



Cypress

David Pestana

Version 1.0.0 2021-03-29

# Contenidos

1. Introducción .....	1
2. Cypress vs Selenium .....	2
3. Requisitos previos .....	3
4. Ecosistema .....	4
5. Mocha .....	5
5.1. Describe .....	5
5.2. It .....	5
5.3. Hooks de Mocha .....	6
5.4. Lab: Mocha .....	7
6. Chai .....	12
6.1. Lab: Chai .....	12
7. Sinon .....	18
8. jQuery .....	19
8.1. Selectores .....	19
9. Instalación .....	22
9.1. Lab: Instalación .....	22
10. Ejecución de Tests .....	25
10.1. Desde la interfaz del test runner .....	25
10.2. Desde el terminal .....	25
10.3. Lab: Ejecución de Tests .....	25
11. Estructura de carpetas .....	32
12. Integración con el editor .....	33
12.1. Enlazar el test runner con el editor .....	33
12.2. Autocompletado y ayuda .....	33
12.3. Plugins .....	33
12.4. Lab: Integración con el editor .....	34
13. Buscar elementos .....	39
13.1. Get .....	39
13.2. Find .....	39
13.3. Contains .....	40
13.4. Lab: Buscar elementos .....	40
14. Navegar por el DOM .....	47
14.1. Lab: Navegar por el DOM .....	47
15. Retry ability .....	56
15.1. Lab: Retry ability .....	56
16. Interacciones con elementos .....	62
16.1. Click .....	62
16.2. Lab: Click .....	62

16.3. Lab: Doble Click .....	80
16.4. Cajas de texto (type y clear) .....	84
16.5. Lab: Cajas de texto .....	85
16.6. Submit .....	98
16.7. Lab: Submit .....	99
16.8. Check y uncheck .....	105
16.9. Lab: Checkboxes .....	106
16.10. Desplegables (Select) .....	111
16.11. Lab: Desplegable .....	112
16.12. Lab: Desplegable múltiple .....	118
16.13. Trigger .....	127
16.14. Lab: Drag and Drop .....	128
16.15. Cookies .....	131
16.16. Lab: Cookies .....	132
16.17. Eventos .....	137
16.18. Lab: Popups del navegador .....	138
16.19. Screenshots .....	152
16.20. Lab: Screenshots .....	153
17. Alias .....	162
18. Fixtures .....	163
18.1. Lab: Fixtures .....	163
19. Mocking .....	177
19.1. Spy .....	177
19.2. Stub .....	177
19.3. Lab: Spy y Stub .....	178
19.4. Intercept .....	188
19.5. Lab: Intercept .....	189
20. Peticiones HTTP: Request .....	195
21. Tick y Clock .....	197
21.1. Lab: Tick y Clock .....	197
22. Comandos .....	210
22.1. Lab: Comandos .....	210
23. Test condicionales .....	218
23.1. Lab: Test condicionales .....	218
24. Tests visuales .....	223
24.1. Lab: Tests visuales .....	223
25. Testing de componentes .....	231
25.1. Lab: Testing de componentes con React .....	231
25.2. Lab: Testing de componentes de React con estado .....	244
26. Cypress Studio .....	255
26.1. Lab: Cypress Studio .....	255

27. Dashboard .....	259
27.1. Organizaciones .....	259
27.2. Proyectos .....	260
27.3. Ejecuciones .....	261
27.4. Usuarios .....	261
27.5. Analíticas .....	262
27.6. Integraciones .....	262
28. Generadores de informes .....	263
28.1. Lab: generadores de informes .....	263
29. Cobertura de código .....	274
29.1. Lab: Cobertura de código .....	275

# Capítulo 1. Introducción

Cypress es una herramienta que nos permite testear aplicaciones web modernas y que se ha pensado para solucionar los típicos problemas que los desarrolladores y personas de testing se encuentran en este tipo de aplicaciones, como por ejemplo el manejo de las esperas. También funciona perfectamente con aplicaciones que no sean SPAs o que no se hayan desarrollado con las librerías y frameworks más modernos.

Cypress está construido sobre Node por lo que solo podremos utilizar JavaScript como lenguaje para escribir nuestros tests.

Las principales características de Cypress son:

- **Viaje en el tiempo:** según se van ejecutando los tests, Cypress saca snapshots de la aplicación antes y después de las acciones para que podamos ir viendo que se ha ido haciendo en cada paso del test, pudiendo ver la ejecución de este paso a paso.
- **Debug:** nos permite debuggear los tests directamente desde el test runner y las herramientas del desarrollador. Muchos de los errores que nos muestra son de gran ayuda y se entienden fácilmente.
- **Esperas automáticas:** con Cypress no hace falta que pongamos instrucciones de espera para hacer que la ejecución del test se pare para dar tiempo a que los elementos con los que queremos interactuar se puedan encontrar.
- **Screenshots y videos:** cuando ejecutamos los tests con el CLI, se graba un video con la ejecución del test y se sacan pantallazos cada vez que falla algún test.
- **Resultados consistentes:** como los tests se ejecutan desde el mismo navegador, se consigue que los resultados sean mucho más fiables ya que no depende de que algún intermediario le diga al navegador como tiene que ejecutar algunas instrucciones.

Algunas de las desventajas que podemos encontrarnos al usar esta herramienta son:

- Los tests **se ejecutan directamente dentro del navegador**, por lo que no tiene soporte (por el momento) para utilizar **múltiples pestañas** o los **popups** del navegador, es decir, no podemos testear funcionalidades nativas de los navegadores.
- Tampoco tiene soporte para el **Shadow DOM**.
- Antes de la versión 4 de Cypress solo soportaba **Chrome** y **Electron**. Actualmente soporta algunos navegadores más.
- No se puede visitar distintos dominios en el mismo test.

# Capítulo 2. Cypress vs Selenium

Cypress es un nuevo competidor de Selenium que ha cambiado la forma de funcionar respecto a este segundo. Vamos a ver algunas de las diferencias principales.

Para empezar, la principal diferencia entre Selenium y Cypress es la comunicación con el navegador para ejecutar las pruebas.

Para esto, Selenium utiliza WebDriver como puente entre nuestros tests y el navegador. Desde nuestros tests y con webdriver (que utiliza los drivers de los navegadores) les iremos mandando una serie de instrucciones a los navegadores para que realicen ciertas acciones.

Por el lado de Cypress, los tests se ejecutan sobre el propio navegador sin necesidad de realizar la comunicación con los drivers como hace Selenium.

Entonces al hilo de la primera diferencia, viene la segunda, que es ¿qué lenguajes de programación podemos utilizar para realizar los tests? Pues dado que Cypress se ejecuta dentro del navegador, solo podremos utilizar JavaScript para desarrollar los tests, mientras que con Selenium podemos utilizar cualquier lenguaje (JS, Ruby, Java, Python...) ya que al final el lenguaje utilizado va a interactuar con los drivers de los navegadores indicandole que tienen que ir haciendo en todo momento.

Debido a esta comunicación que necesita Selenium con el navegador, la ejecución de tests con Selenium será mucho más lenta que en Cypress, donde las instrucciones se ejecutan directamente dentro del propio navegador.

Con Selenium podemos utilizar cualquier librería o framework de testing dependiendo del lenguaje utilizado, mientras que con Cypress estamos atados a los de JavaScript (Mocha, Chai...), lo bueno es que ya los trae integrados y no necesitamos realizar configuraciones complejas.

Por el momento, Cypress funciona bien en Chrome o navegadores basados en el, pero no se lleva bien con otro tipos de navegadores (Firefox, Safari, IE...), aunque seguro que ya están trabajando en ello para solucionarlo. Por el lado de Selenium, debido a que se usan los drivers de los navegadores para realizar las acciones sobre estos, pues tiene un mayor soporte y podemos utilizarlo con todos ellos sin ningún problema más allá de los bugs que tenga Selenium.

# Capítulo 3. Requisitos previos

Para poder empezar a trabajar con Cypress necesitaremos tener instalado lo siguiente:

- Node: <https://nodejs.org/en/> (Recomendado descargar la versión LTS)
- NPM: viene instalado con Node.
- Un editor de código:
  - Visual Studio Code: <https://code.visualstudio.com/> (Recomendado)
  - Sublime Text: <https://www.sublimetext.com/>
  - Atom: <https://atom.io/>
  - WebStorm: <https://www.jetbrains.com/es-es/webstorm/> (De pago. Gratis para estudiantes)

# Capítulo 4. Ecosistema

Cypress está enfocado a tests e2e, pero también incluye una serie de librerías enfocadas a otro tipo de pruebas y que podemos utilizar como:

- **Mocha**: es una librería que se encarga de agrupar los tests y ejecutarlos.
- **Chai**: es una librería que nos proporciona muchas más aserciones de las que podemos utilizar con las que vienen por defecto.
- **Sinon**: es una librería para crear mocks, stubs y spies en nuestros tests.

Además de las librerías para testing de antes, también incluye **jQuery**, una librería que nos permite buscar elementos de DOM e interactuar con ellos.

# Capítulo 5. Mocha

Mocha es un framework de pruebas de JavaScript que se ejecuta tanto en Node como en el navegador. Nos permite crear tests en serie para nuestras aplicaciones. Con él conseguimos ejecutar y organizar las pruebas.

Este framework se puede utilizar con cualquier librería de aserciones.

## 5.1. Describe

Con **describe** generamos un bloque de tests en el que agrupamos aquellas que están relacionadas entre sí.

La función **describe** recibe como parámetros un texto que define qué se va a testear en este bloque y una función en la que vamos a definir estos tests.

```
const { describe } = require('mocha');

describe('Vamos a testear la funcionalidad Y', () => {
  // ...
})
```

## 5.2. It

Con **it** generamos un test para probar que alguna parte de nuestra aplicación funciona correctamente. En estos bloques podemos testear una misma funcionalidad pero siguiendo distintos flujos, por ejemplo, testeamos un registro de usuario con datos correctos, con un email invalido, sin llenar un campo...

La función **it** recibe como parámetros un texto que define qué se va a testear y una función en la que vamos a escribir las instrucciones necesarias para probar una funcionalidad de la aplicación.

```
const { describe, it } = require('mocha');

describe('Vamos a testear la funcionalidad Y', () => {
  it('debería devolver un número si se le pasa un número', () => {
    // ...
  })

  it('debería devolver un string si se le pasa un string', () => {
    // ...
  })
})
```

## 5.3. Hooks de Mocha

Los hooks de Mocha son una serie de funciones que se ejecutan en unos ciertos momentos de la ejecución de los tests y que nos permiten inicializar variables o realizar tareas de limpieza entre otras cosas.

Algunas tareas que se suelen realizar en estas funciones suelen ser:

- Reniciar el estado de la aplicación para que las modificaciones hechas en un test no afecten al siguiente.
- Inicializar datos genericos a los tests.
- Limpiar la BBDD.

Nos encontramos con los siguientes:

- **before**: este hook solo se ejecuta una vez. Justo antes del primer test del bloque.
- **beforeEach**: este hook se ejecuta justo antes de cada uno de los tests.
- **afterEach**: este hook se ejecuta justo después de cada uno de los tests.
- **after**: este hook solo se ejecuta una vez. Justo después del último test del bloque.

Cada una de estas funciones recibe como parámetro una función en la que vamos a añadir el código que queremos ejecutar en el momento en que se ejecuta el hook.

```

describe('Test ...', () => {
  before(() => {
    console.log('Se ejecuta solo 1 vez al principio de todo')
  })

  beforeEach(() => {
    console.log('Se ejecuta 1 vez antes de cada it')
  })

  afterEach(() => {
    console.log('Se ejecuta 1 vez después de cada it')
  })

  after(() => {
    console.log('Se ejecuta solo 1 vez al final de todo')
  })

  it('debería ...', () => {
    console.log('IT 1')
  })

  it('debería ...', () => {
    console.log('IT 2')
  })
})

```

La salida de la ejecución de este archivo de test es la siguiente:

```

Test ...
Se ejecuta solo 1 vez al principio de todo
Se ejecuta 1 vez antes de cada it
IT 1
  ✓ debería ...
Se ejecuta 1 vez después de cada it
Se ejecuta 1 vez antes de cada it
IT 2
  ✓ debería ...
Se ejecuta 1 vez después de cada it
Se ejecuta solo 1 vez al final de todo

```

## 5.4. Lab: Mocha

En este laboratorio vamos a familiarizarnos con la sintaxis de Mocha a la hora de organizar los tests.

Empezamos creando una carpeta **cypress-mocha-lab**.

Entramos a ella desde el terminal y vamos a inicializar un proyecto de NPM con el siguiente

comando:

```
$ npm init -y
```

El siguiente paso es instalar las dependencias necesarias para utilizar mocha, y para ello lanzamos el siguiente comando:

```
$ npm install --save-dev mocha
```

Al lanzar este comando, nuestro archivo **package.json** queda de la siguiente forma:

*/cypress-mocha-lab/package.json*

```
{
  "name": "cypress-mocha-lab",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "mocha": "^8.3.0"
  }
}
```

Ahora vamos a añadir un par de scripts de testing para ejecutar nuestros tests con mocha:

- **test** solo va a ejecutar los tests.
- **test:watch** va a estar pendiente de los cambios que realicemos para ejecutar automáticamente los tests.

/cypress-mocha-lab/package.json

```
{  
  "name": "cypress-mocha-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "mocha './src/**/*spec.js'",  
    "test:watch": "mocha './src/**/*spec.js' --watch"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "mocha": "^8.3.0"  
  }  
}
```

Ya tenemos mocha configurado. Ahora vamos a crearnos alguna función y los tests.

Vamos a crear una carpeta **src** con un archivo **app.js** y otro **app.spec.js**

Dentro del primer archivo vamos a crear una función simple que devuelva **Hola** seguido del nombre que se le pase como parámetro. El nombre va a tener como valor por defecto **Mundo**.

/cypress-mocha-lab/src/app.js

```
function hola(nombre = 'Mundo') {  
  return 'Hola ' + nombre;  
}
```

Además, tenemos que exportar la función para poder importarla después en el archivo de testing.

/cypress-mocha-lab/src/app.js

```
function hola(nombre = 'Mundo') {  
  return 'Hola ' + nombre;  
}  
  
module.exports = {  
  hola  
}
```

Ya tenemos la función que queremos probar. Así que solo nos queda crear los tests para dicha función.

Vamos a empezar por importar aquellas funciones que necesitamos para crear nuestros tests:

- describe

- it
- assert
- hola

/cypress-mocha-lab/src/app.spec.js

```
const assert = require('assert');
const { hola } = require('./app.js');
```

Ahora tenemos que crear un grupo de tests con la función **describe** e indicar que es lo que vamos a testear en dicho bloque de código.

/cypress-mocha-lab/src/app.spec.js

```
const assert = require('assert');
const { hola } = require('./app.js');

describe('Test función hola', () => {

})
```

Dentro de este bloque, vamos a crear un primer caso de prueba en el que comprobaremos que si no le pasamos ningún nombre nos devuelve el string **Hola Mundo**. Para ello, vamos a utilizar la función **it** a la que le pasamos la descripción del test.

/cypress-mocha-lab/src/app.spec.js

```
const assert = require('assert');
const { hola } = require('./app.js');

describe('Test función hola', () => {
  it('debería saludar a Mundo si no le pasamos ningún nombre como argumento', () => {
    assert.ok(hola() === 'Hola Mundo');
  })
})
```

Ahora que tenemos nuestra primera prueba, vamos a ejecutar el siguiente comando para comprobar que se pasa correctamente:

```
$ npm test
```

Nos aparecerá por consola algo como lo siguiente.

### Test función hola

└ debería saludar a Mundo si no le pasamos ningún nombre como argumento

1 passing (6ms)

Podemos probar a cambiar el resultado esperado para comprobar que da error el test.

Vamos a por la segunda prueba. Esta vez vamos a añadir otro bloque **it** para comprobar que si se le pasa un nombre cualquiera como parámetro, el valor que devuelve la función es **Hola** seguido de dicho nombre.

/cypress-mocha-lab/src/app.spec.js

```
const assert = require('assert');
const { hola } = require('./app.js');

describe('Test función hola', () => {
  it('debería saludar a Mundo si no le pasamos ningún nombre como argumento', () => {
    assert.ok(hola() === 'Hola Mundo');
  })

  it('debería saludar a Charly si le pasamos Charly como argumento', () => {
    assert.strictEqual(hola('Charly'), 'Hola Charly');
  })
})
```

Y al igual que antes, vamos a lanzar el script que ejecuta los tests. Esta vez podemos probar a lanzar el otro script para comprobar que se queda mirando los cambios en los archivos para volver a ejecutar los tests cuando se realiza cualquier modificación sobre el código.

\$ npm run test:watch

Esta vez, veremos un mensaje por consola como el siguiente:

### Test función hola

└ debería saludar a Mundo si no le pasamos ningún nombre como argumento  
└ debería saludar a Charly si le pasamos Charly como argumento

2 passing (3ms)

└ [mocha] waiting for changes...

Y con esto ya tenemos unas nociones básicas sobre el funcionamiento de Mocha.

# Capítulo 6. Chai

A la hora de testear nuestras aplicaciones, ya sean test unitarios, de integración, e2e... vamos a necesitar comprobar ciertos predicados, es decir, comprobar que lo que se dice es cierto. Y para esto necesitaremos alguna función que nos permita utilizar aserciones.

NodeJS tiene un módulo de aserciones, **assert** que podríamos utilizar para comprobar que los predicados son correctos. El problema es que dicho módulo es muy simple.

En su lugar, existe una librería de aserciones llamada Chai.js y que está integrada dentro de Cypress, por lo que no necesitamos configurar ni instalar nada para utilizarla junto a él. Esta librería nos proporciona una serie de aserciones de distintos tipos y estilos:

- **TDD**: aserciones más clásicas.
- **BBD**: aserciones que usan un lenguaje más expresivo y legible.

## 6.1. Lab: Chai

En este laboratorio vamos a ver algunas de las aserciones que nos proporciona Chai para utilizar en nuestros tests.

Empezamos creando una carpeta **cypress-chai-lab**.

```
$ mkdir cypress-chai-lab
```

Entramos a ella desde el terminal y vamos a inicializar un proyecto de NPM con el siguiente comando:

```
$ cd cypress-chai-lab  
$ npm init -y
```

Necesitamos instalar las dependencias de mocha y chai con el siguiente comando:

```
$ npm install --save-dev mocha chai
```

Después de instalar las dependencias nuestro archivo **package.json** queda como vemos a continuación:

/cypress-chai-lab/package.json

```
{  
  "name": "cypress-chai-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "chai": "^4.3.0",  
    "mocha": "^8.3.0"  
  }  
}
```

Ahora vamos a agregar los scripts de NPM para testing en el archivo **package.json**.

/cypress-chai-lab/package.json

```
{  
  "name": "cypress-chai-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "mocha"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "chai": "^4.3.0",  
    "mocha": "^8.3.0"  
  }  
}
```

Cuando se ejecutan los tests con Mocha, por defecto busca una carpeta con el nombre **test**, así que vamos a crearla y dentro de ella crearemos también el archivo **aserciones.spec.js**.

Lo primero que haremos dentro de este archivo es importar los dos métodos de aserciones que vamos a utilizar, el **expect** y el **should**.



**should** es una función que se añade a si misma sobre los distintos tipos de objetos de JavaScript.

/cypress-chai-lab/test/aserciones.spec.js

```
var expect = require('chai').expect;
var should = require('chai').should();

describe('Aserciones de Chai', () => {
})
```

Ahora vamos a crear nuestro primer bloque de test para ver algunas aserciones. Empezaremos haciendo pruebas con números.

Utilizaremos el método **expect** pasandole el valor que estamos calculando y luego vamos a encadenar una serie de propiedades y métodos propias de Chai que hacen las aserciones más legibles.

/cypress-chai-lab/test/aserciones.spec.js

```
var expect = require('chai').expect;
var should = require('chai').should();

describe('Aserciones de Chai', () => {
  it('nums', () => {
    const num1 = 3;
    const num2 = 2;
    expect(num1 + num2).to.equal(5);
  });
})
```

Esta misma aserción podemos hacerla también con **should**, donde primero le indicamos el valor a testear, y después vamos concatenando unas propiedades y métodos al igual que antes para completar nuestra aserción.

/cypress-chai-lab/test/aserciones.spec.js

```
var expect = require('chai').expect;
var should = require('chai').should();

describe('Aserciones de Chai', () => {
  it('nums', () => {
    const num1 = 3;
    const num2 = 2;
    expect(num1 + num2).to.equal(5);
    (num1+num2).should.equal(5);
  });
})
```

Podemos probar el test lanzando el script de NPM que habíamos añadido:

```
$ npm test
```

Ahora vamos a comprobar que estas aserciones también funcionan con strings, por lo que comprobaremos que un texto dado se puede convertir en mayúsculas usando las mismas aserciones que con los números.

/cypress-chai-lab/test/aserciones.spec.js

```
var expect = require('chai').expect;
var should = require('chai').should();

describe('Aserciones de Chai', () => {
  it('nums', () => {
    const num1 = 3;
    const num2 = 2;
    expect(num1 + num2).to.equal(5);
    (num1+num2).should.equal(5);
  });

  it('texto', () => {
    const texto = 'hola mundo';
    const textoMayus = texto.toUpperCase();
    expect(textoMayus).to.equal('HOLA MUNDO');
    textoMayus.should.be.equal('HOLA MUNDO');
  })
})
```

Vamos a utilizar alguna aserción distinta con los objetos de tipo array. Vamos a comprobar que la longitud del array es correcta, que el array incluye un valor dado o incluso podemos comprobar que el objeto en si es del tipo array.

```
var expect = require('chai').expect;
var should = require('chai').should();

describe('Aserciones de Chai', () => {
  it('nums', () => {
    const num1 = 3;
    const num2 = 2;
    expect(num1 + num2).to.equal(5);
    (num1+num2).should.equal(5);
  });

  it('texto', () => {
    const texto = 'hola mundo';
    const textoMayus = texto.toUpperCase();
    expect(textoMayus).to.equal('HOLA MUNDO');
    textoMayus.should.be.equal('HOLA MUNDO');
  })

  it('array', () => {
    const mascotas = ['perro', 'gato', 'loro', 'canario'];
    expect(mascotas).to.be.a('array');
    expect(mascotas).to.have.lengthOf(4);
    expect(mascotas).to.include('loro');
    mascotas.should.to.be.a('array');
    mascotas.should.have.lengthOf(4);
    mascotas.should.include('loro');
  })
})
```

Por último, vamos a utilizar un objeto para comprobar que tiene una propiedad dada y que una de sus propiedades es false.

```
var expect = require('chai').expect;
var should = require('chai').should();

describe('Aserciones de Chai', () => {
  it('nums', () => {
    const num1 = 3;
    const num2 = 2;
    expect(num1 + num2).to.equal(5);
    (num1+num2).should.equal(5);
  });

  it('texto', () => {
    const texto = 'hola mundo';
    const textoMayus = texto.toUpperCase();
    expect(textoMayus).to.equal('HOLA MUNDO');
    textoMayus.should.be.equal('HOLA MUNDO');
  })

  it('array', () => {
    const mascotas = ['perro', 'gato', 'loro', 'canario'];
    expect(mascotas).to.be.a('array');
    expect(mascotas).to.have.lengthOf(4);
    expect(mascotas).to.include('loro');
    mascotas.should.to.be.a('array');
    mascotas.should.have.lengthOf(4);
    mascotas.should.include('loro');
  })

  it('objeto', () => {
    const serie = {
      titulo: 'The Walking Dead',
      numTemporadas: 10,
      finalizada: false
    }
    expect(serie).to.have.property('titulo');
    expect(serie.finalizada).to.be.false;
    serie.should.have.property('titulo');
    serie.finalizada.should.be.false;
  })
})
```

Al lanzar los tests deberían de pasarse las aserciones sin problemas.

Siempre está bien comprobar que estas aserciones fallan, negando la aserción o cambiando el valor con el que se va a comparar.

# Capítulo 7. Sinon

Al testear nuestras aplicaciones podemos encontrarnos con casos como:

- Una funcionalidad que estamos testeando depende de un valor que no controlamos nosotros.
- Una funcionalidad que estamos testeando tiene que hacer una llamada a una API externa de pago.
- Una funcionalidad que estamos testeando modifica una BBDD.
- ...

Son casos en los que o no podemos controlar los datos que vamos a recibir y por tanto el test puede seguir un camino que no esperamos, o tenemos que configurar una BBDD para el entorno de pruebas, o estamos usando servicios de pago que pueden llegar a hacer que gastemos dinero por probar nuestra aplicación.

**Sinon** es una librería de JavaScript que nos permite utilizar **spies**, **stubs** y **mocks**, o como se les suele llamar, dobles.

- **Spy**: es una función que nos permite crear un doble de otra función y comprobar cosas, como por ejemplo cuántas veces se ha llamado a la función, o si se ha llamado con los parámetros correctos.
- **Stub**: es una función que nos permite crear un doble de otra y modificar su comportamiento para que funcione como nosotros queremos. Podríamos usarla para hacer que cuando se haga una petición a una API, nosotros le digamos qué respuesta nos tiene que dar, sin que se llegue a realizar la petición.
- **Mock**: es una mezcla de los dos anteriores, y permite predefinir aserciones, como que si se llama a la función con un valor x como parámetro entonces se espera que el resultado sea false.

# Capítulo 8. jQuery

jQuery es una librería de JavaScript que en su día era una de las más usadas en el desarrollo de aplicaciones, concretamente en la parte del frontend ya que proporcionaba a los desarrolladores la posibilidad de hacer los sitios mucho más interactivos.

Con jQuery podemos manipular el DOM sin necesidad de recargar la página, mediante la gestión de eventos y el uso de AJAX para pedir datos al servidor.

En el caso de Cypress, jQuery se va a usar para la búsqueda de los elementos de las aplicaciones con los que queremos interactuar. Por tanto, utilizaremos los selectores de jQuery a la hora de buscar algún elemento con Cypress.

## 8.1. Selectores

Para buscar los elementos con los que queremos interactuar necesitamos conocer los distintos selectores que podemos utilizar.

Los métodos que tiene Cypress para buscar elementos utilizan los selectores de jQuery.

Nombre	Selector	Descripción
Global	*	Selecciona <b>todos los elementos</b> del DOM.
Etiqueta	h1	Selecciona todas las etiquetas <b>h1</b> del DOM.
Id	#titulo	Selecciona el elemento del DOM que tiene un atributo <b>id="titulo"</b> .
Clase	.btn-primary	Selecciona los elementos del DOM que tienen un atributo <b>class="btn-primary"</b> .
Múltiple	h1#titulo, h2.subtitle	Selecciona los elementos <b>h1</b> que tienen el atributo <b>id="titulo"</b> y los elementos <b>h2</b> que tienen el atributo <b>class="subtitle"</b>
Descendiente directo	div > button	Selecciona las etiquetas <b>button</b> que están dentro (al primer nivel) de las etiquetas <b>div</b> .
Descendiente	div button	Selecciona las etiquetas <b>button</b> que están dentro (a cualquier nivel) de las etiquetas <b>div</b> .
Atributo	[alt]	Selecciona las etiquetas del DOM que tienen un atributo <b>alt</b> (sin importar su valor).

Nombre	Selector	Descripción
Atributo con valor	[data-precio='23']	Selecciona las etiquetas del DOM que tienen un atributo <b>data-precio="23"</b> .
Atributo que empieza por	[href^='https']	Selecciona las etiquetas del DOM que tienen un atributo <b>href</b> cuyo valor empieza por <b>https</b> ( <b>href="https://www.google.com"</b> ).
Atributo que termina por	[src\$='.webm']	Selecciona las etiquetas del DOM que tienen un atributo <b>src</b> cuyo valor termina en <b>.webm</b> ( <b>src="imagen.webm"</b> ).
Atributo que contiene	[class*='ctrl']	Selecciona las etiquetas del DOM que tienen un atributo <b>class</b> cuyo valor contiene <b>ctrl</b> ( <b>class="key-ctrl-izq"</b> ).
Primer elemento	ul:first	Selecciona la primera etiqueta <b>ul</b> que aparece en el DOM.
Último elemento	a:last	Selecciona el último enlace <b>a</b> de el DOM.
Tercer elemento	li:eq(2)	Selecciona el tercer elemento <b>li</b> de el DOM.
Elementos pares/impares	tr:even, tr:odd	Selecciona las filas pares/impares.
Primer elemento hijo	ul li:first-child	Selecciona los primeros elementos <b>li</b> que son descendientes de los elementos <b>ul</b> .
Último elemento hijo	tbody tr:last-child	Selecciona los últimos elementos <b>tr</b> que son descendientes de los elementos <b>tbody</b>
Segundo elemento hijo	tr td:nth-child(2)	Selecciona los segundos elementos <b>td</b> que son descendientes de los elementos <b>tr</b>
Elemento con el foco	:focus	Selecciona el elemento que tiene el foco (atributo <b>focus</b> ) actualmente.

Nombre	Selector	Descripción
Elementos deshabilitados	:disabled	Selecciona los elementos que están deshabilitados (atributo <b>disabled</b> ).
Elementos marcados	:checked	Selecciona los elementos que están marcados (atributo <b>checked</b> ).
Elementos seleccionados	:selected	Selecciona los elementos que están seleccionados (atributo <b>selected</b> ).
Elementos ocultos	:hidden	Selecciona los elementos que están ocultos (atributo <b>hidden</b> ).

# Capítulo 9. Instalación

Para instalar Cypress necesitamos tener un proyecto creado en el cual vamos a utilizar esta herramienta para testearlo.

Dentro de la carpeta raíz del proyecto vamos a lanzar, desde un terminal, el siguiente comando para instalar Cypress:

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

Usamos el flag `--save-dev` porque Cypress no es una dependencia de producción. Solo se va a utilizar en fases de pruebas y desarrollo.

## 9.1. Lab: Instalación

En este laboratorio vamos a ver como crear un proyecto básico para instalar Cypress en el y generar la carpeta donde crearemos los tests.

Empezamos por crear una carpeta para el proyecto e inicializaremos dentro de esta el proyecto de NPM inicial con los comandos siguientes:

```
$ mkdir cypress-instalacion-lab  
$ cd cypress-instalacion-lab  
$ npm init -y
```

Una vez inicializado el proyecto de NPM vamos a instalar la dependencia de Cypress.

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

Al terminar la instalación de la dependencia, añadimos un script de NPM dentro del archivo **package.json** para poder abrir el test runner de Cypress y poder ejecutar los tests desde ahí.

/cypress-instalacion-lab/package.json

```
{  
  "name": "cypress-instalacion-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "cy:open": "cypress open"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.8.0"  
  }  
}
```

Ahora que tenemos el script, podemos lanzarlo para comprobar que se abre el test runner desde donde podremos ir ejecutando nuestros tests.

```
$ npm run cy:open
```

Al lanzar el comando anterior, nos genera en el proyecto una carpeta de **cypress** con todos esos tests de ejemplo y algunas cosillas más que ya iremos viendo para que sirven durante el resto del curso.

También se abre una ventana del navegador con la interfaz de Cypress y un listado con los tests de ejemplo.

[Tests](#)[Runs](#)[Settings](#)

Chrome 88 ▾

 Search...

## ▼ INTEGRATION TESTS

[COLLAPSE ALL](#) | [EXPAND ALL](#)[▶ Run 19 integration specs](#)▼  examples

- [!\[\]\(22ddba305b5a4b498db610ebd1cfba41\_img.jpg\) actions.spec.js](#)
- [!\[\]\(e5208740e1fc4ac43de5f0db5ee866f8\_img.jpg\) aliasing.spec.js](#)
- [!\[\]\(703d1daeb388758e502a95d553a44805\_img.jpg\) assertions.spec.js](#)
- [!\[\]\(f4941b2896a663640626439465129624\_img.jpg\) connectors.spec.js](#)
- [!\[\]\(6df753f19f70692ececbfd7b5d19ddba\_img.jpg\) cookies.spec.js](#)
- [!\[\]\(f0fe0adab59bf928b910fd7e5fa2a088\_img.jpg\) cypress\\_api.spec.js](#)
- [!\[\]\(9d50e39e20915a2c199afb51d5c29d9f\_img.jpg\) files.spec.js](#)
- [!\[\]\(7c9aae632293c8da6839ab907b71bc10\_img.jpg\) local\\_storage.spec.js](#)
- [!\[\]\(b91ac9c072ba5d2738980fa7aebba6d9\_img.jpg\) location.spec.js](#)
- [!\[\]\(0605a52ab32c0befc10fe77610f15d97\_img.jpg\) misc.spec.js](#)
- [!\[\]\(33518d54c9348c08b3e7e4ac80f7d2c8\_img.jpg\) navigation.spec.js](#)
- [!\[\]\(b36312b82b8464bce155e00a5dff004b\_img.jpg\) network\\_requests.spec.js](#)
- [!\[\]\(50aeec16de27d61c5f5fbd9fa499c902\_img.jpg\) querying.spec.js](#)
- [!\[\]\(0eb42a7575ed0eaca6f80b170f415fef\_img.jpg\) spies\\_stubs\\_clocks.spec.js](#)
- [!\[\]\(59f4bf4e0f45091f303b6c3676cc0454\_img.jpg\) traversal.spec.js](#)
- [!\[\]\(07493c95abad8333a1df4886927c6c34\_img.jpg\) utilities.spec.js](#)

[Open in IDE](#)

Version 6.4.0

[Changelog](#)

# Capítulo 10. Ejecución de Tests

Para ejecutar los test de Cypress, nos encontramos con dos formas de hacerlo:

- Desde la interfaz de Cypress
- Desde la terminal

## 10.1. Desde la interfaz del test runner

Podemos ejecutar los tests de nuestra aplicación desde el test runner de Cypress, que es una ventana del navegador donde encontraremos todos los tests del proyecto y podremos ver como se van ejecutando.

```
$ cypress open
```

## 10.2. Desde el terminal

Normalmente la opción anterior se utilizará mientras estamos creando nuestros tests, los estamos probando...

Pero cuando ya los tenemos listos, la idea es que nos ahorremos el trabajo de tener que lanzarlos nosotros mismos dejando que su ejecución se realice automáticamente, por ejemplo utilizando alguna herramienta de CD/CI como Jenkins, Circle CI o las Github Actions.

De esta forma, cuando el equipo de desarrollo realice cualquier cambio sobre el código que se encuentra en un repositorio, estas herramientas pueden detectar este evento y ejecutar una serie de scripts automáticamente, donde uno de estos scripts puede ser la ejecución de los tests para comprobar que los cambios no han roto nuestra aplicación.

Entonces, el comando a lanzar en estos casos es el siguiente:

```
$ cypress run
```

Una vez lanzado este comando vamos a ver por el terminal una serie de logs que nos van indicando como va la ejecución de los tests.

## 10.3. Lab: Ejecución de Tests

En este laboratorio vamos a ver como ejecutar los tests de Cypress de las distintas formas posibles.

Vamos a crear un proyecto con unos tests simples que ejecutaremos de las distintas formas que Cypress nos permite hacerlo.

Vamos a empezar creando la carpeta del proyecto e instalando las dependencias necesarias lanzando los siguientes comandos:

```
$ mkdir cypress-ejecucion-de-tests-lab
$ cd cypress-ejecucion-de-tests-lab
$ npm init -y
$ npm install --save-dev cypress
```

Después de lanzar estos comandos deberíamos de tener un archivo **package.json** en nuestro proyecto. Dentro de este archivo vamos a modificar la sección de scripts para añadir los dos scripts que se van a encargar de ejecutar los tests.

*/cypress-ejecucion-de-tests/package.json*

```
{
  "name": "cypress-ejecucion-de-tests-lab",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "cy:open": "cypress open",
    "cy:run": "cypress run"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "cypress": "^6.6.0"
  }
}
```

Ahora vamos a lanzar el primer comando que hemos añadido para que se genere dentro de nuestro proyecto una carpeta de **cypress** donde encontramos una serie de tests de ejemplo.

```
$ npm run cy:open
```

Al abrirse la interfaz de Cypress veremos una carpeta de **examples** con todos los archivos de pruebas y ya podemos ver como ejecutarlos.

Para ejecutar **un único test** solo tenemos que pulsar sobre el archivo que queramos ejecutar, por ejemplo, podemos probar a pulsar sobre **aliasing.spec.js**.

cypress-ejecucion-de-tests-lab

Support Docs Log In

Tests Runs Settings Chrome 88 ▾

Search...

INTEGRATION TESTS COLLAPSE ALL | EXPAND ALL

examples

- actions.spec.js
- aliasing.spec.js**
- assertions.spec.js
- connectors.spec.js
- cookies.spec.js
- cypress\_api.spec.js
- files.spec.js
- local\_storage.spec.js
- location.spec.js
- misc.spec.js

Open in IDE

Version 6.6.0 Changelog

No es seguro | example.cypress.io/\_/#/tests/integration/examples/aliasing.spec.js

Un software automatizado de pruebas está controlando Chrome.

Tests 2 07.15 https://example.cypress.io/commands/aliasing 1000 x 660 (100%)

aliasing

.as()

To alias a route or DOM element for use later, use the `.as()` command.

```
// The following DOM command chain is unwieldy.
// To avoid repeating it, let's use '.as()':
cy.get('.as-table')
  .find('tbody>tr').first()
  .find('td').first()
  .find('button').as('firstBtn')

// To reference the alias we created, we place an
// @ in front of its name
cy.get('@firstBtn').click()

cy.get('firstBtn')
  .should('have.class', 'btn-success')
  .and('contain', 'Changed')

// Alias the route to wait for its response

cy.intercept('GET', '*@comment/*').as('getComment')

// we have code that gets a comment when
// the button is clicked in scripts.js
cy.get('.network-btn').click()

// https://on.cypress.io/wait
cy.wait('@getComment').its('response.statusCode').should('eq', 200)
```

Get Comment

laudantium enim quasi est quidem magnam  
voluptate ipsum eos tempora quo necessitatibus  
dolor quam autem quasi reiciendis et nam sapiente  
accusantium

Column 1	Column 2
Row 1: Cell 1	Row 1: Cell 2
Change	Change
Row 2: Cell 1	Row 2: Cell 2
Change	Change

En lugar de ejecutar un solo test, podemos encontrarnos con algún caso en que queramos ejecutar varios tests que cumplen con algún patrón. En este caso, podemos escribir la parte del nombre de los tests que coincide entre ellos para filtrar, y después podemos pulsar sobre el botón que aparece a la derecha para ejecutar el conjunto de pruebas que cumplen con el filtro.

The screenshot shows the Cypress Test Runner interface. At the top, there are tabs for 'Tests', 'Runs', and 'Settings'. A search bar contains the text 'co'. Below the search bar, a section titled 'INTEGRATION TESTS' shows two files: 'connectors.spec.js' and 'cookies.spec.js'. On the right side of this section, there is a button labeled 'Run 2 integration specs'. The overall theme is dark with light-colored text and icons.

The screenshot shows a browser window with the title 'cypress-ejecucion-de-tests-lab'. The address bar shows 'example.cypress.io/\_/#/tests/\_all'. The page content is about browser cookies. It includes sections for 'cy.clearCookie()' and 'cy.clearCookies()'. Each section provides a brief description and examples of how to use the command. There are also 'Set Cookie' buttons. The browser interface includes standard navigation buttons and a status bar at the bottom.

Y por último, podemos ejecutar todos los tests de una vez pulsando sobre el botón de la derecha sin aplicar ningún filtro.

