.", () => {

    // using fixuter
    cy.fixture('examplelog').then((data)=>{

        const logindata = data.TestCase3

        // using custom command
        cy.login(logindata.username,logindata.password);

    })

    //assertion about error message stating "Invalid credentials"
    cy.xpath("//li[@class='error']").should("have.text", "Invalid credentials");

    //assertion about URL https://end-to-end-v1.onrender.com/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");

});

it("Verify system handles invalid login credentials with an invalid password.", () => {
    // Using Fixuter
    cy.fixture('examplelog').then((data)=>{

        const logindata = data.TestCase4

        // using custom command
        cy.login(logindata.username,logindata.password);

    })

    //assertion about error message stating "Invalid credentials"
    cy.xpath("//li[@class='error']").should("have.text", "Invalid credentials");

    //assertion about URL https://end-to-end-v1.onrender.com/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");

});

```

```
});
```

5.4 PAGE OBJECT PATTERN IN CYPRESS

The Page Object Pattern is a design pattern commonly used in test automation frameworks, including Cypress, to enhance the maintainability and readability of tests. It helps to separate web page element and its action command from actual test case.

Here's how you can implement the Page Object Pattern in Cypress:

1. Create a Page Object file: Start by creating a new JavaScript file (e.g., "LoginPage.js") in a suitable directory (e.g., "Support/PageObjects") that represents the page you want to interact with. In this file, define a **class that represents** the page and its **related actions and elements**.

```
cy.xpath("//input[@id='username']").type(logindata.username);
cy.xpath("//input[@id='password']").type(logindata.password);
cy.xpath("//button[@type='submit']").click();
```

For example:- in the above code test case for login page and on the login page we have 3 elements and their action commands. Let's define those elements on the page object pattern in a JavaScript file called "**LoginPage.js**" and create a JavaScript class to define each element.

LoginPage.js

```
class LoginPage {

  username(username) {
    cy.get('#username').type(username);
  }

  Password(password) {
    cy.get('#password').type(password);
  }

  submit() {
    cy.get('button').click();
  }
}

export default new LoginPage();
```

In the above example, the LoginPage class encapsulates the login page's actions and elements. Each method represents an action that can be performed on the page, such as visiting the page, filling in the username and password fields, and clicking the login button. The `export default new LoginPage()` statement exports an instance of the LoginPage class.

2. Use the Page Object in tests: In your Cypress test files, import the Page Object file and use its methods to interact with the page. For example:

Using POM.

```
import LoginPage from "../../support/pageObjects/LoginPage";

describe("Testing login functionality ", () => {

  beforeEach("visity webpage", () => {
    // visit command
    cy.visit("https://end-to-end-v1.onrender.com/");
  });

  it("Verify that a user can successfully log in with valid credentials.", () => {

    cy.fixture('examplelog').then((data)=>{

      const mylogin_data = data.TestCase1;

      // Using POM
      LoginPage.username(mylogin_data.username)
      LoginPage.Password(mylogin_data.password)
      LoginPage.submit()
    })

    // assertion about URL https://end-to-end-v1.onrender.com/home/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");

    // assertion about element in on home page
    cy.get("h2").should("exist");
    cy.get("h2").should("be.visible");
    cy.get("h2").should("have.text", "Registered Users");
    cy.get("h2")
      .should("exist")
      .and("be.visible")
      .and("have.text", "Registered Users");
  });

  it("Verify that the system behaber with an invalid username and valid password.", () => {

    // Using Fixuter
    cy.fixture('examplelog').then((data)=>{

      const mylogin_data = data.TestCase2;

      // Using POM
      LoginPage.username(mylogin_data.username)
      LoginPage.Password(mylogin_data.password)
      LoginPage.submit()
    })

    //assertion about error message stating "Invalid credentials"
    cy.xpath("//li[@class='error']").should("have.text", "Invalid credentials");

    //assertion about URL https://end-to-end-v1.onrender.com/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");
  });
});
```



