Pruebas funcionales con Cypress

Cypress, instalación, configuración y ejecución

> *"La calidad no es un acto, es un hábito"*
>
> -- ✍ **Aristóteles**

La mínima garantía de calidad es que algo funcione conforme a lo esperado; es decir que pase un test funcional. Pero desde antiguo se sabe que esto sólo es válido si es constante. Para que se convierta en hábito nada mejor que hacerlo **sencillo, cómodo y agradable.**

Las pruebas E2E pueden, y deben, incorporar comprobaciones funcionales. Hasta ahora habíamos visto situaciones muy básica en las que una herramienta libre y gratuita como *Puppeteer* se defendía muy bien.

Pero, cuando probamos funcionalidades complejas se nos complica el uso continuado de la sintaxis asíncrona. Incluso con el moderno *async/await*. Tampoco *Puppeteer* trae de fábrica funciones adaptadas al testing. Se le tienen que añadir ayudas como la Integración con Jest, o con otros frameworks de testing como *Mocha* o *Jasmine*.

Nada de esto contribuye a integrar el hábito de la prueba funcional constante. Y **sin ese hábito no hay calidad**.

Cypress

Cypress es un framework para pruebas funcionales *e2e*. En este sentido es mucho más potente y cómodo de utilizar que *Puppeteer* que no deja de ser un automatizador de navegadores. El fallo es que... *Cypress* es de pago; pero lo bueno es que tiene **una buena parte gratis que es más suficiente** para las pruebas funcionales.

Y desde luego es de lo más sencillo, cómodo y agradable de usar. Ideal para convertirlo en hábito de uso.

Instalar Cypress

Instalarlo es trivial. Partimos de una aplicación *Node* y agregamos la dependencia. Puedes hacerlo con *npm* o con *yarn* y guardarlo como dependencia principal o para desarrollo.

```
yarn add cypress
npm i --save cypress
```

Este proceso tarda un poco porque *Cypress* es mucho más que una librería. Es un *test runner* que necesita su propio servidor y controlador del navegador. La espera merecerá la pena porque además nos genera una estructura de carpetas para que empecemos con nuestros tests.

Así que una vez instalado, se habrá creado una carpeta en la ruta `cypress\integration`. En ella crearemos nuestras especificaciones en ficheros con nombre tipo `mi-prueba.spec.js`

Test funcionales

Las **pruebas funcionales de aplicaciones web**, son un tipo *e2e* que va más allá de las superficialidad de *surfear* con un navegador automatizado. También controlan los navegadores y por supuesto que simulan el comportamiento del usuario en ellos: hacer clic, escribir, desplazarse, etc.

Son consideradas como de integración porque evidentemente para su ejecución deben coordinarse distintos servicios. Pero lo diferencial es que:

> Las pruebas funcionales aseguran que determinados escenarios realmente funcionen desde el punto de vista de un usuario final.

Probar funcionalidades web con Cypress

Como la mayoría de los frameworks de pruebas, *Cypress* nos ofrece la conocida sintaxis `describe it`. Se trata de dos funciones que reciben un primer argumento de texto y en el segundo una función.

Describe it

```
describe('Funcionalidad que se pretende probar', () => {
  // Código de preparación y actuación pruebas
  it('Lo que debería ocurrir', () => {
    // Comprobación mediante aserciones
  });
});
```

Es tan importante, o más, **prestar atención al texto** que reciben en el primer parámetro como al código que se ejecutará. Recuerda que el usuario final de este programa eres tú. Hazte un favor a ti mismo y especifica el texto de la manera más clara posible.

Actuaciones y aserciones

Dentro de las anteriores funciones irá el código propiamente dicho. Ahora ya usaremos *Cypress* como una librería mas y habrá que familiarizarse con su API. Aunque tu editor puede ayudarte en eso.

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

// Actuaciones como un usuario
cy.visit('https://www.bitademy.com');
// Comprobaciones como un programador
cy.title().should('include', 'bitAdemy');
```

Ejecutar las pruebas con Cypress

La ejecución es recomendable lanzarla desde el *package.json*. Os muestro dos alternativas. Podéis instalar *Cypress* y ejecutar las pruebas desde el mismo repositorio que la aplicación que estáis probando. O podéis

crear una aplicación de pruebas independiente.