The screenshot shows the Cypress Test Runner interface. At the top, there's a navigation bar with links for Support, Docs, and Log In. Below that is a toolbar with 'Tests' (selected), 'Runs', 'Settings', and a dropdown for 'Chrome 88'. A search bar is present. The main area shows a tree view under 'INTEGRATION TESTS' with the 'examples' folder expanded, displaying 19 individual test files. To the right of the tree view is a button labeled 'Run 19 integration specs'. At the bottom right of the interface are links for 'Version 6.6.0' and 'Changelog'.

De estas formas podemos ejecutar los tests que hayamos creado desde la interfaz de Cypress.

La otra posibilidad es hacerlo desde nuestro terminal lanzando el comando:

```
$ npm run cy:run
```

Al lanzar este comando se ejecuta nuestros tests sin abrir el navegador como ocurría en las anteriores pruebas, pero si que podemos ir viendo el progreso de la ejecución y más información.

```
=====
(Run Starting)
```

```
| Cypress: 6.6.0
| Browser: Electron 87 (headless)
| Specs: 19 found (examples/actions.spec.js, examples/aliasing.spec.js, examples/assertions
| .spec.js, examples/connectors.spec.js, examples/cookies.spec.js, examples/cypress_
| api.spec.js, examples/files.spec.js, examples/local_storage.spec.js, examples/loca
| tion.spec.js...)
```

```
Running: examples/actions.spec.js
```

```
(1 of 19)
```

#### Actions

- .type() - type into a DOM element (7877ms)
- .focus() - focus on a DOM element (804ms)
- .blur() - blur off a DOM element (2560ms)
- .clear() - clears an input or textarea element (2179ms)

Al finalizar la ejecución de todos los tests nos muestra una tabla con los resultados de estos:

```
=====
(Run Finished)
```

Spec	Tests	Passing	Failing	Pending	Skipped
examples/actions.spec.js	00:36	14	14	-	-
examples/aliasing.spec.js	00:08	2	2	-	-
examples/assertions.spec.js	00:10	9	9	-	-
examples/connectors.spec.js	00:09	8	8	-	-
examples/window.spec.js	00:02	3	3	-	-
All specs passed!	02:54	114	114	-	-

Podemos indicarle un solo archivo a ejecutar pasandole la opción **spec** y la ruta del archivo de test:

```
$ npm run cy:run -- --spec cypress/integration/examples/connectors.spec.js
```

Al lanzar este comando solo debería de ejecutarse el test cuya ruta hemos pasado al comando.

# Capítulo 11. Estructura de carpetas

Al lanzar el comando de ejecución de los tests por primera vez se añade una carpeta **cypress** donde vamos a estar trabajando con Cypress dentro del proyecto. En esta carpeta nos encontramos con las carpetas:

- **fixtures**: archivos json con datos que podemos utilizar en nuestros tests como mockups (datos de mentira).
- **integration**: en esta carpeta es donde vamos a poner nuestros archivos de tests.
- **plugins**: aquí se indica que plugins de Cypress se van a utilizar.
- **support**: aquí podemos crear funciones o comandos que hagan tareas que se pueden reutilizar en distintos tests.
- **downloads**: aquí podemos encontrar los archivos descargados durante la ejecución de los tests.
- **screenshots**: aquí encontramos las capturas de pantalla que se van haciendo durante la ejecución de los tests.
- **videos**: aquí se van a ir guardando los videos grabados de la ejecución de los tests.

# Capítulo 12. Integración con el editor

Cypress se puede integrar con nuestro editor para hacernos más productivos a la hora de escribir nuestros tests. Algunas de las opciones que nos encontramos son:

- Enlazar el test runner con el editor
- Autocompletado o ayuda al escribir los tests
- Utilizar plugins del editor

## 12.1. Enlazar el test runner con el editor

Desde la interfaz de Cypress podemos añadir en los ajustes nuestro editor como herramienta para abrir los archivos de test. Al configurarlo podremos abrir los tests pulsando sobre la opción de **Abrir en el IDE** que aparece junto a ellos.

Cuando un test falla, se muestra justo el archivo y la línea en la que se encuentra el fallo permitiendo también abrirlo directamente en nuestro editor pulsando sobre el nombre del archivo.

## 12.2. Autocompletado y ayuda

Dentro del editor podemos conseguir la ayuda y autocompletado de los comandos e instrucciones de Cypress sin necesidad de instalar ningún plugin o extensión solamente añadiendo una referencia a los tipos de Cypress al principio de nuestros archivos de testing.

Esta referencia la añadimos con la siguiente línea de código:

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

La línea de código anterior se encarga de cargar los tipos de datos que se han definido en el archivo de tipos de Cypress (un archivo que termina con **.d.ts**)

Otra forma de hacer lo mismo, pero evitando tener que copiar esa línea de código en cada test, es añadir esa opción que carga los tipos de Cypress en un archivo de configuración, **jsconfig.json**:

```
{
  "include": [
    "./node_modules/cypress",
    "cypress/**/*.js"
  ]
}
```

## 12.3. Plugins

Algunos **plugins del VS Code** que pueden ayudarnos a trabajar más fácilmente con nuestros tests son:

- **ES6 Mocha Snippets**: snippets para crear los bloques de mocha.
- **Cypress Open**: abre directamente la interfaz de Cypress y ejecuta el test en el que estamos trabajando.

## 12.4. Lab: Integración con el editor

En este laboratorio vamos a ver como preparar todo el entorno de trabajo e integrar Cypress con el editor.

Vamos a empezar por crear el proyecto de NPM donde crearemos un test de ejemplo con Cypress para probar que tenemos todo bien configurado y funcionando.

Lanzamos los siguientes comandos:

```
$ mkdir cypress-integracion-con-el-editor-lab
$ cd cypress-integracion-con-el-editor-lab
$ npm init -y
```

Ahora vamos a instalar la dependencia de Cypress en el proyecto.

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

El siguiente paso es añadir un script de NPM en el archivo **package.json** con el que abriremos la interfaz de Cypress.

*/cypress-integracion-con-el-editor-lab/package.json*

```
{
  "name": "cypress-integracion-con-el-editor-lab",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "cy:open": "cypress open"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "cypress": "^6.6.0"
  }
}
```

Ahora ya podemos lanzar el script que acabamos de añadir para generar la carpeta de **cypress** dentro del proyecto y abrir la interfaz de este.

```
$ npm run cy:open
```

Una vez abierta la interfaz de Cypress, vamos a empezar por indicarle que queremos utilizar VS Code como el editor en el que abrir los tests desde esta herramienta. Entramos en **Settings > File Opener Preference** y seleccionamos **Visual Studio Code**.

The screenshot shows the Cypress Settings interface. At the top, there are tabs for 'Tests', 'Runs', and 'Settings'. The 'Settings' tab is selected. In the main area, under 'File Opener Preference', there is a dropdown menu with options: 'Finder' (radio button), 'Visual Studio Code' (radio button, highlighted with a green border), 'Vim' (radio button), and 'Other:' (input field). Below this, there is a section for 'Experiments'. At the bottom right, there are links for 'Version 6.6.0' and 'Changelog'.

En caso de no aparecer Visual Studio Code como opción, se debe a que no se encuentra en el PATH.

Para añadirlo al PATH, hay que ir a VS Code, entrar en **Ver > Paleta de comandos** y seleccionar el de **Instalar el comando 'code' en PATH**.

The screenshot shows the VS Code Command Palette. A search bar at the top contains the text '>Instalar el comando'. Below it, a list of commands is shown: 'Comando shell: Instalar el comando 'code' en PATH' (recently used) and 'Comando shell: Shell Command: Install 'code' command in PATH'. Below these, another group of commands is listed: 'Comando shell: Desinstalar el comando 'code' de PATH' (other commands) and 'Comando shell: Shell Command: Uninstall 'code' command from PATH'. On the far left, there is a blue circular icon with a white 'i'.

Una vez termina de ejecutar el comando, ya deberíamos de ver la opción de **Visual Studio Code**.

Ahora vamos a crear un archivo de test, **tuw.spec.js**, en **cypress/integration**.

```
describe('The Useless Web', () => {
  it('Visita The Useless Web', () => {
    })
});
```

Podemos ahorrarnos escribir todo ese código con una extensión de VS Code. Vamos a ir a la pestaña de **Extensiones** y vamos a instalar **ES6 Mocha Snippets**.



Una vez instalada, podríamos haber construido esos dos bloques escribiendo **describeAndIt**.

Dentro del bloque **it** podemos escribir **cy**. y veremos que no nos muestra una ayuda o los comandos que podemos utilizar. Vamos a solucionarlo añadiendo en un archivo **jsconfig.json** dos expresiones regulares que se encargan de incluir los tipos de datos de Cypress para que se nos muestre la ayuda según escribimos los comandos de Cypress en los tests.

```
{
  "include": [
    "./node_modules/cypress",
    "cypress/**/*.*"
  ]
}
```

Una vez creado el archivo si probamos a escribir lo mismo que antes, deberíamos de poder ver en el popup las distintas opciones que podemos usar después del comando **cy**.

Ahora vamos a añadir las instrucciones para navegar a la página <https://theuselessweb.com/>, y vamos a buscar un elemento que no exista para hacer que fallen los tests.

```
describe('The Useless Web', () => {
  it('Visita The Useless Web', () => {
    cy.visit('https://theuselessweb.com/');

    cy.get('#no-existe')
      .should('exist');
  })
});
```

Vamos a hacer que se ejecute este test desde la interfaz de Cypress seleccionando el archivo que tiene el mismo nombre que el del test. Una vez que pulsamos sobre el veremos que empieza a ejecutarse y falla. Al haber enlazado Cypress con el VS Code, ahora podemos pulsar sobre el archivo en el que ha ocurrido el fallo y nos lo abrirá en el editor directamente.

The screenshot shows the Cypress Test Runner interface. At the top, there's a summary bar with icons for Tests (green checkmark), Fails (red X), Pending (grey circle), Duration (29.34), and a refresh button. Below the summary is a tree view of tests. A red box highlights the 'Visita The Useless Web' test under 'The Useless Web'. This test has a red X icon and a warning icon. The 'TEST BODY' section shows three steps: 'visit https://theuselessweb.com/' (XHR POST), 'GET 200 /getconfig/sodar?sv=200&tid=gda&tv=r20210309...', and 'get #no-existe'. Step 3 contains an assertion: '- assert expected #no-existe to exist in the DOM'. A pink box highlights this assertion. Below the test body, an 'AssertionError' section shows a timeout message: 'Timed out retrying after 4000ms: Expected to find element: #no-existe, but never found it.' A red box highlights the error message. The code for the failing assertion is shown in a code editor window at the bottom, with line 6 highlighted. The code is:

```
1 | 
2 |   visit https://theuselessweb.com/
3 | 
4 | 
5 |   cy.get('#no-existe')
> 6 |     .should('exist');
|   ^
7 |   });
8 | });
```

At the bottom of the interface are buttons for 'View stack trace' and 'Print to console'.

También podemos abrir los tests desde Cypress aunque no hayan fallado previamente. Solo hay que

pulsar sobre el botón de Open in IDE.

The screenshot shows the Cypress UI interface. At the top, there's a dark header bar with the project name "cypress-integracion-con-el-editor-lab". On the right side of the header are links for "Support", "Docs", and "Log In". Below the header is a navigation bar with three tabs: "Tests" (selected), "Runs", and "Settings". To the right of the navigation bar are two buttons: "Stop" (red) and "Running Chrome 89" (green). Below the navigation bar is a search bar with the placeholder "Search...". Underneath the search bar is a section titled "INTEGRATION TESTS" with a dropdown arrow. Inside this section, there's a list item "tuw.spec.js" followed by a status indicator "Running 1 spec" and a button labeled "Open in IDE" which is highlighted with a black border. At the bottom of the page is a dark footer bar with the text "Version 6.6.0" and "Changelog".

Y con esto ya tenemos configurado nuestro entorno para poder seguir trabajando con nuestros tests de una forma más cómoda.

# Capítulo 13. Buscar elementos

Para poder testear la aplicación y poder simular todas las acciones que realizaría un usuario mientras está en el navegador, es necesario poder acceder a los distintos elementos de las páginas.

Puede que a veces sea complicado obtener algún elemento, pero siempre existe alguna forma para conseguirlo.

A continuación veremos los distintos métodos que se pueden usar para obtener los elementos de la aplicación web.

Para acceder a los elementos de la aplicación web usaremos la variable **cy** que nos proporciona Cypress.

Desde la variable **cy** tenemos acceso a bastantes métodos, pero ahora vamos a centrarnos en aquellos que nos permiten buscar los distintos elementos web con los que queremos interactuar:

- **get**
- **find**
- **contains**

## 13.1. Get

Cypress nos proporciona una serie de métodos para realizar las búsquedas de los elementos con los que queremos interactuar, y el método que más vamos a usar en nuestros tests es el **get**.

El método **get** recibe como parámetro el selector con el que apuntamos a los elementos que queremos.

Este método nos puede devolver uno o varios elementos.

```
cy.get('#mensaje-error');
```

## 13.2. Find

Otro de los métodos que podemos utilizar para acceder a aquellos elementos con los que queremos interactuar es el **find**.

Este método también recibe como parámetro un selector de jQuery y puede retornar un elemento o varios de ellos.

Pero, ¿no es igual que el **get**? **No**, la diferencia es que este método **no podemos utilizarlo desde el cy** sino que lo tenemos que encadenar sobre otra instrucción que nos devuelva algún elemento de la web, como el método **get**.

```
cy.get('#lista-enlaces')
  .find('.active')
```

## 13.3. Contains

Otro método de los que podemos utilizar para la búsqueda de elementos web es el **contains**, al cual le podemos pasar como parámetro el texto entero o parcial que contiene el elemento buscado para poder acceder a él.

Podemos pasárle también un selector para ajustar más la búsqueda del elemento web, además de que podemos utilizarlo tanto desde el propio objeto **cy** como desde la llamada al método **get** visto anteriormente.

Este método nos devolverá el primer elemento que encuentre con los datos que se le han dado.

```
cy.contains('a', 'Google');
```

## 13.4. Lab: Buscar elementos

En este laboratorio vamos a ver como utilizar los métodos de Cypress para localizar elementos web de la aplicación que estamos testeando.

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-buscar-elementos-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-buscar-elementos-lab
$ cd cypress-buscar-elementos-lab
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests
- http-server: servidor de desarrollo local

```
$ npm install --save-dev cypress http-server
```

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

```
{  
  "name": "cypress-buscar-elementos-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "http-server",  
    "cy:open": "cypress open"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.8.0",  
    "http-server": "^0.12.3"  
  }  
}
```

El siguiente paso es crear un archivo **index.html** donde vamos a poner unas cuantas etiquetas para ver como buscarlas.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Buscar elementos web</title>
  <style>
    body {
      height: 100vh;
      width: 100vw;
      background-color: #242424;
      color: white;
    }

    .center {
      height: 100%;
      display: flex;
      flex-direction: column;
      justify-content: center;
      align-items: center;
    }
  </style>
</head>
<body>
  <div class="center">
    <h2 id="titulo">Listados</h2>
    <div>
      <ul id="listaItems">
        <li class="li1">Item 1</li>
        <li class="li2">Item 2</li>
        <li class="li3">Item 3</li>
      </ul>
      <ul id="listaCosas">
        <li class="li1">Cosa 1</li>
        <li class="li2">Cosa 2</li>
        <li class="li3">Cosa 3</li>
      </ul>
    </div>
  </div>
</body>
</html>
```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Además, en otra terminal, vamos a lanzar el otro script para poder acceder con Cypress a la web.

```
$ npm start
```

Como nos indica en la terminal, podemos acceder a la web en <http://localhost:8080/>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **busquedas.spec.js** dentro de la carpeta **integration**.

Añadimos nuestro caso de prueba y visitamos la página donde se encuentra lo que queremos testear. Como vamos a tener 3 pruebas, y las 3 van a hacerse sobre la misma url, vamos a poner la navegación a la aplicación en el **beforeEach**.

/cypress-buscar-elementos-lab/cypress/integration/busquedas.spec.js

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

describe('Búsquedas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

})
```

La primera búsqueda consiste en encontrar el título y comprobar que existe y tiene el texto correcto. Para ello lo buscaremos por su identificador y usaremos varias aserciones por lo que las vamos a encadenar con la función **and**.

/cypress-buscar-elementos-lab/cypress/integration/busquedas.spec.js

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

describe('Búsquedas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('deberías encontrar el título Listados', () => {
    cy.get('#titulo')
      .should('exist')
      .and('have.text', 'Listados')
  })

})
```

Ahora vamos a buscar el elemento **Item 2** que se encuentra en la lista de cosas. Para este caso vamos a utilizar primero el **get** para acceder a la lista de cosas, y después sobre la lista utilizaremos el **find** para buscar el que tiene la clase **li2**.

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

describe('Búsquedas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('deberías encontrar el título Listados', () => {
    cy.get('#titulo')
      .should('exist')
      .and('have.text', 'Listados')
  })

  it('deberías encontrar el elemento 2 del segundo listado', () => {
    cy.get('#listaCosas')
      .find('.li2')
      .should('have.text', 'Cosa 2')
  })
})
```

La última búsqueda consiste en encontrar el elemento **Item 3**, pero esta vez utilizando el método **contains**. Podemos hacer esta búsqueda de bastantes formas. Vamos a buscar primero con el **get** la lista de items, y después buscaremos el elemento **li** que contiene el texto buscado.

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

describe('Búsquedas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('deberías encontrar el título Listados', () => {
    cy.get('#titulo')
      .should('exist')
      .and('have.text', 'Listados')
  })

  it('deberías encontrar el elemento 2 del segundo listado', () => {
    cy.get('#listaCosas')
      .find('.li2')
      .should('have.text', 'Cosa 2')
  })

  it('deberías encontrar el elemento Item 3 (sin get y sin find)', () => {
    cy.get('#listaItems')
      .contains('li', 'Item 3')
      .should('have.text', 'Item 3');
  })
})
```

Otra forma de haberlo hecho hubiese sido buscando directamente el elemento que contiene el texto **Item 3** con el **contains**.

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

describe('Búsquedas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('deberías encontrar el título Listados', () => {
    cy.get('#titulo')
      .should('exist')
      .and('have.text', 'Listados')
  })

  it('deberías encontrar el elemento 2 del segundo listado', () => {
    cy.get('#listaCosas')
      .find('.li2')
      .should('have.text', 'Cosa 2')
  })

  it('deberías encontrar el elemento Item 3 (sin get y sin find)', () => {
    // Busca el Item 3 dentro del #listaItems
    cy.get('#listaItems')
      .contains('li', 'Item 3')
      .should('have.text', 'Item 3');

    // Busca el Item 3
    cy.contains('Item 3')
      .should('have.text', 'Item 3');
  })
})
```

Y con esto ya tendríamos una idea de como utilizar estos 3 métodos de búsqueda de elementos que iremos usando durante el resto del curso.

# Capítulo 14. Navegar por el DOM

Cuando queremos testear listas, tablas o elementos web que contienen varias etiquetas del mismo tipo repetidas en su interior, nos encontramos con que todos estos elementos tienen las mismas clases o atributos y puede ser un poco complicado acceder directamente a ellos utilizando selectores como el de los identificadores (a cada elemento de una lista no se le suele poner un identificador único).

En estos casos tendremos que empezar a utilizar comandos que se encargan de filtrar, recorrer listas de elementos, acceder a los elementos hijos... que nos facilitarán nuestra tarea.

Algunos de estos elementos son:

- **children**: devuelve los elementos hijos del elemento sobre el que se aplica este método.
- **parent**: devuelve el elemento padre del elemento sobre el que se aplica este método.
- **first**: devuelve el primer elemento de una lista de elementos.
- **last**: devuelve el último elemento de una lista de elementos.
- **eq**: devuelve un elemento dada su posición dentro de una lista de elementos.
- **each**: itera sobre un array de elementos y nos los va pasando de uno en uno (como elementos de jQuery) en la función de callback que recibe como parámetro.
- **filter**: permite filtrar una lista de elementos devolviéndonos aquellos que cumplen con el selector que se le pasa como parámetro.
- ...

## 14.1. Lab: Navegar por el DOM

En este laboratorio vamos a ver como utilizar algunos de los métodos que nos permiten recorrer el DOM y buscar elementos de la aplicación para realizar una serie de comprobaciones sobre una tabla de datos. A continuación nos encontramos con los objetivos:

- Navegar a [http://www.w3schools.com/html/html\\_tables.asp](http://www.w3schools.com/html/html_tables.asp)
- Encontrar la tabla de la página que tiene la cabecera "HTML Table Example"
- Comprueba que la tabla existe
- Comprueba que tiene el número de filas correcto
- Comprueba que la última fila tiene el número de celdas correcto
- Comprueba que después de la quinta fila, hay dos filas más
- Comprueba que todas las celdas tienen contenido

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-navegar-por-el-dom-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-navegar-por-el-dom-lab  
$ cd cypress-navegar-por-el-dom-lab  
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests

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

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

*/cypress-navegar-por-el-dom-lab/package.json*

```
{  
  "name": "cypress-navegar-por-el-dom-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "cy:open": "cypress open"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.8.0",  
    "http-server": "^0.12.3"  
  }  
}
```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **tabla.spec.js** dentro de la carpeta **integration**.

/cypress-navegar-por-el-dom-lab/cypress/integration/tabla.spec.js

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

describe('Tabla', () => {
})
```

Vamos a crearnos el bloque it y empezaremos yendo a la página que nos indican.

/cypress-navegar-por-el-dom-lab/cypress/integration/tabla.spec.js

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

describe('Tabla', () => {
  it('debería tener los datos correctos la tabla', () => {
    cy.visit('https://www.w3schools.com/html/html_tables.asp')
  })
})
```

Ahora vamos a buscar la tabla para comprobar que existe.

/cypress-navegar-por-el-dom-lab/cypress/integration/tabla.spec.js

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

describe('Tabla', () => {
  it('debería tener los datos correctos la tabla', () => {
    cy.visit('https://www.w3schools.com/html/html_tables.asp')

    cy.get('#customers')
      .should('exist');
  })
})
```

No hemos buscado por la etiqueta **table** porque hay varias tablas, y con el identificador nos aseguramos de acceder a la que queremos.

Ahora vamos a buscar las filas que hay dentro de la tabla para comprobar que tienen el número correcto de filas. Primero pedimos la tabla, y después usamos el método **find** para buscar las filas que hay dentro de la tabla.

/cypress-navegar-por-el-dom-lab/cypress/integration/tabla.spec.js

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

describe('Tabla', () => {
  it('debería tener los datos correctos la tabla', () => {
    cy.visit('https://www.w3schools.com/html/html_tables.asp')

    cy.get('#customers')
      .should('exist');

    cy.get('#customers')
      .find('tr')
      .should('have.length', 7);
  })
})
```

La siguiente comprobación nos dice que miremos que la última fila tiene 3 celdas. Para este caso, vamos a buscar todas las filas, y vamos a utilizar el método **last** para acceder a la última. Dentro de este último **tr** vamos a buscar los elementos **td** que hay dentro usando el método **find** para ello.

/cypress-navegar-por-el-dom-lab/cypress/integration/tabla.spec.js

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

describe('Tabla', () => {
  it('debería tener los datos correctos la tabla', () => {
    cy.visit('https://www.w3schools.com/html/html_tables.asp')

    cy.get('#customers')
      .should('exist');

    cy.get('#customers')
      .find('tr')
      .should('have.length', 7);

    cy.get('#customers tr')
      .last()
      .find('td')
      .should('have.length', 3);
  })
})
```

La siguiente comprobación se complica un poco, porque esta vez vamos a necesitar el método **each** para recorrer todas las filas de la tabla y así ir incrementando una variable por cada fila cuya posición en el array de filas sea superior a 4.

Después de terminar de recorrer el array de filas, vamos a añadir la aserción utilizando el método **then** y dentro el **expect** de Chai.

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

describe('Tabla', () => {
  it('debería tener los datos correctos la tabla', () => {
    cy.visit('https://www.w3schools.com/html/html_tables.asp')

    cy.get('#customers')
      .should('exist');

    cy.get('#customers')
      .find('tr')
      .should('have.length', 7);

    cy.get('#customers tr')
      .last()
      .find('td')
      .should('have.length', 3);

    let numFilas = 0;
    cy.get('#customers tr')
      .each(($tr, index) => {
        if (index > 4) {
          numFilas += 1;
        }
      })
      .then(() => {
        expect(numFilas).to.be.equal(2);
      })
  })
})
```

Por último, la aserción que nos queda nos dice que comprobemos que todas las celdas tienen texto, por lo que vamos a volver a utilizar el método **each** pero esta vez para acceder a los elementos de jQuery, obtener el texto que contienen, y comprobar que no está vacío.

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

describe('Tabla', () => {
  it('debería tener los datos correctos la tabla', () => {
    cy.visit('https://www.w3schools.com/html/html_tables.asp')

    cy.get('#customers')
      .should('exist');

    cy.get('#customers')
      .find('tr')
      .should('have.length', 7);

    cy.get('#customers tr')
      .last()
      .find('td')
      .should('have.length', 3);

    let numFilas = 0;
    cy.get('#customers tr')
      .each(($tr, index) => {
        if (index > 4) {
          numFilas += 1;
        }
      })
      .then(() => {
        expect(numFilas).to.be.equal(2);
      })

    cy.get('#customers td')
      .each($td => {
        const textoCelda = $td.text();
        expect(textoCelda).not.to.be.empty;
      })
  })
})
```

Con esto ya tenemos todas las aserciones anteriores completadas, podemos probar a cambiar las condiciones que se comprueban para ver que fallan los tests y no tenemos falsos positivos.

Ahora vamos a mejorar un poco las aserciones anteriores y vamos a ver como podríamos encadenar unas cuantas de ellas para evitar tener que buscar varias veces los mismos elementos.

Vamos a crear otro bloque it, en el que encadenaremos las tres primeras aserciones.

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

describe('Tabla', () => {
  it('debería tener los datos correctos la tabla', () => {
    cy.visit('https://www.w3schools.com/html/html_tables.asp')

    cy.get('#customers')
      .should('exist');

    cy.get('#customers')
      .find('tr')
      .should('have.length', 7);

    cy.get('#customers tr')
      .last()
      .find('td')
      .should('have.length', 3);

    let numFilas = 0;
    cy.get('#customers tr')
      .each(($tr, index) => {
        if (index > 4) {
          numFilas += 1;
        }
      })
      .then(() => {
        expect(numFilas).to.be.equal(2);
      })

    cy.get('#customers td')
      .each($td => {
        const textoCelda = $td.text();
        expect(textoCelda).not.to.be.empty;
      })
  })

  it('debería poder unir las tres primeras comprobaciones en una sola', () => {
    cy.visit('https://www.w3schools.com/html/html_tables.asp')

    cy.get('#customers')
      .should('exist')
      .find('tr')
      .should('have.length', 7)
      .last()
      .find('td')
      .should('have.length', 3);
  })
})
```

Hemos podido encadenarlas porque los comandos de Cypress van retornando el objeto buscado o sobre el que se realizan acciones hacia los comandos siguientes.

También podemos encadenar las dos últimas aserciones, así que vamos a crear otro bloque it para ver como quedarían.

/cypress-navegar-por-el-dom-lab/cypress/integration/tabla.spec.js

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

describe('Tabla', () => {
  it('debería tener los datos correctos la tabla', () => {
    cy.visit('https://www.w3schools.com/html/html_tables.asp')

    cy.get('#customers')
      .should('exist');

    cy.get('#customers')
      .find('tr')
      .should('have.length', 7);

    cy.get('#customers tr')
      .last()
      .find('td')
      .should('have.length', 3);

    let numFilas = 0;
    cy.get('#customers tr')
      .each(($tr, index) => {
        if (index > 4) {
          numFilas += 1;
        }
      })
      .then(() => {
        expect(numFilas).to.be.equal(2);
      })

    cy.get('#customers td')
      .each($td => {
        const textoCelda = $td.text();
        expect(textoCelda).not.to.be.empty;
      })
  })

  it('debería poder unir las tres primeras comprobaciones en una sola', () => {
    cy.visit('https://www.w3schools.com/html/html_tables.asp')

    cy.get('#customers')
      .should('exist')
      .find('tr')
      .should('have.length', 7)
```

```
.last()
.find('td')
.should('have.length', 3);
})

it('debería poder unir las dos últimas comprobaciones en una sola', () => {
  cy.visit('https://www.w3schools.com/html/html_tables.asp')

  let numFilas = 0;
  cy.get('#customers tr')
    .each(($tr, index) => {
      if (index > 4) {
        numFilas += 1;
      }
    })
    .then(() => {
      expect(numFilas).to.be.equal(2);
    })
    .find('td')
    .each($td => {
      const textoCelda = $td.text();
      expect(textoCelda).not.to.be.empty;
    })
  })
})
```

# Capítulo 15. Retry ability

Cypress viene con la funcionalidad de que el va a reintentar hacer ciertas acciones por un máximo de 4 segundos (es el valor por defecto). Si en estos 4 segundos no ha conseguido realizar la acción que necesitaba entonces el test va a fallar.

Esta funcionalidad está muy bien porque muchas veces en las aplicaciones algunos elementos con los que necesitamos interactuar no se muestran en el mismo momento y tardan un poco en aparecer por distintos motivos como que estos elementos pinten datos que vienen de una petición a una API, y esta API tarde algo más de lo necesario en enviárnoslos.

Entonces, cuando buscamos un elemento, si este se encuentra pues se sigue con la ejecución del test, pero si no es así, intenta otra vez a buscar el elemento, y así varias veces hasta que se acaba el tiempo de reinicio que es cuando falla el test.

Este tiempo de reinicio se puede cambiar pasándole, a los comandos, la propiedad **timeout** con el nuevo valor en milisegundos dentro del objeto de opciones.

Otra forma de cambiarlo, y que afectaría a todos los tests es definirlo en el archivo de **cypress.json** con la propiedad **defaultCommandTimeout** y asignándole el nuevo tiempo en milisegundos como antes.

Esta funcionalidad solo se aplica a comandos de Cypress que se encargan de buscar elementos, es decir, **get**, **contains**, **find**... Esto se debe a que si reintenta un comando de interacción como un **type** o un **click**, puede afectar al comportamiento de la aplicación y tener efectos colaterales sobre el test.

## 15.1. Lab: Retry ability

En este laboratorio vamos a ver como Cypress es capaz de esperar un tiempo hasta que puede interactuar con el elemento buscado, además de como modificar este tiempo de espera.

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-retry-ability-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-retry-ability-lab  
$ cd cypress-retry-ability-lab  
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests

- http-server: servidor de desarrollo local

```
$ npm install --save-dev cypress http-server
```

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

*/cypress-retry-ability-lab/package.json*

```
{
  "name": "cypress-retry-ability-lab",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "http-server",
    "cy:open": "cypress open"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "cypress": "^6.6.0",
    "http-server": "^0.12.3"
  }
}
```

El siguiente paso es crear un archivo **index.html** en el que vamos a hacer que se generen dos botones pasados 3.5 segundos y 5.5 segundos respectivamente.

*/cypress-retry-ability-lab/index.html*

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Retry ability</title>
  <style>
    body {
      height: 100vh;
      width: 100vw;
      background-color: #242424;
    }

    .center {
      height: 100%;
      display: flex;
      justify-content: center;
    }
  </style>
</head>
<body>
  <div class="center">
    <button>3.5 seconds</button>
    <br>
    <button>5.5 seconds</button>
  </div>
</body>
</html>
```

```

        align-items: center;
    }

    button {
        background-color: black;
        width: 100%;
        color: white;
        border: 1px solid white;
        border-radius: 5px;
        padding: 10px;
        margin: 20px 0;
        display: block;
    }

    button:hover {
        border: 1px solid #999999;
        color: #999999;
    }

```

</style>

</head>

<body>

```

    <div class="center">
        <div id="caja-retry-ability"></div>
    </div>

```

```

<script>
    const btn3500 = document.createElement('button');
    btn3500.id = "btn-lazy-3500";
    btn3500.innerText = 'Soy un botón perezoso';

    setTimeout(() => {
        document.getElementById('caja-retry-ability').appendChild(btn3500);
    }, 3500);

    const btn5500 = document.createElement('button');
    btn5500.id = "btn-lazy-5500";
    btn5500.innerText = 'Soy un botón más perezoso todavía';

    setTimeout(() => {
        document.getElementById('caja-retry-ability').appendChild(btn5500);
    }, 5500);

```

</script>

</body>

</html>

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Además, en otra terminal, vamos a lanzar el otro script para poder acceder con Cypress a la web.

```
$ npm start
```

Como nos indica en la terminal, podemos acceder a la web en <http://localhost:8080/>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **retry-ability.spec.js** dentro de la carpeta **integration**.

Para este laboratorio vamos a empezar por comprobar que el primer botón es visible antes de que termine el tiempo de reintento de búsqueda que tiene definido Cypress por defecto.

Añadimos el bloque describe con un beforeEach para navegar a la página web en cada test y vamos a añadir el primer bloque de caso de prueba.

```
/cypress-retry-ability-lab/cypress/integration/retry-ability.spec.js
```

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

describe('Retry ability', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080');
  })

  it('debería encontrar el botón aunque tarde 4seg en salir', () => {
    })
})
```

Dentro del test vamos a buscar el botón que tarda 3.5 segundos en aparecer para comprobar que existe y es visible.

```
/cypress-retry-ability-lab/cypress/integration/retry-ability.spec.js
```

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

describe('Retry ability', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080');
  })

  it('debería encontrar el botón aunque tarde 4seg en salir', () => {
    cy.get('#btn-lazy-3500')
      .should('exist')
      .and('have.text', 'Soy un botón perezoso')
      .and('be.visible');
  })
})
```

El siguiente test que vamos a añadir es para comprobar que con el tiempo por defecto que usa Cypress para reintentar buscar el elemento no consigue encontrar el botón que tarda 5.5 segundos en crearse.

Creamos el nuevo bloque it y en el vamos a indicar que el botón de 5.5 segundos no existe. Para esperar a que pase el tiempo antes de comprobar la condición usaremos la función de **wait** con un tiempo que esté entre 4 segundos y los 5.5 segundos que tarda en aparecer el botón.

/cypress-retry-ability-lab/cypress/integration/retry-ability.spec.js

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

describe('Retry ability', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080');
  })

  it('debería encontrar el botón aunque tarde 4seg en salir', () => {
    cy.get('#btn-lazy-3500')
      .should('exist')
      .and('have.text', 'Soy un botón perezoso')
      .and('be.visible');
  })

  it('no debería encontrar el botón si tarda más de 4seg en salir', () => {
    cy.wait(4500);

    cy.get('#btn-lazy-5500')
      .should('not.exist');
  })
})
```

Para finalizar, vamos a añadir un último test en el que vamos a cambiar el tiempo de reintento para que sea capaz de encontrar el segundo botón. Para modificar el tiempo de reintento solo tenemos que pasarle un objeto de opciones al get y utilizar la propiedad **timeout** con el nuevo tiempo en milisegundos.

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

describe('Retry ability', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080');
  })

  it('debería encontrar el botón aunque tarde 4seg en salir', () => {
    cy.get('#btn-lazy-3500')
      .should('exist')
      .and('have.text', 'Soy un botón perezoso')
      .and('be.visible');
  })

  it('no debería encontrar el botón si tarda más de 4seg en salir', () => {
    cy.wait(4500);

    cy.get('#btn-lazy-5500')
      .should('not.exist');
  })

  it('debería encontrar el botón si tarda más de 4seg en salir (cambiando el timeout)', () => {
    cy.get('#btn-lazy-5500', { timeout: 6000 })
      .should('be.visible');
  })
})
```

Ya tendríamos nuestros tests, y deberían de pasarse correctamente.

# Capítulo 16. Interacciones con elementos

Ahora vamos a ver algunos de los comandos que nos permiten interactuar con los diferentes elementos de la web como pueden ser las cajas de texto, los botones, los checkboxes, los desplegables...

## 16.1. Click

Cuando queremos pulsar sobre algún elemento podemos utilizar la función **click** de Cypress. Esta función no se puede llamar directamente desde **cy**, sino que primero tenemos que indicarle sobre qué elemento hay que pulsar.

```
cy.get('#miBtn')
  .click();
```

Cuando queramos lanzar un doble click usaremos en su lugar la función **dblclick**.

```
cy.get('#miBtn')
  .dblclick();
```

## 16.2. Lab: Click

En este laboratorio vamos a probar que un panel de teclas funciona correctamente. Probaremos los siguientes casos:

- Si introducimos el código correcto (6710) nos muestra en el display el mensaje "CODE OK".
- Si pulsamos números y pulsamos el botón de "CLD" los borra.
- Si pulsamos números y pulsamos el botón de "DEL", elimina el último número introducido.
- No deja introducir un código de más de 4 números.

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-interacciones-con-elementos-click-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-interacciones-con-elementos-click-lab
$ cd cypress-interacciones-con-elementos-click-lab
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests
- http-server: servidor de desarrollo local

```
$ npm install --save-dev cypress http-server
```

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

*/cypress-interacciones-con-elementos-click-lab/package.json*

```
{
  "name": "cypress-interacciones-con-elementos-click-lab",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "http-server",
    "cy:open": "cypress open"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "cypress": "^6.6.0",
    "http-server": "^0.12.3"
  }
}
```

El siguiente paso es crear un archivo **index.html** donde vamos a poner el panel de teclas para introducir el código correcto.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Click</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <div class="center">
    <div id="panel-codigo-secreto">
      <div id="pantalla-codigo"></div>
      <div id="teclado">
        <div class="fila-teclas">
          <div class="tecla tecla-num" id="p1">1</div>
          <div class="tecla tecla-num" id="p2">2</div>
          <div class="tecla tecla-num" id="p3">3</div>
        </div>
        <div class="fila-teclas">
          <div class="tecla tecla-num" id="p4">4</div>
          <div class="tecla tecla-num" id="p5">5</div>
          <div class="tecla tecla-num" id="p6">6</div>
        </div>
        <div class="fila-teclas">
          <div class="tecla tecla-num" id="p7">7</div>
          <div class="tecla tecla-num" id="p8">8</div>
          <div class="tecla tecla-num" id="p9">9</div>
        </div>
        <div class="fila-teclas">
          <div class="tecla tecla-clear" id="pclear">CLD</div>
          <div class="tecla tecla-num" id="p0">0</div>
          <div class="tecla tecla-del" id="pdel">DEL</div>
        </div>
      </div>
    </div>
  </div>

  <script src="app.js"></script>
</body>
</html>
```

Ahora vamos a crear un archivo **app.js** al mismo nivel que el **index.html** donde vamos a añadir la parte del javascript que hace que el panel de teclas funcione.

```
const display = document.getElementById('pantalla-codigo');

function checkCodigoSecreto(codigo) {
    if (codigo.length < 4) {
        return [false, false];
    }
    const codigoSecreto = 6710;
    return [codigoSecreto == codigo, codigo.length == 5];
}

function borrarUltimoNum() {
    const codigoActual = display.innerText;
    display.innerText = codigoActual.slice(0, codigoActual.length-1);
}

function borrarCodigo() {
    display.innerText = '';
}

function introducirNum(num) {
    const codigoActual = display.innerText;
    const codigoNuevo = codigoActual + num;
    const [esCorrecto, longitudMaxima] = checkCodigoSecreto(codigoNuevo);
    if (esCorrecto) {
        display.innerText = 'CODE OK';
    } else {
        display.innerText = longitudMaxima ? codigoActual : codigoNuevo;
    }
}

document.getElementById('teclado').addEventListener('click', (event) => {
    const num = event.target.innerText;
    switch (num) {
        case 'DEL':
            borrarUltimoNum();
            break;
        case 'CLD':
            borrarCodigo();
            break;
        default:
            introducirNum(num);
    }
})
```

También tenemos que crear el **style.css** que le da los estilos a nuestro panel de teclas.