```

it("Verify that the system behavior with an invalid username and valid password.", () => {

    // Using Fixturer
    cy.fixture('examplelog').then((data)=>{

        const mylogin_data = data.TestCase3;

        // Using POM
        LoginPage.username(mylogin_data.username)
        LoginPage.Password(mylogin_data.password)
        LoginPage.submit()
    })

    //assertion about error message stating "Invalid credentials"
    cy.xpath("//li[@class='error']").should("have.text", "Invalid credentials");

    //assertion about URL https://end-to-end-v1.onrender.com/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");

});

it("Verify system handles invalid login credentials with an invalid password.", () => {

    // Using Fixturer
    cy.fixture('examplelog').then((data)=>{

        const mylogin_data = data.TestCase4;

        // Using POM
        LoginPage.username(mylogin_data.username)
        LoginPage.Password(mylogin_data.password)
        LoginPage.submit()
    })

    //assertion about error message stating "Invalid credentials"
    cy.xpath("//li[@class='error']").should("have.text", "Invalid credentials");

    //assertion about URL https://end-to-end-v1.onrender.com/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");

});

});

```

4.5 ENVIRONMENT VARIABLES

Environment variables are variables that hold values related to the environment in which an application or system is running. They are used to store configuration settings, sensitive information, or any other data that may vary depending on the environment.

In the context of Cypress, environment variables can be used to store values that you want to use in your tests, such as API keys, database credentials, or URLs. Environment variables provide a way to keep sensitive information separate from your code and allow

for easy configuration across different environments (e.g., development, staging, production).

Here's how you can work with environment variables in Cypress:

1. **Set environment variables:-** Environment variables can be set in various ways depending on your operating system or CI/CD system. Here are a few common methods:

Using the `cypress.json` file:

You can set environment variables in the `cypress.json` file by adding a "env" property.

```
{
  "env": {
    "URL": "https://end-to-end-v1.onrender.com/"
  }
}
```

2. **Access environment variables:-** in Cypress tests: To access environment variables in your Cypress tests, you can use the `Cypress.env()` function.

For example:

In the above example, `Cypress.env("URL")` retrieves the value of the environment variable named "MY_VARIABLE" and assigns it to the `apiKey` variable.

Let's update the code using environment variables.

```
import LoginPage from "../../support/pageObjects/LoginPage";

describe("Testing login functionality ", () => {

  beforeEach("visity webpage", () => {
    // using environment var
    cy.visit(Cypress.env('URL'));
  });

  it("Verify that a user can successfully log in with valid credentials.", () => {

    cy.fixture('examplelog').then((data)=>{

      const mylogin_data = data.TestCase1;

      // Using POM
      LoginPage.username(mylogin_data.username)
      LoginPage.Password(mylogin_data.password)
      LoginPage.submit()
    })

    // assertion about URL https://end-to-end-v1.onrender.com/home/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");
  });
});
```

```

    // assertion about element in on home page
    cy.get("h2").should("exist");
    cy.get("h2").should("be.visible");
    cy.get("h2").should("have.text", "Registered Users");
    cy.get("h2")
      .should("exist")
      .and("be.visible")
      .and("have.text", "Registered Users");
  });

  it("Verify that the system behavior with an invalid username and valid password.", () => {

    // Using Fixturer
    cy.fixture('examplelog').then((data)=>{

      const mylogin_data = data.TestCase2;

      // Using POM
      LoginPage.username(mylogin_data.username)
      LoginPage.Password(mylogin_data.password)
      LoginPage.submit()
    })

    //assertion about error message stating "Invalid credentials"
    cy.xpath("//li[@class='error']").should("have.text", "Invalid credentials");

    //assertion about URL https://end-to-end-v1.onrender.com/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");

  });

  it("Verify that the system behavior with an invalid username and valid password.", () => {

    // Using Fixturer
    cy.fixture('examplelog').then((data)=>{

      const mylogin_data = data.TestCase3;

