El Test Runner de Cypress es Mocha JS. Util para probar FrontEnd applications, ya que puede interactuar con el browser directamente y ejecuta el script en el browser como una aplicación de código. Una vez ejecutado en el browser, cypress puede hacer automation dando clicks, encontrando elementos y obteniendo texto de los elementos.

Feature:

1. Se toman snapshots de la aplicación en cada comando; por lo tanto, después de que un caso de prueba sea ejecutado, se verá un command log con los correspondientes screenshot de la aplicación.
2. Tiene la habilidad de stub and spy, lo cual permite tener un control del comportamiento de la función y respuesta con servidores. Se puede modificar el tiempo de ejecución de la aplicación usando funciones de reloj para manipular cosas como set timers, set interval, date and so on.
3. Se puede debugear para ver el estado de la aplicación antes y después de los comandos de ejecución.
4. No es necesario poner esperas adicionales o custom wait (pero se puede agregar si se necesita), puesto que Cypress tendrá cuidado de las esperas necesarias para las assertions.
5. Consistent results como este interactua directamente con la aplicación y el browser.
6. Puedes obtener screenshot y videos.
7. Puedes ejecutar cross browser testing usando remote systems or cloud services.

Cypress se puede usar para ejecutar Unit Tests, Integration Tests, and E2E Tests.

Cypress se diferencia de Selenium por las siguientes características:

* No es compatible con otros lenguajes.
* Todo lo que necesitas está en el mismo framework.
* Tanto desarrolladorees y QA engineers lo pueden usar.
* Native access (acceso a todos los objetos en la página web)
* Trabaja para cualquier frontend framework como Vanilla JS, Angular, VUE JS, REACT JS.
* Está escrito completamente con JavaScript.
* Cypress ocupa solamente css selectors, no se ocupan los otros identificadores como en Selenium. Se tiene que integrar plugins para ocupar xpaths.

Requerimientos de instalación de Cypress:

* Última versión de Node JS con NPM (Node Package Manager).
* Visual Studio Code.
  + Cypress Snippets

Para instalar:

npm init

npm install cypress --save-dev

Se creará la carpeta node\_modules, donde están todos los paquetes instalados. Para validar la instalación de cypress:

node\_modules\.bin\cypress open

Estructura del proyecto de Cypress V7

Cypress

Fixtures Cargar archivo externo a la prueba

Integration\examples Es donde se hacen las pruebas

Plugins Cargar plugins

Support

Node\_modules No se sube al repositorio

Cypress.json

Package-lock.json Son las dependencias instaladas

Package.json Este y el anterior se usan para instalar dependencias de cypress.

Electron es el browser que trae por default cypress. La versión más actualizada es la 130. Está basado en Chromium.

Estructura del proyecto de Cypress 14:

Cypress

downloads Aquí se guardan archivos usando cypress

e2e integration (aquí van los casos de prueba)

fixtures Test data en formato json para los TC

support Aquí vienen los comandos (se colocan comandos para no usar el Page Object Model, sino las prácticas de Cypress. En sí es crear comandos en funciones para ser ocupado fácilmente con herencia) y los index (configuración global como evitar que falle por los js en la página web)

plugins personalizar la experiencia en cypress en un archivo index.js

screenshots

videos

node\_modules Aquí van guardados todos los módulos de npm (librerías)

cypress.json Habilitar o deshabilitar funcionalidades experimentales

cypress.config.js

package-lock.json

package.json Definimos los scripts que se van a ejecutar, dependencia de cypress (version), dirección de GIT, información de usuario.

Cuando se ejecuta:

* Cualquier falla interna de JavaScript en la página provocará que el test falle. Para evitar eso, tenemos que ir al archivo index.js

import './commands'

Cypress.on('uncaught:exception', (err, runnable) => {

// returning false here prevents Cypress from

// failling the test

return false

})

**Ejecutar distintos browser en segundo plano:**

cy:run:chrome --- en el archivo package.js vamos a agregar:

“scripts”:{

“cy:run”: “cypress run”

}

Poner en la línea de comando:

npm run cy:run -b Chrome

“scripts”:{

“cy:run:chrome”: “cypress run -b chrome”,

“cy:run:firefox”: “cypress run -b firefox”

}

Poner en la línea de comando:

npm run cy:run:chrome ó npm run cy:run:firefox

**Forma ideal para seleccionar elementos de la UI:**

La mejor manera es agregar atributos de tipo data-cy. De otra manera, hay que elegir el elemento id, si es que no es dinámico. Por ejemplo:

cy.get(‘[data.cy=username]’).click()

**¿Cómo interacturar con elementos en Cypress?**

Tenemos los parent commands:

cy.visit – Usado para acceder a una URL

cy.get – Usado para encontrar elementos en la UI.

cy.request – Usado para obtener la respuesta HTTP (API Testing)

cy.exec – Ejecuta comandos del sistema.

cy.route – Maneja el comportamiento de la respuesta de la red.