```
body {  
  height: 100vh;  
  width: 100vw;  
  background-color: #242424;  
}  
  
.center {  
  height: 100%;  
  display: flex;  
  justify-content: center;  
  align-items: center;  
}  
  
#panel-codigo-secreto {  
  width: 160px;  
  border: 1px solid black;  
  display: flex;  
  flex-direction: column;  

```

Con estos tres archivos ya tenemos nuestra aplicación para la que vamos a crear los tests a continuación.

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Además, en otra terminal, vamos a lanzar el otro script para poder acceder con Cypress a la web.

```
$ npm start
```

Como nos indica en la terminal, podemos acceder a la web en <http://localhost:8080>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **click.spec.js** dentro de la carpeta **integration**.

Añadimos nuestro caso de prueba y visitamos la página donde se encuentra lo que queremos testear. Como vamos a tener 4 pruebas, y las 4 van a hacerse sobre la misma url, vamos a poner la navegación a la aplicación en el **beforeEach**.

/cypress-interacciones-con-elementos-click-lab/cypress/integration/click.spec.js

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

describe('Panel de teclas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

})
```

Ahora vamos a crear la primera prueba, con la que vamos a comprobar que si introducimos el código correcto, entonces nos muestra en la pantalla el mensaje de "CODE OK".

/cypress-interacciones-con-elementos-click-lab/cypress/integration/click.spec.js

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

describe('Panel de teclas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar CODE OK cuando ponemos el código correcto - 6710', () => {
  })

})
```

Empezamos por buscar las 4 teclas por sus identificadores y llamando a la función **click** sobre ellas para que realice las pulsaciones y marque los números del código.

/cypress-interacciones-con-elementos-click-lab/cypress/integration/click.spec.js

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

describe('Panel de teclas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar CODE OK cuando ponemos el código correcto - 6710', () => {
    cy.get('#p6').click();
    cy.get('#p7').click();
    cy.get('#p1').click();
    cy.get('#p0').click();

  })
})
```

Ahora que hemos escrito el código correcto, vamos a comprobar que el texto que se muestra en la pantalla es el correcto. Pedimos la pantalla por su identificador y comprobamos que contiene el texto **CODE OK**.

/cypress-interacciones-con-elementos-click-lab/cypress/integration/click.spec.js

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

describe('Panel de teclas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar CODE OK cuando ponemos el código correcto - 6710', () => {
    cy.get('#p6').click();
    cy.get('#p7').click();
    cy.get('#p1').click();
    cy.get('#p0').click();

    cy.get('#pantalla-codigo')
      .should('have.text', 'CODE OK');

  })
})
```

Con esto ya tenemos nuestro primer test, ahora toca comprobar que se pasa correctamente.

La siguiente prueba será en la que vamos a comprobar que al pulsar el botón de **CLD** elimina de la

pantalla todos los números en los que hemos pulsado.

/cypress-interacciones-con-elementos-click-lab/cypress/integration/click.spec.js

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

describe('Panel de teclas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar CODE OK cuando ponemos el código correcto - 6710', () => {
    cy.get('#p6').click();
    cy.get('#p7').click();
    cy.get('#p1').click();
    cy.get('#p0').click();

    cy.get('#pantalla-codigo')
      .should('have.text', 'CODE OK');
  })

  it('debería borrar todos los números si pulsamos el botón de CLD', () => {
  })
})
```

Dentro de este caso de prueba, empezamos pulsando unos números cualquiera, buscando estos por sus identificadores. Después buscaremos el botón de CLD, también por su identificador y pulsaremos sobre él.

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

describe('Panel de teclas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar CODE OK cuando ponemos el código correcto - 6710', () => {
    cy.get('#p6').click();
    cy.get('#p7').click();
    cy.get('#p1').click();
    cy.get('#p0').click();

    cy.get('#pantalla-codigo')
      .should('have.text', 'CODE OK');
  })

  it('debería borrar todos los números si pulsamos el botón de CLD', () => {
    cy.get('#p4').click();
    cy.get('#p9').click();
    cy.get('#pclear').click();
  })
})
```

Una vez que hemos realizado los clicks sobre los distintos botones, vamos a añadir la aserción al test. En ella vamos a acceder a la pantalla para comprobar que está vacía o que el texto es un string vacío, cualquiera de las dos aserciones nos sirve en este caso.

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

describe('Panel de teclas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar CODE OK cuando ponemos el código correcto - 6710', () => {
    cy.get('#p6').click();
    cy.get('#p7').click();
    cy.get('#p1').click();
    cy.get('#p0').click();

    cy.get('#pantalla-codigo')
      .should('have.text', 'CODE OK');
  })

  it('debería borrar todos los números si pulsamos el botón de CLD', () => {
    cy.get('#p4').click();
    cy.get('#p9').click();
    cy.get('#pclear').click();

    cy.get('#pantalla-codigo')
      .should('have.text', '')
      .and('be.empty');
  })
})
```

Con esto ya tenemos nuestro test y debería de pasarse correctamente.

Podemos simplificar un poco más el código que hemos puesto haciendo que se pulse sobre los números en una misma linea. Para ello, solo tenemos que usar un selector que nos permita obtener las dos teclas numéricas y llamar al método **click** sobre ellas pasandole la opción **multiple: true** a esta función.

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

describe('Panel de teclas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar CODE OK cuando ponemos el código correcto - 6710', () => {
    cy.get('#p6').click();
    cy.get('#p7').click();
    cy.get('#p1').click();
    cy.get('#p0').click();

    cy.get('#pantalla-codigo')
      .should('have.text', 'CODE OK');
  })

  it('debería borrar todos los números si pulsamos el botón de CLD', () => {
    cy.get('#p4, #p9').click({ multiple: true });
    cy.get('#pclear').click();

    cy.get('#pantalla-codigo')
      .should('have.text', '')
      .and('be.empty');
  })
})
```

Esta forma de hacer pulsaciones sobre múltiples elementos no nos sirve en el primer caso de prueba porque no se nos garantiza que los clicks se realicen en el mismo orden en que hemos puesto los selectores para buscar los elementos.

Otro test menos. Nos quedan dos.

El siguiente consiste en probar que el botón **DEL** elimina los números introducidos de uno en uno. Empezamos por crear el caso de prueba.

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

describe('Panel de teclas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar CODE OK cuando ponemos el código correcto - 6710', () => {
    cy.get('#p6').click();
    cy.get('#p7').click();
    cy.get('#p1').click();
    cy.get('#p0').click();

    cy.get('#pantalla-codigo')
      .should('have.text', 'CODE OK');
  })

  it('debería borrar todos los números si pulsamos el botón de CLD', () => {
    cy.get('#p4, #p9').click({ multiple: true });
    cy.get('#pclear').click();

    cy.get('#pantalla-codigo')
      .should('have.text', '')
      .and('be.empty');
  })

  it('debería borrar el último número al pulsar el botón DEL', () => {
  })
})
```

Al igual que hemos hecho en el anterior test, vamos a pulsar sobre unos números, para después pulsar sobre la tecla de **DEL**. Buscamos todos ellos por sus identificadores.

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

describe('Panel de teclas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar CODE OK cuando ponemos el código correcto - 6710', () => {
    cy.get('#p6').click();
    cy.get('#p7').click();
    cy.get('#p1').click();
    cy.get('#p0').click();

    cy.get('#pantalla-codigo')
      .should('have.text', 'CODE OK');
  })

  it('debería borrar todos los números si pulsamos el botón de CLD', () => {
    cy.get('#p4, #p9').click({ multiple: true });
    cy.get('#pclear').click();

    cy.get('#pantalla-codigo')
      .should('have.text', '')
      .and('be.empty');
  })

  it('debería borrar el último número al pulsar el botón DEL', () => {
    cy.get('#p4').click();
    cy.get('#p9').click();
    cy.get('#p3').click();
    cy.get('#pdel').click();

    })
  })
})
```

Ahora solo nos queda comprobar que en la pantalla solo se muestran los dos primeros números sobre los que se ha pulsado.

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

describe('Panel de teclas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar CODE OK cuando ponemos el código correcto - 6710', () => {
    cy.get('#p6').click();
    cy.get('#p7').click();
    cy.get('#p1').click();
    cy.get('#p0').click();

    cy.get('#pantalla-codigo')
      .should('have.text', 'CODE OK');
  })

  it('debería borrar todos los números si pulsamos el botón de CLD', () => {
    cy.get('#p4, #p9').click({ multiple: true });
    cy.get('#pclear').click();

    cy.get('#pantalla-codigo')
      .should('have.text', '')
      .and('be.empty');
  })

  it('debería borrar el último número al pulsar el botón DEL', () => {
    cy.get('#p4').click();
    cy.get('#p9').click();
    cy.get('#p3').click();
    cy.get('#pdel').click();

    cy.get('#pantalla-codigo')
      .should('have.text', '49');
  })
})
```

Y por último, vamos a añadir otro caso de prueba más para comprobar que no se puede introducir más de 4 números. En caso de hacerlo, los últimos números pulsados no se muestran.

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

describe('Panel de teclas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar CODE OK cuando ponemos el código correcto - 6710', () => {
    cy.get('#p6').click();
    cy.get('#p7').click();
    cy.get('#p1').click();
    cy.get('#p0').click();

    cy.get('#pantalla-codigo')
      .should('have.text', 'CODE OK');
  })

  it('debería borrar todos los números si pulsamos el botón de CLD', () => {
    cy.get('#p4, #p9').click({ multiple: true });
    cy.get('#pclear').click();

    cy.get('#pantalla-codigo')
      .should('have.text', '')
      .and('be.empty');
  })

  it('debería borrar el último número al pulsar el botón DEL', () => {
    cy.get('#p4').click();
    cy.get('#p9').click();
    cy.get('#p3').click();
    cy.get('#pdel').click();

    cy.get('#pantalla-codigo')
      .should('have.text', '49');
  })

  it('no debería permitir introducir más de 4 números', () => {
  })
})
```

Dentro de este bloque it vamos a realizar más de 4 clicks sobre los botones del panel.

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

describe('Panel de teclas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar CODE OK cuando ponemos el código correcto - 6710', () => {
    cy.get('#p6').click();
    cy.get('#p7').click();
    cy.get('#p1').click();
    cy.get('#p0').click();

    cy.get('#pantalla-codigo')
      .should('have.text', 'CODE OK');
  })

  it('debería borrar todos los números si pulsamos el botón de CLD', () => {
    cy.get('#p4, #p9').click({ multiple: true });
    cy.get('#pclear').click();

    cy.get('#pantalla-codigo')
      .should('have.text', '')
      .and('be.empty');
  })

  it('debería borrar el último número al pulsar el botón DEL', () => {
    cy.get('#p4').click();
    cy.get('#p9').click();
    cy.get('#p3').click();
    cy.get('#pdel').click();

    cy.get('#pantalla-codigo')
      .should('have.text', '49');
  })

  it('no debería permitir introducir más de 4 números', () => {
    cy.get('#p4').click();
    cy.get('#p9').click();
    cy.get('#p5').click();
    cy.get('#p1').click();
    cy.get('#p0, #p3').click({ multiple: true });
  })
})
```

Ahora debemos de comprobar que el número que aparece es el que se compone con los primeros 4 números pulsados. Otra forma de realizar esta comprobación sería mirar que la longitud del valor que se muestra en pantalla es de 4 (si hacemos esta comprobación, hay que sacar el texto del

elemento que obtenemos por el identificador).

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

describe('Panel de teclas', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar CODE OK cuando ponemos el código correcto - 6710', () => {
    cy.get('#p6').click();
    cy.get('#p7').click();
    cy.get('#p1').click();
    cy.get('#p0').click();

    cy.get('#pantalla-codigo')
      .should('have.text', 'CODE OK');
  })

  it('debería borrar todos los números si pulsamos el botón de CLD', () => {
    cy.get('#p4, #p9').click({ multiple: true });
    cy.get('#pclear').click();

    cy.get('#pantalla-codigo')
      .should('have.text', '')
      .and('be.empty');
  })

  it('debería borrar el último número al pulsar el botón DEL', () => {
    cy.get('#p4').click();
    cy.get('#p9').click();
    cy.get('#p3').click();
    cy.get('#pdel').click();

    cy.get('#pantalla-codigo')
      .should('have.text', '49');
  })

  it('no debería permitir introducir más de 4 números', () => {
    cy.get('#p4').click();
    cy.get('#p9').click();
    cy.get('#p5').click();
    cy.get('#p1').click();
    cy.get('#p0, #p3').click({ multiple: true });

    cy.get('#pantalla-codigo')
      .should('have.text', '4951')
      .invoke('text') // devuelve el texto del div que pinta el código introducido
      .and('have.length', 4);
  })
})
```

Ya tenemos todos los tests y todos ellos deberían de pasarse correctamente.

Deberíamos de comprobar que fallan para asegurarnos de que los tests no dan falsos positivos.

## 16.3. Lab: Doble Click

En este laboratorio tenemos que testear que al realizar un doble click sobre la caja que aparece en la aplicación web, esta cambia su color de fondo.

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-interacciones-con-elementos-doble-click-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-interacciones-con-elementos-doble-click-lab  
$ cd cypress-interacciones-con-elementos-doble-click-lab  
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests
- http-server: servidor de desarrollo local

```
$ npm install --save-dev cypress http-server
```

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

```
{  
  "name": "cypress-interacciones-con-elementos-doble-click-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "http-server",  
    "cy:open": "cypress open"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.6.0",  
    "http-server": "^0.12.3"  
  }  
}
```

El siguiente paso es crear un archivo **index.html** donde vamos a poner una caja con un script que cambia el color de esta cuando se hace doble click sobre ella.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Click</title>
  <style>
    body {
      height: 100vh;
      width: 100vw;
      background-color: #242424;
    }

    .center {
      height: 100%;
      display: flex;
      justify-content: center;
      align-items: center;
    }

    #caja-doble-click {
      width: 100px;
      height: 100px;
      background-color: blue;
    }

    #caja-doble-click.db-clicked {
      background-color: yellow;
    }
  </style>
</head>
<body>
  <div class="center">
    <div id="caja-doble-click"></div>
  </div>

  <script>
    document.getElementById('caja-doble-click').addEventListener('dblclick', (event) => {
      event.target.className = event.target.className.includes('db-clicked') ? '' : 'db-clicked';
    })
  </script>
</body>
</html>
```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Además, en otra terminal, vamos a lanzar el otro script para poder acceder con Cypress a la web.

```
$ npm start
```

Como nos indica en la terminal, podemos acceder a la web en <http://localhost:8080/>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **doble-click.spec.js** dentro de la carpeta **integration**.

Añadimos nuestro caso de prueba y visitamos la página donde se encuentra lo que queremos testear.

```
/cypress-interacciones-con-elementos-doble-click-lab/cypress/integration/doble-click.spec.js
```

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

describe('Doble Click', () => {
  it('debería cambiar el color de azul a amarillo si se hace doble click', () => {
    cy.visit('http://localhost:8080')

  })
})
```

Ahora vamos a buscar la caja sobre la que hacer el doble click. La buscamos por su identificador.

Después ejecutaremos el doble click llamando a la función **dblclick** sobre el elemento que se nos devuelve.

```
/cypress-interacciones-con-elementos-doble-click-lab/cypress/integration/doble-click.spec.js
```

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

describe('Doble Click', () => {
  it('debería cambiar el color de azul a amarillo si se hace doble click', () => {
    cy.visit('http://localhost:8080')

    cy.get('#caja-doble-click')
      .dblclick();
  })
})
```

Para finalizar con el test, vamos a comprobar que tiene como color de fondo el color amarillo. Para ello, miraremos la propiedad **background-color** de css de este elemento.

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

describe('Doble Click', () => {
  it('debería cambiar el color de azul a amarillo si se hace doble click', () => {
    cy.visit('http://localhost:8080')

    cy.get('#caja-doble-click')
      .dblclick()
      .should('have.css', 'background-color', 'rgb(255, 255, 0)');
  })
})
```

Ahora que tenemos el test, podemos ejecutarlo y comprobar que pasa la prueba.

También deberíamos probar que si le cambiamos el valor del color de fondo el test falla, para evitar encontrarnos con un falso positivo.

## 16.4. Cajas de texto (type y clear)

Cypress nos proporciona un método **type** con el que podremos escribir texto dentro de los campos de texto como los **inputs** o **textareas**.

Este método recibe como parámetro el texto a escribir dentro del campo, y tenemos que llamarlo desde el propio elemento del campo de texto.

```
cy.get('#inputDNI')
  .type('00000000T');
```

Algunas veces necesitaremos realizar alguna pulsación de una tecla como un **intro**, la tecla **control**, las **flechas** del teclado... Le podemos pasar el nombre de estas teclas entre llaves para que las pulse.

Algunos nombres de las teclas que podemos utilizar son:

enter	backspace	del	esc
downarrow	leftarrow	rightarrow	uparrow
alt	shift	ctrl	cmd

```
cy.get('#inputDNI')
  .type('00000000T{enter}');
```

Cuando vamos a escribir un texto en un campo input, es posible que nos encontremos con que ya tiene texto introducido por defecto, como por ejemplo cuando entramos en un formulario con la intención de editar datos.

Podemos utilizar el método **clear** sobre el propio elemento del input para borrar su contenido.

```
cy.get('#inputDNI')
  .clear()
  .type('00000000T');
```

## 16.5. Lab: Cajas de texto

En este laboratorio vamos a ver como trabajar con un campo de texto para añadir tareas en la página de <http://todomvc.com/examples/vue/> y poder interactuar con ellas. Haremos unos tests que prueben las siguientes acciones:

- Añadir 3 tareas y deberían de mostrarse las 3 además de los botones de activas y completadas.
- Añadir 3 tareas y marcar 1 como completada.
- Añadir 3 tareas, completar todas y eliminar las completadas.

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-interacciones-con-elementos-cajas-de-texto-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a seguir los siguientes comandos:

```
$ mkdir cypress-interacciones-con-elementos-cajas-de-texto-lab
$ cd cypress-interacciones-con-elementos-cajas-de-texto-lab
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests

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

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

```
{  
  "name": "cypress-interacciones-con-elementos-cajas-de-texto-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "cy:open": "cypress open"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.8.0"  
  }  
}
```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **todomvc.spec.js** dentro de la carpeta **integration**. Vamos a añadir el bloque **describe** y el primer test donde primero de todo vamos a visitar la página de **todomvc**.

```
/// <reference types="Cypress" />  
  
describe('TodoMVC', () => {  
  it('añadir 3 tareas y deberían de mostrarse las 3 además de los botones de activas y completadas', () => {  
    cy.visit('https://todomvc.com/examples/vue/');  
  })  
})
```

Ahora vamos a añadir las 3 tareas, para esto necesitamos buscar el elemento input y escribir 3 veces el nombre de la tarea seguida de una pulsación sobre la tecla Intro ya que es la forma en que podemos añadirlas en esta aplicación.

/cypress-interacciones-con-elementos-cajas-de-texto-lab/cypress/integration/todomvc.spec.js

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

describe('TodoMVC', () => {
  it('añadir 3 tareas y deberían de mostrarse las 3 además de los botones de activas y completadas', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

  })
})
```

El siguiente paso es comprobar que se han creado 3 tareas, para lo que buscaremos estos elementos y comprobaremos que la longitud es de 3.

/cypress-interacciones-con-elementos-cajas-de-texto-lab/cypress/integration/todomvc.spec.js

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

describe('TodoMVC', () => {
  it('añadir 3 tareas y deberían de mostrarse las 3 además de los botones de activas y completadas', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    cy.get('li.todo')
      .should('have.length', 3);

  })
})
```

Hasta aquí todo debería de ir bien si probamos a ejecutar el test.

Para terminar con este test, vamos a comprobar que aparecen los botones que se nos pide. Si miramos con el inspector de herramientas, veremos que no tienen ninguna clase ni identificador por el que podamos buscarlos. Para este caso vamos a utilizar el **contains** buscando por el texto que esperamos que tengan.

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

describe('TodoMVC', () => {
  it('añadir 3 tareas y deberían de mostrarse las 3 además de los botones de activas y completadas', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    cy.get('li.todo')
      .should('have.length', 3);

    cy.contains('a', 'Active')
      .should('exist');
    cy.contains('a', 'Completed')
      .should('exist');

  })
})
```

Ya tenemos el primer test. Vamos a crear el segundo bloque y realizamos la navegación y creación de tareas como antes.

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

describe('TodoMVC', () => {
  it('añadir 3 tareas y deberían de mostrarse las 3 además de los botones de activas y completadas', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    cy.get('li.todo')
      .should('have.length', 3);

    cy.contains('a', 'Active')
      .should('exist');
    cy.contains('a', 'Completed')
      .should('exist');
  })

  it('añadir 3 tareas y marcar 1 como completada', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

  })
})
```

Vamos a empezar marcando la tercera tarea, por lo que tendremos que acceder a ella y hacer un click. Como es un input de tipo de checkbox, vamos a buscar todos los inputs de este tipo, accederemos al que está en la posición 2, y realizamos el click.

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

describe('TodoMVC', () => {
  it('añadir 3 tareas y deberían de mostrarse las 3 además de los botones de activas y completadas', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    cy.get('li.todo')
      .should('have.length', 3);

    cy.contains('a', 'Active')
      .should('exist');
    cy.contains('a', 'Completed')
      .should('exist');
  })

  it('añadir 3 tareas y marcar 1 como completada', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    cy.get('input.toggle[type="checkbox"]')
      .eq(2)
      .click();

  })
})
})
```

Con esto ya tenemos la tarea completada y las otras dos sin completar. Ahora nos toca comprobar que si filtramos por tareas activas y completadas se muestran correctamente según su estado.

Pulsamos sobre el botón de tareas activas y comprobamos que solo se muestran 2 de ellas.

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

describe('TodoMVC', () => {
  it('añadir 3 tareas y deberían de mostrarse las 3 además de los botones de activas y completadas', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    cy.get('li.todo')
      .should('have.length', 3);

    cy.contains('a', 'Active')
      .should('exist');
    cy.contains('a', 'Completed')
      .should('exist');
  })

  it('añadir 3 tareas y marcar 1 como completada', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    cy.get('input.toggle[type="checkbox"]')
      .eq(2)
      .click();

    cy.contains('a', 'Active')
      .click();
    cy.get('li.todo')
      .should('have.length', 2);
  })
})
```

Para la última comprobación de este test vamos a mirar que en la sección de tareas completadas solo hay 1.

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

describe('TodoMVC', () => {
  it('añadir 3 tareas y deberían de mostrarse las 3 además de los botones de activas y completadas', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    cy.get('li.todo')
      .should('have.length', 3);

    cy.contains('a', 'Active')
      .should('exist');
    cy.contains('a', 'Completed')
      .should('exist');
  })

  it('añadir 3 tareas y marcar 1 como completada', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    cy.get('input.toggle[type="checkbox"]')
      .eq(2)
      .click();

    cy.contains('a', 'Active')
      .click();
    cy.get('li.todo')
      .should('have.length', 2);

    cy.contains('a', 'Completed')
      .click();
    cy.get('li.todo')
      .should('have.length', 1);
  })
})
```

Ya tenemos el segundo test que debería de pasarse correctamente.

Vamos a crear el tercer y último test que consiste en añadir varias tareas, completar todas ellas y después eliminar las completadas.

Empezamos igual que los otros dos anteriores, navegando a la página y añadiendo las tareas.

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

describe('TodoMVC', () => {
  it('añadir 3 tareas y deberían de mostrarse las 3 además de los botones de activas y completadas', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    cy.get('li.todo')
      .should('have.length', 3);

    cy.contains('a', 'Active')
      .should('exist');
    cy.contains('a', 'Completed')
      .should('exist');
  })

  it('añadir 3 tareas y marcar 1 como completada', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    cy.get('input.toggle[type="checkbox"]')
      .eq(2)
      .click();

    cy.contains('a', 'Active')
      .click();
    cy.get('li.todo')
      .should('have.length', 2);

    cy.contains('a', 'Completed')
      .click();
    cy.get('li.todo')
      .should('have.length', 1);
  })

  it('añadir 3 tareas, completar todas y eliminar las completadas', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    })
  })
})
```

Ahora vamos a buscar los distintos checkboxes y vamos a pulsar sobre ellos para marcar las tareas como completadas. Cuando vamos a realizar un **click** sobre varios elementos, tenemos que añadirle la opción **multiple: true** en el comando click. Otra solución sería recorrer la lista de

checkboxes y pulsarlos de uno en uno.

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

describe('TodoMVC', () => {
  it('añadir 3 tareas y deberían de mostrarse las 3 además de los botones de activas y completadas', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    cy.get('li.todo')
      .should('have.length', 3);

    cy.contains('a', 'Active')
      .should('exist');
    cy.contains('a', 'Completed')
      .should('exist');
  })

  it('añadir 3 tareas y marcar 1 como completada', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    cy.get('input.toggle[type="checkbox"]')
      .eq(2)
      .click();

    cy.contains('a', 'Active')
      .click();
    cy.get('li.todo')
      .should('have.length', 2);

    cy.contains('a', 'Completed')
      .click();
    cy.get('li.todo')
      .should('have.length', 1);
  })

  it('añadir 3 tareas, completar todas y eliminar las completadas', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    cy.get('input.toggle[type="checkbox"]')
      .click({ multiple: true });

    })
  })
})
```

Una vez completadas las tareas, vamos a ir a eliminar todas las tareas completadas pulsando sobre el botón **Clear completed**. Y después comprobamos que ya no tenemos ninguna tarea.

/cypress-interacciones-con-elementos-cajas-de-texto-lab/cypress/integration/todomvc.spec.js

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

describe('TodoMVC', () => {
  it('añadir 3 tareas y deberían de mostrarse las 3 además de los botones de activas y completadas', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    cy.get('li.todo')
      .should('have.length', 3);

    cy.contains('a', 'Active')
      .should('exist');
    cy.contains('a', 'Completed')
      .should('exist');
  })

  it('añadir 3 tareas y marcar 1 como completada', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    cy.get('input.toggle[type="checkbox"]')
      .eq(2)
      .click();

    cy.contains('a', 'Active')
      .click();
    cy.get('li.todo')
      .should('have.length', 2);

    cy.contains('a', 'Completed')
      .click();
    cy.get('li.todo')
      .should('have.length', 1);
  })

  it('añadir 3 tareas, completar todas y eliminar las completadas', () => {
    cy.visit('https://todomvc.com/examples/vue/');

    cy.get('.new-todo')
      .type('Tarea 1{enter}')
      .type('Tarea 2{enter}')
      .type('Tarea 3{enter}')

    cy.get('input.toggle[type="checkbox"]')
      .click({ multiple: true });

    cy.get('button.clear-completed')
```

```
.click();
cy.get('li.todo')
  .should('have.length', 0);
})
})
```

Como nos hemos podido dar cuenta, hay unas cuantas instrucciones que se repiten en todos nuestros tests, así que podríamos sacarlas al **beforeEach**.

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

describe('TodoMVC', () => {
  beforeEach(() => {
    cy.visit('https://todomvc.com/examples/vue/');
  })

  it('añadir 3 tareas y deberían de mostrarse las 3 además de los botones de activas y completadas', () => {
    cy.get('li.todo')
      .should('have.length', 3);

    cy.contains('a', 'Active')
      .should('exist');
    cy.contains('a', 'Completed')
      .should('exist');
  })

  it('añadir 3 tareas y marcar 1 como completada', () => {
    cy.get('input.toggle[type="checkbox"]')
      .eq(2)
      .click();

    cy.contains('a', 'Active')
      .click();
    cy.get('li.todo')
      .should('have.length', 2);

    cy.contains('a', 'Completed')
      .click();
    cy.get('li.todo')
      .should('have.length', 1);
  })

  it('añadir 3 tareas, completar todas y eliminar las completadas', () => {
    cy.get('input.toggle[type="checkbox"]')
      .click({ multiple: true });

    cy.get('button.clear-completed')
      .click();
    cy.get('li.todo')
      .should('have.length', 0);
  })
})
```

Y con esto ya tenemos nuestros tests completados y funcionando.

## 16.6. Submit

Los formularios podemos enviarlos de dos formas:

- Una de ellas es pulsando sobre el botón que se encarga de enviar los datos.
- La otra es utilizar el evento submit de los formularios (que es lo que normalmente se emite cuando se pulsa el botón).

Podemos lanzar el evento **submit** del formulario utilizando la función de Cypress que lleva el mismo nombre, sobre el elemento del formulario ya que es el que tiene que detectar el evento para realizar el envío de datos.

```
cy.get('#formulario')
  .submit();
```

## 16.7. Lab: Submit

En este laboratorio vamos a ver como realizar una búsqueda de "Ironman" en la página [wikipedia](#) desde el formulario que aparece en la esquina superior derecha.

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-interacciones-con-elementos-submit-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-interacciones-con-elementos-submit-lab
$ cd cypress-interacciones-con-elementos-submit-lab
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests

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

Ahora añadimos el script que vamos a utilizar después dentro del archivo **package.json**.

```
{  
  "name": "cypress-interacciones-con-elementos-submit-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "cy:open": "cypress open"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.8.0"  
  }  
}
```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **submit.spec.js** dentro de la carpeta **integration**. Como los dos tests se van a realizar en la misma página, vamos a añadir dentro del bloque **describe** el hook de **beforeEach** para realizar la visita a la página donde vamos a realizar la búsqueda.

Además vamos a declarar el texto a buscar en una variable dentro del describe, y que inicializaremos en el hook para utilizarla después en cada test.

```
/// <reference types="Cypress" />  
  
describe('Wiki', () => {  
  let txtBuscar;  
  beforeEach(() => {  
    txtBuscar = 'Hulk'  
    cy.visit('https://es.wikipedia.org/wiki/Wikipedia:Portada')  
  })  
})
```

Vamos a añadir nuestro primer bloque **it**, en el que realizaremos la búsqueda utilizando el método **submit** de Cypress. El primer paso es buscar el campo del texto y escribir sobre el aquello que vamos a buscar.

/cypress-interacciones-con-elementos-submit-lab/cypress/integration/submit.spec.js

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

describe('Wiki', () => {
  let txtBuscar;
  beforeEach(() => {
    txtBuscar = 'Hulk'
    cy.visit('https://es.wikipedia.org/wiki/Wikipedia:Portada')
  })

  it('debería buscar cypress en la wikipedia (enviando el form)', () => {
    cy.get('#searchInput')
      .clear()
      .type(txtBuscar);

  })
})
```

Ahora tenemos que enviar el formulario, pero para ello tenemos que buscar el elemento del formulario ya que es desde este elemento web desde el que hay que llamar al método **submit**.

/cypress-interacciones-con-elementos-submit-lab/cypress/integration/submit.spec.js

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

describe('Wiki', () => {
  let txtBuscar;
  beforeEach(() => {
    txtBuscar = 'Hulk'
    cy.visit('https://es.wikipedia.org/wiki/Wikipedia:Portada')
  })

  it('debería buscar cypress en la wikipedia (enviando el form)', () => {
    cy.get('#searchInput')
      .clear()
      .type(txtBuscar);

    cy.get('#searchform')
      .submit();

  })
})
```

Para terminar este test, vamos a comprobar que el texto en la pestaña del navegador es el texto buscado seguido de "- Wikipedia, la enciclopedia libre". Para acceder al título de la pestaña utilizamos el método de Cypress **title**.

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

describe('Wiki', () => {
  let txtBuscar;
  beforeEach(() => {
    txtBuscar = 'Hulk'
    cy.visit('https://es.wikipedia.org/wiki/Wikipedia:Portada')
  })

  it('debería buscar cypress en la wikipedia (enviando el form)', () => {
    cy.get('#searchInput')
      .clear()
      .type(txtBuscar);

    cy.get('#searchform')
      .submit();

    cy.title()
      .should('equal', `${txtBuscar} - Wikipedia, la enciclopedia libre`)
  })
})
```

Ya tendríamos el primer test, que si ejecutamos debería de funcionar correctamente.

Ahora vamos a realizar este mismo test, pero sin utilizar el método **submit**. ¿De que otra forma podemos hacer la búsqueda? Pulsando sobre el botón del formulario que emite el evento **submit**.

Volvemos a buscar el campo de texto y lo rellenamos de nuevo.

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

describe('Wiki', () => {
  let txtBuscar;
  beforeEach(() => {
    txtBuscar = 'Hulk'
    cy.visit('https://es.wikipedia.org/wiki/Wikipedia:Portada')
  })

  it('debería buscar cypress en la wikipedia (enviando el form)', () => {
    cy.get('#searchInput')
      .clear()
      .type(txtBuscar);

    cy.get('#searchform')
      .submit();

    cy.title()
      .should('equal', `${txtBuscar} - Wikipedia, la enciclopedia libre`)
  })

  it('debería buscar cypress en la wikipedia (pulsando el botón)', () => {
    cy.get('#searchInput')
      .clear()
      .type(txtBuscar);

  })
})
```

Ahora buscamos el botón del tipo submit que envía el formulario y pulsamos sobre el. Esta acción debería llevarnos a la misma página que antes, por lo que vamos a añadir la misma aserción que en nuestro test anterior.

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

describe('Wiki', () => {
  let txtBuscar;
  beforeEach(() => {
    txtBuscar = 'Hulk'
    cy.visit('https://es.wikipedia.org/wiki/Wikipedia:Portada')
  })

  it('debería buscar cypress en la wikipedia (enviando el form)', () => {
    cy.get('#searchInput')
      .clear()
      .type(txtBuscar);

    cy.get('#searchform')
      .submit();

    cy.title()
      .should('equal', `${txtBuscar} - Wikipedia, la enciclopedia libre`)
  })

  it('debería buscar cypress en la wikipedia (pulsando el botón)', () => {
    cy.get('#searchInput')
      .clear()
      .type(txtBuscar);

    cy.get('#searchButton')
      .click();

    cy.title()
      .should('eq', `${txtBuscar} - Wikipedia, la enciclopedia libre`)
  })
})
```

Como podemos observar, hay algunas instrucciones que podemos añadir también dentro del bloque **beforeEach** ya que se repiten en ambos tests. Finalmente quedaría de la siguiente forma.

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

describe('Wiki', () => {
  let txtBuscar;
  beforeEach(() => {
    txtBuscar = 'Hulk'
    cy.visit('https://es.wikipedia.org/wiki/Wikipedia:Portada')

    cy.get('#searchInput')
      .clear()
      .type(txtBuscar);
  })

  it('debería buscar cypress en la wikipedia (enviando el form)', () => {
    cy.get('#searchform')
      .submit();

    cy.title()
      .should('equal', `${txtBuscar} - Wikipedia, la enciclopedia libre`)
  })

  it('debería buscar cypress en la wikipedia (pulsando el botón)', () => {
    cy.get('#searchButton')
      .click();

    cy.title()
      .should('eq', `${txtBuscar} - Wikipedia, la enciclopedia libre`)
  })
})
```

Con esto ya tendríamos nuestros tests.

## 16.8. Check y uncheck

A la hora de trabajar con inputs de los tipos checkbox y radio button podemos utilizar las siguientes funciones de Cypress:

- **check**: marca el input.
- **uncheck**: desmarca el input.

Ambos métodos se pueden usar directamente sobre el elemento input para marcarlo o desmarcarlo.

Pero también se puede usar sobre una lista de elementos y pasarle como parámetro el valor de la opción a marcar/desmarcar, o una lista con estos valores para marcar/desmarcar varios a la vez.

```
cy('[type="checkbox"]').check(['check1', 'check2']);
cy('[type="checkbox"]').uncheck('check4');
```

## 16.9. Lab: Checkboxes

En este laboratorio vamos a ver como interactuar con los inputs del tipo checkbox para marcar o desmarcar las distintas opciones que se muestran.

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-interacciones-con-elementos-checkboxes-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a seguir los siguientes comandos:

```
$ mkdir cypress-interacciones-con-elementos-checkboxes-lab
$ cd cypress-interacciones-con-elementos-checkboxes-lab
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests
- http-server: servidor de desarrollo local

```
$ npm install --save-dev cypress http-server
```

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

```
{  
  "name": "cypress-interacciones-con-elementos-checkboxes-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "http-server",  
    "cy:open": "cypress open"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.6.0",  
    "http-server": "^0.12.3"  
  }  
}
```

El siguiente paso es crear un archivo **index.html** donde vamos a poner un script que crea una cookie en la página web.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Checkboxes</title>
  <style>
    body {
      height: 100vh;
      width: 100vw;
      background-color: #242424;
    }

    .center {
      height: 100%;
      display: flex;
      justify-content: center;
      align-items: center;
    }

    div {
      color: white;
    }
  </style>
</head>
<body>
  <div class="center">
    <div id="hobbies" data-cy="hobbies">
      <div><input type="checkbox" id="hobby1" name="hobby" value="cine">Cine</div>
      <div><input type="checkbox" id="hobby2" name="hobby" value="musica">Música</div>
      <div><input type="checkbox" id="hobby3" name="hobby" value="series" checked>Series</div>
      <div><input type="checkbox" id="hobby4" name="hobby" value="tenis" checked>Tenis</div>
      <div><input type="checkbox" id="hobby5" name="hobby" value="baloncesto">Baloncesto</div>
    </div>
  </div>
</body>
</html>
```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Además, en otra terminal, vamos a lanzar el otro script para poder acceder con Cypress a la web.

```
$ npm start
```

Como nos indica en la terminal, podemos acceder a la web en <http://localhost:8080/>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **checkboxes.spec.js** dentro de la carpeta **integration**. Dentro de este archivo empezamos por crear el bloque describe y la navegación a la página principal dentro del beforeEach.

/cypress-interacciones-con-elementos-checkboxes-lab/cypress/integration/checkboxes.spec.js

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

describe('Checkboxes', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

})
```

Empezamos con el primer test, en el que vamos a comprobar que hay dos hobbies seleccionados inicialmente. Podemos obtener estos elementos buscando los inputs que tienen la pseudo-clase **checked**.

/cypress-interacciones-con-elementos-checkboxes-lab/cypress/integration/checkboxes.spec.js

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

describe('Checkboxes', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('deberían de estar seleccionados dos hobbies inicialmente', () => {
    cy.get('[data-cy=hobbies] > input:checked')
      .should('have.length', 2);
  })
})
```

La siguiente prueba consiste en comprobar que podemos seleccionar más hobbies usando el método **check**.

Añadimos el bloque it y seleccionamos un hobby más buscando el elemento por su identificador.

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

describe('Checkboxes', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('deberían de estar seleccionados dos hobbies inicialmente', () => {
    cy.get('[data-cy=hobbies] > input:checked')
      .should('have.length', 2);
  })

  it('debería de poder seleccionar más hobbies al pulsar sobre ellos', () => {
    cy.get('#hobby1')
      .check();

    cy.get('[data-cy=hobbies] input:checked')
      .should('have.length', 3);
  })
})
```

La última prueba a realizar consiste en comprobar que si usamos el **uncheck** sobre un checkbox que ya se encuentra seleccionado, quitamos la selección. Para ello vamos a buscar el checkbox de Series por su identificador y vamos a utilizar la función anterior para comprobar que solo queda uno seleccionado.