      // Using POM
      LoginPage.username(mylogin_data.username)
      LoginPage.Password(mylogin_data.password)
      LoginPage.submit()
    })

    //assertion about error message stating "Invalid credentials"
    cy.xpath("//li[@class='error']").should("have.text", "Invalid credentials");

    //assertion about URL https://end-to-end-v1.onrender.com/.
    cy.url().should("eq", "https://end-to-end-v1.onrender.com/");

  });

  it("Verify system handles invalid login credentials with an invalid password.", () => {

    // Using Fixturer
    cy.fixture('examplelog').then((data)=>{

      const mylogin_data = data.TestCase4;

```

```
// Using POM
LoginPage.username(mylogin_data.username)
LoginPage.Password(mylogin_data.password)
LoginPage.submit()
})

//assertion about error message stating "Invalid credentials"
cy.xpath("//li[@class='error']").should("have.text", "Invalid credentials");

//assertion about URL https://end-to-end-v1.onrender.com/.
cy.url().should("eq", "https://end-to-end-v1.onrender.com/");

});

});
```

CHAPTER - 6

RUNNING CYPRESS AND REPORTING

6.1 RUNNING CYPRESS FROM THE TERMINAL

Running the test cases in UI mode is more suitable when the development of test cases is in progress. But once the development is complete, the user would want to run the test cases in headless mode. Because running test cases in UI are always slower than running in headless mode. Cypress fulfils all these needs and provides ways to execute the test cases from CLI.

1. Running all test case

```
npx cypress run.
```

This command will run your all Cypress tests in the Electron browser by default. The tests will be executed, and the results will be displayed in the terminal.

2. Running single test case.

```
npx cypress run --spec cypress run --spec "path/to/your/test-file.spec.js"
```

This command will run your single Cypress test code in the Electron browser by default. The tests will be executed, and the results will be displayed in the terminal.

3. Running with Specific Browsers

```
npx cypress run --browser <browser-name>
```

Replace <browser-name> with the name of the desired browser. Cypress supports several browser options, including Chrome, Firefox, and Electron (default). For example, to run tests in Chrome, use:

To run in Chrome browser, use the following command.

```
npx cypress run --browser chrome
```

To run in Firefox browser, use the following command.

```
npx cypress run --browser Firefox
```


To run in edge browser, use the following command.

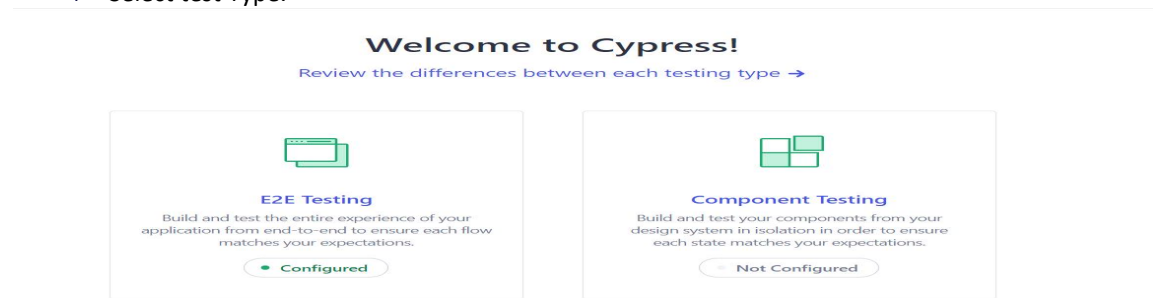
```
npx cypress run --browser edge
```


2. RUNNING CYPRESS FROM TEST RUNNER

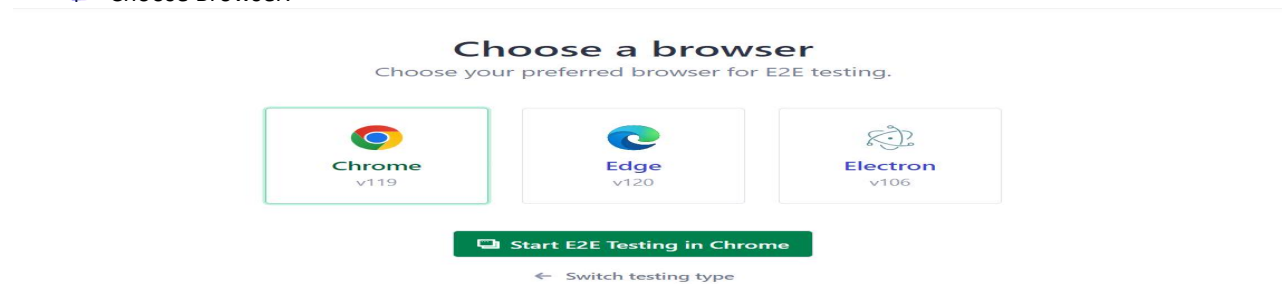
To run with cypress test runner. Use the following command and follow the following steps.