Tenemos los child commands:

.click() – Dar click en la página.

.type() – Agregamos un valor dentro de un inputtext

.find() – Usado para encontrar elementos hijo de un elemento padre seteado con el comando get.

.should() – Usado para hacer assertions.

Tenemos los Dual Commands:s

cy.contains() – Encontramos un elemento a partir de un texto.

cy.screenshot() -

cy.scrollTo() -

cy.wait() -

**Diversas maneras de localizar elementos con css**

Css selector lo tienen todos los atributos del elemento, además del id, name, atributo, clase, innerText, substring, etc. Algunos ejemplos:

Obtener un elemento input:

cy.get(‘input’).should(‘be.disabled’)

cy.get(‘input[name]=”firstName”]’).should(‘have.value’, ‘Homer’)

Encontrar la primera li descendiente dentro de una ul:

cy.get(‘ul li:first’).should(‘have.class’, ‘active’)

Hacer un click en un dropdown:

cy.get(‘.dropdown-menu’).click()

Encontrar 5 elementos con los datos del atributo:

cy.get(‘[data-test-id=”test-example”]’).should(‘have.length’, 5)

Encontrar un elemento con un href que contenga una palabra:

cy.get(‘a[href\*=”questions”]’).click()

Encontrar un elemento que su id comience con una palabra:

cy.get(‘[id^=local-]’)

Encontrar un elemento que su id termine con una palabra:

cy.get(‘[id$=-remote]’)

Encontrar un elemento que empiece con una palabra y termine con otra:

cy.get[‘id^=local-’][id$=-remote]’)

Encontrar elementos con un id que tenga caracteres que se usan en css como “.”, “:”:

cy.get(‘#id\\.\\.\\.1234’)

Encontrar elementos dentro de un determinado elemento, de tal manera que podamos buscar desde ese pivote:

cy.get(‘form’).within(() => {

cy.get(‘input’).type(‘Pamela’)

cy.get(‘textarea’).type(‘is a developer’) })

El uso de Get es para localizar un elemento de todo el documento, haciendo match con el primer elemento. Si no usamos el .within() entonces no podríamos acceder fácilmente a los elementos hijo del elemento. Para ello, podemos ocupar .find para encontrar un elemento a partir del elemento actual.

<div class="test-title">cy.get vs .find</div>  
<section id="comparison">  
 <div class="feature">Both are querying commands</div>  
</section>

cy.get('#comparison')  
 .get('div')  
 // finds the div.test-title outside the #comparison  
 // and the div.feature inside  
 .should('have.class', 'test-title')  
 .and('have.class', 'feature')  
cy.get('#comparison')  
 .find('div')  
 // the search is limited to the tree at #comparison element  
 // so it finds div.feature only  
 .should('have.length', 1)  
 .and('have.class', 'feature')

También podemos usar Alias, los cuales ayudan a no tener que escribir largos css:

cy.get(‘ul#todos’).as(‘todos’)

cy.get(‘@todos’)

Otro ejemplo, para un botón:

beforeEach(() = > {

cy.get(‘button[type=submit]’).as(‘submitBtn’)

})

It(‘disables on click’, () => {

cy.get(‘@submitBtn’).should(be.disabled)

})

Tomar un alias de un archivo de datos:

beforeEach(() => {

cy.fixture(‘user.json’).as(‘users’)

})

it(‘disables on click’, () =>{

cy.get(‘@users’).then((users) =>{

const user = user[0]

cy.get(‘header’).contains(user.name)

})

})

Podemos agregar un plugin para leer elementos por medio de xpath, así como en Selenium. Comando:

npm install -D cypress-xpath

en la ruta support/e2e.js (index.js) agregamos:

require(‘cypress-xpath’)

**Filtros**

Si busco un botón, puedo poner después del css los dos puntos y poner si está o no habilitado el botón que necesito. Ejemplo:

Cy.get(‘#adjfapisfdjapiofas:disabled’) ----- esto ayuda a encontrar el elemento, no es lo mismo a:

Cy.get(‘#adjfapisfdjapiofas’).should(‘not.be.disabled’) ---- esto es un assertion

Pueden ser:

disabled

empty (elemento que no tiene hijos)

checked (para radio buttons de selección)

visible (ayuda por si tiene algún elemento que está oculto, pero habilitado. Solo si se ve)

**Assertions**

Más de 1 argumento:

have.length ayuda a ver cuántos elementos se encontraron:

cy.get('.sc-fEOKFM > :nth-child(3) > .sc-dJkDXt').should('have.length', 11)

have.class ayuda a validar que un elemento tenga una clase x:

cy.get('#email-form-name').should('have.class', 'sc-jdabBK jjJSyU')

have.text ayuda a validar un texto en un elemento:

cy.get('[data-testid="header-container"]').should('have.text', 'Blog')

De solo 1 argumento:

have.value

be.visible

not.be.visible

exist Aunque no se vea, por el hecho de existir en el DOM.

not.exist

be.checked Validar si el radio button está seleccionado.

be.disabled

**¿Cómo debuggear con Cypress?**

log(): Nos ayuda a visualizar un log de lo que está pasando en el script.

cy.log(‘String’)

pause(): Nos ayuda a ir deteniendo la ejecución entre pausas para ir viendo y pausando.

cy.get(‘’).type(‘…..’).pause()

cy.get(‘’).pause().type(‘…..’)

debbug(): Se muestra en consola el valor que nos devuelve o información util para levantar defectos. Nos ayuda a ver qué nos devuelve un comando en la consola de la DevTool.

**Funciones especiales**

Cuando la página abra una segunda ventana, tenemos que evitarlo para continuar con el test. No se puede probar en una segunda ventana, por lo que ocuparemos el siguiente comando para que no nos abra una segunda ventana si damos clic en algún elemento y nos redirecciona. El invoke ejecuta funciones de acuerdo a la necesidad:

cy.contains(‘.----.’) .invoke('removeAttr', 'target').click()

Con ShadowDOM podemos usar lo siguiente:

cy.get(‘css’).shadow().should(‘have.text’, ‘Test’)

cy.get(‘css’).shadow().click()

Para el manejo de elementos dinámicos tenemos las etiquetas de .first(), .last(), .eq(1)

**Assertions implícitas y explícitas**

Validación implícita están dados por el should, y and. Las validaciones explícitas vienen dadas por el expect.

**Promesas**

Handler asociadas a un evneto asíncrono y espera si el evento sale bien o sale mal. Es una especie de try-catch pero flexible. Esperan por ser resueltas o rechazadas de manera asíncrona. cy.promese(), .then, await.

**Callback**

Función que se ejecuta después de que otra función se ejecutó. Sirven para controlar el orden en que se ejecutan las acciones. En el siguiente ejemplo, al botón 1 se le dará click después de que se haya dado click en el botón2.

cy.get(“boton1”).click().then(() =>{

cy.get(“boton2”).click()

})

Un ejemplo de las promesas con callback:

// Define la acción que se pasará como callback

action = () => {

cy.get(‘button’).click()

}

// Define la función que recibirá el callback

performActionWithCallback = (callback) => {

cy.visit(‘www.pagina.com’)

callback();

}

// Llamada a la función con el callback

performActionWithCallback(action)

**cy.task()**

Esto nos permite ejecutar código en el entorno de Node.js donde se ejecuta Cypress. Esto es útil para interactuar con el backend de la aplicación o realizar operaciones fuera del contexto del navegador. Puede ser usado para interactuar on el sistema de archivos, bases de datos, y otros servicios en el backend. Esto está definido en el archivo cypress.config.js

**JQuery**

Cypress ocupa JQuery. Pero con JQuery vuelve al elemento web síncrono, así que no espera al elemento como en Cypress (este es asíncrono). Esto permite que con JQuery podamos crear constantes de elementos, algo que con Cypress no se puede.

$ se puede ocupar para acciones fuera y dentro del DOM

$$ solo hace cosas dentro del DOM

Creación de un test case:

Describe – esto es el equivalente al nombre del feature de cucumber

It --- es cada caso de prueba

beforeEach --- es el background, para que se ejecute antes de cada test case.

TypeScript obliga a indicar el tipo de dato declarado, al contrario de JavaScript, el cual no hay problema si se declara la variable como:

let mensaje;

mensaje = 12; mensaje = ’12’

alert(mensaje);