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

describe('Checkboxes', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('deberían de estar seleccionados dos hobbies inicialmente', () => {
    cy.get('[data-cy=hobbies] > input:checked')
      .should('have.length', 2);
  })

  it('debería de poder seleccionar más hobbies al pulsar sobre ellos', () => {
    cy.get('#hobby1')
      .check();

    cy.get('[data-cy=hobbies] input:checked')
      .should('have.length', 3);
  })

  it('debería de poder deseleccionar hobbies al pulsar sobre ellos cuando ya están seleccionados', () => {
    cy.get('#hobby3')
      .uncheck();

    cy.get('[data-cy=hobbies] input:checked')
      .should('have.length', 1);
  })
})
```

En los tests de arriba hemos usado los métodos de **check** y **uncheck**, aunque también podríamos haber utilizado el método **click**.

La diferencia es que con los dos primeros nos aseguramos que solo realizamos una acción sobre el checkbox, o se marca o se desmarca. Mientras que con el click estaríamos haciendo las dos acciones dependiendo del estado en el que se encuentra el checkbox, es decir, si está marcado lo desmarcamos y viceversa.

## 16.10. Desplegables (Select)

Para interactuar con un desplegable, el usuario tiene que realizar distintas acciones, primero pulsar sobre el propio desplegable para mostrar las opciones, y después volver a pulsar sobre alguna de estas opciones.

Pero en Cypress han pensado en hacer este tipo de interacciones fáciles, por lo que nos proporcionan un método **select** al cual se le va a pasar como parámetro el valor (atributo value) o texto visible de la opción que queremos seleccionar.

```
cy.get('#miDesplegable')
  .select('Opción 7');
```

A veces nos podemos encontrar con desplegables que permiten seleccionar varias de las opciones que se muestran. Estos desplegables son los que llevan el atributo **multiple** en la etiqueta select.

La función a utilizar con estos desplegables es la misma que antes. La principal diferencia es que ahora en lugar de pasarle como parámetro un único valor (sea el texto visible o el value del option) se le va a pasar un array con estos valores a seleccionar.

```
cy.get('#miDesplegableMultiple')
  .select(['opcion1', 'opcion4']);
```

## 16.11. Lab: Desplegable

En este laboratorio vamos a ver como interactuar con el desplegable de una página web.

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-interacciones-con-elementos-select-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a seguir los siguientes comandos:

```
$ mkdir cypress-interacciones-con-elementos-select-lab
$ cd cypress-interacciones-con-elementos-select-lab
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests
- http-server: servidor de desarrollo local

```
$ npm install --save-dev cypress http-server
```

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

```
{  
  "name": "cypress-interacciones-con-elementos-select-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "http-server",  
    "cy:open": "cypress open"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.5.0",  
    "http-server": "^0.12.3"  
  }  
}
```

El siguiente paso es crear un archivo **index.html** donde vamos a poner el desplegable que queremos probar.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Select</title>
  <style>
    body {
      height: 100vh;
      width: 100vw;
      background-color: #242424;
    }

    .center {
      height: 100%;
      display: flex;
      justify-content: center;
      align-items: center;
    }
  </style>
</head>
<body>
  <div class="center">
    <select id="select-coches-electricos">
      <option value="polestar-2">Polestar 2</option>
      <option value="nio-et7">Nio eT7</option>
      <option value="tesla-model-3">Tesla Model 3</option>
      <option value="xpeng-p7">Xpeng P7</option>
    </select>
  </div>
</body>
</html>
```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Además, en otra terminal, vamos a lanzar el otro script para poder acceder con Cypress a la web.

```
$ npm start
```

Como nos indica en la terminal, podemos acceder a la web en <http://localhost:8080/>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo

**select.spec.js** dentro de la carpeta **integration**. Como todos los tests se van a realizar en la misma página, vamos a añadir dentro del bloque **describe** el hook de **beforeEach** para realizar la visita a la página donde se levanta nuestra página web.

/cypress-interacciones-con-elementos-select-lab/cypress/integration/select.spec.js

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

describe('Desplegable', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })
})
```

Nuestro primer test va a comprobar que por defecto aparece seleccionado el Polestar 2, es decir, la primera opción que hay en el desplegable.

Tenemos que buscar el desplegable, por ejemplo utilizando el identificador que tiene puesto y comprobar que el valor de la etiqueta es el correcto, el de la primera opción que hay.

/cypress-interacciones-con-elementos-select-lab/cypress/integration/select.spec.js

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

describe('Desplegable', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería utilizar la primera opción como valor inicial', () => {
    cy.get('#select-coches-electricos')
      .should('have.value', 'polestar-2');
  })
})
```

Si entramos al test desde la ventana de Cypress, veremos que se pasa.

Ahora estamos comprobando que el valor del select es el de la primera opción, pero ¿cómo podemos hacer para comprobar que el texto que se está mostrando es el correcto?, Polester 2 en nuestro caso.

Esta vez vamos a usar **have.text** y tenemos que ver como llegar a obtener la opción que está seleccionada.

En CSS hay unas pseudo-clases que permiten acceder a etiquetas que tienen un estado que les indican que están marcadas, tienen el foco... Uno de ellos nos permite saber cuáles están seleccionadas, y es el que vamos a utilizar, el **selected**.

Entonces nuestro test queda de la siguiente forma, pedimos todas las opciones filtrando por la que está seleccionada, y después comprobamos que el texto es el que esperamos.

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

describe('Desplegable', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería utilizar la primera opción como valor inicial', () => {
    cy.get('#select-coches-electricos')
      .should('have.value', 'polestar-2');
  })

  it('debería utilizar la primera opción como valor inicial (comprobamos el texto)', () => {
    cy.get('option:selected')
      .should('have.text', 'Polestar 2');
  })
})
```

Podemos comprobar que este test también se está pasando desde dentro del archivo de **select.spec.js** en la pantalla de Cypress.

El siguiente test consiste en cambiar la selección, por ejemplo al Nio. Como ya hemos comentado, Cypress nos da el método de **select** para poder indicar la opción que queremos seleccionar, así que vamos a decirle que seleccione esa utilizando su valor. Para ello solo tenemos que pasarle a la función de select el valor del atributo value del option que queremos seleccionar.

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

describe('Desplegable', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería utilizar la primera opción como valor inicial', () => {
    cy.get('#select-coches-electricos')
      .should('have.value', 'polestar-2');
  })

  it('debería utilizar la primera opción como valor inicial (comprobamos el texto)', () => {
    cy.get('option:selected')
      .should('have.text', 'Polestar 2');
  })

  it('debería cambiar el valor seleccionado al pulsar sobre otra opción (usando el valor)', () => {
    const valorDeLaOpcion = 'nio-et7';

    cy.get('#select-coches-electricos')
      .select(valorDeLaOpcion)
      .should('have.value', valorDeLaOpcion);
  })
})
```

Otra forma de seleccionar una opción es pasandole como parámetro de la función select el texto que es visible al usuario, es decir, el texto que vemos al desplegar las opciones.

Por tanto, vamos a crear otro test, en el que vamos a seleccionar la opción del Tesla utilizando el texto visible.

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

describe('Desplegable', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería utilizar la primera opción como valor inicial', () => {
    cy.get('#select-coches-electricos')
      .should('have.value', 'polestar-2');
  })

  it('debería utilizar la primera opción como valor inicial (comprobamos el texto)', () => {
    cy.get('option:selected')
      .should('have.text', 'Polestar 2');
  })

  it('debería cambiar el valor seleccionado al pulsar sobre otra opción (usando el valor)', () => {
    const valorDeLaOpcion = 'nio-et7';

    cy.get('#select-coches-electricos')
      .select(valorDeLaOpcion)
      .should('have.value', valorDeLaOpcion);
  })

  it('debería cambiar el valor seleccionado al pulsar sobre otra opción (usando el texto visible)', () => {
    cy.get('#select-coches-electricos')
      .select('Tesla Model 3')
      .should('have.value', 'tesla-model-3');
  })
})
```

Ahora podemos comprobar que todos los tests se están pasando correctamente.

## 16.12. Lab: Desplegable múltiple

En este laboratorio vamos a ver como interactuar con el desplegable de opciones múltiples de una página web.

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-interacciones-con-elementos-select-multiple-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-interacciones-con-elementos-select-multiple-lab
$ cd cypress-interacciones-con-elementos-select-multiple-lab
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests
- http-server: servidor de desarrollo local

```
$ npm install --save-dev cypress http-server
```

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

*/cypress-interacciones-con-elementos-select-multiple-lab/package.json*

```
{  
  "name": "cypress-interacciones-con-elementos-select-multiple-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "http-server",  
    "cy:open": "cypress open"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.5.0",  
    "http-server": "^0.12.3"  
  }  
}
```

El siguiente paso es crear un archivo **index.html** donde vamos a poner el desplegable de múltiples opciones que queremos probar.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Select múltiple</title>
  <style>
    body {
      height: 100vh;
      width: 100vw;
      background-color: #242424;
    }

    .center {
      height: 100%;
      display: flex;
      justify-content: center;
      align-items: center;
    }

    #select-colores {
      width: 150px;
      height: 160px;
    }
  </style>
</head>
<body>
  <div class="center">
    <select id="select-colores" multiple>
      <option value="negro">Negro</option>
      <option value="rojo">Rojo</option>
      <option value="azul">Azul</option>
      <option value="amarillo">Amarillo</option>
      <option value="blanco">Blanco</option>
    </select>
  </div>
</body>
</html>
```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Además, en otra terminal, vamos a lanzar el otro script para poder acceder con Cypress a la web.

```
$ npm start
```

Como nos indica en la terminal, podemos acceder a la web en <http://localhost:8080/>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **select-multiple.spec.js** dentro de la carpeta **integration**. Como todos los tests se van a realizar en la misma página, vamos a añadir dentro del bloque **describe** el hook de **beforeEach** para realizar la visita a la página donde se levanta nuestra página web.

*/cypress-interacciones-con-elementos-select-multiple-lab/cypress/integration/select-multiple.spec.js*

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

describe('Desplegable múltiple', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })
})
```

El primer test debería de comprobar que este select tiene el atributo **multiple** y por tanto vamos a poder seleccionar varias opciones. Creamos el primer bloque **it** donde buscaremos la etiqueta del select por su identificador y miramos si tiene el atributo **multiple**.

*/cypress-interacciones-con-elementos-select-multiple-lab/cypress/integration/select-multiple.spec.js*

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

describe('Desplegable múltiple', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería tener el atributo multiple', () => {
    cy.get('#select-colores')
      .should('have.attr', 'multiple');
  })
})
```

Con esto ya tenemos nuestro primer test y debería de ejecutarse sin problemas.

El siguiente test que vamos a añadir es en el que vamos a seleccionar varios elementos para comprobar que se seleccionan bien. En este primero vamos a seleccionarlas por el texto visible, por lo que vamos a crearnos un array con el texto de las opciones a seleccionar. También vamos a crearnos un array con esas mismas opciones en minúsculas porque las necesitaremos en la aserción.

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

describe('Desplegable múltiple', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería tener el atributo multiple', () => {
    cy.get('#select-colores')
      .should('have.attr', 'multiple');
  })

  it('debería seleccionar varias opciones (usando el texto visible)', () => {
    const colores = ['Azul', 'Amarillo', 'Negro']
    const coloresLowerCase = colores.map(color => color.toLowerCase());
  })
})
```

Ahora vamos a buscar el desplegable para llamar a la función **select** sobre él, a la que le vamos a pasar el array de opciones a seleccionar.

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

describe('Desplegable múltiple', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería tener el atributo multiple', () => {
    cy.get('#select-colores')
      .should('have.attr', 'multiple');
  })

  it('debería seleccionar varias opciones (usando el texto visible)', () => {
    const colores = ['Azul', 'Amarillo', 'Negro']
    const coloresLowerCase = colores.map(color => color.toLowerCase());

    cy.get('#select-colores')
      .select(colores);
  })
})
```

Ahora vamos a comprobar que el valor del select contiene un array con los valores (atributo value de las opciones seleccionadas) y para ello tenemos que invocar el valor de la propiedad **val** y realizar la aserción sobre este. Para esta aserción no hay que mirar que el orden de las opciones sea el mismo, por tanto vamos a usar **have.members** que comprueba que dos arrays tienen los mismos

miembros.

/cypress-interacciones-con-elementos-select-multiple-lab/cypress/integration/select-multiple.spec.js

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

describe('Desplegable múltiple', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería tener el atributo multiple', () => {
    cy.get('#select-colores')
      .should('have.attr', 'multiple');
  })

  it('debería seleccionar varias opciones (usando el texto visible)', () => {
    const colores = ['Azul', 'Amarillo', 'Negro']
    const coloresLowerCase = colores.map(color => color.toLowerCase());

    cy.get('#select-colores')
      .select(colores)
      .invoke('val')
      .should('have.members', coloresLowerCase);
  })
})
```

Otra posible aserción sería comprobar que el número de opciones que se han seleccionado es igual a la longitud del array. Podemos usar la pseudo clase `:selected` para obtener las opciones seleccionadas.

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

describe('Desplegable múltiple', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería tener el atributo multiple', () => {
    cy.get('#select-colores')
      .should('have.attr', 'multiple');
  })

  it('debería seleccionar varias opciones (usando el texto visible)', () => {
    const colores = ['Azul', 'Amarillo', 'Negro']
    const coloresLowerCase = colores.map(color => color.toLowerCase());

    cy.get('#select-colores')
      .select(colores)
      .invoke('val')
      .should('have.members', coloresLowerCase);

    cy.get('#select-colores > option:selected')
      .should('have.length', 3);
  })
})
```

Ya tenemos el segundo test.

Vamos a por el último en el que vamos a comprobar que las opciones se pueden seleccionar utilizando el valor de estas en lugar del texto visible. Empezamos por crear el bloque it y definir el array con los valores de las opciones que queremos seleccionar.

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

describe('Desplegable múltiple', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería tener el atributo multiple', () => {
    cy.get('#select-colores')
      .should('have.attr', 'multiple');
  })

  it('debería seleccionar varias opciones (usando el texto visible)', () => {
    const colores = ['Azul', 'Amarillo', 'Negro']
    const coloresLowerCase = colores.map(color => color.toLowerCase());

    cy.get('#select-colores')
      .select(colores)
      .invoke('val')
      .should('have.members', coloresLowerCase);

    cy.get('#select-colores > option:selected')
      .should('have.length', 3);
  })

  it('debería seleccionar varias opciones (usando el valor)', () => {
    const valorDeLaOpcion = ['blanco'];
  })
})
```

Al igual que antes, buscamos el select por su identificador y llamamos a la función **select** pasandole el array de selecciones a realizar.

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

describe('Desplegable múltiple', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería tener el atributo multiple', () => {
    cy.get('#select-colores')
      .should('have.attr', 'multiple');
  })

  it('debería seleccionar varias opciones (usando el texto visible)', () => {
    const colores = ['Azul', 'Amarillo', 'Negro']
    const coloresLowerCase = colores.map(color => color.toLowerCase());

    cy.get('#select-colores')
      .select(colores)
      .invoke('val')
      .should('have.members', coloresLowerCase);

    cy.get('#select-colores > option:selected')
      .should('have.length', 3);
  })

  it('debería seleccionar varias opciones (usando el valor)', () => {
    const valorDeLaOpcion = ['blanco'];

    cy.get('#select-colores')
      .select(valorDeLaOpcion);
  })
})
```

Por último añadimos las aserciones. Una de ellas consiste en invocar el valor del desplegable y comprobar que el array que obtenemos es igual al que contenía la opción a seleccionar.

Por otro lado, podemos buscar las opciones seleccionadas y comprobar que la lista tiene una longitud igual a la del array de opciones a seleccionar.

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

describe('Desplegable múltiple', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería tener el atributo multiple', () => {
    cy.get('#select-colores')
      .should('have.attr', 'multiple');
  })

  it('debería seleccionar varias opciones (usando el texto visible)', () => {
    const colores = ['Azul', 'Amarillo', 'Negro']
    const coloresLowerCase = colores.map(color => color.toLowerCase());

    cy.get('#select-colores')
      .select(colores)
      .invoke('val')
      .should('have.members', coloresLowerCase);

    cy.get('#select-colores > option:selected')
      .should('have.length', 3);
  })

  it('debería seleccionar varias opciones (usando el valor)', () => {
    const valorDeLaOpcion = ['blanco'];

    cy.get('#select-colores')
      .select(valorDeLaOpcion)
      .invoke('val')
      .should('deep.equal', valorDeLaOpcion);

    cy.get('#select-colores > option:selected')
      .should('have.length', valorDeLaOpcion.length);
  })
})
```

Y con esto ya hemos terminado con estos tests. Podemos comprobar que se pasan todos ellos correctamente.

## 16.13. Trigger

El comando **trigger** de Cypress permite lanzar un evento sobre un elemento del DOM. Este comando nos va a permitir realizar algunas acciones que no son tan fáciles de realizar como llamar a un comando que te escribe en un input o pulsa sobre un elemento.

Este comando se suele utilizar con eventos del **ratón**, como un **mousemove**, **mousedown** o incluso

eventos para cambiar valores de otros elementos web más complejos como un input de tipo **range** cuyo valor cambia al desplazar la barrita que se muestra, donde podríamos indicarle que lance el evento **change** y evitar así tener que desplazarla lo que sería una interacción más difícil de conseguir.

```
cy.get('#teclaH')
  .trigger('mousedown', { which: 1 })
  .trigger('mouseup');
```

## 16.14. Lab: Drag and Drop

En este laboratorio vamos a ver como realizar la acción de arrastrar un elemento y soltarlo sobre otro elemento distinto.

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-interacciones-con-elementos-drag-and-drop-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-interacciones-con-elementos-drag-and-drop-lab
$ cd cypress-interacciones-con-elementos-drag-and-drop-lab
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar la dependencia de Cypress.

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

Ahora añadimos el script que vamos a utilizar después dentro del archivo **package.json**.

```
{  
  "name": "cypress-interacciones-con-elementos-drag-and-drop-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "cy:open": "cypress open"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.6.0",  
    "http-server": "^0.12.3"  
  }  
}
```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Como nos indica en la terminal, podemos acceder a la web en <http://localhost:8080/>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **drag-and-drop.spec.js** dentro de la carpeta **integration**.

Añadimos nuestro caso de prueba donde comprobaremos que si arrastramos un elemento sobre otro, el texto del contenedor donde hemos soltado el primer elemento cambia.

```
/// <reference types="Cypress" />  
  
describe('Drag and Drop', () => {  
  it('debería cambiar el texto si arrastramos la primera caja sobre la segunda', () => {  
    cy.visit('http://cookbook.seleniumacademy.com/DragDropDemo.html');  
  
  })  
})
```

Empezamos por disparar un evento **mousedown** para pulsar sobre el elemento que queremos arrastrar. Vamos a pasarle como segundo parámetro un objeto de opciones donde le vamos a indicar que el botón del ratón que se tiene que pulsar es el izquierdo.

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

describe('Drag and Drop', () => {
  it('debería cambiar el texto si arrastramos la primera caja sobre la segunda', () => {
    cy.visit('http://cookbook.seleniumacademy.com/DragDropDemo.html');

    cy.get('#draggable')
      .trigger('mousedown', { which: 1 })
  })
})
```

Ya tenemos el botón del ratón pulsado sobre el elemento que queremos arrastrar, el siguiente paso es arrastrarlo hasta ponerlo encima del elemento contenedor. Para ello vamos a disparar el evento **mousemove** y le vamos a pasar como segundo parámetro del **trigger** un objeto con las coordenadas (**pageX** y **pageY**) de la página donde se encuentra el elemento contenedor.

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

describe('Drag and Drop', () => {
  it('debería cambiar el texto si arrastramos la primera caja sobre la segunda', () => {
    cy.visit('http://cookbook.seleniumacademy.com/DragDropDemo.html');

    cy.get('#draggable')
      .trigger('mousedown', { which: 1 })
      .trigger('mousemove', { pageX: 180, pageY: 55 })

  })
})
```

Por último, una vez que hemos movido el ratón hasta una posición específica hay que dejar de pulsar el botón para poder soltar el elemento que estamos moviendo sobre el elemento contenedor, por lo que vamos a disparar un **mouseup**.

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

describe('Drag and Drop', () => {
  it('debería cambiar el texto si arrastramos la primera caja sobre la segunda', () => {
    cy.visit('http://cookbook.seleniumacademy.com/DragDropDemo.html');

    cy.get('#draggable')
      .trigger('mousedown', { which: 1 })
      .trigger('mousemove', { pageX: 180, pageY: 55 })
      .trigger('mouseup');

  })
})
```

Una vez que tenemos la acción completa y ya se puede mover un elemento sobre el otro, vamos a comprobar que el texto del contenedor ha cambiado y muestra Dropped!.

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

describe('Drag and Drop', () => {
  it('debería cambiar el texto si arrastramos la primera caja sobre la segunda', () => {
    cy.visit('http://cookbook.seleniumacademy.com/DragDropDemo.html');

    cy.get('#draggable')
      .trigger('mousedown', { which: 1 })
      .trigger('mousemove', { pageX: 180, pageY: 55 })
      .trigger('mouseup');

    cy.get('#droppable > p')
      .should('have.text', 'Dropped!');

  })
})
```

Con esto ya tendríamos nuestro test y debería de estar ejecutandose correctamente.

## 16.15. Cookies

Cypress nos permite interactuar con las cookies del navegador, accediendo a ellas para ver su valor, crear nuevas cookies o incluso borrarlas. Para todo ello usaremos los métodos que nos proporciona:

Función	Descripción
getCookies()	Devuelve todas las cookies.
getCookie(nombre)	Accede a la cookie que tiene el nombre indicado.

Función	Descripción
setCookie(nombre, valor, {...opciones})	Este método añade una cookie dándole un nombre, su valor y si es necesario un objeto con el resto de valores que configuran una cookie.
clearCookie(nombre)	Elimina una cookie dado su nombre.
clearCookies()	Elimina todas las cookies.

Un caso en el que podríamos utilizar la creación de una cookie sería para acelerar los tests creando una cookie de sesión que autentique a un usuario y evitar de esta forma tener que llenar el formulario de login en todos los casos de prueba que vayamos a crear y para los cuales se necesite estar logueado.

## 16.16. Lab: Cookies

En este laboratorio vamos a ver como acceder a las cookies de las aplicaciones web y ver como interactuar con ellas.

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-interacciones-con-elementos-cookies-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-interacciones-con-elementos-cookies-lab
$ cd cypress-interacciones-con-elementos-cookies-lab
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests
- http-server: servidor de desarrollo local

```
$ npm install --save-dev cypress http-server
```

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

```
{  
  "name": "cypress-interacciones-con-elementos-cookies-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "http-server",  
    "cy:open": "cypress open"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.6.0",  
    "http-server": "^0.12.3"  
  }  
}
```

El siguiente paso es crear un archivo **index.html** donde vamos a poner un script que crea una cookie en la página web.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Screenshot</title>
  <style>
    body {
      height: 100vh;
      width: 100vw;
      background-color: #242424;
    }

    .center {
      height: 100%;
      display: flex;
      justify-content: center;
      align-items: center;
      color: white;
    }
  </style>
</head>
<body>
  <div class="center">
    Mira las cookies desde las "Herramientas del desarrollador > Application > Cookies"
  </div>
  <script>
    const valorCookie = 'Cookies, cookies...';
    document.cookie = `miCookie=${valorCookie}`;
  </script>
</body>
</html>
```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Además, en otra terminal, vamos a lanzar el otro script para poder acceder con Cypress a la web.

```
$ npm start
```

Como nos indica en la terminal, podemos acceder a la web en <http://localhost:8080/>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo

**cookies.spec.js** dentro de la carpeta **integration**. Dentro de este archivo empezamos por crear el bloque describe.

/cypress-interacciones-con-elementos-cookies-lab/cypress/integration/cookies.spec.js

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

describe('Cookies', () => {
})
```

Vamos a crear un test en el que empezaremos por definir dos constantes que van a almacenar:

- La primera, el valor de la cookie que se crea con el script de la página web.
- La segunda, el valor de la cookie que vamos a crear desde el propio Cypress.

/cypress-interacciones-con-elementos-cookies-lab/cypress/integration/cookies.spec.js

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

describe('Cookies', () => {
  it('debería de tener una cookie inicialmente y otra que vamos a crear aquí', () => {
    const valorCookieCy = 'La cookie de Cypress';
    const valorMiCookie = 'Cookies, cookies...';

    cy.visit('http://localhost:8080');

  })
})
```

Empezaremos comprobando que en la página hay una sola cookie (la que se crea desde el **index.html**) inicialmente, y que esta cookie tiene el valor que esperamos que tenga.

Utilizaremos la llamada a **getCookies** para comprobar que solo hay 1. Además buscaremos la cookie por el nombre con el método **getCookie** y comprobaremos que el objeto que obtenemos tiene una propiedad **value** con el valor de **valorMiCookie**.

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

describe('Cookies', () => {
  it('debería de tener una cookie inicialmente y otra que vamos a crear aquí', () => {
    const valorCookieCy = 'La cookie de Cypress';
    const valorMiCookie = 'Cookies, cookies...';

    cy.visit('http://localhost:8080');

    cy.getCookies()
      .should('have.length', 1);

    cy.getCookie('miCookie')
      .should('have.property', 'value', valorMiCookie);
  })
})
```

El siguiente paso es crear nuestra cookie desde este test usando la función de **setCookie** a la cual le vamos a pasar como parámetro el nombre de la cookie y su valor. Justo después podemos comprobar que se ha creado.

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

describe('Cookies', () => {
  it('debería de tener una cookie inicialmente y otra que vamos a crear aquí', () => {
    const valorCookieCy = 'La cookie de Cypress';
    const valorMiCookie = 'Cookies, cookies...';

    cy.visit('http://localhost:8080');

    cy.getCookies()
      .should('have.length', 1);

    cy.getCookie('miCookie')
      .should('have.property', 'value', valorMiCookie);

    cy.setCookie('cookie-cypress', valorCookieCy);

    cy.getCookie('cookie-cypress')
      .should('have.property', 'value', valorCookieCy);
  })
})
```

Comprobamos también que tenemos 2 cookies en total, y después vamos a ir eliminandolas con los métodos de **clearCookie** y **clearCookies** para comprobar que se eliminan de la página.

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

describe('Cookies', () => {
  it('debería de tener una cookie inicialmente y otra que vamos a crear aquí', () => {
    const valorCookieCy = 'La cookie de Cypress';
    const valorMiCookie = 'Cookies, cookies...';

    cy.visit('http://localhost:8080');

    cy.getCookies()
      .should('have.length', 1);

    cy.getCookie('miCookie')
      .should('have.property', 'value', valorMiCookie);

    cy.setCookie('cookie-cypress', valorCookieCy);

    cy.getCookie('cookie-cypress')
      .should('have.property', 'value', valorCookieCy);

    cy.getCookies()
      .should('have.length', 2);

    cy.clearCookie('miCookie');
    cy.getCookies()
      .should('have.length', 1);

    cy.clearCookies();
    cy.getCookies()
      .should('be.empty');
  })
})
```

Si ejecutamos el test debería de pasarse correctamente.

## 16.17. Eventos

En Cypress se han definido una serie de eventos que se van emitiendo mientras se ejecuta en nuestro navegador.

Con estos eventos podemos controlar algunos elementos de la aplicación a testear con los cuales no se puede interactuar con comandos como los que ya hemos visto anteriormente. Se pueden utilizar tanto para controlar el comportamiento de la aplicación, como para debuggearla.

Algunos de estos eventos son:

- **uncaught:exception**: se emite cuando ocurre una excepción en la aplicación.

- **window:alert**: se emite cuando se muestra el popup del navegador de tipo Alert para mostrar un texto informativo. Nos proporciona el texto del popup.
- **window:confirm**: se emite cuando se muestra el popup del navegador de tipo Confirm para aceptar o cancelar lo que se indica en el. Nos proporciona el texto del popup. Si devolvemos un false le estamos indicando que queremos cancelar el Confirm.
- **url:changed**: se emite cuando cambia la URL de la aplicación. Nos proporciona la nueva URL.

Para detectar los eventos, usaremos el comando **on** al cual se le pasa como parámetro el nombre del evento a detectar.

Podemos acceder a los valores que se han descrito en el callback que se le pasa al **then**.

```
cy.on('url:changed')
  .then(newURL => {
    expect(newURL).to.eq('/nueva/url');
  })
```

## 16.18. Lab: Popups del navegador

En este laboratorio vamos a ver como trabajar con los distintos popups que nos muestra el navegador:

- Alert
- Confirm
- Prompt

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-interacciones-con-elementos-popups-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-interacciones-con-elementos-popups-lab
$ cd cypress-interacciones-con-elementos-popups-lab
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests
- http-server: servidor de desarrollo local

```
$ npm install --save-dev cypress http-server
```

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

/cypress-interacciones-con-elementos-popups-lab/package.json

```
{  
  "name": "cypress-interacciones-con-elementos-popups-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "http-server",  
    "cy:open": "cypress open"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.6.0",  
    "http-server": "^0.12.3"  
  }  
}
```

El siguiente paso es crear un archivo **index.html** donde vamos a poner 3 botones donde cada uno nos muestra un popup distinto, además de un par de párrafos que cambian según los botones que pulsemos de los popups.

/cypress-interacciones-con-elementos-popups-lab/index.html

```
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <meta charset="UTF-8">  
    <meta http-equiv="X-UA-Compatible" content="IE=edge">  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <title>Popups</title>  
    <link rel="stylesheet" href="style.css">  
  </head>  
  <body>  
    <div class="center">  
      <div id="js-popups">  
        <button type="button" id="btn-alert">Mostrar un Alert</button>  
        <button type="button" id="btn-confirm">Quitar el mensaje</button>  
        <p id="mensaje-confirm">Mensaje: <span id="confirm-nombre">The Marathon Continues</span></p>  
        <button type="button" id="btn-prompt">Mostrar tu nombre</button>  
        <p id="mensaje-prompt">Nombre: <span id="prompt-nombre"></span></p>  
      </div>  
    </div>  
  
    <script src="app.js"></script>  
  </body>  
</html>
```

Ahora creamos el archivo de estilos **style.css** que se está importando en la página html.

/cypress-interacciones-con-elementos-popups-lab/style.css

```
body {  
    height: 100vh;  
    width: 100vw;  
    background-color: #242424;  
}  
  
.center {  
    height: 100%;  
    display: flex;  
    justify-content: center;  
    align-items: center;  
}  
  
button {  
    background-color: black;  
    width: 100%;  
    color: white;  
    border: 1px solid white;  
    border-radius: 5px;  
    padding: 10px;  
    margin: 20px 0;  
    display: block;  
}  
  
button:hover {  
    border: 1px solid #999999;  
    color: #999999;  
}  
  
p {  
    color: white;  
}
```

Y también creamos el archivo **app.js** donde pondremos la lógica de los botones.

/cypress-interacciones-con-elementos-popups-lab/app.js

```
document.getElementById('btn-alert').addEventListener('click', () => {
  alert('Hola mundo!!!');
})

document.getElementById('btn-confirm').addEventListener('click', () => {
  const borrarMsg = confirm('¿Quieres borrar el mensaje?');
  if (borrarMsg) {
    document.getElementById('confirm-nombre').innerText = '';
  }
})

document.getElementById('btn-prompt').addEventListener('click', () => {
  const nombre = prompt('Introduce tu nombre:');
  console.log({nombre})
  if (nombre) {
    document.getElementById('prompt-nombre').innerText = nombre;
  }
})
```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Además, en otra terminal, vamos a lanzar el otro script para poder acceder con Cypress a la web.

```
$ npm start
```

Como nos indica en la terminal, podemos acceder a la web en <http://localhost:8080/>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **popups.spec.js** dentro de la carpeta **integration**.

Dentro de este archivo empezamos por crear el bloque **describe** y añadir en el **beforeEach** la instrucción para navegar a la página que hemos creado.

/cypress-interacciones-con-elementos-popups-lab/cypress/integration/popups.spec.js

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

describe('Alerts', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })
})
```

Vamos a empezar por comprobar que al pulsar el botón que muestra el popup del alert, muestra el texto correcto.

/cypress-interacciones-con-elementos-popups-lab/cypress/integration/popups.spec.js

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

describe('Alerts', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar el alert con el Hola mundo', () => {
    })
})
```

Empezamos buscando el botón por su identificador para pulsar sobre él. A continuación, vamos a decirle a Cypress que tiene que ponerse a escuchar el evento **window:alert** para ejecutar a continuación una función de callback en la que vamos a recibir el texto de este popup.

/cypress-interacciones-con-elementos-popups-lab/cypress/integration/popups.spec.js

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

describe('Alerts', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar el alert con el Hola mundo', () => {
    cy.get('#btn-alert')
      .click();

    cy.on('window:alert', (textoAlert) => {
      })
    })
})
```

Dentro de la función de callback es donde pondremos la aserción para comprobar que el texto es el que esperamos.

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

describe('Alerts', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar el alert con el Hola mundo', () => {
    cy.get('#btn-alert')
      .click();

    cy.on('window:alert', (textoAlert) => {
      expect(textoAlert).to.equal('Hola mundo!!!')
    })
  })
})
```

El siguiente caso de prueba es para comprobar que si aceptamos el popup de tipo confirm, el mensaje que se está mostrando justo debajo, desaparece.

Empezamos creando el test, y pulsando sobre el botón que muestra este popup.

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

describe('Alerts', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar el alert con el Hola mundo', () => {
    cy.get('#btn-alert')
      .click();

    cy.on('window:alert', (textoAlert) => {
      expect(textoAlert).to.equal('Hola mundo!!!')
    })
  })

  it('debería quitar el mensaje si aceptamos el confirm', () => {
    cy.get('#btn-confirm')
      .click();

    })
})
```

Una vez que tenemos la parte del botón, vamos a añadir el evento **window:confirm** y le vamos a

pasar la función de callback en la que vamos a devolver un **true** para indicar que se tiene que aceptar el popup, es decir, que se tiene que pulsar sobre el botón de **Aceptar**.

/cypress-interacciones-con-elementos-popups-lab/cypress/integration/popups.spec.js

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

describe('Alerts', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar el alert con el Hola mundo', () => {
    cy.get('#btn-alert')
      .click();

    cy.on('window:alert', (textoAlert) => {
      expect(textoAlert).to.equal('Hola mundo!!!')
    })
  })

  it('debería quitar el mensaje si aceptamos el confirm', () => {
    cy.get('#btn-confirm')
      .click();

    cy.on('window:confirm', () => true);
  })
})
```

Para terminar con este test, ahora hay que comprobar que el mensaje no contiene el texto que antes estaba mostrando. Para ello obtenemos el párrafo por su identificador y utilizamos alguna de las aserciones que nos sirven para este caso.

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

describe('Alerts', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar el alert con el Hola mundo', () => {
    cy.get('#btn-alert')
      .click();

    cy.on('window:alert', (textoAlert) => {
      expect(textoAlert).to.equal('Hola mundo!!!')
    })
  })

  it('debería quitar el mensaje si aceptamos el confirm', () => {
    cy.get('#btn-confirm')
      .click();

    cy.on('window:confirm', () => true);
    cy.get('#confirm-nombre')
      .should('not.have.text', 'The Marathon Continues');
  })
})
```

Con esto ya tendríamos este test. Debería de pasarse correctamente. Podemos comprobar también que si cambiamos la aserción, el test falla.

El siguiente caso de prueba es justo el contrario al anterior. Tenemos que comprobar que si se pulsa sobre el botón **Cancelar** del popup, el texto del mensaje no desaparece.

Para indicar que hay que rechazar el popup, esta vez hay que devolver un **false** en la función de callback que se le pasa al evento.

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

describe('Alerts', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar el alert con el Hola mundo', () => {
    cy.get('#btn-alert')
      .click();

    cy.on('window:alert', (textoAlert) => {
      expect(textoAlert).to.equal('Hola mundo!!!')
    })
  })

  it('debería quitar el mensaje si aceptamos el confirm', () => {
    cy.get('#btn-confirm')
      .click();

    cy.on('window:confirm', () => true);
    cy.get('#confirm-nombre')
      .should('not.have.text', 'The Marathon Continues');
  })

  it('no debería quitar el mensaje si cancelamos el confirm', () => {
    cy.get('#btn-confirm')
      .click();

    cy.on('window:confirm', () => false);
    cy.get('#confirm-nombre')
      .should('have.text', 'The Marathon Continues');
  })
})
```

Con esto ya tendríamos la parte del mensaje probada.

Ahora toca ponernos a testear el último popup, el prompt, que permite introducir al usuario un valor y nos lo devuelve en caso de que se pulse sobre el botón **Aceptar**. En el caso contrario se devuelve un **null**.

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

describe('Alerts', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar el alert con el Hola mundo', () => {
    cy.get('#btn-alert')
      .click();

    cy.on('window:alert', (textoAlert) => {
      expect(textoAlert).to.equal('Hola mundo!!!')
    })
  })

  it('debería quitar el mensaje si aceptamos el confirm', () => {
    cy.get('#btn-confirm')
      .click();

    cy.on('window:confirm', () => true);
    cy.get('#confirm-nombre')
      .should('not.have.text', 'The Marathon Continues');
  })

  it('no debería quitar el mensaje si cancelamos el confirm', () => {
    cy.get('#btn-confirm')
      .click();

    cy.on('window:confirm', () => false);
    cy.get('#confirm-nombre')
      .should('have.text', 'The Marathon Continues');
  })

  it('debería escribir el nombre si aceptamos el prompt', () => {
  })
})
```

Este caso es distinto a los anteriores, porque no hay un evento que nos permita detectar cuando se va a abrir el popup del tipo **prompt**, y por tanto no podremos indicarle que texto queremos escribir para retornarlo. Por tanto, la solución pasa por crear un **stub** para indicarle que cuando ejecute la función **prompt** del objeto **window**, nos devuelva un valor dado por nosotros.

Para ello, tenemos que poder acceder a la ventana usando el **cy.window** que nos da el objeto **window** en la instrucción **then**.

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

describe('Alerts', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar el alert con el Hola mundo', () => {
    cy.get('#btn-alert')
      .click();

    cy.on('window:alert', (textoAlert) => {
      expect(textoAlert).to.equal('Hola mundo!!!')
    })
  })

  it('debería quitar el mensaje si aceptamos el confirm', () => {
    cy.get('#btn-confirm')
      .click();

    cy.on('window:confirm', () => true);
    cy.get('#confirm-nombre')
      .should('not.have.text', 'The Marathon Continues');
  })

  it('no debería quitar el mensaje si cancelamos el confirm', () => {
    cy.get('#btn-confirm')
      .click();

    cy.on('window:confirm', () => false);
    cy.get('#confirm-nombre')
      .should('have.text', 'The Marathon Continues');
  })

  it('debería escribir el nombre si aceptamos el prompt', () => {
    cy.window()
      .then(windowObj => {
        });

    })
  })
})
```

Ahora ya podemos crear el stub para indicarle que devuelva el valor **Octavia Blake**. Justo a continuación de crear el stub, vamos a provocar que el popup se lance pulsando sobre el último botón.