```
npx cypress open
```

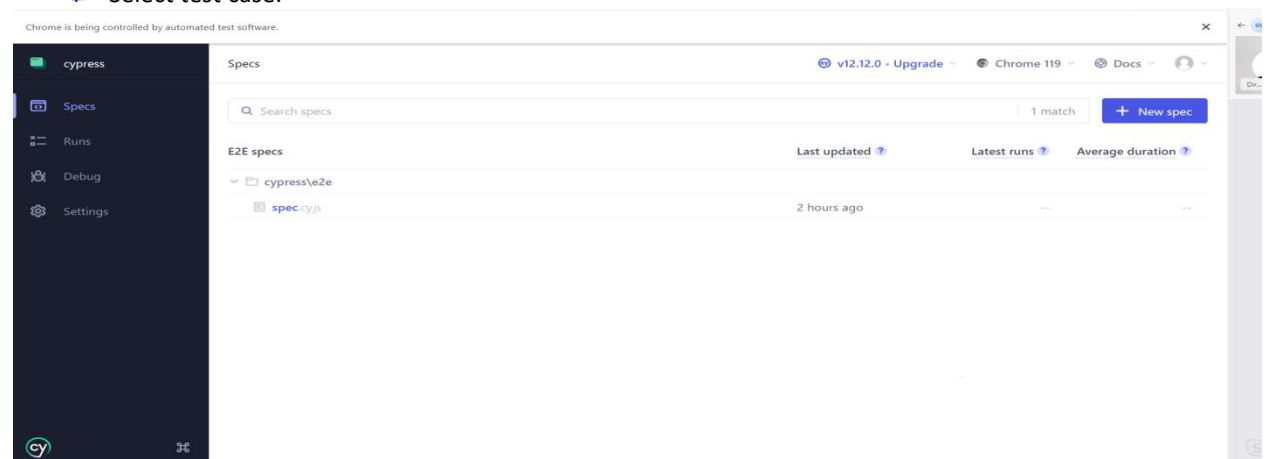
 Select test Type.




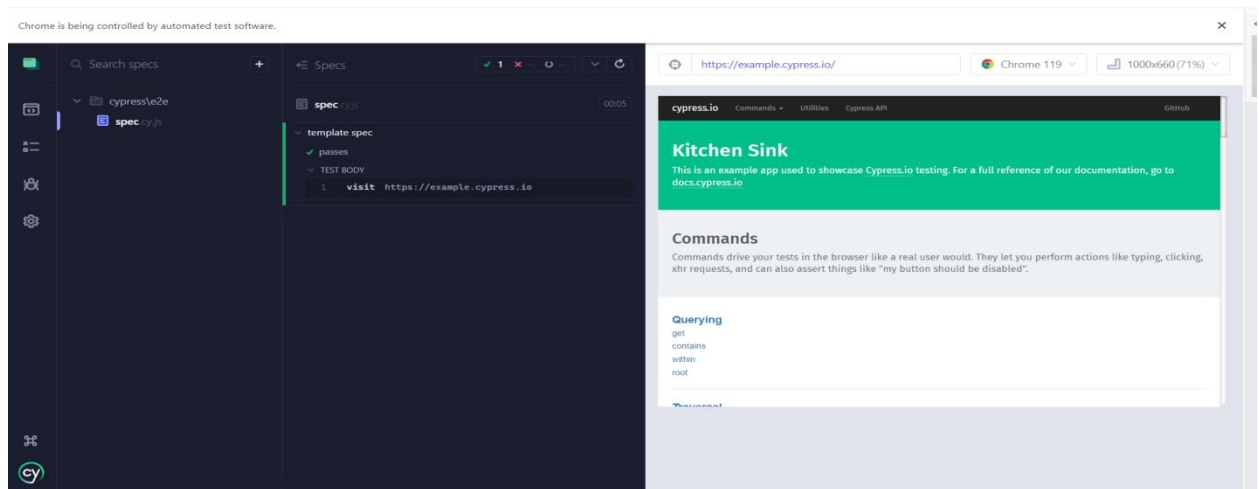
 Choose Browser.



 Select test case.



 This is when running in test runner.



3. RUNNING CYPRESS FROM DASHBOARD

What is Cypress dashboard?

Cypress Dashboard is a web-based component that provides various features related to projects and test runs in Cypress. Cypress Dashboard increases test velocity while giving total