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

describe('Alerts', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar el alert con el Hola mundo', () => {
    cy.get('#btn-alert')
      .click();

    cy.on('window:alert', (textoAlert) => {
      expect(textoAlert).to.equal('Hola mundo!!!')
    })
  })

  it('debería quitar el mensaje si aceptamos el confirm', () => {
    cy.get('#btn-confirm')
      .click();

    cy.on('window:confirm', () => true);
    cy.get('#confirm-nombre')
      .should('not.have.text', 'The Marathon Continues');
  })

  it('no debería quitar el mensaje si cancelamos el confirm', () => {
    cy.get('#btn-confirm')
      .click();

    cy.on('window:confirm', () => false);
    cy.get('#confirm-nombre')
      .should('have.text', 'The Marathon Continues');
  })

  it('debería escribir el nombre si aceptamos el prompt', () => {
    cy.window()
      .then(windowObj => {
        cy.stub(windowObj, 'prompt').returns('Octavia Blake');
      });

    cy.get('#btn-prompt')
      .click();
  })
})
```

Ahora solo nos queda comprobar que el nombre que devuelve el stub es el que aparece en nuestra aplicación.

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

describe('Alerts', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar el alert con el Hola mundo', () => {
    cy.get('#btn-alert')
      .click();

    cy.on('window:alert', (textoAlert) => {
      expect(textoAlert).to.equal('Hola mundo!!!')
    })
  })

  it('debería quitar el mensaje si aceptamos el confirm', () => {
    cy.get('#btn-confirm')
      .click();

    cy.on('window:confirm', () => true);
    cy.get('#confirm-nombre')
      .should('not.have.text', 'The Marathon Continues');
  })

  it('no debería quitar el mensaje si cancelamos el confirm', () => {
    cy.get('#btn-confirm')
      .click();

    cy.on('window:confirm', () => false);
    cy.get('#confirm-nombre')
      .should('have.text', 'The Marathon Continues');
  })

  it('debería escribir el nombre si aceptamos el prompt', () => {
    cy.window()
      .then(win => {
        cy.stub(win, 'prompt').returns('Octavia Blake');
      });

    cy.get('#btn-prompt')
      .click();

    cy.get('#prompt-nombre')
      .should('have.text', 'Octavia Blake')
  })
})
```

Vamos a terminar este laboratorio con este mismo ejemplo, pero simulando que cancelamos el popup del prompt, y por lo tanto no se muestra ningún nombre en el párrafo.

Ahora en lugar de hacer que el stub retorne el nombre, vamos a hacer que retorne un **null**. El resto del test es parecido.

/cypress-interacciones-con-elementos-popups-lab/cypress/integration/popups.spec.js

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

describe('Alerts', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080')
  })

  it('debería mostrar el alert con el Hola mundo', () => {
    cy.get('#btn-alert')
      .click();

    cy.on('window:alert', (textoAlert) => {
      expect(textoAlert).to.equal('Hola mundo!!!')
    })
  })

  it('debería quitar el mensaje si aceptamos el confirm', () => {
    cy.get('#btn-confirm')
      .click();

    cy.on('window:confirm', () => true);
    cy.get('#confirm-nombre')
      .should('not.have.text', 'The Marathon Continues');
  })

  it('no debería quitar el mensaje si cancelamos el confirm', () => {
    cy.get('#btn-confirm')
      .click();

    cy.on('window:confirm', () => false);
    cy.get('#confirm-nombre')
      .should('have.text', 'The Marathon Continues');
  })

  it('debería escribir el nombre si aceptamos el prompt', () => {
    cy.window()
      .then(win => {
        cy.stub(win, 'prompt').returns('Octavia Blake');
      });

    cy.get('#btn-prompt')
      .click();
  })
})
```

```

    cy.get('#prompt-nombre')
      .should('have.text', 'Octavia Blake')
  })

it('no debería escribir el nombre si cancelamos el prompt', () => {
  cy.window()
    .then(win => {
      cy.stub(win, 'prompt').returns(null);
    });

  cy.get('#btn-prompt')
    .click();

  cy.get('#prompt-nombre')
    .should('have.text', '');
})
})
}

```

Ya tenemos testeados los distintos casos con los que nos encontramos en la aplicación de ejemplo. Todos ellos deberían de pasarse correctamente.

## 16.19. Screenshots

Con Cypress podemos hacer capturas de pantalla que luego nos pueden servir para ver cual es el motivo por el que un test está fallando, o para guardar alguna captura como prueba visual de que se muestra en la página.

Para sacar un screenshot con Cypress solo hay que llamar a la función **screenshot** pasandole el nombre con el que queremos guardar la imagen.

```
cy.screenshot('nombre-imagen');
```

Todos los pantallazos se van a guardar en una carpeta **screenshots** automáticamente que se genera sola.

Al sacar un screenshot por defecto se hace de toda la ventana, pero esto podemos cambiarlo llamando al comando **screenshot** desde el elemento del cual queremos obtener la imagen.

```
cy.get('#miPerfil')
  .screenshot('perfil');
```

Para evitar sacar información sensible en los pantallazos, como puede ser un número de cuenta, el dni de una persona, el email... Cypress da la opción de ocultar esta información utilizando la opción de **blackout**. Esta es una propiedad que se pone en un objeto que le pasamos a la instrucción **screenshot** como parámetro. Esta propiedad recibe como valor un array con los selectores que queremos ocultar, y Cypress pondrá una caja de color negro sobre estos elementos.

```
cy.get('#miPerfil')
  .screenshot('perfil', {
    blackout: ['#dni', '#numero-cuenta']
 });
```

Cuando lanzamos los tests con el comando de **cypress run**, por defecto se obtiene un pantallazo cada vez que alguno de los tests falla, para poder ver que se estaba mostrando en el instante del error. Esto no ocurre al ejecutar los tests desde la interfaz de Cypress ya que al ejecutar el test, estamos viendo que se muestra en la página y el posible error.

Esta opción se puede deshabilitar desde la configuración de Cypress (archivo **cypress.json**) añadiendo la opción de **screenshotOnRunFailure: false**.

## 16.20. Lab: Screenshots

En este laboratorio vamos a ver como podemos realizar capturas de pantalla y como ocultar datos que pueden ser sensibles.

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-screenshots-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-screenshots-lab
$ cd cypress-screenshots-lab
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests
- http-server: servidor de desarrollo local

```
$ npm install --save-dev cypress http-server
```

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

```
{  
  "name": "cypress-screenshots-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "http-server",  
    "cy:open": "cypress open"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.6.0",  
    "http-server": "^0.12.3"  
  }  
}
```

El siguiente paso es crear un archivo **index.html** donde vamos a poner los datos de los cuales queremos sacar un pantallazo ocultando aquellos que son sensibles.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Screenshot</title>
  <style>
    body {
      height: 100vh;
      width: 100vw;
      background-color: #242424;
    }

    .center {
      height: 100%;
      display: flex;
      justify-content: center;
      align-items: center;
    }

    #dashboard-screenshot {
      color: rgb(184, 184, 184);
    }
  </style>
</head>
<body>
  <div class="center">
    <div id="dashboard-screenshot">
      <div>
        <h3>Datos de Charly Falco</h3>
        <p>Email: <span id="email">cfalco@gmail.com</span></p>
        <p>DNI: <span id="dni">00000000T</span></p>
        <p>Saldo invertido: <span id="saldo">30000€</span></p>
      </div>
      <ul id="inversiones">
        <li>50% - Hooly, Inc</li>
        <li>30% - Pied Piper, Inc</li>
        <li>20% - Aviato, Inc</li>
      </ul>
    </div>
  </div>
</body>
</html>
```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Además, en otra terminal, vamos a lanzar el otro script para poder acceder con Cypress a la web.

```
$ npm start
```

Como nos indica en la terminal, podemos acceder a la web en <http://localhost:8080/>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **screenshots.spec.js** dentro de la carpeta **integration**.

/cypress-screenshots-lab/cypress/integration/screenshots.spec.js

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

describe('Screenshots', () => {

})
```

Con nuestro primer test vamos a sacar dos pantallazos sobre la página que hemos usado anteriormente en el laboratorio del **Drag and Drop**. Queremos sacar un primer pantallazo antes de arrastrar el elemento y otro después de arrastrarlo. Podemos copiar el test del laboratorio.

/cypress-screenshots-lab/cypress/integration/screenshots.spec.js

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

describe('Screenshots', () => {
  it('sacar un pantallazo antes de arrastrar el elemento y despues de soltarlo', () => {
    cy.visit('http://cookbook.seleniumacademy.com/DragDropDemo.html')

    cy.get('#draggable')
      .trigger('mousedown', { which: 1 })
      .trigger('mousemove', { pageX: 180, pageY: 55 })
      .trigger('mouseup');

    cy.get('#droppable')
      .should('include.text', 'Dropped!');
  })
})
```

Ahora que tenemos el test, vamos a añadir las instrucciones para sacar el pantallazo antes y después. Para ello, usamos el comando de **screenshot** pasandole el nombre con el que queremos guardar la imagen que se saca.

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

describe('Screenshots', () => {
  it('sacar un pantallazo antes de arrastrar el elemento y despues de soltarlo', () => {
    cy.visit('http://cookbook.seleniumacademy.com/DragDropDemo.html')

    cy.screenshot('beforeDrag')

    cy.get('#draggable')
      .trigger('mousedown', { which: 1 })
      .trigger('mousemove', { pageX: 180, pageY: 55 })
      .trigger('mouseup');

    cy.screenshot('afterDrop')

    cy.get('#droppable')
      .should('include.text', 'Dropped!');
  })
})
```

Si ejecutamos este test, podemos ver que se genera automáticamente una carpeta **cypress/screenshots** con las dos imágenes dentro.

Con el siguiente test queremos obtener un pantallazo de un dashboard que muestra los datos de un inversor pero ocultando la información que puede ser sensible como el email, el dni y el saldo invertido. Para este vamos a usar la página web que hemos creado al principio.

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

describe('Screenshots', () => {
  it('sacar un pantallazo antes de arrastrar el elemento y despues de soltarlo', () => {
    cy.visit('http://cookbook.seleniumacademy.com/DragDropDemo.html')

    cy.screenshot('beforeDrag')

    cy.get('#draggable')
      .trigger('mousedown', { which: 1 })
      .trigger('mousemove', { pageX: 180, pageY: 55 })
      .trigger('mouseup');

    cy.screenshot('afterDrop')

    cy.get('#droppable')
      .should('include.text', 'Dropped!');
  })

  it('sacar un pantallazo ocultando los datos sensibles', () => {
    cy.visit('http://localhost:8080')

  })
})
```

Esta vez, queremos solo un pantallazo de un cierto elemento de toda la página web, por tanto vamos a empezar por buscar este elemento por su identificador y vamos a utilizar la función de **screenshot** sobre él.

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

describe('Screenshots', () => {
  it('sacar un pantallazo antes de arrastrar el elemento y despues de soltarlo', () => {
    cy.visit('http://cookbook.seleniumacademy.com/DragDropDemo.html')

    cy.screenshot('beforeDrag')

    cy.get('#draggable')
      .trigger('mousedown', { which: 1 })
      .trigger('mousemove', { pageX: 180, pageY: 55 })
      .trigger('mouseup');

    cy.screenshot('afterDrop')

    cy.get('#droppable')
      .should('include.text', 'Dropped!');
  })

  it('sacar un pantallazo ocultando los datos sensibles', () => {
    cy.visit('http://localhost:8080')

    cy.get('#dashboard-screenshot')
      .screenshot('dashboard');
  })
})
```

El siguiente paso es hacer que no se muestren los datos sensibles, por lo que vamos a pasarle un objeto con la opción de **blackout** a la función de **screenshot**. El blackout espera recibir un array con los selectores de aquellos elementos que no queremos que se muestren en el pantallazo, por lo que le pasamos los selectores de los identificadores del dni, email y saldo.

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

describe('Screenshots', () => {
  it('sacar un pantallazo antes de arrastrar el elemento y despues de soltarlo', () => {
    cy.visit('http://cookbook.seleniumacademy.com/DragDropDemo.html')

    cy.screenshot('beforeDrag')

    cy.get('#draggable')
      .trigger('mousedown', { which: 1 })
      .trigger('mousemove', { pageX: 180, pageY: 55 })
      .trigger('mouseup');

    cy.screenshot('afterDrop')

    cy.get('#droppable')
      .should('include.text', 'Dropped!');
  })

  it('sacar un pantallazo ocultando los datos sensibles', () => {
    cy.visit('http://localhost:8080')

    cy.get('#dashboard-screenshot')
      .screenshot('dashboard', {
        blackout: ['#saldo', '#dni', '#email']
      });
  })
})
```

Al ejecutar el test, veremos que se genera una nueva imagen en la carpeta de **screenshots**. Esta imagen se verá de la siguiente forma:

# Datos de Charly Falco

Email: [REDACTED]

DNI: [REDACTED]

Saldo invertido: [REDACTED]

- 50% - Hooly, Inc
- 30% - Pied Piper, Inc
- 20% - Aviato, Inc

# Capítulo 17. Alias

Los **alias** de Cypress nos van a permitir crear referencias a ciertos elementos con los que queremos interactuar varias veces y así evitar tener que buscarlos en varias ocasiones teniendo código duplicado.

Para crear un alias usamos el comando **as(alias)** justo después de la instrucción que nos da el elemento al cual queremos la referencia.

```
cy.get('#mi-lista').as('lista');
cy.get('#mi-lista').first().as('primer-elemento-de-la-lista');
```

Una vez creados los alias, tenemos que acceder a ellos para poder usarlos. Solo hay que utilizar el comando **get** y pasarle el nombre del alias al que queremos acceder precedido del símbolo de @.

```
cy.get('lista'); // Obtenemos una etiqueta con el nombre de lista
cy.get('@lista'); // Obtenemos la referencia a la lista
```

Ahora ya podemos utilizar el resto de comandos que hemos visto hasta ahora para interactuar con estos elementos que nos devuelven los alias.

```
cy.get('@lista')
  .should(...)
```

Algunas veces necesitaremos acceder al propio objeto al que hace referencia un alias, y para ello podemos usar el comando **then** y pasarle una función de callback donde recibir el elemento como parámetro.

```
cy.get('@lista')
  .then($lista => { ... })
```

Los alias también nos permiten acceder a datos o elementos que se mencionan en los hooks que se ejecutan antes de los tests para evitar repetir el mismo código en todos los bloques **it**.

Otra de las cosas que podemos hacer con los alias es hacer que el test espere hasta que aquello a lo que se hace referencia se encuentre disponible, por ejemplo para cuando se hace una petición a una API y esta tarda en respondernos.

```
cy.wait('@peticionGetAPI');
```

# Capítulo 18. Fixtures

En la mayor parte de los tests vamos a necesitar utilizar una serie de datos, como datos de usuarios para hacer un login o un registro, un listado de objetos para guardarlos en nuestra aplicación, una imagen para subirla al servidor...

Cypress nos da la funcionalidad de los **fixtures** para leer estos datos que necesitaremos de archivos JSON, o directamente los propios archivos (imagen, video...) a utilizar y así no tener que tenerlos hardcodeados por los diferentes tests pudiéndolos reutilizar en cualquier momento.

Estos archivos se guardan dentro de la carpeta **cypress/fixtures**.

Para leer estos datos dentro de un test utilizamos la función **fixture** y le pasamos el nombre del archivo que queremos cargar como primer parámetro. Podemos pasarle como segundo parámetro el tipo de encoding a utilizar al leer los archivos.

Los tipos de encoding que soporta Cypress son: ascii, base64, binary, hex, utf8, utf16le, ucs2 y latin1.

Cypress puede inferir sin problema el tipo de encoding a partir de la extensión del archivo, por lo que no tendremos que preocuparnos por ello.

```
cy.fixture('misDatos.json')
  .then(datos => {})
```

## 18.1. Lab: Fixtures

En este laboratorio vamos a ver como utilizar en nuestro test unos datos que ya tenemos guardados en un archivo JSON.

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-fixtures-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a seguir los siguientes comandos:

```
$ mkdir cypress-fixtures-lab
$ cd cypress-fixtures-lab
$ npm init -y
```

Una vez lanzados, deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests
- express: framework para crear nuestro servidor

```
$ npm install --save-dev cypress  
$ npm install --save express
```

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

*/cypress-fixtures-lab/package.json*

```
{  
  "name": "cypress-fixtures-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "node app.js",  
    "cy:open": "cypress open"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.6.0"  
  },  
  "dependencies": {  
    "express": "^4.17.1"  
  }  
}
```

Ahora vamos a crear los archivos de las páginas de inicio, de login y el archivo con nuestro servidor.

En la página de inicio solo mostraremos un mensaje de bienvenida.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Fixtures</title>
  <style>
    body {
      height: 100vh;
      width: 100vw;
      background-color: #242424;
      color: white;
      text-align: center;
    }
  </style>
</head>
<body>
  <div class="center">
    <h1>Bienvenido a la página</h1>
  </div>
</body>
</html>
```

En la página de login vamos a crear un formulario para mandar el email y la contraseña para ver si podemos entrar a la aplicación o no.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Login Fixtures</title>
  <style>
    body {
      height: 100vh;
      width: 100vw;
      background-color: #242424;
    }

    .center {
      height: 100%;
      display: flex;
      justify-content: center;
      align-items: center;
      color: white;
    }

    form input {
      margin-bottom: 15px;
    }
  </style>
</head>
<body>
  <div class="center">
    <form method="POST">
      <div>
        <label for="email">Email</label>
        <input type="email" id="email" name="email">
      </div>
      <div>
        <label for="password">Password</label>
        <input type="password" id="password" name="password">
      </div>
      <button type="submit">Sign In</button>
    </form>
  </div>
</body>
</html>
```

En el siguiente archivo vamos a crear nuestro backend con express para recibir las distintas peticiones GET y POST y devolver los archivos que nos están pidiendo desde el cliente.

```
/cypress-fixtures-lab/app.js
```

```
const http = require('http');
const express = require('express');
const path = require('path');

const app = express();

const usuariosValidos = [
  {email: 'cfalco@gmail.com', password: 'cfalco'},
];

app.use(express.urlencoded({extended: false}));

app.get('/login', (req, res, next) => {
  res.sendFile(path.join(__dirname, 'login.html'));
});

app.post('/login', (req, res, next) => {
  const params = req.body;

  const usuario = usuariosValidos.find(u => {
    return ((u.password === params.password) && (u.email === params.email));
  });

  if (usuario) {
    res.redirect('/');
  } else {
    res.redirect('/login');
  }
});

app.get('/', (req, res, next) => {
  res.sendFile(path.join(__dirname, 'index.html'));
});

const server = http.createServer(app);
server.listen('8080');
```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Además, en otra terminal, vamos a lanzar el otro script para poder acceder con Cypress a la web.

```
$ npm start
```

Nuestro servidor está escuchando las peticiones del cliente en <http://localhost:8080>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **fixtures.spec.js** dentro de la carpeta **integration**. Dentro de este archivo empezamos por crear el bloque describe y la navegación a la página principal (**/login**) dentro del beforeEach.

*/cypress-fixtures-lab/cypress/integration/fixtures.spec.js*

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

describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080/login');
  })

})
```

Como en este laboratorio vamos a utilizar unos datos para loguearnos, antes de empezar a crear nuestros casos de prueba, vamos a definir estos datos en un archivo **datosUsuarios.json** dentro de la carpeta **fixtures**.

Dentro vamos a definir dos usuarios, uno con el que podremos loguearnos y otro con el que no.

*/cypress-fixtures-lab/cypress/fixtures/datosUsuarios.json*

```
{
  "usuarioOk": {
    "email": "cfalco@gmail.com",
    "password": "cfalco"
  },
  "usuarioKo": {
    "email": "mike.koz@gmail.com",
    "password": "1234"
  }
}
```

Vamos a crear un test en el que vamos a loguearnos con un usuario válido para ver si conseguimos llegar a la página de inicio.

Utilizamos la función de **fixture** para cargar el archivo que contiene los datos de los usuarios y que hemos creado anteriormente.

/cypress-fixtures-lab/cypress/integration/fixtures.spec.js

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

describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080/login');
  })

  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {
    cy.fixture('datosUsuarios.json')
  })
})
```

Los datos que nos devuelve la función del **fixture** los vamos a encontrar en la función de callback del **then**, y ahí mismo vamos a aprovechar para obtener el **usuarioOk** utilizando la desestructuración (operador spread).

Una vez que tenemos los datos, podemos llenar los campos del formulario y enviarlo.

/cypress-fixtures-lab/cypress/integration/fixtures.spec.js

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

describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080/login');
  })

  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {
    cy.fixture('datosUsuarios.json')
      .then(({ usuarioOk }) => {
        cy.get('input#email')
          .type(usuarioOk.email);
        cy.get('input#password')
          .type(usuarioOk.password);
        cy.get('button')
          .click();
      })
  })
})
```

Si todo va bien, como debería de ocurrir, ahora deberíamos de estar en la página de inicio, y por tanto vamos a comprobar que el path de la URL es el raíz, y además para asegurarnos podemos comprobar también que aparece el título de la página dándonos la bienvenida.

Para obtener el path, usaremos el comando de **location** al que le vamos a pasar como parámetro, cual es el dato que queremos obtener, que es el **pathname**.

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

describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080/login');
  })

  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {
    cy.fixture('datosUsuarios.json')
      .then(({ usuarioOk }) => {
        cy.get('input#email')
          .type(usuarioOk.email);
        cy.get('input#password')
          .type(usuarioOk.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/');

        cy.get('h1')
          .should('have.text', 'Bienvenido a la página');
      })
  })
})
```

Ya tenemos el primer test, y debería de pasarse correctamente. Estaría bien comprobar que si cambiamos alguna aserción el test falla, de esta forma nos aseguramos que no es un falso positivo.

Ahora vamos a crear el segundo caso de prueba, en el que usaremos los datos del usuario que no es válido (**usuarioKo**) para comprobar que vuelve a la página del login y no consigue entrar al inicio.

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

describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080/login');
  })

  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {
    cy.fixture('datosUsuarios.json')
      .then(({ usuarioOk }) => {
        cy.get('input#email')
          .type(usuarioOk.email);
        cy.get('input#password')
          .type(usuarioOk.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/');

        cy.get('h1')
          .should('have.text', 'Bienvenido a la página');
      })
  })

  it('debería volver a la página de login si se loguea con un usuario no válido', () => {
  })
})
```

Los primeros pasos del test son igual que antes, hay que leer el archivo de fixtures, pero esta vez, en la desestructuración nos vamos a quedar con los datos del otro usuario (**usuarioKo**).

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

describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080/login');
  })

  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {
    cy.fixture('datosUsuarios.json')
      .then(({ usuarioOk }) => {
        cy.get('input#email')
          .type(usuarioOk.email);
        cy.get('input#password')
          .type(usuarioOk.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/');

        cy.get('h1')
          .should('have.text', 'Bienvenido a la página');
      })
  })

  it('debería volver a la página de login si se loguea con un usuario no válido', () => {
    cy.fixture('datosUsuarios.json')
      .then(({ usuarioKo }) => {
        })
      })
  })
})
```

Ahora rellenamos con estos datos el formulario y lo enviamos pulsando sobre el botón.

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

describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080/login');
  })

  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {
    cy.fixture('datosUsuarios.json')
      .then(({ usuarioOk }) => {
        cy.get('input#email')
          .type(usuarioOk.email);
        cy.get('input#password')
          .type(usuarioOk.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/');

        cy.get('h1')
          .should('have.text', 'Bienvenido a la página');
      })
  })

  it('debería volver a la página de login si se loguea con un usuario no válido', () => {
    cy.fixture('datosUsuarios.json')
      .then(({ usuarioKo }) => {
        cy.get('input#email')
          .type(usuarioKo.email);
        cy.get('input#password')
          .type(usuarioKo.password);
        cy.get('button')
          .click();

      })
  })
})
```

Y ya podemos comprobar que hemos vuelto a la página de login. Para ello, podemos mirar que esta vez el pathname es **/login**, y que el formulario existe en la página.

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

describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080/login');
  })

  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {
    cy.fixture('datosUsuarios.json')
      .then(({ usuarioOk }) => {
        cy.get('input#email')
          .type(usuarioOk.email);
        cy.get('input#password')
          .type(usuarioOk.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/');

        cy.get('h1')
          .should('have.text', 'Bienvenido a la página');
      })
  })

  it('debería volver a la página de login si se loguea con un usuario no válido', () => {
    cy.fixture('datosUsuarios.json')
      .then(({ usuarioKo }) => {
        cy.get('input#email')
          .type(usuarioKo.email);
        cy.get('input#password')
          .type(usuarioKo.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/login');

        cy.get('form')
          .should('exist');
      })
  })
})
```

Si nos fijamos, en ambos tests leemos el archivo de **datosUsuarios.json** para acceder a los datos de los usuarios. Esta tarea la podemos realizar en el **beforeEach** al igual que la navegación a la página de login.

Vamos a poner esta instrucción en el hook, y para poder acceder a los datos que lee, vamos a

añadirle un alias con el comando **as** y pasándole el nombre del alias con el cual pediremos estos datos en los tests.

/cypress/fixtures-lab/cypress/integration/fixtures.spec.js

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

describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080/login');
    cy.fixture('datosUsuarios.json').as('datosUsuarios');
  })

  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {
    cy.fixture('datosUsuarios.json')
      .then(({ usuarioOk }) => {
        cy.get('input#email')
          .type(usuarioOk.email);
        cy.get('input#password')
          .type(usuarioOk.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/');

        cy.get('h1')
          .should('have.text', 'Bienvenido a la página');
      })
  })

  it('debería volver a la página de login si se loguea con un usuario no válido', () => {
    cy.fixture('datosUsuarios.json')
      .then(({ usuarioKo }) => {
        cy.get('input#email')
          .type(usuarioKo.email);
        cy.get('input#password')
          .type(usuarioKo.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/login');

        cy.get('form')
          .should('exist');
      })
  })
})
```

Ahora solo tenemos que cambiar la llamada al fixture de los tests por un **get(@datosUsuarios)**

para poder utilizar el then y acceder a los datos que queramos del archivo JSON.

/cypress-fixtures-lab/cypress/integration/fixtures.spec.js

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

describe('Fixtures', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080/login');
    cy.fixture('datosUsuarios.json').as('datosUsuarios');
  })

  it('debería ir a la pantalla de inicio si se loguea con un usuario válido', () => {
    cy.get('@datosUsuarios')
      .then(({ usuarioOk }) => {
        cy.get('input#email')
          .type(usuarioOk.email);
        cy.get('input#password')
          .type(usuarioOk.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/');
      })
      .then(() => {
        cy.get('h1')
          .should('have.text', 'Bienvenido a la página');
      })
  })

  it('debería volver a la página de login si se loguea con un usuario no válido', () => {
    cy.get('@datosUsuarios')
      .then(({ usuarioKo }) => {
        cy.get('input#email')
          .type(usuarioKo.email);
        cy.get('input#password')
          .type(usuarioKo.password);
        cy.get('button')
          .click();

        cy.location('pathname')
          .should('eq', '/login');

        cy.get('form')
          .should('exist');
      })
  })
})
```

Si probamos a lanzar los tests, estos deberían de seguir pasándose correctamente.

# Capítulo 19. Mocking

Aunque utilicemos Cypress para crear tests end-to-end, hay algunos casos en los que necesitaremos crear mocks de alguna función o una funcionalidad que no nos devuelve algún resultado predecible para poder controlar los tests y que no fallen unas veces y otras salgan como correctos.

A continuación vamos a ver los métodos de **spy**, **stub** e **intercept** que nos ayudan con estas tareas.

## 19.1. Spy

El método **spy** nos permite envolver una función con el para poder registrar las llamadas a esta y que parámetros se utilizan en cada llamada de tal forma que podamos comprobar que a una función se le llaman las veces que son necesarias y ninguna más.

A este método le podemos pasar como parámetros el objeto donde se encuentra la función a envolver y el nombre de esta.

```
cy.spy(serie, 'verCapitulo');
```

## 19.2. Stub

Con la función de **stub** podemos reemplazar una función, tener un control de su uso al igual que podemos hacer son el método **spy** y controlar su funcionamiento.

Al stub le podemos pasar como parámetros el objeto en el que se encuentra la función a mockear, y un string con el nombre de dicha función. De esta forma podemos utilizar aserciones como:

- Se ha llamado X veces
- Se ha llamado con X parámetros
- ...

```
cy.stub(serie, 'verCapitulo');
```

Si a esta función se le pasa un tercer parámetro, estaríamos reemplazando la función mockeada por aquella que se le envía como parámetro.

```
cy.stub(serie, 'verCapitulo', () => {  
  return 5;  
});
```

Cuando vamos a mockear métodos del objeto **window**, deberíamos de realizar ese cambio antes de que se cargue la página para que funcionen los mocks como esperamos. Para ello, se suele utilizar la propiedad **onBeforeLoad** de las opciones del método **visit** en la que pondremos nuestras instrucciones de stub.

```
cy.visit('http://mi-pagina-web.com', {
  onBeforeLoad: (win) => {
    cy.stub(win, 'prompt').returns('Texto escrito en el popup del navegador.');
  }
});
```

## 19.3. Lab: Spy y Stub

En este laboratorio vamos a ver como utilizar los métodos de spy y stub para comprobar que:

- Cuando pedimos la ubicación en la que nos encontramos solo se llama una vez a la función que pinta la información.
- Cuando hacemos que la función que nos devuelve la ubicación nos de las coordenadas de Fargo, la ciudad que se muestra es la misma.
- Cuando hacemos que la función que nos devuelve la ubicación nos de las coordenadas de Harlan, la ciudad que se muestra es la misma.
- Cuando hacemos que la función que nos devuelve la ubicación nos de unas coordenadas vacías, la ciudad que se muestra es "Una ciudad cualquiera".

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-mocking-spy-y-stub-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-mocking-spy-y-stub-lab
$ cd cypress-mocking-spy-y-stub-lab
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests
- express: framework para crear webs

```
$ npm install --save-dev cypress
$ npm install --save express
```

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

```
{  
  "name": "cypress-mocking-spy-y-stub-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "cy:open": "cypress open",  
    "start": "node server.js"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.8.0"  
  },  
  "dependencies": {  
    "express": "^4.17.1"  
  }  
}
```

El siguiente paso es crear un archivo **index.html** donde vamos a poner el código para mostrar la ciudad en la que nos encontramos en función de nuestra ubicación.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Spy y Stub</title>
</head>
<body>
  <div>
    <button type="button" id="btn-ubicacion">¿Dónde estoy?</button>
    <p>Ubicación: <span id="ciudad"></span></p>
  </div>

  <script>
    document.getElementById('btn-ubicacion').addEventListener('click', () => {
      navigator.geolocation.getCurrentPosition(position => {
        const { coords } = position;
        const { longitude, latitude } = coords;

        fetch(`http://localhost:8080/get-city?lat=${latitude}&lon=${longitude}`)
          .then(resp => resp.json())
          .then(({ciudad}) => {
            document.getElementById('ciudad').innerText = ciudad;
          })
      })
    })
  </script>
</body>
</html>
```

Ahora añadimos el archivo **server.js** con la API que nos devolverá el nombre de una ciudad dadas sus coordenadas.

```
const http = require('http');
const express = require('express');
const path = require('path');

const app = express();

app.get('/get-city', (req, res, next) => {
  const { lat, lon } = req.query;
  if (lat === '46.874396' && lon === '-96.835556') {
    res.json({ciudad: 'Fargo'});
  } else if (lat === '36.848044' && lon === '-83.320589') {
    res.json({ciudad: 'Harlan'});
  } else if (lat === '52.485973' && lon === '-1.890715') {
    res.json({ciudad: 'Birmingham'});
  } else if (lat === '35.110816' && lon === '-106.668173') {
    res.json({ciudad: 'Albuquerque'});
  } else {
    res.json({ciudad: 'Una ciudad cualquiera...'});
  }
})

app.get('/', (req, res, next) => {
  res.sendFile(path.join(__dirname, 'index.html'));
});

const server = http.createServer(app);
server.listen('8080');
```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

En otra terminal distinta a la anterior vamos a lanzar el comando que levanta el servidor de express con la API.

```
$ npm start
```

Podemos entrar a la página de la aplicación desde <http://localhost:8080>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **geolocalizacion.spec.js** dentro de la carpeta **integration**.

Empezamos por crear el bloque **describe** con nuestro primer test.

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

describe('Geolocalización', () => {
  it('debería de llamar una sola vez a paintCity cuando se pulsa el botón', () => {
    })
})
```

Vamos a empezar por indicar a que página tiene que entrar, y el comando **visit**, a parte de la URL también recibe un segundo parámetro de opciones donde podemos añadir una propiedad **onBeforeLoad** cuyo valor es una función en la que recibimos el objeto de la ventana y que podemos utilizar para crear un **spy** sobre el método **fetch**.

Para ello, le vamos a pasar al método **spy** el objeto de la ventana y un string con el nombre de la función que queremos espiar. También le añadiremos un alias para poder realizar las comprobaciones necesarias más tarde.

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

describe('Geolocalización', () => {
  it('debería de llamar una sola vez a paintCity cuando se pulsa el botón', () => {
    cy.visit('http://localhost:8080/', {
      onBeforeLoad: (win) => {
        cy.spy(win, 'fetch').as('spyFetch');
      }
    });
  })
})
```

Ahora que tenemos creado el spy, ya podemos pulsar sobre el botón que pide nuestra ubicación al navegador, y justo después buscaremos el spy por el alias que le habíamos dado para comprobar que la función del **fetch** solo se ha llamado una sola vez.

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

describe('Geolocalización', () => {
  it('debería de llamar una sola vez a paintCity cuando se pulsa el botón', () => {
    cy.visit('http://localhost:8080/', {
      onBeforeLoad: (win) => {
        cy.spy(win, 'fetch').as('spyFetch');
      }
    });

    cy.get('#btn-ubicacion')
      .click();

    cy.get('@spyFetch')
      .should('be.calledOnce')
  })
})
```

Ya tenemos nuestro primer test.

Vamos a por el siguiente con el que en lugar de crear un spy, vamos a crear un stub sobre la función que pide la ubicación de nuestro dispositivo. Al método **stub** le pasamos como primer parámetro el objeto donde se encuentra la función de la que queremos crear el stub, y como siguiente parámetro la función.

Después, sobre el objeto que devuelve este stub, vamos a añadir la función **callsFake** para indicarle que cuando vaya a ejecutar el callback que recibe el **getCurrentPosition**, los valores que queremos que nos devuelva son los que se han definido previamente.

Por último, también le vamos a añadir un alias.

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

describe('Geolocalización', () => {
  it('debería de llamar una sola vez a paintCity cuando se pulsa el botón', () => {
    cy.visit('http://localhost:8080/', {
      onBeforeLoad: (win) => {
        cy.spy(win, 'fetch').as('spyFetch');
      }
    });

    cy.get('#btn-ubicacion')
      .click();

    cy.get('@spyFetch')
      .should('be.calledOnce')
  })

  it('debería mostrar "Fargo" cuando se le pasan sus coordenadas', () => {
    const coords = {latitude: '46.874396', longitude: '-96.835556'};
    cy.visit('http://localhost:8080/', {
      onBeforeLoad: win => {
        cy.stub(win.navigator.geolocation, 'getCurrentPosition')
          .callsFake(callback => {
            return callback({coords})
          })
          .as('stubGeo')
      }
    })
  })
})
```

Una vez que tenemos el stub, pulsamos el botón, y vamos a comprobar que para estas coordenadas, la ciudad es **Fargo**, y que la función del **getCurrentPosition** solo se ha llamado una vez.

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

describe('Geolocalización', () => {
  it('debería de llamar una sola vez a paintCity cuando se pulsa el botón', () => {
    cy.visit('http://localhost:8080/', {
      onBeforeLoad: (win) => {
        cy.spy(win, 'fetch').as('spyFetch');
      }
    });

    cy.get('#btn-ubicacion')
      .click();

    cy.get('@spyFetch')
      .should('be.calledOnce')
  })

  it('debería mostrar "Fargo" cuando se le pasan sus coordenadas', () => {
    const coords = {latitude: '46.874396', longitude: '-96.835556'};
    cy.visit('http://localhost:8080/', {
      onBeforeLoad: win => {
        cy.stub(win.navigator.geolocation, 'getCurrentPosition')
          .callsFake(callback => {
            return callback({coords})
          })
          .as('stubGeo')
      }
    })

    cy.get('#btn-ubicacion')
      .click();

    cy.get('#ciudad')
      .should('have.text', 'Fargo');

    cy.get('@stubGeo')
      .should('be.calledOnce');
  })
})
```

En el siguiente test, vamos a realizar una comprobación similar a la anterior, solo que ahora comprobaremos que la ciudad que se muestra es **Harlan**.

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

```

describe('Geolocalización', () => {
  it('debería de llamar una sola vez a paintCity cuando se pulsa el botón', () => {
    cy.visit('http://localhost:8080/', {
      onBeforeLoad: (win) => {
        cy.spy(win, 'fetch').as('spyFetch');
      }
    });

    cy.get('#btn-ubicacion')
      .click();

    cy.get('@spyFetch')
      .should('be.calledOnce')
  })

  it('debería mostrar "Fargo" cuando se le pasan sus coordenadas', () => {
    const coords = {latitude: '46.874396', longitude: '-96.835556'};
    cy.visit('http://localhost:8080/', {
      onBeforeLoad: win => {
        cy.stub(win.navigator.geolocation, 'getCurrentPosition')
          .callsFake(callback => {
            return callback({coords})
          })
          .as('stubGeo')
      }
    })

    cy.get('#btn-ubicacion')
      .click();

    cy.get('#ciudad')
      .should('have.text', 'Fargo');

    cy.get('@stubGeo')
      .should('be.calledOnce');
  })

  it('debería mostrar "Harlan" cuando se le pasan sus coordenadas', () => {
    const coords = {latitude: '36.848044', longitude: '-83.320589'};

    cy.visit('http://localhost:8080/', {
      onBeforeLoad: (win) => {
        cy.stub(win.navigator.geolocation, 'getCurrentPosition')
          .callsFake(callback => {
            return callback({coords})
          })
      }
    })

    cy.get('#btn-ubicacion')
      .click();
  })
})

```

```

        cy.get('#ciudad')
          .should('have.text', 'Harlan');
    })
}

```

Por último, comprobaremos también que cuando las coordenadas son vacias o aleatorias, la ciudad que se nos muestra es **Una ciudad cualquiera....**

*/cypress-mocking-spy-y-stub-lab/cypress/integration/geolocalizacion.spec.js*

```

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

describe('Geolocalización', () => {
  it('debería de llamar una sola vez a paintCity cuando se pulsa el botón', () => {
    cy.visit('http://localhost:8080/', {
      onBeforeLoad: (win) => {
        cy.spy(win, 'fetch').as('spyFetch');
      }
    });

    cy.get('#btn-ubicacion')
      .click();

    cy.get('@spyFetch')
      .should('be.calledOnce')
  })

  it('debería mostrar "Fargo" cuando se le pasan sus coordenadas', () => {
    const coords = {latitude: '46.874396', longitude: '-96.835556'};
    cy.visit('http://localhost:8080/', {
      onBeforeLoad: win => {
        cy.stub(win.navigator.geolocation, 'getCurrentPosition')
          .callsFake(callback => {
            return callback(coords)
          })
          .as('stubGeo')
      }
    });

    cy.get('#btn-ubicacion')
      .click();

    cy.get('#ciudad')
      .should('have.text', 'Fargo');

    cy.get('@stubGeo')
      .should('be.calledOnce');
  })
}

```

```

it('debería mostrar "Harlan" cuando se le pasan sus coordenadas', () => {
  const coords = {latitude: '36.848044', longitude: '-83.320589'};

  cy.visit('http://localhost:8080/', {
    onBeforeLoad: (win) => {
      cy.stub(win.navigator.geolocation, 'getCurrentPosition')
        .callsFake(callback => {
          return callback({coords})
        })
    }
  })

  cy.get('#btn-ubicacion')
    .click();

  cy.get('#ciudad')
    .should('have.text', 'Harlan');
})

it('debería mostrar la ciudad comodín cuando se le pasan sus coordenadas', () => {
  const coords = {latitude: '', longitude: ''};

  cy.visit('http://localhost:8080/', {
    onBeforeLoad: (win) => {
      cy.stub(win.navigator.geolocation, 'getCurrentPosition')
        .callsFake(callback => {
          return callback({coords})
        })
    }
  })

  cy.get('#btn-ubicacion')
    .click();

  cy.get('#ciudad')
    .should('have.text', 'Una ciudad cualquiera...');

})
}

```

Y con esto ya tendríamos los tests que deberían de pasarse correctamente.