visibility into tests running in your CI pipeline. Additionally, it provides the visual representation of the test runs, their reports, and status on a single web window.

Cypress dashboard is helpful.

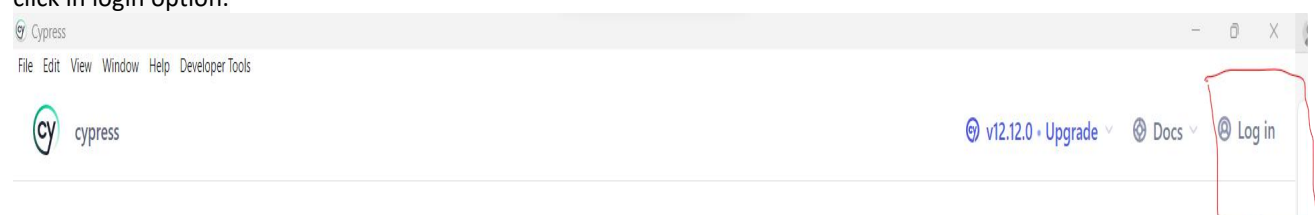
1. Test Stats We can view the number and details of all failed, passed, pending, and skipped tests.
2. Stack Trace We can view the complete stack trace of the failed test cases.
3. View Screenshots We can view all the screenshots taken for the test cases during the test run.
4. View video We can watch a complete video of the test run. Additionally, you can also watch just a clip out of the entire test run.
5. Parallel Tests We can run multiple tests in parallel while running them on CI.
6. Grouping Tests, we can group tests based on specific parameters and run them in one test run.

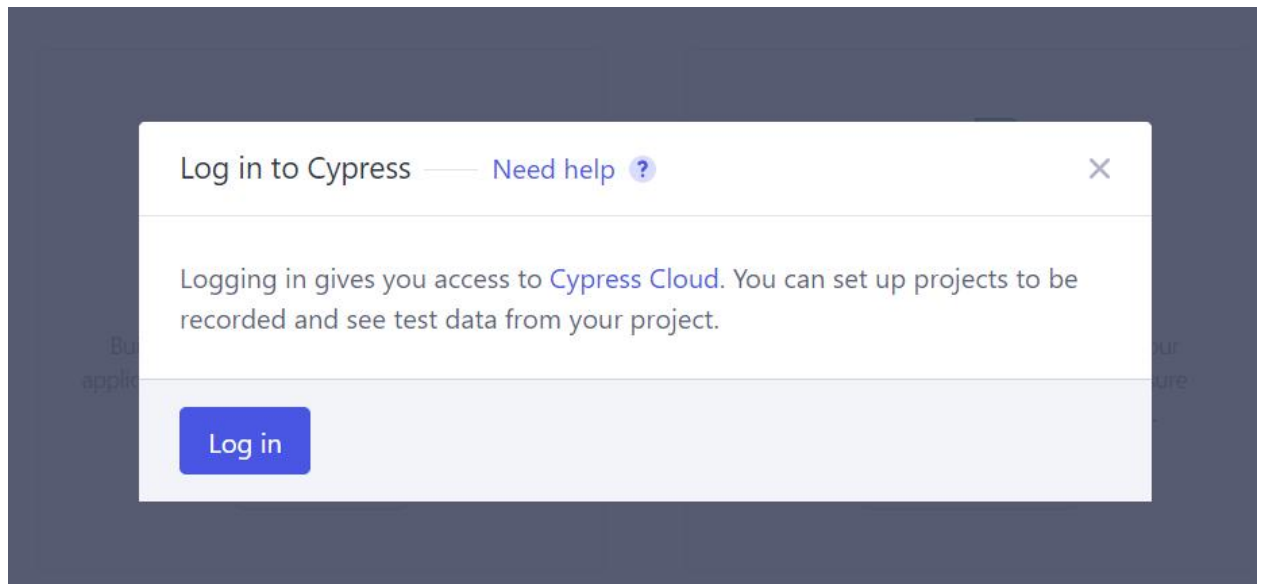
To run in dashboard, we follow the following steps.