## 19.4. Intercept

Cypress nos permite utilizar el método **intercept** para interceptar nuestras peticiones HTTP.

Pasandole un primer parámetro con la URL a la que se hace la petición no estaremos mockeando nada, pero si que nos serviría para indicar que se espere a que se haya obtenido una respuesta a la petición interceptada.

```
cy.intercept('http://mi-api.com/get-response').as('peticion');

cy.wait('@peticion');
```

Pero si le pasamos **como segundo parámetro una respuesta o un fixture** entonces la petición no se va a realizar al servidor, devolviendo el valor indicado. Con esto si que estaríamos creando un stub a partir de una petición HTTP para controlar la respuesta que se recibe cuando esta puede no devolvernos un resultado predecible.

```
cy.intercept('http://mi-api.com/get-response', { fixture: 'datos.json' }).
```

Si queremos interceptar una petición pero filtrando por el método de esta, podemos pasar como primer parámetro un string con el tipo de método.

## 19.5. Lab: Intercept

En este laboratorio vamos a ver como interceptar una petición HTTP para devolver una respuesta mockeada cuando se quieren testear funcionalidades que no devuelven un valor previsible. El objetivo del laboratorio es probar que:

- Cuando una API del tiempo devuelve el texto 'soleado', se pinta el emoji del sol en la aplicación.
- Cuando una API del tiempo devuelve el texto 'nevado', se pinta el emoji de la nube con nieve en la aplicación.

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-mocking-intercept-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-mocking-intercept-lab
$ cd cypress-mocking-intercept-lab
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests
- express: framework para crear webs

```
$ npm install --save-dev cypress
$ npm install --save express
```

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

/cypress-mocking-intercept-lab/package.json

```
{  
  "name": "cypress-mocking-intercept-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "cy:open": "cypress open",  
    "start": "node server.js"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.8.0"  
  },  
  "dependencies": {  
    "express": "^4.17.1"  
  }  
}
```

El siguiente paso es crear un archivo **index.html** donde vamos a poner el código para mostrar los emojis dependiendo de la respuesta que obtengamos de una API.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Intercept</title>
</head>
<body>
  <p>Tiempo para hoy: <span id="tiempo"></span></p>

  <script>
    function getEmoji() {
      return fetch('http://localhost:8080/get-weather')
        .then(resp => resp.json())
        .then(({ weather }) => {
          switch(weather) {
            case 'soleado': return '☀';
            case 'parcialmente-nublado': return '🌤';
            case 'nublado': return '☁';
            case 'lluvioso': return '🌧';
            case 'tormenta': return '⛈';
            case 'nevado': return '❄';
          }
        });
    }

    getEmoji().then(val => {
      document.getElementById('tiempo').innerText = val;
    })
  </script>
</body>
</html>
```

Ahora añadimos el archivo **server.js** con la API a la que vamos a realizar las peticiones para obtener el tiempo y que nos va a enviar la página inicial de la aplicación.

```
const http = require('http');
const express = require('express');
const path = require('path');

function getWeather() {
  const weathers = ['soleado', 'parcialmente-nublado', 'nublado', 'lluvioso', 'tormenta', 'nevado'];
  const pos = Math.floor(Math.random() * weathers.length);
  return weathers[pos];
}

const app = express();

app.get('/get-weather', (req, res, next) => {
  res.json({ weather: getWeather() });
})

app.get('/', (req, res, next) => {
  res.sendFile(path.join(__dirname, 'index.html'));
});

const server = http.createServer(app);
server.listen('8080');
```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

En otra terminal distinta a la anterior vamos a lanzar el comando que levanta el servidor de express con la API.

```
$ npm start
```

Podemos entrar a la página de la aplicación desde <http://localhost:8080>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **intercept.spec.js** dentro de la carpeta **integration**.

Empezamos por crear el bloque **describe** con nuestro primer test.

/cypress-mocking-intercept-lab/cypress/integration/intercept.spec.js

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

describe('Tiempo', () => {
  it('debería pintar el sol si el tiempo es soleado', () => {
    })
})
```

Empezamos por navegar a la página inicial de la aplicación, y añadimos la comprobación que habíamos indicado al principio, es decir, que el span que hay dentro del párrafo muestre el emoji del sol cuando la API nos indique que el tiempo es soleado.

/cypress-mocking-intercept-lab/cypress/integration/intercept.spec.js

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

describe('Tiempo', () => {
  it('debería pintar el sol si el tiempo es soleado', () => {
    cy.visit('http://localhost:8080/')

    cy.get('#tiempo')
      .should('have.text', '☀')
  })
})
```

Si probamos a ejecutar varias veces este test, veremos que algunas se ejecuta correctamente y otras no. Esto se debe a que la API nos devuelve una respuesta que no es predecible en base a unos parámetros. Que nos devuelve el string **soleado** depende del tiempo que haga durante el día que se están ejecutando los tests, y un día puede estarlo, pero otro día puede estar nublado.

Entonces en este caso, para solucionar este problema, la solución es utilizar el método **intercept** para interceptar la llamada a la API, e indicarle la respuesta que nos viene bien que se retorne, es decir, estamos mockeando una respuesta de una API.

Al método intercept le vamos a pasar como parámetros el path de la API que tiene que interceptar y el objeto que queremos obtener como respuesta. Este método tiene que ejecutarse antes de llegar a entrar en la página.

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

describe('Tiempo', () => {
  it('debería pintar el sol si el tiempo es soleado', () => {
    cy.intercept('/get-weather', { weather: 'soleado' })
    cy.visit('http://localhost:8080/')

    cy.get('#tiempo')
      .should('have.text', '☀')
  })
})
```

Ahora ya debería de pasarse nuestro test correctamente.

El siguiente test es prácticamente igual, solo tenemos que cambiar la respuesta y la aserción, utilizando el texto **nevado** y el emoji de la nube con nieve.

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

describe('Tiempo', () => {

  it('debería pintar el sol si el tiempo es soleado', () => {
    cy.intercept('/get-weather', { weather: 'soleado' });
    cy.visit('http://localhost:8080/')

    cy.get('#tiempo')
      .should('have.text', '☀')
  })

  it('debería pintar la nube con nieve si el tiempo es nevado', () => {
    cy.intercept('/get-weather', { weather: 'nevado' })
    cy.visit('http://localhost:8080/')

    cy.get('#tiempo')
      .should('have.text', '☃')
  })
})
```

Ahora ya deberíamos de pasar las dos pruebas correctamente.

# Capítulo 20. Peticiones HTTP: Request

Cuando hacemos bastantes tests que pasan por las mismas partes de la aplicación, como por ejemplo un **login** que ya estaría más que testeado, lo único que conseguimos es que nuestros tests tarden más tiempo del necesario en ejecutarse.

Por ejemplo, para realizar un login, tenemos que seguir los siguientes pasos:

- Buscar el campo de email/usuario
- Escribir el email/usuario
- Buscar el campo de la contraseña
- Escribir la contraseña
- Buscar el botón de Login
- Pulsar sobre el botón

Todas estas acciones podemos reducirlas a enviar una petición HTTP al servidor que se encargue de realizar este login.



Ojo, no quiere decir que haya que hacer la petición en todos nuestros tests. Deberíamos de tener algún test que si que realice esos 6 pasos para comprobar que el formulario funciona correctamente y podemos loguearnos. Pero si esto ya lo hemos hecho, en el resto de test lo podemos sustituir por una petición HTTP.

Otro de los casos en el que podríamos utilizar este comando es para **rellenar una BBDD con datos de prueba** para realizar los tests justo antes de cada uno de estos, realizando una petición a un servicio que se encargue de introducirlos.

En Cypress el comando que nos permite realizar este tipo de peticiones es el **request**. Este comando recibe como parámetros la url, y por defecto va a realizar una petición GET.

```
cy.request('http://mi-api.com/v1/perfil/3');
```

Cuando necesitamos realizar una petición distinta de un GET, entonces podemos pasarle el tipo como primer parámetro.

```
cy.request('DELETE', 'http://mi-api.com/v1/tarea/3');
```

Y en el caso de una petición que necesita enviar unos datos en el cuerpo de esta, entonces los vamos a poner como último parámetro de la función.

```
const ofertaTrabajo = { puesto: 'Experto en ciberseguridad', empresa: 'E-Corp, Inc', lugar: 'remoto', salario: 25000 }
cy.request('POST', 'http://mi-api.com/v1/oferta-trabajo', ofertaTrabajo);
```

Para poder acceder a las respuestas de estas peticiones tendremos que utilizar el **then** donde las

vamos a recibir. El cuerpo de las respuestas ya viene serializado a JSON por lo que no hace falta llamar a la función **resp.body()** como si ocurre con métodos como el **fetch**.

# Capítulo 21. Tick y Clock

Cuando tenemos aplicaciones en las que el tiempo es algo importante, ya que la aplicación depende de los segundos, minutos... que vayan pasando, como por ejemplo aplicaciones de gestión de tiempo que usan técnicas como pomodoro, nos encontramos con que son difíciles de testear por el tiempo en si, ya que cuando ejecutamos los tests, el tiempo también irá pasando y puede ser difícil cuadrar las aserciones.

Aquí Cypress nos viene a proporcionar dos funciones: **tick** y **clock**.

Con la función **clock** vamos a poder sobrescribir las funciones nativas del tiempo (setTimeout, setInterval...) para luego poder controlar síncronamente el tiempo que va a ir pasando durante el test.

Luego tenemos la función **tick**, que es la que nos va a permitir mover el tiempo pasandoselo como parámetro en milisegundos.

Esta función solo se puede llamar después que hayamos sobreescrito las funciones nativas del tiempo con el **clock**.

```
cy.clock()  
  
cy.tick(1000 * 4) // Adelantamos el tiempo 4 segundos  
cy.tick(1000 * 10 * 60) // Adelantamos el tiempo 10 minutos
```



La función **tick** es una función a la que no se le puede concatenar otros comandos de utilidad ni ningún tipo de aserción.

## 21.1. Lab: Tick y Clock

En este laboratorio vamos a ver como utilizar las funciones de **tick** y **clock** para simular el paso del tiempo en nuestra aplicación y así poder testear aplicaciones que tienen algún contador o cuya funcionalidad depende del tiempo que pasa.

Para este laboratorio vamos a testear la aplicación <https://resting.onrender.com/> y vamos a comprobar los siguientes casos:

- Debería estar deshabilitado el botón de Start si no se añade un título al descanso
- Debería crear la cuenta atrás con los datos correctos
- Debería quedarse en 0 la cuenta atrás si pasa el tiempo seleccionado

Vamos a empezar creando la carpeta del proyecto e instalando las dependencias necesarias lanzando los siguientes comandos:

```
$ mkdir cypress-tick-clock-lab
$ cd cypress-tick-clock-lab
$ npm init -y
$ npm install --save-dev cypress
```

Después de lanzar estos comandos deberíamos de tener un archivo **package.json** en nuestro proyecto. Dentro de este archivo vamos a modificar la sección de scripts para añadir los dos scripts que se van a encargar de ejecutar los tests.

/cypress-tick-clock/package.json

```
{
  "name": "cypress-tick-clock-lab",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "cy:open": "cypress open",
    "cy:run": "cypress run"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "cypress": "^8.4.1"
  }
}
```

Ahora vamos a lanzar el primer comando que hemos añadido para que se genere dentro de nuestro proyecto una carpeta de **cypress** donde encontramos una serie de tests de ejemplo.

```
$ npm run cy:open
```

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **resting.spec.js** dentro de la carpeta **integration**. Vamos a añadir el bloque **describe** y el primer bloque **it** para el primer test.

Como en todas las pruebas vamos a ir a la misma página, vamos a meter la instrucción de visitar la aplicación dentro de un bloque **beforeEach**.

/cypress-tick-clock-lab/cypress/integration/resting.spec.js

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

describe('Resting', () => {
  beforeEach(() => {
    cy.visit('http://resting.onrender.com/')
  })

  it('debería estar deshabilitado el botón de Start si no se añade un título al descanso', () => {
    })
})
```

Ahora buscamos el botón de Start de la aplicación y vemos que una de las posibles formas para buscarlo es utilizando la clase que lleva la etiqueta, por lo que usamos esta clase, y después vamos a comprobar que tiene el atributo **disabled**.

/cypress-tick-clock-lab/cypress/integration/resting.spec.js

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

describe('Resting', () => {
  beforeEach(() => {
    cy.visit('http://resting.onrender.com/')
  })

  it('debería estar deshabilitado el botón de Start si no se añade un título al descanso', () => {
    cy.get('.css-i0fnhq')
      .should('have.attr', 'disabled')
  })
})
```

Una vez tenemos el primer test, vamos al segundo, en el que tendremos que empezar buscando el campo de texto para introducir el título del descanso.

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

describe('Resting', () => {
  beforeEach(() => {
    cy.visit('http://resting.onrender.com/')
  })

  it('debería estar deshabilitado el botón de Start si no se añade un título al descanso', () => {
    cy.get('.css-i0fnhq')
      .should('have.attr', 'disabled')
  })

  it('debería crear la cuenta atrás con los datos correctos', () => {
    const tituloDescanso = 'Curso de Cypress'

    cy.get('.css-qzovtw')
      .type(tituloDescanso);

  })
})
```

El siguiente paso va a ser cambiar el tiempo del descanso a 15 min. Por tanto tenemos que buscar el botón adecuado, y después pulsar sobre él. Tenemos que hacer lo mismo con el botón de Start para que nos lleve a la página con la cuenta atrás.

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

describe('Resting', () => {
  beforeEach(() => {
    cy.visit('http://resting.onrender.com/')
  })

  it('debería estar deshabilitado el botón de Start si no se añade un título al descanso', () => {
    cy.get('.css-i0fnhq')
      .should('have.attr', 'disabled')
  })

  it('debería crear la cuenta atrás con los datos correctos', () => {
    const tituloDescanso = 'Curso de Cypress'

    cy.get('.css-qzovtw')
      .type(tituloDescanso);

    cy.get('.css-nhk4on > :nth-child(2) > :nth-child(4)')
      .click();

    cy.get('.css-i0fnhq')
      .click();
  })
})
```

Una vez que estamos en la página correcta, vamos a usar la función **clock** para sobrescribir las funciones de tiempo y poder controlarlo nosotros de forma síncrona.

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

describe('Resting', () => {
  beforeEach(() => {
    cy.visit('http://resting.onrender.com/')
  })

  it('debería estar deshabilitado el botón de Start si no se añade un título al descanso', () => {
    cy.get('.css-i0fnhq')
      .should('have.attr', 'disabled')
  })

  it('debería crear la cuenta atrás con los datos correctos', () => {
    const tituloDescanso = 'Curso de Cypress'

    cy.get('.css-qzovtw')
      .type(tituloDescanso);

    cy.get('.css-nhk4on > :nth-child(2) > :nth-child(4)')
      .click();

    cy.get('.css-i0fnhq')
      .click();

    cy.clock();
  })
})
```

En este caso de prueba, con el tiempo parado, vamos a comprobar que el título es el correcto, el tiempo es 15:00 y que tenemos una imagen de fondo que se ha puesto mediante la propiedad de css background-image.

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

describe('Resting', () => {
  beforeEach(() => {
    cy.visit('http://resting.onrender.com/')
  })

  it('debería estar deshabilitado el botón de Start si no se añade un título al descanso', () => {
    cy.get('.css-i0fnhq')
      .should('have.attr', 'disabled')
  })

  it('debería crear la cuenta atrás con los datos correctos', () => {
    const tituloDescanso = 'Curso de Cypress'

    cy.get('.css-qzovtw')
      .type(tituloDescanso);

    cy.get('.css-nhk4on > :nth-child(2) > :nth-child(4)')
      .click();

    cy.get('.css-i0fnhq')
      .click();

    cy.clock();

    cy.get(':nth-child(2) > .gradient-text')
      .should('have.text', tituloDescanso);

    cy.get('.css-nhk4on > .cristal > .gradient-text')
      .should('have.text', '15 : 00');

    cy.get('[data-cy=container]')
      .should('have.css', 'background-image')
      .and('contain', 'https://images.unsplash.com/photo-');
  })
})
```

Por último, vamos a realizar el último caso de test en el quearemos lo mismo que en el caso anterior hasta llegar a la página de la cuenta atrás. Pero una vez estemos ahí vamos a comprobar que si hacemos que pase el tiempo de forma controlada con la función **tick**, llegamos a tener al final como cuenta 00:00.

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

describe('Resting', () => {
  beforeEach(() => {
    cy.visit('http://resting.onrender.com/')
  })

  it('debería estar deshabilitado el botón de Start si no se añade un título al descanso', () => {
    cy.get('.css-i0fnhq')
      .should('have.attr', 'disabled')
  })

  it('debería crear la cuenta atrás con los datos correctos', () => {
    const tituloDescanso = 'Curso de Cypress'

    cy.get('.css-qzovtw')
      .type(tituloDescanso);

    cy.get('.css-nhk4on > :nth-child(2) > :nth-child(4)')
      .click();

    cy.get('.css-i0fnhq')
      .click();

    cy.clock();

    cy.get(':nth-child(2) > .gradient-text')
      .should('have.text', tituloDescanso);

    cy.get('.css-nhk4on > .cristal > .gradient-text')
      .should('have.text', '15 : 00');

    cy.get('[data-cy=container]')
      .should('have.css', 'background-image')
      .and('contain', 'https://images.unsplash.com/photo-');
  })

  it('debería quedarse en 0 la cuenta atrás si pasa el tiempo seleccionado', () => {
    const tituloDescanso = 'Curso de Cypress';

    cy.get('.css-qzovtw')
      .type(tituloDescanso);

    cy.get('.css-nhk4on > :nth-child(2) > :nth-child(4)')
      .click();

    cy.get('.css-i0fnhq')
      .click();

    cy.clock();
  })
})
```

Ahora que tenemos el tiempo parado necesitamos esperar a que todos los elementos de la página se hayan cargado antes de empezar a controlar el tiempo con el **tick**.

Aquí tenemos varias opciones:

- Podríamos interceptar la petición a la nueva página, es decir, interceptar el GET a '/resting/15/min/in/Curso%20de%20Cypress', pero tenemos un problema, y es que no se hace ninguna petición porque la aplicación es una SPA, es decir, lo que cambia es el componente que se muestra, no se vuelve a descargar del servidor una nueva página.
- La segunda opción es poner una pequeña espera con **cy.wait()**, y con esto nos serviría, solo que queda un poco feo tener que hacer este tipo de esperas.
- La última opción y que veo más correcta en este caso es esperar a que se cargue el elemento que más puede tardar en descargarse. Este elemento es la imagen de fondo que viene desde la API de Unsplash, por lo que podemos interceptar esta petición y hacer que el test se espere hasta que esta imagen ya se haya descargado.

/cypress-tick-clock-lab/cypress/integration/resting.spec.js

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

describe('Resting', () => {
  beforeEach(() => {
    cy.visit('http://resting.onrender.com/')
  })

  it('debería estar deshabilitado el botón de Start si no se añade un título al descanso', () => {
    cy.get('.css-i0fnhq')
      .should('have.attr', 'disabled')
  })

  it('debería crear la cuenta atrás con los datos correctos', () => {
    const tituloDescanso = 'Curso de Cypress'

    cy.get('.css-qzovtw')
      .type(tituloDescanso);

    cy.get('.css-nhk4on > :nth-child(2) > :nth-child(4)')
      .click();

    cy.get('.css-i0fnhq')
      .click();

    cy.clock();

    cy.get(':nth-child(2) > .gradient-text')
      .should('have.text', tituloDescanso);

    cy.get('.css-nhk4on > .cristal > .gradient-text')
      .should('have.text', '15 : 00');

    cy.get('[data-cy=container]')
      .should('have.css', 'background-image')
      .and('contain', 'https://images.unsplash.com/photo-');
  })
})
```

```

})
it('debería quedarse en 0 la cuenta atrás si pasa el tiempo seleccionado', () => {
  const tituloDescanso = 'Curso de Cypress';

  cy.get('.css-qzovtw')
    .type(tituloDescanso);

  cy.get('.css-nhk4on > :nth-child(2) > :nth-child(4)')
    .click();

  cy.get('.css-i0fnhq')
    .click();

  cy.clock();

  // cy.intercept('/resting/15/min/in/Curso%20de%20Cypress').as('secondPage');
  // cy.wait(100)
  cy.intercept('https://images.unsplash.com/photo-*').as('imagenFondo')

})
})

```

Ahora ya podemos comprobar que el tiempo es 15:00, además de darle un alias a este elemento para usarlo después para volver a comprobar el tiempo.

*/cypress-tick-clock-lab/cypress/integration/resting.spec.js*

```

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

describe('Resting', () => {
  beforeEach(() => {
    cy.visit('http://resting.onrender.com/')
  })

  it('debería estar deshabilitado el botón de Start si no se añade un título al descanso', () => {
    cy.get('.css-i0fnhq')
      .should('have.attr', 'disabled')
  })

  it('debería crear la cuenta atrás con los datos correctos', () => {
    const tituloDescanso = 'Curso de Cypress'

    cy.get('.css-qzovtw')
      .type(tituloDescanso);

    cy.get('.css-nhk4on > :nth-child(2) > :nth-child(4)')
      .click();

    cy.get('.css-i0fnhq')
      .click();

    cy.clock();
  })
})

```

```

    cy.get(':nth-child(2) > .gradient-text')
      .should('have.text', tituloDescanso);

    cy.get('.css-nhk4on > .cristal > .gradient-text')
      .should('have.text', '15 : 00');

    cy.get('[data-cy=container]')
      .should('have.css', 'background-image')
      .and('contain', 'https://images.unsplash.com/photo-');
  })

it('debería quedarse en 0 la cuenta atrás si pasa el tiempo seleccionado', () => {
  const tituloDescanso = 'Curso de Cypress';

  cy.get('.css-qzovtw')
    .type(tituloDescanso);

  cy.get('.css-nhk4on > :nth-child(2) > :nth-child(4)')
    .click();

  cy.get('.css-i0fnhq')
    .click();

  cy.clock();

  // cy.intercept('/resting/15/min/in/Curso%20de%20Cypress').as('secondPage');
  // cy.wait(100)
  cy.intercept('https://images.unsplash.com/photo-*').as('imagenFondo')

  cy.get('.css-nhk4on > .cristal > .gradient-text')
    .as('countdownText')
    .should('have.text', '15 : 00');

})
}

})

```

Ahora vamos a indicar que se espere el test hasta que la petición de la imagen que habíamos interceptado se haya finalizado.

Y justo después ya podremos llamar al **tick** pasandole una cantidad de segundos para ir comprobando que el tiempo pasa según los segundos que le pasamos a la función anterior y que la vista se actualiza correctamente.

*/cypress-tick-clock-lab/cypress/integration/resting.spec.js*

```

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

describe('Resting', () => {
  beforeEach(() => {
    cy.visit('http://resting.onrender.com/')
  })

  it('debería estar deshabilitado el botón de Start si no se añade un título al descanso', () => {
    cy.get('.css-i0fnhq')
      .should('have.attr', 'disabled')
  })
}

```

```

})

it('debería crear la cuenta atrás con los datos correctos', () => {
  const tituloDescanso = 'Curso de Cypress'

  cy.get('.css-qzovtw')
    .type(tituloDescanso);

  cy.get('.css-nhk4on > :nth-child(2) > :nth-child(4)')
    .click();

  cy.get('.css-i0fnhq')
    .click();

  cy.clock();

  cy.get(':nth-child(2) > .gradient-text')
    .should('have.text', tituloDescanso);

  cy.get('.css-nhk4on > .cristal > .gradient-text')
    .should('have.text', '15 : 00');

  cy.get('[data-cy=container]')
    .should('have.css', 'background-image')
    .and('contain', 'https://images.unsplash.com/photo-');
}

it('debería quedarse en 0 la cuenta atrás si pasa el tiempo seleccionado', () => {
  const tituloDescanso = 'Curso de Cypress';

  cy.get('.css-qzovtw')
    .type(tituloDescanso);

  cy.get('.css-nhk4on > :nth-child(2) > :nth-child(4)')
    .click();

  cy.get('.css-i0fnhq')
    .click();

  cy.clock();

  // cy.intercept('/resting/15/min/in/Curso%20de%20Cypress').as('secondPage');
  // cy.wait(100)
  cy.intercept('https://images.unsplash.com/photo-*').as('imagenFondo')

  cy.get('.css-nhk4on > .cristal > .gradient-text')
    .as('countdownText')
    .should('have.text', '15 : 00');

  // Si falla, comprobar que no se haya cacheado (pulsamos sobre el checkbox de "disable cache" en Network)
  cy.wait('@imagenFondo');

  cy.tick(1000);
  cy.get('@countdownText')
    .should('have.text', '14 : 50');

  cy.tick(5000);
  cy.get('@countdownText')
    .should('have.text', '14 : 00');

  cy.tick(1000 * 14 * 60);
}

```

```
    cy.get('@countdownText')
      .should('have.text', '00 : 00');
  })
})
```

Y con esto ya deberíamos dar por probada la aplicación de **resting**.

# Capítulo 22. Comandos

Para interactuar con el navegador hemos estado usando comandos que vienen predefinidos con Cypress, pero Cypress nos da la opción de extender su funcionalidad añadiendo nuevos comandos.

Es decir, que podemos crear nuestros propios comandos para evitar tener código repetido entre varios tests y distintos archivos de testing.

Estos nuevos comandos los podemos crear directamente en el archivo **cypress/support/commands.js**, o en cualquier otro archivo que finalmente se esté importando en el **index.js** de la carpeta **cypress/support**.

Para crear un comando, usamos el método **Cypress.Commands.add** al que le pasamos como primer parámetro el nombre del comando, y como segundo parámetro la función que se va a ejecutar cuando lo llamemos.

```
Cypress.Commands.add('pulsarBoton', (textoBoton) => {
  return cy.contains('button', textoBoton)
    .click();
});
```

## 22.1. Lab: Comandos

En este laboratorio vamos a ver como crearnos nuestros propios comandos para reutilizar funciones dentro de nuestros tests. Crearemos los siguientes comandos:

- Pulsar un botón dado su identificador
- Rellenar un campo dado su name y el texto que hay que escribir

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-comandos-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-comandos-lab
$ cd cypress-comandos-lab
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests
- express: framework para crear webs

```
$ npm install --save-dev cypress  
$ npm install --save express
```

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

*/cypress-comandos-lab/package.json*

```
{  
  "name": "cypress-comandos-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "cy:open": "cypress open",  
    "start": "node server.js"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.8.0"  
  },  
  "dependencies": {  
    "express": "^4.17.1"  
  }  
}
```

Ahora vamos a crear un archivo **server.js** donde crearemos con express una API muy básica para simular que podemos hacer login y que si este es correcto se nos devuelve un token (un número aleatorio).

/cypress-comandos-lab/server.js

```
const http = require('http');
const express = require('express');
const path = require('path');

const usuariosValidos = [
  { email: 'cfalco@gmail.com', password: '1234' },
  { email: 'admin@gmail.com', password: 'admin' },
  { email: 'kozinsky@gmail.com', password: 'password' },
]

const app = express();

app.use(express.json())
app.use(express.urlencoded())

app.post('/login', (req, res, next) => {
  const { email, password } = req.body;
  const usuario = usuariosValidos.find(u => (u.email === email && u.password === password))
  if (!usuario) {
    return res.status(401).json({msg: 'Invalid email or password'});
  }

  const token = Math.random().toString().slice(2);
  return res.status(200).json({ token });
})

app.get('/', (req, res, next) => {
  res.sendFile(path.join(__dirname, 'index.html'));
});

const server = http.createServer(app);
server.listen('8080');
```

El siguiente paso es crear un archivo **index.html** que nos devuelve el servidor y que tendrá nuestra página con el formulario de login.

/cypress-comandos-lab/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Comandos</title>
<style>
  body {
    height: 100vh;
    width: 100vw;
    background-color: #242424;
    color: white;
  }
</style>
```

```

.center {
  height: 100%;
  display: flex;
  justify-content: center;
  align-items: center;
}

.error {
  color: red;
}

</style>
</head>
<body>
  <div class="center">
    <div id="container">
      <h1>Login</h1>
      <div>
        <label for="email">Email</label>
        <input type="email" name="email" id="email">
      </div>
      <div>
        <label for="password">Password</label>
        <input type="password" name="password" id="password">
      </div>
      <button type="button" id="btn-login">Login</button>
    </div>
  </div>

<script>
  document.getElementById('btn-login').addEventListener('click', () => {
    const email = document.getElementById('email').value;
    const password = document.getElementById('password').value;
    const usuario = {email, password};
    fetch('http://localhost:8080/login', {
      method: 'POST',
      body: JSON.stringify(usuario),
      headers: {
        'Content-Type': 'application/json'
      }
    })
    .then(resp => {
      if (resp.status === 401) {
        throw new Error(resp.statusText)
      }
      return resp.json();
    })
    .then(data => {
      localStorage.setItem('auth-token', data.token);
      document.getElementById('container').innerHTML = '<h1>Bienvenido</h1>';
    })
    .catch(err => {
      document.getElementById('container').innerHTML += '<p class="error">Error: email o contraseña erroneo</p>';
    })
  })
</script>
</body>
</html>

```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cypress:open
```

Además, en otra terminal, vamos a lanzar el otro script para poder acceder con Cypress a la web.

```
$ npm start
```

Podemos acceder a la web en <http://localhost:8080/>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **comandos.spec.js** dentro de la carpeta **integration**.

Empezamos añadiendo el bloque **describe**, además de un **beforeEach** para visitar la página inicial, ya que tendremos dos tests que van a dicha página.

/cypress-comandos-lab/cypress/integration/comandos.spec.js

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

describe('Comandos', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080');
  })

});
```

En el primer test vamos a comprobar que si rellenamos los campos con los datos de un usuario válido, podemos loguearnos y se nos muestra un mensaje de bienvenida.

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

describe('Comandos', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080');
  })

  it('deberíamos de poder hacer login con un usuario válido', () => {
    cy.get('[name="email"]')
      .type('cfalco@gmail.com');

    cy.get('[name="password"]')
      .type('1234');

    cy.get('#btn-login')
      .click();

    cy.get('h1')
      .should('exist')
      .and('have.text', 'Bienvenido');
  });
});
```

En el segundo test vamos a comprobar que si nos logueamos con un usuario invalido, se muestra un error en la página de login.

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

describe('Comandos', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080');
  })

  it('deberíamos de poder hacer login con un usuario válido', () => {
    cy.get('[name="email"]')
      .type('cfalco@gmail.com');

    cy.get('[name="password"]')
      .type('1234');

    cy.get('#btn-login')
      .click();

    cy.get('h1')
      .should('exist')
      .and('have.text', 'Bienvenido');
  });

  it('deberíamos de mostrarse un error si hacemos login con un usuario invalido', () => {
    cy.get('[name="email"]')
      .type('noexiste@gmail.com');
    cy.get('[name="password"]')
      .type('noexiste');
    cy.get('#btn-login')
      .click();
    cy.get('.error')
      .should('exist')
      .and('have.text', 'Error: email o contraseña erroneo');
  });
});
```

Si probamos a ejecutar los test podemos ver que funcionan correctamente.

Pero si nos fijamos tenemos bastante código repetido en ambos tests. Para hacer los test más simples y más legibles, vamos a crearnos dos comandos en el archivo de **cypress/support/commands.js**:

- Uno para buscar un elemento por su identificador.
- Otro para llenar un campo dado su atributo name con un texto.

/cypress-comandos-lab/cypress/support/commands.js

```
// ...

Cypress.Commands.add('pulsarBotonPorId', (id) => {
  return cy.get(`button#${id}`)
    .click();
});

Cypress.Commands.add('rellenarCampo', (name, texto) => {
  return cy.get(`input[name='${name}']`)
    .type(texto);
});
```

Una vez añadidos estos dos comandos, podemos dejar nuestros test más limpios como hemos dicho.

/cypress-comandos-lab/cypress/integration/comandos.spec.js

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

describe('Comandos', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080/');
  })

  it('deberíamos de poder hacer login con un usuario válido', () => {
    cy.rellenarCampo('email', 'cfalco@gmail.com');
    cy.rellenarCampo('password', '1234');
    cy.pulsarBotonPorId('btn-login');

    cy.get('h1')
      .should('exist')
      .and('have.text', 'Bienvenido');
  });

  it('deberíamos de mostrarse un error si hacemos login con un usuario invalido', () => {
    cy.rellenarCampo('email', 'noexiste@gmail.com');
    cy.rellenarCampo('password', 'noexiste');
    cy.pulsarBotonPorId('btn-login');

    cy.get('.error')
      .should('exist')
      .and('have.text', 'Error: email o contraseña erroneo');
  });
});
```

Y si ejecutamos el test debería de seguir pasándose correctamente.

Podemos cambiar las aserciones para hacer que fallen y comprobar que no son falsos positivos.

# Capítulo 23. Test condicionales

El **test condicional** consiste en poder realizar una prueba u otra dependiendo de una condición, como por ejemplo, si aparece este elemento primero hay que pulsar el botón y después escribimos este texto, pero si no aparece el elemento entonces escribimos el texto directamente.

Se nos podría ocurrir hacer algo como lo siguiente:

```
const elemento = cy.get('#elemento');

if (elemento) {
  cy.get('button').click();
}

cy.get('input').type('Hola mundo')
```

Pero eso no nos va a funcionar con Cypress, ya que los métodos de Cypress devuelven objetos del tipo Chainer, y por tanto no obtenemos el elemento que buscamos en la constante.

Entonces la opción que podemos utilizar es realizar la comprobación con objetos de jQuery en lugar de con los objetos de Cypress.

Podemos buscar un elemento de la aplicación en el que se encuentre contenido el elemento que puede aparecer o no aparecer en la web, y encadenarle un **then** para acceder al elemento de jQuery. Después ya podríamos utilizar los métodos de jQuery para buscar el elemento y realizar alguna acción necesaria para que el test pueda continuar sin dar un error, ya que si con jQuery no se obtiene el elemento, el test no va a fallar, como ocurriría con un método de Cypress.

```
cy.get('body')
  .then(body => {
    if (body.find('#elemento')) {
      cy.get('button').click();
    }
  });

cy.get('input').type('Hola mundo')
```

## 23.1. Lab: Test condicionales

En este laboratorio vamos a ver como crear un test condicional que compruebe si hay que habilitar la edición de los campos de texto antes de poder escribir en ellos o si ya estaba habilitado.

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-tests-condicionales-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-tests-condicionales-lab  
$ cd cypress-tests-condicionales-lab  
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests
- http-server: servidor de desarrollo local

```
$ npm install --save-dev cypress http-server
```

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

/cypress-tests-condicionales-lab/package.json

```
{  
  "name": "cypress-tests-condicionales-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "http-server",  
    "cy:open": "cypress open"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.8.0",  
    "http-server": "^0.12.3"  
  }  
}
```

El siguiente paso es crear un archivo **index.html** donde vamos a poner la caja de texto con el checkbox que controla si se puede escribir en ella o no.

/cypress-tests-condicionales-lab/index.html

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
  <meta charset="UTF-8">  
  <meta http-equiv="X-UA-Compatible" content="IE=edge">  
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```

<title>Test condicionales</title>
<style>
  body {
    height: 100vh;
    width: 100vw;
    background-color: #242424;
    color: white;
  }

  .center {
    height: 100%;
    display: flex;
    justify-content: center;
    align-items: center;
    flex-direction: column;
  }
</style>
</head>
<body>
  <div class="center">
    <div id="caja-edicion">
      <label for="modoEdicion">Edición deshabilitada:</label>
      <input type="checkbox" id="modoEdicion">
    </div>
    <div>
      <label for="nombre">Nombre</label>
      <input type="text" name="nombre" id="nombre">
    </div>
  </div>

  <script>
    const modoEdicionDesactivado = Math.floor(Math.random() * 3) === 0;
    const checkEdicion = document.getElementById('modoEdicion');
    const inputNombre = document.getElementById('nombre');
    checkEdicion.checked = modoEdicionDesactivado;
    inputNombre.disabled = modoEdicionDesactivado;

    checkEdicion.addEventListener('click', () => {
      const checked = checkEdicion.checked;
      checkEdicion.checked = checked;
      inputNombre.disabled = checked;
    })
  </script>
</body>
</html>

```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Además, en otra terminal, vamos a lanzar el otro script para poder acceder con Cypress a la web.

```
$ npm start
```

Como nos indica en la terminal, podemos acceder a la web en <http://localhost:8080/>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **condicional.spec.js** dentro de la carpeta **integration**.

Añadimos nuestro caso de prueba y visitamos la página donde se encuentra lo que queremos testear.

/cypress-tests-condicionales-lab/cypress/integration/condicional.spec.js

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

describe('Condicional', () => {
  it('deberíamos de poder escribir en el campo de texto aunque el modo edición esté desactivado', () => {
    cy.visit('http://localhost:8080');

  });
});
```

Ahora viene la parte del test en el que tenemos que ver si hay que realizar una acción previa o no a las que haríamos para realizar el test.

En este caso, tenemos que comprobar primero si el checkbox está marcado o no, para habilitar el modo edición. Para ello, empezamos por buscar el propio checkbox y utilizaremos el **then** para acceder a la etiqueta que nos devuelve jQuery.

/cypress-tests-condicionales-lab/cypress/integration/condicional.spec.js

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

describe('Condicional', () => {
  it('deberíamos de poder escribir en el campo de texto aunque el modo edición esté desactivado', () => {
    cy.visit('http://localhost:8080');

    cy.get('#modoEdicion')
      .then(checkEdicion => {

    })
  });
});
```

El siguiente paso es obtener el valor de la propiedad **checked** de esta etiqueta y comprobar si es true o false. En el caso de ser true, tendremos que realizar un click sobre el elemento, si es false no

hay que hacer nada porque ya podríamos escribir en el campo.

/cypress-tests-condicionales-lab/cypress/integration/condicional.spec.js

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

describe('Condisional', () => {
  it('deberíamos de poder escribir en el campo de texto aunque el modo edición esté desactivado', () => {
    cy.visit('http://localhost:8080');

    cy.get('#modoEdicion')
      .then(checkEdicion => {
        const isChecked = checkEdicion.prop('checked');
        if (isChecked) {
          checkEdicion.trigger('click');
        }
      })
    });

  });
});
```

Por último, ahora que ya sabemos de seguro que la edición en los campos de texto está habilitada, podemos escribir nuestra prueba como hemos venido haciendo durante el curso.

Vamos a buscar el campo de texto, escribiremos cualquier nombre, y comprobaremos que el valor del input es lo que se había escrito justo antes.

/cypress-tests-condicionales-lab/cypress/integration/condicional.spec.js

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

describe('Condisional', () => {
  it('deberíamos de poder escribir en el campo de texto aunque el modo edición esté desactivado', () => {
    cy.visit('http://localhost:8080');

    cy.get('#modoEdicion')
      .then(checkEdicion => {
        const isChecked = checkEdicion.prop('checked');
        if (isChecked) {
          checkEdicion.trigger('click');
        }
      })

    const nombre = 'Charly Falco';
    cy.get('#nombre')
      .type(nombre)
      .should('have.value', nombre);
  });
});
```

Con esto ya tenemos nuestro test y debería de pasarse correctamente al ejecutarlo.