To run with cypress test runner. Use the following command and follow the following steps.

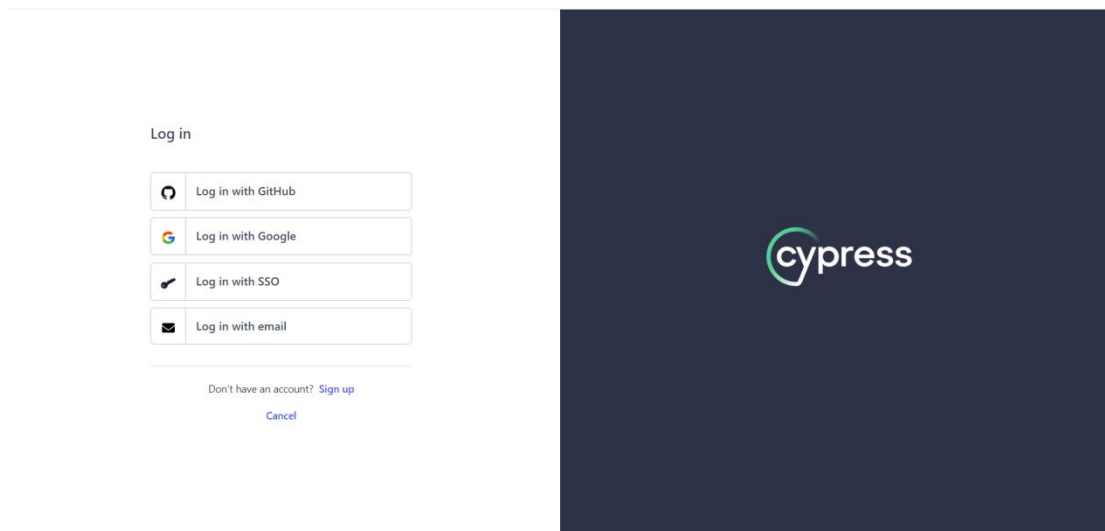
```
npx cypress open
```

click in login option.

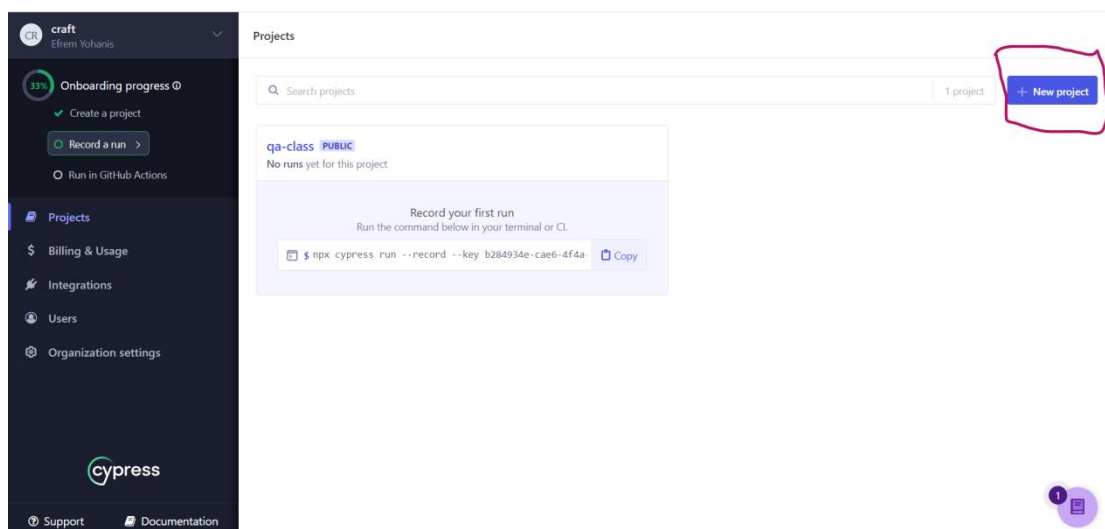




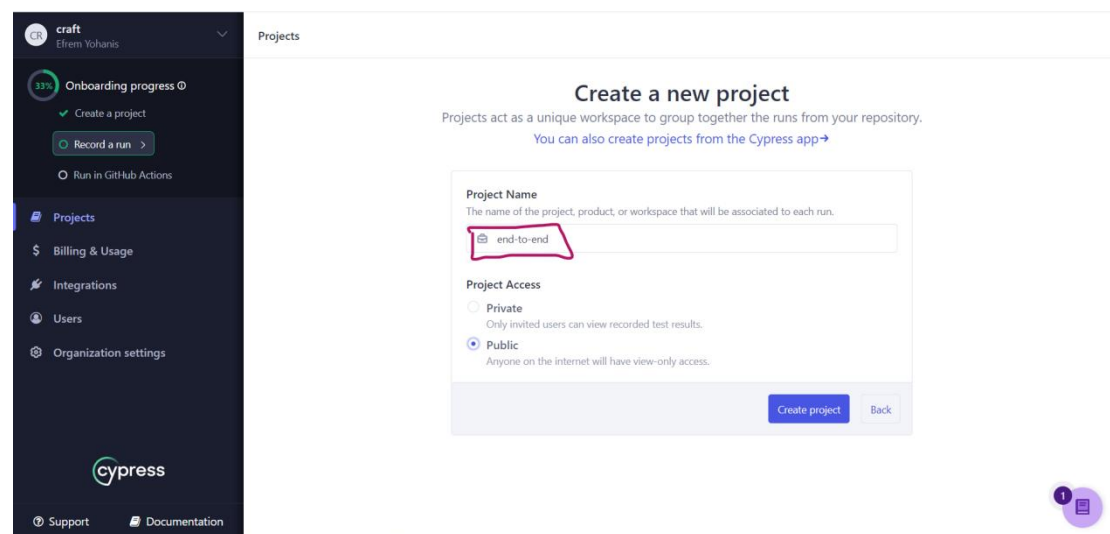
Loing using given option



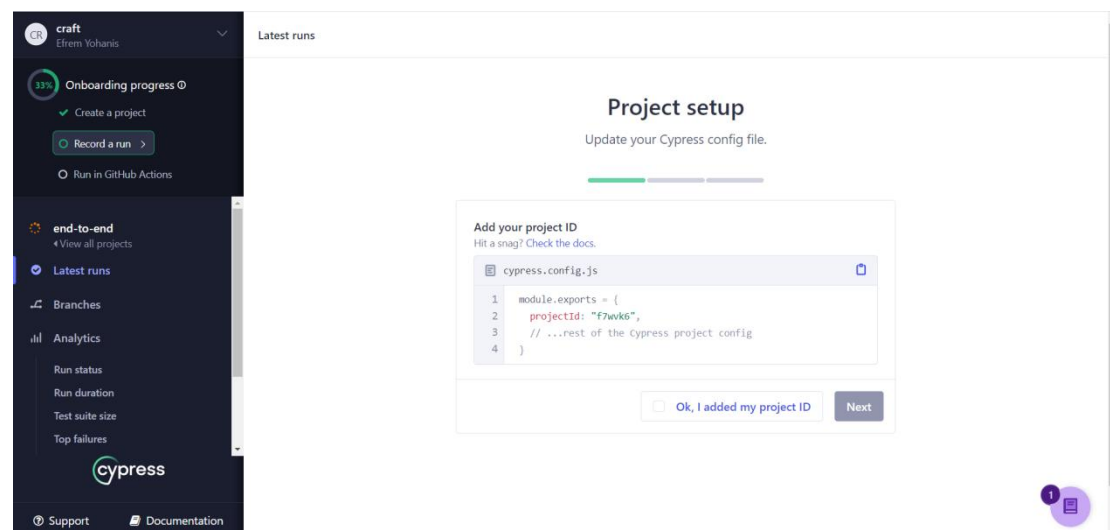
Create new project



Give Project Name.



PROJECT ID



add the project id in to cypress conf file.