# Capítulo 24. Tests visuales

Desde Cypress podemos comprobar que los elementos tienen las propiedades CSS correctas para que se pinten como se tienen que pintar. Pero a veces queremos saber si al actualizar alguna librería o cambiar los estilos de los elementos, se ha modificado algo que hace que la aplicación cambie su apariencia y no se vea como esperamos.

Una solución sería añadir aserciones con todas las propiedades CSS de los elementos, pero esto sería una tarea muy lenta de hacer y que no podemos permitirnos.

La otra solución es utilizar el **testing visual**, el cual consiste en comparar capturas de pantalla de nuestra aplicación o de los elementos sueltos.

Se saca un screenshot de la página en el momento en que la vemos que está perfecta, y este screenshot será la prueba visual con la que se compararán los siguientes screenshots que se obtengan al ejecutar los tests de la aplicación.

Cuando se modifique la web y sepamos que ya se muestra todo correctamente tendremos que actualizar el screenshot principal.

Esta tipo de pruebas son más lentas que las que hemos visto hasta ahora, ya que no es lo mismo realizar unas interacciones sobre la web y comprobar que un texto aparezca, que comparar dos imágenes.

Para realizar este tipo de pruebas podemos utilizar algunas herramientas como Applitools o Percy, o alguna dependencia que nos permita comparar imágenes.

Este tipo de pruebas las utilizan por ejemplo en Google.

## 24.1. Lab: Tests visuales

En este laboratorio vamos a ver como podemos realizar pruebas visuales de nuestras aplicaciones para comprobar que los elementos que se pintan en la aplicación no han cambiado.

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-tests-visuales-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-tests-visuales-lab  
$ cd cypress-tests-visuales-lab  
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir un script para levantar la aplicación en un servidor de desarrollo local y otro para abrir el panel de Cypress.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests
- http-server: servidor de desarrollo local

```
$ npm install --save-dev cypress http-server
```

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

*/cypress-tests-visuales-lab/package.json*

```
{
  "name": "cypress-tests-visuales-lab",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "http-server",
    "cy:open": "cypress open"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "cypress": "^6.8.0",
    "http-server": "^0.12.3"
  }
}
```

El siguiente paso es crear un archivo **index.html** donde vamos a poner una caja con un color.

/cypress-tests-visuales-lab/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Test visual</title>
  <style>
    body {
      height: 100vh;
      width: 100vw;
      background-color: #242424;
    }

    .center {
      height: 100%;
      display: flex;
      justify-content: center;
      align-items: center;
    }

    .caja {
      width: 100px;
      height: 100px;
    }

    .bg-gold {
      background-color: gold;
    }
  </style>
</head>
<body>
  <div class="center">
    <div class="caja bg-gold"></div>
  </div>
</body>
</html>
```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Además, en otra terminal, vamos a lanzar el otro script para poder acceder con Cypress a la web.

```
$ npm start
```

Como nos indica en la terminal, podemos acceder a la web en <http://localhost:8080/>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **caja-gold.spec.js** dentro de la carpeta **integration**.

Añadimos nuestro caso de prueba y visitamos la página donde se encuentra lo que queremos testear.

```
/cypress-tests-visuales-lab/cypress/integration/caja-gold.spec.js
```

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

describe('Caja Gold', () => {
  it('debería pintar una caja de color oro', () => {
    cy.visit('http://localhost:8080');

  });
});
```

Podemos comprobar que el color de la caja es el correcto utilizando la aserción de **have.css**.

```
/cypress-tests-visuales-lab/cypress/integration/caja-gold.spec.js
```

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

describe('Caja Gold', () => {
  it('debería pintar una caja de color oro', () => {
    cy.visit('http://localhost:8080');
    cy.get('.caja')
      .should('have.css', 'background-color', 'rgb(255, 215, 0)');
  });
});
```

Pero si tuviéramos que realizar comprobaciones de muchas propiedades CSS, el test se puede volver demasiado complejo. Por eso, vamos a utilizar una prueba visual para pasar este test.

Estas pruebas consisten en sacar una imagen de la página que se va a ir comparando con las siguientes imágenes que se saquen cuando ejecutemos el test en el futuro. Así que necesitamos utilizar alguna herramienta para que se encargue de estas tareas, realizar las capturas y la comprobación entre imágenes.

Vamos a utilizar el paquete **cypress-image-snapshot**, por lo que vamos a instalarlo en el proyecto utilizando NPM:

```
$ npm install --save-dev cypress-image-snapshot
```

Una vez instalado, nuestro **package.json** queda de la siguiente forma:

/cypress-tests-visuales-lab/package.json

```
{  
  "name": "cypress-tests-visuales-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "http-server",  
    "cy:open": "cypress open"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.8.0",  
    "cypress-image-snapshot": "^4.0.1",  
    "http-server": "^0.12.3"  
  }  
}
```

Una vez instalado, vamos a seguir los pasos que se nos indican en la documentación de esta dependencia para poder utilizarla.

Vamos a añadir el plugin en la configuración de Cypress, por lo que dentro del archivo **cypress/plugins/index.js** vamos a importar el plugin que viene con la dependencia y lo vamos a ejecutar dentro del módulo que se está exportando.

/cypress-tests-visuales-lab/cypress/plugins/index.js

```
/// <reference types="cypress" />  
// *****  
// ...  
// *****  
const { addMatchImageSnapshotPlugin } = require('cypress-image-snapshot/plugin');  
  
/**  
 * @type {Cypress.PluginConfig}  
 */  
// eslint-disable-next-line no-unused-vars  
module.exports = (on, config) => {  
  // `on` is used to hook into various events Cypress emits  
  // `config` is the resolved Cypress config  
  addMatchImageSnapshotPlugin(on, config)  
  
  return config;  
}
```

Una vez añadido aquí el plugin, nos toca añadir el comando para que podemos utilizar la funcionalidad desde el objeto **cy**. Vamos a llamar a la función que se encarga de añadir el comando

**matchImageSnapshot** dentro del archivo **cypress/support/commands.js** además de configurarlo con el porcentaje de fallo que consideramos aceptable para que un test se pase correctamente.

/cypress-tests-visuales-lab/cypress/plugins/index.js

```
// ****
// ...
// ****
//
// ...

import { addMatchImageSnapshotCommand } from 'cypress-image-snapshot/command';

addMatchImageSnapshotCommand({
  failureThresholdType: 'percent'
});
```

Ahora ya podemos utilizar el comando de Cypress **matchImageSnapshot** pasándole como parámetro el nombre que le queremos dar al snapshot.

/cypress-tests-visuales-lab/cypress/integration/caja-gold.spec.js

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

describe('Caja Gold', () => {
  it('debería pintar una caja de color oro', () => {
    cy.visit('http://localhost:8080');
    cy.matchImageSnapshot('cuadrado-de-oro');
  });
});
```

Con esto ya tendríamos nuestro test visual. Vamos a ejecutarlo una vez para sacar el snapshot inicial.

Después de que se haya ejecutado, podremos ver una carpeta **cypress/snapshots/caja-gold.spec.js** con el snapshot que se ha sacado.

Vamos a cambiar los estilos de la caja.

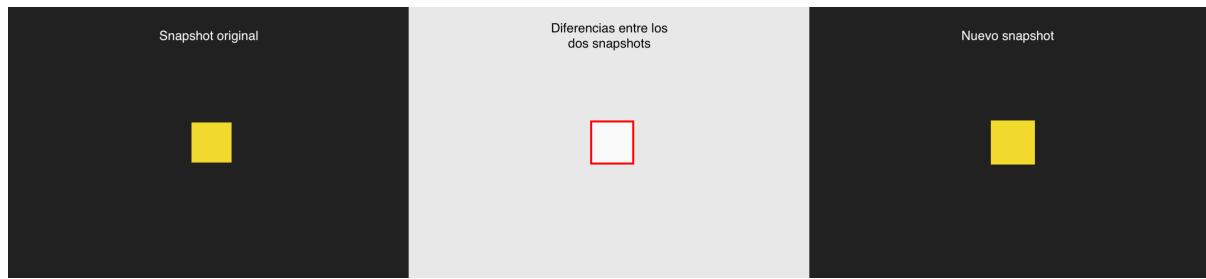
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Test visual</title>
  <style>
    body {
      height: 100vh;
      width: 100vw;
      background-color: #242424;
    }

    .center {
      height: 100%;
      display: flex;
      justify-content: center;
      align-items: center;
    }

    .caja {
      width: 110px;
      height: 110px;
      /* width: 100px;
      height: 100px; */
    }

    .bg-gold {
      background-color: gold;
    }
  </style>
</head>
<body>
  <div class="center">
    <div class="caja bg-gold"></div>
  </div>
</body>
</html>
```

Ahora volvemos a ejecutar el test para comprobar que falla y nos genera la imagen con las diferencias entre el nuevo snapshot y el snapshot original.



# Capítulo 25. Testing de componentes

El testing de componentes consiste en probar que los componentes se renderizan correctamente y por tanto la funcionalidad que encapsulan funciona como se espera.

Este tipo de pruebas vienen bien ya que sirven:

- De documentación para que cualquier persona que esté en el proyecto sepa como se tiene que comportar el componente o que versiones de este se van a pintar dependiendo de los datos que reciba.
- Además de que nos proporcionan una mayor seguridad a la hora de modificar el componente, ya que si luego no pasan los tests, sabremos que nuestros cambios pueden estar mal hechos.
- Y sobre todo que con ellos vamos a ahorrar tiempo probando manualmente que se comportan como lo tienen que hacer.

Estas pruebas unitarias deberían de ser fáciles de entender, ejecutar y solo deberían de probar una funcionalidad en cada prueba, y de la misma forma en que un usuario interactuaría con el componente.

Los tests de componentes en Cypress todavía no están en una fase estable y por el momento solo se pueden realizar pruebas sobre React ([@cypress/react](#)) y Vue ([@cypress/vue](#)).

## 25.1. Lab: Testing de componentes con React

En este laboratorio vamos a ver como testear unos componentes de React con Cypress y realizar la configuración necesaria para ello.

Empezamos por crear un proyecto de React utilizando la herramienta de **create-react-app** con el siguiente comando:

```
$ npx create-react-app cypress-testing-de-componentes-react-lab
```

Una vez generado el proyecto, vamos a instalar las dependencias necesarias:

```
$ npm install --save-dev cypress @cypress/react
```

Una vez instaladas, nuestro **package.json** quedará de la siguiente forma:

```
{  
  "name": "cypress-testing-de-componentes-react-lab",  
  "version": "0.1.0",  
  "private": true,  
  "dependencies": {  
    "@testing-library/jest-dom": "^5.11.9",  
    "@testing-library/react": "^11.2.5",  
    "@testing-library/user-event": "^12.7.2",  
    "react": "^17.0.1",  
    "react-dom": "^17.0.1",  
    "react-scripts": "4.0.3",  
    "web-vitals": "^1.1.0"  
  },  
  "scripts": {  
    "start": "react-scripts start",  
    "build": "react-scripts build",  
    "test": "react-scripts test",  
    "eject": "react-scripts eject"  
  },  
  "eslintConfig": {  
    "extends": [  
      "react-app",  
      "react-app/jest"  
    ]  
  },  
  "browserslist": {  
    "production": [  
      ">0.2%",  
      "not dead",  
      "not op_mini all"  
    ],  
    "development": [  
      "last 1 chrome version",  
      "last 1 firefox version",  
      "last 1 safari version"  
    ]  
  },  
  "devDependencies": {  
    "@cypress/react": "^5.0.1",  
    "cypress": "^6.5.0"  
  }  
}
```

Después de instalar las dependencias, tenemos que añadir los scripts de NPM que levantan Cypress.

```
{  
  "name": "cypress-testing-de-componentes-react-lab",  
  "version": "0.1.0",  
  "private": true,  
  "dependencies": {  
    "@testing-library/jest-dom": "^5.11.9",  
    "@testing-library/react": "^11.2.5",  
    "@testing-library/user-event": "^12.7.2",  
    "react": "^17.0.1",  
    "react-dom": "^17.0.1",  
    "react-scripts": "4.0.3",  
    "web-vitals": "^1.1.0"  
  },  
  "scripts": {  
    "start": "react-scripts start",  
    "build": "react-scripts build",  
    "test": "react-scripts test",  
    "eject": "react-scripts eject",  
    "cy:open": "cypress open",  
    "cy:run": "cypress run"  
  },  
  "eslintConfig": {  
    "extends": [  
      "react-app",  
      "react-app/jest"  
    ]  
  },  
  "browserslist": {  
    "production": [  
      ">0.2%",  
      "not dead",  
      "not op_mini all"  
    ],  
    "development": [  
      "last 1 chrome version",  
      "last 1 firefox version",  
      "last 1 safari version"  
    ]  
  },  
  "devDependencies": {  
    "@cypress/react": "^5.0.1",  
    "cypress": "^6.5.0"  
  }  
}
```

El siguiente paso es lanzar el comando que abre la ventana de Cypress y nos genera la carpeta de cypress en nuestro proyecto:

```
$ npm run cy:open
```

Antes de continuar tenemos que configurar la dependencia de **@cypress/react** para poder renderizar los componentes de React dentro de la ventana de pruebas de Cypress. Dentro del archivo de **cypress/support/index.js** hay que importar la carpeta **support** de la dependencia.

/cypress-testing-de-componentes-react-lab/cypress/support/index.js

```
// ****
// ...
// ****

// Import commands.js using ES2015 syntax:
import './commands'
import '@cypress/react/support';
```

El siguiente paso es indicarle a cypress cual es la configuración que se encarga de hacer el bundle de la aplicación, para que Cypress pueda cargar correctamente los componentes. Para ello vamos al archivo de **cypress/plugins/index.js** y añadimos dentro de la función el siguiente código.

/cypress-testing-de-componentes-react-lab/cypress/plugins/index.js

```
/// <reference types="cypress" />
// ****
// ...
// ****

/**
 * @type {Cypress.PluginConfig}
 */
module.exports = (on, config) => {
  // `on` is used to hook into various events Cypress emits
  // `config` is the resolved Cypress config
  require('@cypress/react/plugins/react-scripts')(on, config);

  return config;
}
```

Para terminar con la configuración de la dependencia, tenemos que activar dentro del **cypress.json** la funcionalidad experimental del testing de componentes, ademas de indicarle donde se encuentran los tests de componentes que vamos a crear.

/cypress-testing-de-componentes-react-lab/cypress.json

```
{
  "experimentalComponentTesting": true,
  "componentFolder": "cypress/components"
}
```

Y ahora vamos a crear los componentes que vamos a testear. Empezamos por crear una carpeta **components** dentro de **src** donde vamos a crear estos componentes.

Dentro de ella creamos un archivo **Tarea.jsx**, y vamos a añadir el siguiente código.

/cypress-testing-de-componentes-react-lab/src/components/Tarea.jsx

```
import React from 'react';

const Tarea = ({ id, titulo, completada }) => {
  let styles = {
    backgroundColor: completada ? 'gray' : 'white',
    textDecoration: completada ? 'line-through' : 'none',
    margin: '10px',
    padding: '15px',
    border: '1px solid black',
    borderRadius: '5px',
  };

  return (
    <div
      className="tarea"
      style={styles}
      data-cy={'tarea' + id}
    >
      <span>{titulo}</span>
    </div>
  )
}

export default Tarea;
```

Aquí tenemos un componente Tarea que puede tener dos estados:

- No está completada: aparece el título de la tarea.
- Está completada: aparece el título de la tarea tachado y el fondo de color gris.

Así que ya sabemos que dos pruebas tenemos que crear para este componente.

Creamos dentro de **cypress/components** un archivo **Tarea.spec.js**, donde vamos a importar lo necesario para poder ejecutar este test:

- React
- El componente Tarea
- La función mount que se encarga de renderizar el componente que queremos testear

/cypress-testing-de-componentes-react-lab/cypress/components/Tarea.spec.js

```
import React from 'react'
import { mount } from '@cypress/react';
import Tarea from '../../src/components/Tarea.jsx';

describe('Componente Tarea', () => {

});
```

Ahora añadimos el primer caso de prueba (**it**) en el que vamos a comprobar que cuando se le pasa una tarea como propiedad al componente, este se renderiza correctamente mostrandonos el título de la tarea.

/cypress-testing-de-componentes-react-lab/cypress/components/Tarea.spec.js

```
import React from 'react'
import { mount } from '@cypress/react';
import Tarea from '../../src/components/Tarea.jsx';

describe('Componente Tarea', () => {
  it('debería renderizarse el componente Tarea', () => {
    })
});
```

Lo primero que vamos a hacer es crear el objeto de propiedades que se le van a pasar al componente, en el que tendremos los datos de la tarea (id, titulo y completada).

/cypress-testing-de-componentes-react-lab/cypress/components/Tarea.spec.js

```
import React from 'react'
import { mount } from '@cypress/react';
import Tarea from '../../src/components/Tarea.jsx';

describe('Componente Tarea', () => {
  it('debería renderizarse el componente Tarea', () => {
    const propsTarea = {
      id: 283,
      titulo: 'Testear este componente',
      completada: false
    };

    })
});
```

El siguiente paso es renderizar el componente, para lo que vamos a utilizar la función de **mount** que hemos importado antes y le vamos a pasar el componente con las propiedades de la tarea.

```
/cypress-testing-de-componentes-react-lab/cypress/components/Tarea.spec.js
```

```
import React from 'react'
import { mount } from '@cypress/react';
import Tarea from '../../src/components/Tarea.jsx';

describe('Componente Tarea', () => {
  it('debería renderizarse el componente Tarea', () => {
    const propsTarea = {
      id: 283,
      titulo: 'Testear este componente',
      completada: false
    };

    mount(<Tarea {...propsTarea} />);

  })
});
```

El siguiente paso es comprobar que el título de la tarea se muestra en el componente. Para ello usamos lo que ya hemos visto con Cypress para realizar esta comprobación.

```
/cypress-testing-de-componentes-react-lab/cypress/components/Tarea.spec.js
```

```
import React from 'react'
import { mount } from '@cypress/react';
import Tarea from '../../src/components/Tarea.jsx';

describe('Componente Tarea', () => {
  it('debería renderizarse el componente Tarea', () => {
    const propsTarea = {
      id: 283,
      titulo: 'Testear este componente',
      completada: false
    };

    mount(<Tarea {...propsTarea} />);

    cy.get('span').should('have.text', propsTarea.titulo);
  })
});
```

Tenemos el primer test, ahora toca probar que este test se pasa. Vamos a lanzar el comando que abre el panel de Cypress y vamos a pulsar sobre el test de **Tarea.spec.js**.

```
$ npm run cy:open
```

The screenshot shows the Cypress Testing interface. At the top, there are tabs for 'Tests', 'Runs', and 'Settings'. On the right, there are buttons for 'Stop' (red) and 'Running Chrome 88' (green). Below the tabs is a search bar with placeholder text 'Search...'. Underneath, there are sections for 'INTEGRATION TESTS' and 'COMPONENT TESTS'. The 'Tarea.spec.js' file is highlighted with a red border. To the right of the test list, it says 'Running 1 spec'.

Pulsamos sobre el test y se empieza a ejecutar en una nueva ventana.

The screenshot shows a browser window with the title 'cypress-testing-de-componentes-react-lab'. The address bar shows the URL 'http://localhost:49880/\_/#/tests/component/Tarea.spec.js'. The page content is a component testing interface for 'Tarea.spec.js'. It includes a header with test counts (1 green checkmark, 0 red X, 0 yellow circle), execution time (00.81), and a status bar indicating 1000 x 660 (78%). The main area shows the test code structure:

```


    ✓ Componente Tarea
      ✓ debería renderizarse el componente Tarea
        ✓ BEFORE ALL
          1 Coverage Reset [@cypress/code-coverage]
        ✓ TEST BODY
          1 mount <Tarea ... />
          2 get span
          3 -assert expected <span> to have text
            Testear este componente
        ✓ AFTER EACH
          1 log Saving code coverage for
            /__cypress/iframes/component/Tarea
            .spec.js [@cypress/code-coverage]
        ✓ AFTER ALL (1)
          1 log Saving code coverage for unit
            [@cypress/code-coverage]
        ✓ AFTER ALL (2)
          1 Coverage Generating report [@cypress/code-...
    

```

A button labeled 'Testear este componente' is visible on the right side of the interface.

Si se muestra el siguiente error al seleccionar el test del componente: "Error: ENOENT: no such file or directory, stat './VolumeIcon.icns'", vamos a mirar los errores en nuestra terminal. Si dentro de la terminal se muestra algo como "Module not found: Error: Can't resolve 'cypress-react-selector' in ..." entonces tendremos que instalar la dependencia que nos dicen.

En este caso instalamos la dependencia **cypress-react-selector** con el comando **npm install --save-dev cypress-react-selector**.

Ya tenemos nuestro primer test de componente, ahora vamos a por el segundo con el que vamos a comprobar que si la tarea está completada entonces el título de la tarea tiene que aparecer tachado y con el fondo en gris.

Empezamos creando un nuevo caso de prueba.

```
import React from 'react'
import { mount } from '@cypress/react';
import Tarea from '../../src/components/Tarea.jsx';

describe('Componente Tarea', () => {
  it('debería renderizarse el componente Tarea', () => {
    const propsTarea = {
      id: 283,
      titulo: 'Testear este componente',
      completada: false
    };

    mount(<Tarea {...propsTarea} />);

    cy.get('span').should('have.text', propsTarea.titulo);
  })

  it('debería aparecer tachada y con el fondo sombreado si la tarea está completada', () => {
  })
});
```

Ahora vamos a crear las propiedades de la tarea donde el valor de completada lo vamos a poner a **true**, y vamos a montar el componente pasandole estas propiedades al igual que antes.

```
import React from 'react'
import { mount } from '@cypress/react';
import Tarea from '../../../../../src/components/Tarea.jsx';

describe('Componente Tarea', () => {
  it('debería renderizarse el componente Tarea', () => {
    const propsTarea = {
      id: 283,
      titulo: 'Testear este componente',
      completada: false
    };

    mount(<Tarea {...propsTarea} />);

    cy.get('span').should('have.text', propsTarea.titulo);
  })

  it('debería aparecer tachada y con el fondo sombreado si la tarea está completada', () => {
    const propsTarea = {
      id: 283,
      titulo: 'Testear este componente',
      completada: true
    };

    mount(<Tarea {...propsTarea} />);
  })
});
```

Ahora toca añadir las aserciones necesarias. Podemos empezar comprobando que el color del fondo es gris, para lo que necesitamos obtener el div que envuelve a la tarea que es al que se le está añadiendo la propiedad que le da el color gris.

Podemos buscar este elemento de varias formas, pero como le hemos puesto un atributo **data-cy** vamos a buscar por dicho atributo.

En cuanto a la aserción, como necesitamos comprobar una propiedad css, utilizaremos **have.css** pasandole el nombre de la propiedad y el valor esperado.

```
import React from 'react'
import { mount } from '@cypress/react';
import Tarea from '../../../../../src/components/Tarea.jsx';

describe('Componente Tarea', () => {
  it('debería renderizarse el componente Tarea', () => {
    const propsTarea = {
      id: 283,
      titulo: 'Testear este componente',
      completada: false
    };

    mount(<Tarea {...propsTarea} />);

    cy.get('span').should('have.text', propsTarea.titulo);
  })

  it('debería aparecer tachada y con el fondo sombreado si la tarea está completada', () => {
    const propsTarea = {
      id: 283,
      titulo: 'Testear este componente',
      completada: true
    };

    mount(<Tarea {...propsTarea} />);

    cy.get(`[data-cy=tarea${propsTarea.id}]`)
      .should('have.css', 'background-color', 'rgb(128, 128, 128)');
  })
});
```

Si nos fijamos en la pantalla donde se ejecuta el test, deberíamos de ver que se está pasando correctamente.

El siguiente paso es añadir otra aserción que compruebe que aparece tachado el texto, es decir, que contienen **line-through** como valor de la propiedad CSS **text-decoration**.

```
import React from 'react'
import { mount } from '@cypress/react';
import Tarea from '../../../../../src/components/Tarea.jsx';

describe('Componente Tarea', () => {
  it('debería renderizarse el componente Tarea', () => {
    const propsTarea = {
      id: 283,
      titulo: 'Testear este componente',
      completada: false
    };

    mount(<Tarea {...propsTarea} />);

    cy.get('span').should('have.text', propsTarea.titulo);
  })

  it('debería aparecer tachada y con el fondo sombreado si la tarea está completada', () => {
    const propsTarea = {
      id: 283,
      titulo: 'Testear este componente',
      completada: true
    };

    mount(<Tarea {...propsTarea} />);

    cy.get(`[data-cy=tarea${propsTarea.id}]`)
      .should('have.css', 'background-color', 'rgb(128, 128, 128)');

    cy.get(`[data-cy=tarea${propsTarea.id}]`)
      .should('have.css', 'text-decoration', 'line-through')
  })
});
```

Al ejecutar este test, vemos que falla, y esto se debe a que la propiedad **text-decoration** está compuesta por 4 propiedades (line, style, color y thickness), por tanto el valor que espera es el valor de esas 4 propiedades.

Esto podemos solucionarlo indicando exactamente cual de las 4 propiedades es la que tiene que tener como valor **line-through**.

```
import React from 'react'
import { mount } from '@cypress/react';
import Tarea from '../../../../../src/components/Tarea.jsx';

describe('Componente Tarea', () => {
  it('debería renderizarse el componente Tarea', () => {
    const propsTarea = {
      id: 283,
      titulo: 'Testear este componente',
      completada: false
    };

    mount(<Tarea {...propsTarea} />);

    cy.get('span').should('have.text', propsTarea.titulo);
  })

  it('debería aparecer tachada y con el fondo sombreado si la tarea está completada', () => {
    const propsTarea = {
      id: 283,
      titulo: 'Testear este componente',
      completada: true
    };

    mount(<Tarea {...propsTarea} />);

    cy.get(`[data-cy=tarea${propsTarea.id}]`)
      .should('have.css', 'background-color', 'rgb(128, 128, 128)');

    cy.get(`[data-cy=tarea${propsTarea.id}]`)
      .should('have.css', 'text-decoration-line', 'line-through')
  })
});
```

Otra forma de haber resuelto este problema sería encadenando dos comprobaciones, una primera donde se le dice que debería tener la propiedad CSS **text-decoration**, y una segunda en la que le indicamos que dicha propiedad CSS tiene que contener el valor **line-through**.

```
import React from 'react'
import { mount } from '@cypress/react';
import Tarea from '../../../../../src/components/Tarea.jsx';

describe('Componente Tarea', () => {
  it('debería renderizarse el componente Tarea', () => {
    const propsTarea = {
      id: 283,
      titulo: 'Testear este componente',
      completada: false
    };

    mount(<Tarea {...propsTarea} />);

    cy.get('span').should('have.text', propsTarea.titulo);
  })

  it('debería aparecer tachada y con el fondo sombreado si la tarea está completada', () => {
    const propsTarea = {
      id: 283,
      titulo: 'Testear este componente',
      completada: true
    };

    mount(<Tarea {...propsTarea} />);

    cy.get(`[data-cy=tarea${propsTarea.id}]`)
      .should('have.css', 'background-color', 'rgb(128, 128, 128)');

    cy.get(`[data-cy=tarea${propsTarea.id}]`)
      .should('have.css', 'text-decoration-line', 'line-through');

    cy.get(`[data-cy=tarea${propsTarea.id}]`)
      .should('have.css', 'text-decoration')
      .and('contain', 'line-through');
  })
});
```

Ahora ya debería de pasar nuestros tests correctamente, y como nuestro componente Tarea solo tiene estas dos formas de pintarse, ya lo tendríamos testeado.

## 25.2. Lab: Testing de componentes de React con estado

En este laboratorio vamos a ver como testear con Cypress unos componentes de React que tienen estado.

Empezamos por crear un proyecto de React utilizando la herramienta de **create-react-app** con el siguiente comando:

```
$ npx create-react-app cypress-testing-de-componentes-react-con-estado-lab
```

Una vez generado el proyecto, vamos a instalar las dependencias necesarias:

```
$ npm install --save-dev cypress @cypress/react
```

Una vez instaladas, nuestro **package.json** quedará de la siguiente forma:

```
{  
  "name": "cypress-testing-de-componentes-react-con-estado-lab",  
  "version": "0.1.0",  
  "private": true,  
  "dependencies": {  
    "@testing-library/jest-dom": "^5.11.9",  
    "@testing-library/react": "^11.2.5",  
    "@testing-library/user-event": "^12.7.2",  
    "react": "^17.0.1",  
    "react-dom": "^17.0.1",  
    "react-scripts": "4.0.3",  
    "web-vitals": "^1.1.0"  
  },  
  "scripts": {  
    "start": "react-scripts start",  
    "build": "react-scripts build",  
    "test": "react-scripts test",  
    "eject": "react-scripts eject"  
  },  
  "eslintConfig": {  
    "extends": [  
      "react-app",  
      "react-app/jest"  
    ]  
  },  
  "browserslist": {  
    "production": [  
      ">0.2%",  
      "not dead",  
      "not op_mini all"  
    ],  
    "development": [  
      "last 1 chrome version",  
      "last 1 firefox version",  
      "last 1 safari version"  
    ]  
  },  
  "devDependencies": {  
    "@cypress/react": "^5.0.1",  
    "cypress": "^6.5.0"  
  }  
}
```

Después de instalar las dependencias, tenemos que añadir los scripts de NPM que levantan Cypress.

```
{  
  "name": "cypress-testing-de-componentes-react-con-estado-lab",  
  "version": "0.1.0",  
  "private": true,  
  "dependencies": {  
    "@testing-library/jest-dom": "^5.11.9",  
    "@testing-library/react": "^11.2.5",  
    "@testing-library/user-event": "^12.7.2",  
    "react": "^17.0.1",  
    "react-dom": "^17.0.1",  
    "react-scripts": "4.0.3",  
    "web-vitals": "^1.1.0"  
  },  
  "scripts": {  
    "start": "react-scripts start",  
    "build": "react-scripts build",  
    "test": "react-scripts test",  
    "eject": "react-scripts eject",  
    "cy:open": "cypress open",  
    "cy:run": "cypress run"  
  },  
  "eslintConfig": {  
    "extends": [  
      "react-app",  
      "react-app/jest"  
    ]  
  },  
  "browserslist": {  
    "production": [  
      ">0.2%",  
      "not dead",  
      "not op_mini all"  
    ],  
    "development": [  
      "last 1 chrome version",  
      "last 1 firefox version",  
      "last 1 safari version"  
    ]  
  },  
  "devDependencies": {  
    "@cypress/react": "^5.0.1",  
    "cypress": "^6.5.0"  
  }  
}
```

El siguiente paso es lanzar el comando que abre la ventana de Cypress y nos genera la carpeta de cypress en nuestro proyecto:

```
$ npm run cy:open
```

Antes de continuar tenemos que configurar la dependencia de **@cypress/react** para poder renderizar los componentes de React dentro de la ventana de pruebas de Cypress. Dentro del archivo de **cypress/support/index.js** hay que importar la carpeta **support** de la dependencia.

*/cypress-testing-de-componentes-react-con-estado-lab/cypress/support/index.js*

```
// ****
// ...
// *****

// Import commands.js using ES2015 syntax:
import './commands'
import '@cypress/react/support';
```

El siguiente paso es indicarle a cypress cual es la configuración que se encarga de hacer el bundle de la aplicación, para que Cypress pueda cargar correctamente los componentes. Para ello vamos al archivo de **cypress/plugins/index.js** y añadimos dentro de la función el siguiente código.

*/cypress-testing-de-componentes-react-con-estado-lab/cypress/plugins/index.js*

```
/// <reference types="cypress" />
// ****
// ...
// *****

/**
 * @type {Cypress.PluginConfig}
 */
module.exports = (on, config) => {
  // `on` is used to hook into various events Cypress emits
  // `config` is the resolved Cypress config
  require('@cypress/react/plugins/react-scripts')(on, config);

  return config;
}
```

Para terminar con la configuración de la dependencia, tenemos que activar dentro del **cypress.json** la funcionalidad experimental del testing de componentes, ademas de indicarle donde se encuentran los tests de componentes que vamos a crear.

*/cypress-testing-de-componentes-react-con-estado-lab/cypress.json*

```
{
  "experimentalComponentTesting": true,
  "componentFolder": "cypress/components"
}
```

Y ahora vamos a crear los componentes que vamos a testear. Empezamos por crear una carpeta **components** dentro de **src** donde vamos a crear estos componentes.

Dentro de ella creamos un archivo **Contador.jsx**, y vamos a añadir el siguiente código.

/cypress-testing-de-componentes-react-con-estado-lab/src/components/Contador.jsx

```
import React, { useState } from 'react';

const Contador = () => {
  const [cuenta, setCuenta] = useState(0)

  return (
    <div data-cy="contador">
      <button type="button" id="btnDec" onClick={() => setCuenta(cuenta - 1)}>-</button>
      <span id="cuenta">Cuenta: {cuenta}</span>
      <button type="button" id="btnInc" onClick={() => setCuenta(cuenta + 1)}>+</button>
    </div>
  )
}

export default Contador;
```

Aquí tenemos un componente Contador que tiene dos botones y muestra la cuenta actual (que se va guardando en el estado del componente), por tanto, aquí podemos encontrarnos tres casos a probar:

- Inicialmente la cuenta es 0
- Si pulsamos el botón del + la cuenta aumenta
- Si pulsamos el botón del - la cuenta desciende

Sabiendo las pruebas que podemos crear, vamos a ello, y empezamos creando dentro de **cypress/components** un archivo **Contador.spec.js**, donde vamos a importar lo necesario para poder ejecutar este test:

- React
- El componente Contador
- La función `mount` que se encarga de renderizar el componente que queremos testear

/cypress-testing-de-componentes-react-con-estado-lab/cypress/components/Contador.spec.js

```
import React from 'react'
import { mount } from '@cypress/react';
import Contador from '../src/components/Contador.jsx';

describe('Componente Contador', () => {
});
```

Ahora añadimos el primer caso de prueba (**it**) en el que vamos a comprobar que cuando se renderiza el componente la cuenta inicial es 0.

/cypress-testing-de-componentes-react-con-estado-lab/cypress/components/Contador.spec.js

```
import React from 'react'
import { mount } from '@cypress/react';
import Contador from '../../src/components/Contador.jsx';

describe('Componente Contador', () => {
  it('debería mostrar la cuenta a 0 en el primer renderizado del componente', () => {
    });
  });
});
```

Dentro del test vamos a montar primero el componente de Contador, y después vamos a obtener el span que muestra la cuenta para comprobar que contiene un 0.

/cypress-testing-de-componentes-react-con-estado-lab/cypress/components/Contador.spec.js

```
import React from 'react'
import { mount } from '@cypress/react';
import Contador from '../../src/components/Contador.jsx';

describe('Componente Contador', () => {
  it('debería mostrar la cuenta a 0 en el primer renderizado del componente', () => {
    mount(<Contador />);

    cy.get('#cuenta')
      .should('have.text', 'Cuenta: 0');
  });
});
```

Ya tenemos el primer test. Con los siguientes tests vamos a comprobar que el estado del componente cambia según esperamos que lo haga, es decir, que si pulsamos uno de los botones, la cuenta cambia.

Empezamos haciendo el test que comprueba que la cuenta aumenta al pulsar el botón correcto. Ponemos el caso de prueba y montamos el componente al igual que antes.

/cypress-testing-de-componentes-react-con-estado-lab/cypress/components/Contador.spec.js

```
import React from 'react'
import { mount } from '@cypress/react';
import Contador from '../../src/components/Contador.jsx';

describe('Componente Contador', () => {
  it('debería mostrar la cuenta a 0 en el primer renderizado del componente', () => {
    mount(<Contador />);

    cy.get('#cuenta')
      .should('have.text', 'Cuenta: 0');
  });

  it('debería incrementar la cuenta cuando se pulsa sobre el botón del +', () => {
    mount(<Contador />);

  });
});
```

Ahora tenemos que buscar el botón del + para pulsar sobre él. Podemos utilizar el identificador que tiene puesto para acceder de una forma sencilla a él.

/cypress-testing-de-componentes-react-con-estado-lab/cypress/components/Contador.spec.js

```
import React from 'react'
import { mount } from '@cypress/react';
import Contador from '../../src/components/Contador.jsx';

describe('Componente Contador', () => {
  it('debería mostrar la cuenta a 0 en el primer renderizado del componente', () => {
    mount(<Contador />);

    cy.get('#cuenta')
      .should('have.text', 'Cuenta: 0');
  });

  it('debería mostrar la cuenta a 0 en el primer renderizado del componente', () => {
    mount(<Contador />);

    cy.get('#btnInc')
      .click();

  });
});
```

Y por último, comprobamos que el span tiene la cuenta correcta.

```
import React from 'react'
import { mount } from '@cypress/react';
import Contador from '../../src/components/Contador.jsx';

describe('Componente Contador', () => {
  it('debería mostrar la cuenta a 0 en el primer renderizado del componente', () => {
    mount(<Contador />);

    cy.get('#cuenta')
      .should('have.text', 'Cuenta: 0');
  });

  it('debería mostrar la cuenta a 0 en el primer renderizado del componente', () => {
    mount(<Contador />);

    cy.get('#btnInc')
      .click();

    cy.get('#cuenta')
      .should('have.text', 'Cuenta: 1');
  });
});
```

Podemos ejecutar el test en Cypress para ver que de verdad se están pasando correctamente.

Por último, vamos a terminar con el otro test, en este caso para comprobar que la cuenta también se decrementa correctamente.

Esta vez pulsaremos sobre el botón del - y comprobaremos que la cuenta es un -1.

```
import React from 'react'
import { mount } from '@cypress/react';
import Contador from '../../src/components/Contador.jsx';

describe('Componente Contador', () => {
  it('debería mostrar la cuenta a 0 en el primer renderizado del componente', () => {
    mount(<Contador />);

    cy.get('#cuenta')
      .should('have.text', 'Cuenta: 0');
  });

  it('debería mostrar la cuenta a 0 en el primer renderizado del componente', () => {
    mount(<Contador />);

    cy.get('#btnInc')
      .click();

    cy.get('#cuenta')
      .should('have.text', 'Cuenta: 1');
  });

  it('debería decrementar la cuenta cuando se pulsa sobre el botón del -', () => {
    mount(<Contador />);

    cy.get('#btnDec')
      .click();

    cy.get('#cuenta')
      .should('have.text', 'Cuenta: -1');
  });
});
```

Ahora ya deberían de pasarse correctamente los tests del Contador y con ello ya tendríamos testeado el componente con estado.

Pero todavía podemos mejorar los tests, ya que en las 3 pruebas lo primero que hacemos es montar el componente Contador y siempre se renderiza de la misma forma, podemos extraer esta linea al hook del beforeEach de tal forma que dejamos de repetir esta linea de código en todos los tests para tenerla en un único sitio.

```
import React from 'react'
import { mount } from '@cypress/react';
import Contador from '../../src/components/Contador.jsx';

describe('Componente Contador', () => {
  beforeEach(() => {
    mount(<Contador />);
  })

  it('debería mostrar la cuenta a 0 en el primer renderizado del componente', () => {
    cy.get('#cuenta')
      .should('have.text', 'Cuenta: 0');
  });

  it('debería incrementar la cuenta cuando se pulsa sobre el botón del +', () => {
    cy.get('#btnInc')
      .click();

    cy.get('#cuenta')
      .should('have.text', 'Cuenta: 1');
  });

  it('debería decrementar la cuenta cuando se pulsa sobre el botón del -', () => {
    cy.get('#btnDec')
      .click();

    cy.get('#cuenta')
      .should('have.text', 'Cuenta: -1');
  });
});
```

Y con esto ya tendríamos nuestros tests para el componente Contador con estado.

# Capítulo 26. Cypress Studio

Cypress Studio es una herramienta en fase experimental que nos permite **generar los tests de una forma visual** desde el test runner , grabando las interacciones que realizamos sobre los distintos elementos de las aplicaciones.

A partir de estas interacciones se genera código de JavaScript que se incluirá automáticamente en nuestro archivo de test que hemos abierto con Cypress Studio.

De momento, esta herramienta solo graba las siguientes interacciones:

- type
- click
- check/uncheck
- select

Para poder habilitarla tenemos que añadir la propiedad **experimentalStudio** dentro del archivo de configuración de Cypress, **cypress.json**.

## 26.1. Lab: Cypress Studio

En este laboratorio vamos a ver como generar un caso de prueba desde Cypress Studio.

Empezamos creando un proyecto lanzando los siguientes comandos:

```
$ mkdir cypress-cypress-studio-lab  
$ cd cypress-cypress-studio-lab  
$ npm init -y
```

Ahora instalamos la dependencia de Cypress con el siguiente comando:

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

Y vamos a añadir un script de NPM en el archivo **package.json** para abrir el test runner de Cypress y que se genere en el proyecto la carpeta de **cypress** y el archivo de configuración.

/cypress-cypress-studio-lab/package.json

```
{  
  "name": "cypress-cypress-studio-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "cy:open": "cypress open"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.6.0"  
  }  
}
```

Ahora vamos a lanzar el comando que acabamos de añadir para poder empezar a configurar y utilizar Cypress Studio.

```
$ npm run cy:open
```

Ya que se han generado los archivos de Cypress ya podemos empezar con la configuración de Cypress Studio.

Lo primero que vamos a hacer es habilitar la opción experimental de Cypress Studio, para lo que tenemos que añadir en el archivo de configuración de Cypress la propiedad **experimentalStudio**.

/cypress-cypress-studio-lab/cypress.json

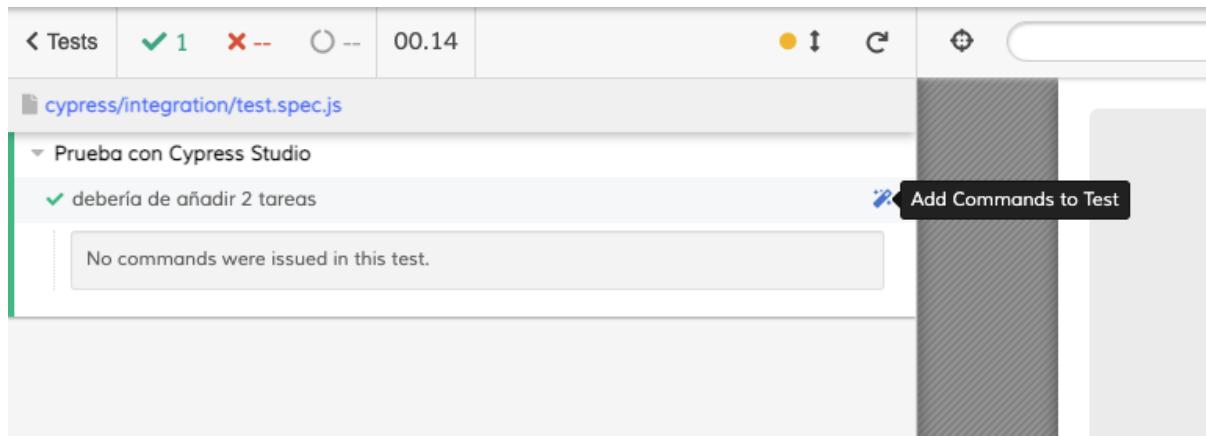
```
{  
  "experimentalStudio": true  
}
```

El siguiente paso es crear un archivo de test con su bloque **describe** y un bloque **it** donde Cypress Studio añadirá el código generado a partir de las interacciones que vayamos a realizar.

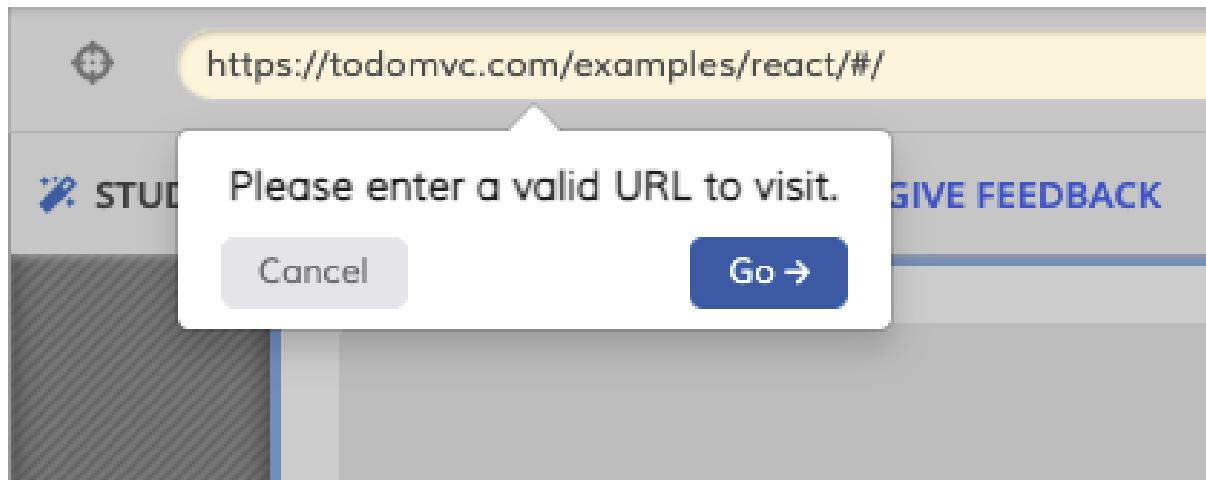
/cypress-cypress-studio-lab/cypress/integration/todomvc.spec.js

```
describe('Prueba con Cypress Studio', () => {  
  it('debería de añadir 2 tareas', () => {  
  })  
});
```

Ya podemos irnos al test runner donde vamos a abrir este archivo de test, y el primer paso va a ser pulsar sobre el botón de **Add Commands to Test**.

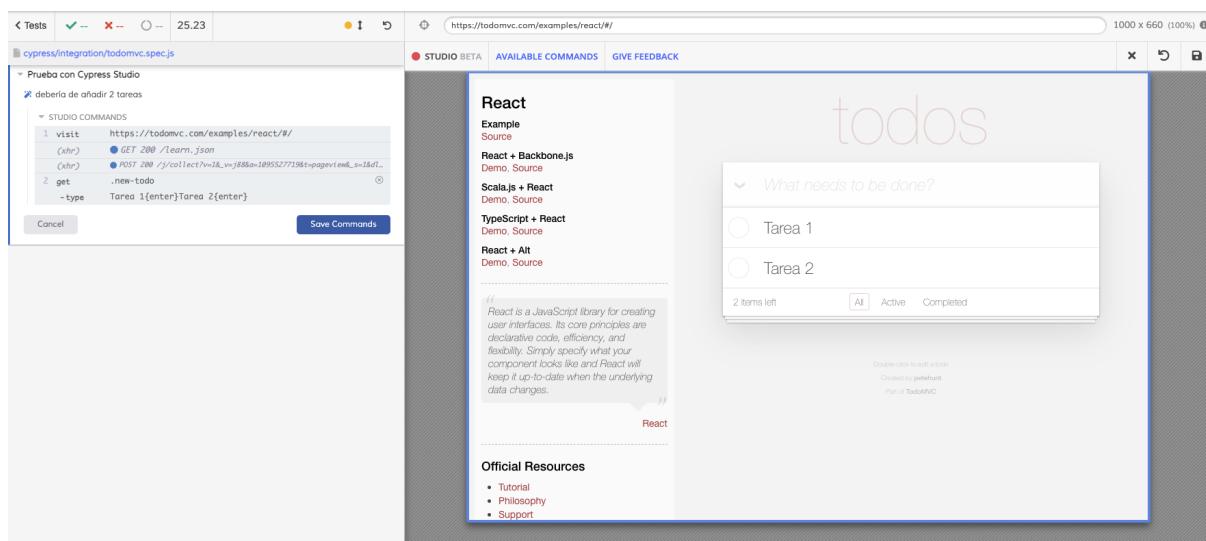


Como no detecta que hayamos puesto en el test la página a visitar para realizar las interacciones sobre ella, nos indica que la añadamos en la barra de URL.



Ahora que ha navegado hasta la página que le hemos indicado, ya deberíamos de ver la página e interactuar con ella. Vamos a añadir dos tareas como lo haríamos nosotros si estuviésemos utilizando la aplicación.

Después de haberlas introducido, vamos a guardar el test, y para ello vamos a pulsar sobre el botón de **Save Commands**.



En este momento ya deberíamos de tener el código en nuestro archivo de test para terminar de rellenarlo con las aserciones necesarias.

/cypress-cypress-studio-lab/cypress/integration/todomvc.spec.js

```
describe('Prueba con Cypress Studio', () => {
  it('debería de añadir 2 tareas', () => {
    /* ===== Generated with Cypress Studio ===== */
    cy.visit('https://todomvc.com/examples/react/#/');
    cy.get('.new-todo').type('Tarea 1{enter}Tarea 2{enter}');
    /* ===== End Cypress Studio ===== */
  })
});
```

Vamos a añadir una aserción para comprobar que se han creado correctamente las 2 tareas.

/cypress-cypress-studio-lab/cypress/integration/todomvc.spec.js

```
describe('Prueba con Cypress Studio', () => {
  it('debería de añadir 2 tareas', () => {
    /* ===== Generated with Cypress Studio ===== */
    cy.visit('https://todomvc.com/examples/react/#/');
    cy.get('.new-todo').type('Tarea 1{enter}Tarea 2{enter}');
    /* ===== End Cypress Studio ===== */

    cy.get('ul.todo-list')
      .children()
      .should('have.length', 2);
  })
});
```

Y de esta forma podemos utilizar Cypress Studio para generar los tests de una forma más rápida, aunque luego deberemos de modificarlos para añadir aserciones o comandos más complejos que todavía no se pueden utilizar.

# Capítulo 27. Dashboard

**Cypress Dashboard** es un servicio que nos proporciona una serie de información que va guardando durante la ejecución de nuestros tests, sobre todo cuando estos se ejecutan en entornos de integración continua, donde los equipos no están pendientes de ejecutar estos tests, sino que todo se hace de forma automática.

Las funcionalidades que nos proporciona este servicio son:

- Gestionar las ejecuciones de los tests y ver sus resultados.
- Nos muestra analíticas.
- Organizar los proyectos.
- Gestionar los usuarios.
- Se puede integrar con otras herramientas como Slack o Github.

Este servicio tiene una versión gratuita que nos permite ejecutar hasta 500 tests al mes. Luego tenemos las versiones de pago que nos dan un margen mucho mayor.

## 27.1. Organizaciones

Dentro del Dashboard de Cypress podemos crear organizaciones que es la forma de agrupar los proyectos para gestionar el acceso a estos.

Dentro de una organización podemos ver los proyectos, los usuarios, las integraciones, la facturación y el uso de este servicio.

Podemos crear nuevas organizaciones al pulsar sobre el icono de nuestro perfil en el panel lateral. En el menú que se despliega aparece un botón **Create new organization....**

Los datos que nos pide para crear una nueva organización son:

- Nombre de la organización
- Página web de la organización
- Es un proyecto personal

Después de añadir esta información podemos añadir los miembros que van a trabajar en ella.

Desde el menú lateral podemos ver los ajustes de las configuraciones en **Organization settings**. Donde encontramos información como:

- El nombre.
- La web de la organización o si esta organización es para un proyecto personal.
- El logo.
- El identificador (se usa para temas de facturación y soporte).
- Botón para eliminar la organización.

## 27.2. Proyectos

Dentro de una organización tenemos proyectos o podemos crearlos en caso de no tener ninguno todavía. Podemos ver todos ellos pulsando sobre **Projects**, o podemos ir directamente a uno en concreto pulsando sobre el de la lista que hay justo debajo de la opción Projects en el menú lateral.

Dentro de cada proyecto podemos ver toda la información que se recoge de la ejecución de los tests de un proyecto. Tenemos distintas secciones:

- **Latest runs:** aquí encontramos las últimas ejecuciones que se han realizado de los tests, y podemos ver:
  - Un resumen de como ha salido la ejecución ya que se muestra el número de tests que se han pasado, los que han fallado, el tiempo que ha tardado la ejecución...
  - También tenemos unos filtros que nos pueden ayudar a buscar ciertas ejecuciones pasadas, que salieron mal...
- **Analytics:** en esta sección podemos ver analíticas de las ejecuciones de los tests.
- **Project settings:** aquí encontramos la información y ajustes del proyecto como:
  - El nombre.
  - El identificador (se pone en **cypress.json** para saber a que proyecto pertenece la ejecución de los tests).
  - La visibilidad del proyecto.
  - Las claves (se añaden en el comando de ejecución y le indica al proyecto que puede grabar la ejecución de los tests en el Dashboard).
  - Las integraciones con otras plataformas (Github, Gitlab, Slack).
  - Y tenemos los botones para cambiar de propietario de proyecto, o eliminar el proyecto.

The screenshot shows a section titled "Waiting on your first run..." with the sub-instruction "To record your first run, complete these 2 steps." Step 1, "Insert ProjectID into your cypress.json file", shows a code snippet with a placeholder for a project ID: { "projectId": "z [REDACTED]" }. Step 2, "Run cypress while passing the record key", shows a terminal command: "cypress run --record --key 31t". Below the terminal is a link "Learn more about using record keys".

## 27.3. Ejecuciones

Desde el servicio de Cypress Dashboard podemos ver en detalle la información de cada ejecución de los tests que se ha llevado a cabo.

Entre la información que se muestra podemos ver:

- Los tests pasados, pendientes, saltados y con errores.
- La versión de Cypress.
- El navegador y el SO donde se han ejecutado los tests.
- Información del repositorio asociado a los tests ejecutados.

Si entramos al detalle de las ejecuciones podremos ver los **logs de la consola**, los **vídeos** y **pantallazos** que se generan por cada uno de los tests para ver si todo ha ido como se esperaba y sobre todo para ver que ha ido mal cuando ha fallado algún test.

 examples/actions.spec.js <span>New</span>	 14	⌚ 00:27	  
 examples/aliasing.spec.js <span>New</span>	 2	⌚ 00:03	  
 examples/assertions.spec.js <span>New</span>	 9	⌚ 00:05	  

## 27.4. Usuarios

Dentro de la sección de una organización nos encontramos con una pestaña de **usuarios** en la que podemos añadir nuevos usuarios a la organización que tenemos creada en Cypress Dashboard.

Al entrar en **Usuarios** podemos ver todos los usuarios que están dentro de la organización y que por tanto pueden gestionarla de una forma u otra. Entre estos usuarios podemos encontrar 3 tipos o 3 roles:

- **Member**: puede ver los proyectos, las ejecuciones y las claves de la organización.
- **Admin**: a parte de los permisos que tiene el **member**, puede gestionar usuarios y los pagos de la organización.
- **Owner**: a parte de los permisos que tiene el **admin**, puede transferir o eliminar proyectos y eliminar la organización.

Podemos añadir más usuarios desde el botón de **Invite User** que tenemos en la esquina superior derecha. Solo nos pide:

- **Email** del usuario al que queremos invitar.
- **Rol** que le vamos a dar al usuario.

## Users

USER	PROVIDER	ROLE	
Álvaro alv[REDACTED]	Password	Member	<a href="#">Leave</a>
Angel and[REDACTED]	Password	Owner	

## 27.5. Analíticas

Dentro de los proyectos tenemos una sección de **analíticas** en la que podemos ver una serie de métricas que nos pueden dar una idea de las ejecuciones de los tests, como por ejemplo:

- **Estado:** muestra el número de ejecuciones que se han grabado con este servicio mostrando tanto los tests que se han pasado como los que han fallado.
- **Duración media:** aquí vemos la duración media de la ejecución de los tests del proyecto. Este valor solo se calcula con los tests que se pasan correctamente.
- **Tamaño de la suite:** nos muestra como va creciendo a lo largo del tiempo nuestra suite de tests.
- **Top de fallos:** aquí veremos los tests que más han dado errores a la hora de ejecutarlos.
- **Tests más lentos:** muestra los tests que tardan más tiempo en ejecutarse dentro de una suite de tests.
- **Errores más comunes:** este servicio es capaz de detectar los tipos de errores que nos dan cuando se ejecutan los tests, y podemos ver cuales son los que más se repiten dentro de esta sección.

## 27.6. Integraciones

Cypress Dashboard nos permite integrar la ejecución de nuestros tests con distintas herramientas como:

- Slack
- Github
- Gitlab
- Bitbucket
- Jira

Al integrarlo con estas herramientas conseguimos que se nos muestre la información la ejecución de los tests por mensaje a algún canal de Slack o en una pull request en los repositorios donde se encuentran los tests. También podríamos crear una issue en Jira desde la propia ejecución de un test fallido.

# Capítulo 28. Generadores de informes

Cypress nos permite generar informes con los resultados de la ejecución de los tests, y al estar construido sobre Mocha ya tiene disponibles todas aquellas opciones que podemos usar con él. Entre ellas tenemos:

- **spec**: es la opción que se usa por defecto y nos muestra la lista de tests con un símbolo cuando el test se pasa, y otro símbolo distinto cuando falla.
- **nyan**: nos muestra un gato que deja una estela de colores según se van pasando los tests.
- **progress**: nos va mostrando una barra de progreso en el terminal mientras se van ejecutando los tests.
- **json**: nos muestra un json con la información del informe obtenida durante la ejecución de los tests.
- ...

Pero a parte de los que vienen por defecto y que se van mostrando por terminal sin generar un archivo con la información, nosotros podemos indicarle a Cypress que utilice otras librerías generadoras de informes como:

- **mochawesome** que nos genera una página HTML con todos los resultados obtenidos tras la ejecución de los tests.
- **junit** que nos genera un archivo XML al estilo de JUnit, con todos los resultados obtenidos tras la ejecución de los tests.

## 28.1. Lab: generadores de informes

En este laboratorio vamos a ver como generar los informes de la ejecución de nuestros tests, tanto los que vienen por defecto con mocha, como uno externo .

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-generadores-de-informes-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-generadores-de-informes-lab  
$ cd cypress-generadores-de-informes-lab  
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir una serie de scripts.

Empezamos por instalar las dependencias necesarias:

- **cypress**: la librería de tests
- **http-server**: servidor de desarrollo local

```
$ npm install --save-dev cypress http-server
```

Ahora añadimos los dos scripts que vamos a utilizar después dentro del archivo **package.json**.

*/cypress-generadores-de-informes-lab/package.json*

```
{
  "name": "cypress-generadores-de-informes-lab",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "http-server",
    "cy:open": "cypress open"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "cypress": "^6.6.0",
    "http-server": "^0.12.3"
  }
}
```

El siguiente paso es crear un archivo **index.html** donde vamos a poner el contador que vamos a testear.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Generador de informes</title>
  <style>
    body {
      height: 100vh;
      width: 100vw;
      background-color: #242424;
      color: white;
    }

    .center {
      height: 100%;
      display: flex;
      justify-content: center;
      align-items: center;
    }

    button {
      background-color: black;
      width: 50px;
      color: white;
      border: 1px solid white;
      border-radius: 5px;
      padding: 10px;
      margin: 0 20px;
      display: block;
    }

    button:hover {
      border: 1px solid #999999;
      color: #999999;
    }
  </style>
</head>
<body>
  <div class="center">
    <button type="button" id="btnDec">-</button>
    <span id="cuenta">0</span>
    <button type="button" id="btnInc">+</button>
  </div>

  <script src="app.js"></script>
</body>
</html>
```

Ahora añadimos el código de JavaScript del archivo **app.js** que se está importando en nuestra página.

/cypress-generadores-de-informes-lab/app.js

```
const btnDec = document.getElementById('btnDec');
const btnInc = document.getElementById('btnInc');
const cuentaElem = document.getElementById('cuenta');

btnDec.addEventListener('click', () => {
  cuentaElem.innerText = Number(cuentaElem.innerText) - 1;
})

btnInc.addEventListener('click', () => {
  cuentaElem.innerText = Number(cuentaElem.innerText) + 1;
})
```

El siguiente paso es crear la carpeta de cypress donde vamos a añadir nuestro test. Para esto, solo tenemos que lanzar el script de cypress que hemos añadido antes.

```
$ npm run cy:open
```

Además, en otra terminal, vamos a lanzar el otro script para poder acceder con Cypress a la web.

```
$ npm start
```

Como nos indica en la terminal, podemos acceder a la web en <http://localhost:8080/>.

Ahora que tenemos la carpeta de **cypress** en nuestro proyecto, vamos a crear un archivo **contador.spec.js** dentro de la carpeta **integration**.

Añadimos nuestro caso de prueba y visitamos la página donde se encuentra lo que queremos testear.

/cypress-generadores-de-informes-lab/cypress/integration/contador.spec.js

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

describe('Contador', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080');
  })

})
```

Lo primero que vamos a comprobar es que la cuenta empieza en 0.

/cypress-generadores-de-informes-lab/cypress/integration/contador.spec.js

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

describe('Contador', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080');
  })

  it('debería empezar con la cuenta a 0', () => {
    cy.get('#cuenta')
      .should('have.text', 0);
  })
})
```

Ahora vamos a añadir un segundo caso de prueba en el que comprobaremos que si pulsamos sobre el botón del -, la cuenta se decrementa en 1.

/cypress-generadores-de-informes-lab/cypress/integration/contador.spec.js

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

describe('Contador', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080');
  })

  it('debería empezar con la cuenta a 0', () => {
    cy.get('#cuenta')
      .should('have.text', 0);
  })

  it('debería decrementar la cuenta', () => {
    cy.get('#btnDec')
      .click();

    cy.get('#cuenta')
      .should('have.text', -1);
  })
})
```

El último test es justo el contrario, comprobar que se incrementa la cuenta cuando se pulsa sobre el +.

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

describe('Contador', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080');
  })

  it('debería empezar con la cuenta a 0', () => {
    cy.get('#cuenta')
      .should('have.text', 0);
  })

  it('debería decrementar la cuenta', () => {
    cy.get('#btnDec')
      .click();

    cy.get('#cuenta')
      .should('have.text', -1);
  })

  it('debería incrementar la cuenta', () => {
    cy.get('#btnInc')
      .click();

    cy.get('#cuenta')
      .should('have.text', '1');
  })
})
```

Una vez que tenemos los tests, probamos a ejecutarlos para ver que todos se pasan correctamente.

Si es así, vamos a configurar el proyecto para poder generar informes con los resultados de los tests.

Como Cypress utiliza Mocha, podemos utilizar los reportes de Mocha sin necesidad de añadir más librerías ni configurar nada añadiéndole a nuestro comando de Cypress el tipo de reporte que queremos generar.

Vamos a añadir un nuevo script en nuestro **package.json** para lanzar el comando de **cypress run** pero mostrando por la consola un informe de la ejecución de los tests. Para mostrar por consola un informe hay que añadirle la opción **--reporter** seguida del tipo de informe que queremos mostrar.

```
{  
  "name": "cypress-generadores-de-informes-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "http-server",  
    "cy:open": "cypress open",  
    "cy:rep-spec": "cypress run --reporter spec"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.6.0",  
    "http-server": "^0.12.3"  
  }  
}
```

Ahora podemos probar a lanzar este nuevo script para ver que información se muestra:

```
$ npm run cy:rep-spec
```

Al terminar la ejecución de los tests, podremos ver por la consola el siguiente informe:

```
npm run cy:rep-spec  
  
> cypress-generadores-de-informes-lab@1.0.0 cy:rep-spec /cypress-generadores-de-informes-lab  
> cypress run --reporter spec  
  
=====  
  
(Run Starting)  
  
[  
  | Cypress: 6.6.0  
  | Browser: Electron 87 (headless)  
  | Specs: 1 found (contador.spec.js)  
]  
  
[  
]  
  
[  
]  
  
Running: contador.spec.js  
          (1 of 1)  
  
Contador  
  ✘ debería empezar con la cuenta a 0 (135ms)  
  ✘ debería decrementar la cuenta (157ms)  
  ✘ debería incrementar la cuenta (170ms)
```

3 passing (1s)

(Results)

```
| Tests:      3
| Passing:    3
| Failing:   0
| Pending:   0
| Skipped:   0
| Screenshots: 0
| Video:     true
| Duration:  1 second
| Spec Ran:  contador.spec.js
```

(Video)

- Started processing: Compressing to 32 CRF
  - Finished processing: /cypress-generadores-de-informes-lab/cypress/videos/contador.spec.mp4 (1 second)
- 

(Run Finished)

Spec

Tests Passing Failing Pending Skipped

```
| 0 contador.spec.js          00:01      3      3      -      -      -      - |
```

```
| All specs passed!          00:01      3      3      -      -      -      - |
```

Este informe que acabamos de sacar, es el que se utiliza por defecto cuando lanzamos el comando **cypress run**, así que no era necesario de añadirle la opción anterior porque sin añadirle nada Cypress ya nos estaba generando estos informes.

Ahora si que vamos a cambiar el tipo de informe por otro de los que vienen con Mocha. Esta vez añadiremos otro comando para mostrar por el terminal el informe con la opción de **nyan**.

## /cypress-generadores-de-informes-lab/package.json

```
{  
  "name": "cypress-generadores-de-informes-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "http-server",  
    "cy:open": "cypress open",  
    "cy:rep-spec": "cypress run --reporter spec",  
    "cy:rep-nyan": "cypress run --reporter nyan"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.6.0",  
    "http-server": "^0.12.3"  
  }  
}
```

Lanzamos desde el terminal el nuevo script:

```
$ npm run cypress:open
```

Y cuando termina se nos muestra un informe por consola como el siguiente:

```
npm run cy:rep-ryan

> cypress-generadores-de-informes-lab@1.0.0 cy:rep-spec /cypress-generadores-de-informes-lab
> CYPRESS_E2E --reporter:ryan
```

(Run Starting)

```
| Cypress:    6.6.0  
| Browser:   Electron 87 (headless)  
| Specs:     1 found (contador.spec.js)
```

Running: contador.spec.js

(1 of 1)

```
3 passing (2s)
```

(Results)

```
| Tests:      3
| Passing:    3
| Failing:   0
| Pending:   0
| Skipped:   0
| Screenshots: 0
| Video:     true
| Duration:  1 second
| Spec Ran:  contador.spec.js
```

(Video)

- Started processing: Compressing to 32 CRF
- Finished processing: /cypress-generadores-de-informes-lab/cypress/videos/contador.spec.mp4 (1 second)

(Run Finished)

Spec	Tests	Passing	Failing	Pending	Skipped
------	-------	---------	---------	---------	---------

④ contador.spec.js	00:01	3	3	-	-	-
--------------------	-------	---	---	---	---	---

④ All specs passed!	00:01	3	3	-	-	-
---------------------	-------	---	---	---	---	---

Podemos ir cambiando el tipo de informes en los scripts de NPM, pero ahora vamos a utilizar una de las librerías más utilizadas para generar informes de testing con Mocha que no solo nos muestra datos por la consola, sino que también nos genera un informe en HTML con los resultados.

Esta librería es **Mochawesome**.

Antes de nada, vamos a instalar esta dependencia en nuestro proyecto lanzando el siguiente comando:

```
$ npm install --save-dev mochawesome
```

Utilizar esta librería es tan sencillo como poner el nombre como valor del **--reporter** y ya está. Así que vamos a añadir un nuevo script de NPM con esta nueva opción.

```
{  
  "name": "cypress-generadores-de-informes-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "http-server",  
    "cy:open": "cypress open",  
    "cy:run": "cypress run",  
    "cy:rep-spec": "cypress run --reporter spec",  
    "cy:rep-nyan": "cypress run --reporter nyan",  
    "cy:rep-mochawesome": "cypress run --reporter mochawesome"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "cypress": "^6.6.0",  
    "http-server": "^0.12.3",  
    "mochawesome": "^6.2.2"  
  }  
}
```

Ahora lanzamos el comando añadido:

```
$ npm run cy:rep-mochawesome
```

Y cuando termina de ejecutarse deberíamos de encontrar una carpeta **mochawesome-report** en la raíz del proyecto, donde encontraremos la información de los tests con sus resultados en un archivo HTML y en un archivo JSON.

Si abrimos el archivo HTML, nos encontramos con un informe como el que podemos ver a continuación:



# Capítulo 29. Cobertura de código

Podemos tener muchos tests creados que prueban nuestras aplicaciones, pero si no medimos cuánto código se cubre con estos tests no sabremos si es necesario generar más tests o si los que hay están pasando por el código de aquellas funcionalidades más importantes de la aplicación.



Ojo, que la cobertura de código nos indique que los tests pasan por todas las líneas que hay en nuestra aplicación, no significa que nuestra aplicación funcione correctamente, quizás se están probando unas funcionalidades que pasan por todo el código de la aplicación, pero hay alguna que no se ha llegado a testear.

Esta medida nos puede ayudar a ver si necesitamos generar más tests o con los que hay generados es suficiente por el momento.

Por ejemplo, al lanzar nuestros tests y ver los resultados de la cobertura de código, podríamos ver que una función de las más importantes de la aplicación no se cubre con los tests que tenemos hasta ahora. Esto nos estaría indicando que tenemos que añadir algún test que cubra esa parte del código.

Para poder llevar a cabo esta medición, primero hay que **instrumentar** nuestro código. Esto consiste en coger nuestro código y generar un código equivalente en el que se añaden una serie de contadores que van a ir incrementándose cada vez que nuestros tests pasen por ahí.

Esta tarea de instrumentación se puede realizar utilizando el plugin de **istanbul** con **Babel**. Otra herramienta que nos permite hacer esto es **NYC**.

Una vez instrumentado el código, al lanzar los tests se irán aumentando los contadores según la ejecución pasa por las líneas de nuestro código, y al terminar se generan los archivos con la información recogida.

Para poder recoger esta información al lanzar los tests de Cypress, necesitamos instalar el siguiente plugin:

```
$ npm install --save-dev @cypress/code-coverage
```

Y también hay que añadir el siguiente código en los archivos indicados:

```
/cypress/support/index.js
```

```
import '@cypress/code-coverage/support'
```

/cypress/plugins/index.js

```
module.exports = (on, config) => {
  require('@cypress/code-coverage/task')(on, config)

  return config
}
```

Una vez configurado todo, y lanzados los tests, deberíamos de ver una carpeta **.nyc\_output** con la información recogida guardada en un JSON y que se puede utilizar para generar los informes. Además el plugin anterior genera automáticamente el informe en **coverage/code-coverage**.

## 29.1. Lab: Cobertura de código

En este laboratorio vamos a ver como sacar un informe con el código de la aplicación que están cubriendo nuestros tests.

Vamos a empezar por crear el proyecto, y para ello vamos a crear una carpeta **cypress-cobertura-de-código-lab** y dentro inicializaremos un proyecto de NPM. Para esto vamos a lanzar los siguientes comandos:

```
$ mkdir cypress-cobertura-de-código-lab
$ cd cypress-cobertura-de-código-lab
$ npm init -y
```

Una vez lanzados ya deberíamos de tener un proyecto de NPM inicializado con el archivo **package.json** que va a almacenar las dependencias necesarias y donde vamos a añadir unos scripts.

Empezamos por instalar las dependencias necesarias:

- cypress: la librería de tests
- @cypress/code-coverage: plugin de cypress para generar los informes de la cobertura de código.
- nyc: herramienta para instrumentar el código
- http-server: servidor de desarrollo local

```
$ npm install --save-dev cypress http-server nyc @cypress/code-coverage
```

Ahora añadimos los scripts que vamos a utilizar después dentro del archivo **package.json**.

```
{  
  "name": "cypress-cobertura-de-codigo-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "http-server",  
    "cy:open:cov": "nyc --reporter=lcov cypress open",  
    "cy:run:cov": "nyc --reporter=lcov cypress run",  
    "instrument": "nyc instrument --compact=false . instrumented"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "@cypress/code-coverage": "^3.9.4",  
    "cypress": "^6.8.0",  
    "http-server": "^0.12.3",  
    "nyc": "^15.1.0"  
  },  
  "nyc": {  
    "include": [  
      "app.js"  
    ]  
  }  
}
```

El siguiente paso es crear un archivo **index.html** donde vamos a poner el HTML del contador que vamos a testear.

/cypress-cobertura-de-codigo-lab/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Cobertura de código</title>
</head>
<body>
  <div>
    <button type="button" id="btnDec">-</button>
    <span id="cuenta">0</span>
    <button type="button" id="btnInc">+</button>
  </div>

  <script src="app.js"></script>
</body>
</html>
```

Ahora vamos a crear el archivo **app.js** que estamos importando en nuestra página y que contiene el código que va modificando la cuenta según vamos pulsando sobre los botones.

/cypress-cobertura-de-codigo-lab/app.js

```
const btnDec = document.getElementById('btnDec');
const btnInc = document.getElementById('btnInc');
const cuentaElem = document.getElementById('cuenta');

btnDec.addEventListener('click', () => {
  cuentaElem.innerText = Number(cuentaElem.innerText) - 1;
});

btnInc.addEventListener('click', () => {
  cuentaElem.innerText = Number(cuentaElem.innerText) + 1;
});
```

Una vez tenemos los archivos ya podemos lanzar el script que levanta el servidor de desarrollo y nos sirve la página anterior.

```
$ npm start
```

Como nos indica en la terminal, podemos acceder a la web en <http://localhost:8080/>.

Ahora vamos a configurar **nyc** para indicarle que archivos tiene que instrumentar. Esta tarea consiste en añadir sobre nuestro código una serie de instrucciones que llevan la cuenta de cuantas veces se ejecuta cada instrucción.

Lo primero de todo es añadir la propiedad **nyc** dentro del **package.json** y decirle que incluya nuestro archivo **app.js**.

/cypress-cobertura-de-codigo-lab/package.json

```
{  
  "name": "cypress-cobertura-de-codigo-lab",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "start": "http-server",  
    "cy:open:cov": "nyc --reporter=lcov cypress open",  
    "cy:run:cov": "nyc --reporter=lcov cypress run",  
    "instrument": "nyc instrument --compact=false . instrumented"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "devDependencies": {  
    "@cypress/code-coverage": "^3.9.4",  
    "cypress": "^6.8.0",  
    "http-server": "^0.12.3",  
    "nyc": "^15.1.0"  
  },  
  "nyc": {  
    "include": [  
      "app.js"  
    ]  
  }  
}
```

Y ahora que ya tenemos la configuración necesaria, vamos a lanzar el script de **instrument**.

```
$ npm run instrument
```

Al lanzar el script se genera una carpeta **instrumented** en la raíz del proyecto donde se encuentra el código con los contadores que van a medir la cobertura.

También tenemos que configurar el plugin de **@cypress/code-coverage**, así que abriremos el archivo **\*\*** y vamos a añadir la siguiente importación:

/cypress-cobertura-de-codigo-lab/cypress/support/index.js

```
// ****
// ...
// ****

// Import commands.js using ES2015 syntax:
import './commands'

// Alternatively you can use CommonJS syntax:
// require('./commands')
import '@cypress/code-coverage/support'
```

También tenemos que añadir el siguiente código en **cypress/plugins/index.js**

/cypress-cobertura-de-codigo-lab/cypress/support/index.js

```
/// <reference types="cypress" />
// ****
// ...
// ****

/**
 * @type {Cypress.PluginConfig}
 */
// eslint-disable-next-line no-unused-vars
module.exports = (on, config) => {
  // `on` is used to hook into various events Cypress emits
  // `config` is the resolved Cypress config
  require('@cypress/code-coverage/task')(on, config);
  return config;
}
```

Una vez tenemos el código instrumentado y el plugin configurado, vamos a crear nuestro archivo de test **contador.spec.js** dentro de la carpeta **cypress/integration** del proyecto.

Crearemos tres tests, uno para comprobar que la cuenta empieza en 0, otro para comprobar que se incrementa y el otro para comprobar que se decrementa.

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

describe('Contador', () => {
  beforeEach(() => {
    cy.visit('http://localhost:8080');
  })

  it('debería empezar con la cuenta a 0', () => {
    cy.get('#cuenta')
      .should('have.text', '0');
  })

  it('debería decrementar la cuenta', () => {
    cy.get('#btnDec')
      .click();

    cy.get('#cuenta')
      .should('have.text', '-1');
  })

  it('debería incrementar la cuenta', () => {
    cy.get('#btnInc')
      .click();

    cy.get('#cuenta')
      .should('have.text', '1');
  })
})
```

Ahora ya podemos lanzar cualquiera de los dos scripts de ejecución que tenemos en el **package.json**.

```
$ npm run cy:run:cov
```

Cuando termina de ejecutarse deberíamos de ver una carpeta **coverage/lcov-report** en la raíz del proyecto donde deberíamos de encontrar un **index.html**. Si lo abrimos en el navegador veremos algo parecido a lo que vemos en la siguiente imagen:

## All files

Unknown% Statements 0/0 Unknown% Branches 0/0 Unknown% Functions 0/0 Unknown% Lines 0/0

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File ▾    Statements ▾    Branches ▾    Functions ▾    Lines ▾

Pero, ¿por qué aparece la página sin información? Esto se debe a que no hemos cargado nuestros archivos instrumentados que son los que al ejecutarse van a ir guardando los datos de la cobertura. Por tanto, podemos solucionarlo cambiando la importación en el **index.html** del archivo **app.js** por el que se ha generado dentro de la carpeta **instrumented**.

/cypress-cobertura-de-codigo-lab/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Cobertura de código</title>
</head>
<body>
  <div>
    <button type="button" id="btnDec">-</button>
    <span id="cuenta">0</span>
    <button type="button" id="btnInc">+</button>
  </div>

  <script src="instrumented/app.js"></script>
</body>
</html>
```

Ahora ya podemos volver a lanzar el comando anterior para ejecutar los tests, y tras hacerlo si volvemos a entrar en la página con el informe de la cobertura deberíamos de ver algo parecido a lo que se muestra a continuación.

## All files

100% Statements 7/7 100% Branches 0/0 100% Functions 2/2 100% Lines 7/7

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

File	Statements	Branches	Functions	Lines
app.js	100%	7/7	100%	100%

Si pulsamos sobre el nombre del archivo de **app.js** veremos por que líneas de código se ha pasado al ejecutar los tests y por cuales no.

## All files app.js

100% Statements 7/7 100% Branches 0/0 100% Functions 2/2 100% Lines 7/7

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
1 4x const btnDec = document.getElementById('btnDec');
2 4x const btnInc = document.getElementById('btnInc');
3 4x const cuentaElem = document.getElementById('cuenta');
4
5 4x btnDec.addEventListener('click', () => {
6 1x   cuentaElem.innerText = Number(cuentaElem.innerText) - 1;
7 });
8
9 4x btnInc.addEventListener('click', () => {
10 1x   cuentaElem.innerText = Number(cuentaElem.innerText) + 1;
11 });
12
```

Ahora mismo nuestros tests pasan por todas las líneas, pero podemos comentar algún test para comprobar como se muestra cuando hay código no cubierto por estos.