```
npx cypress run --record --key 71953534-5f05-416d-96cb-a00b9ad4de28
```

4. CAPTURE SCREENSHOTS, RECORD VIDEO

4.1 Capturing Screenshots:

Cypress automatically captures screenshots by default when a test fails. However, you can also capture screenshots programmatically during your test execution using the `cy.screenshot()` command.

To capture a screenshot at a specific point in your test, you can use the following command:

```
cy.screenshot('filename');
```

The `filename` parameter is optional. If provided, it specifies the name of the screenshot file. If not provided, Cypress will generate a unique filename.

By default, the screenshots are saved in the `cypress/screenshots` folder.

4.2 Recording Video:

Cypress allows you to record videos of your entire test run. By default, it records videos when running tests in interactive mode (with the Cypress Test Runner GUI).

To enable video recording in headless mode, you need to set the video configuration option in your `cypress.json` file or through the command line.

Add the following configuration to your `cypress.json` file:

```
videos: {  
  enabled: true,  
  onTestFailure: true,  
  onTestFinish: false,  
},  
});
```

The `enabled` property is set to `true` to enable video recording.

The `onTestFailure` property is set to `true` to record videos only when a test fails. You
The `onTestFailure` property is set to `true` if you want to record videos for all test runs.
The `onTestFinish` property is set to `false`. If set to `true`, it would record videos for all test runs, regardless of whether they pass or fail.

5. HTML REPORTING

Cypress does not have built-in reporting capabilities for reporting. However, you can use additional tools and plugins to generate reports for your Cypress tests. One popular plugin for generating reports is **mochawesome**.

Here's a **step-by-step** guide on how to prepare a report in Cypress using the **mochawesome** plugin:

step 1: Install the **mochawesome** plugin.

```
npm i --save-dev cypress-mochawesome-reporter
```

step 2: Change cypress reporter & setup hooks

```
const { defineConfig } = require("cypress");

module.exports = defineConfig({
  reporter: 'cypress-mochawesome-reporter',
  video: true,

  e2e: {
    setupNodeEvents(on, config) {
      require('cypress-mochawesome-reporter/plugin')(on);
    },

  },
});
```

Add to cypress/support/e2e.js

```
import 'cypress-mochawesome-reporter/register'
```

Run cypress. With terminal.

```
npx cypress run
```