```
{
  "scripts": {
    "test:e2e": "cypress open",
    "start": "cypress open"
  }
}
```

En el primer caso, la aplicación arranca en *start* y las pruebas en algo tipo *test:e2e*. Pero si lo hacéis en un repo aparte... entonces el test se ejecuta directamente desde *start*.

En el laboratorio he optado por esta aproximación, un repositorio independiente para la prueba funcional. Adaptadlo a vuestro gusto o situación particular.

Hola mundo con Cypress

Para que lo veas todo junto te propongo que te crees un fichero `cypress\integration\examples\0-hello-world\00-basic.spec.js` y escribas esto en él.

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

describe('Visiting the url https://www.bitademy.com', () => {
  it('should have _bitAdemy_ on its title', () => {
    cy.visit('https://www.bitademy.com');
    cy.title().should('include', 'bitAdemy');
  });
});
```

Si no lo habías hecho antes lanza el *runner* con el comando `cypress open` o el script equivalente... y espera que te muestre el panel de administrador con todos los tests disponibles.

Selecciona el `0-hello-world/00-basic.spec` y disfruta.

Más acciones y comprobaciones

Para familiarizarte un poco más con la sintaxis de *Cypress* te dejo la versión extendida de este *Hola Mundo*. Incluye las acciones más comunes como el simular clicks, navegar entre páginas, rellenar formularios... y comprobar contenidos.

De los textos que se incluyen en las funciones se deduce claramente **la intención del desarrollador**. Es un caso dónde la documentación forma parte del programa. **No es un comentario, es un dato.**

Estos tests son de muy alto nivel y normalmente van conducidos por especificaciones *behavior driven* como esta

```
FEATURE:     have web site with a title
As a:        visitor
I want to:   view the title of a site
In order to: be more confident

Scenario:
  Given: the url https://www.bitademy.com
  When: I visit it
  Then: should have BitAdemy on its title
```

Este es el código que realiza la prueba

```
describe('Visiting the url https://www.bitademy.com', () => {
  it('should have _bitAdemy_ on its title', () => {
    cy.visit('https://www.bitademy.com');
    cy.title().should('include', 'bitAdemy');
  });
});

const sutUrl = 'https://www.bitademy.com';
describe(`GIVEN: the url ${sutUrl}`, () => {
  context(`WHEN: I visit it`, () => {
    before(() => cy.visit(sutUrl));
    const expected = 'bitAdemy';
    it(`THEN: should have _${expected}_ on its title`, () => {
      cy.title().should('include', expected);
    });
  });
});
```

Incidiremos más en esta parte en el siguiente tema especialmente dedicado al comportamiento.

Before and after

Si no tienes experiencia previa con frameworks de test, te habrá sorprendido la función `before()`. Esta y sus hermanas *beforeAll, after y afterAll* ejecutan la función que reciben como *callback* en los momentos adecuados. Sus nombres no dejan lugar a dudas.

Se usan para establecer un escenario en el que se desarrollarán las pruebas, definiendo o inicializando variables. No reciben textos informativos. Si quieres dejar rastro tienes que escribir en la consola, en un log...

 Pruebas de comportamiento

Cypress y Behavior Driven Development.

> *"Los ordenadores son muy buenos siguiendo instrucciones, pero muy malos leyendo mentes"*
>
> -- ✍ **Donald Knuth**

En el tema anterior presentamos *Cypress* con el típico *Hola Mundo*. En este tema vamos a incorporar características que nos permitan agrupar y gestionar mejor las pruebas de comportamiento.

Veremos algunos **convenios de nombrado** y acabaremos con una batería de pruebas que, además de testear. documenta la funcionalidad de la aplicación.

Funcionalidad

Y hablando de funcionalidad. Toda aplicación se construye para resolver un problema de un usuario. Si no ¿para qué? En las metodologías ágiles se ha extendido el término **historia de usuario** como un mecanismo estándar para definir la funcionalidad que resuelve un problema.

Así que no es mala idea atar las pruebas de comportamiento con las historias de los usuarios de forma que la prueba acompañe al comportamiento desarrollado. Formalmente los practicantes de metodologías *Agile* lo llaman **behavior-driven development (BDD)**.

Pero tanto si odias o amas los procedimientos y las metodologías ágiles te pido que aceptes este consejo: **estructura y documenta muy bien tus pruebas funcionales**. Si no se te ocurre nada mejor usa este convenio.

Role Feature Reason

Antes de desarrollar la prueba, de hecho antes de desarrollar el software, siempre parto de una mínima documentación, lo imprescindible. Una tarjeta que incluye la descripción genérica de la funcionalidad, y que responde a tres preguntas básicas: el rol (**quién**), la solución (**qué**) y la razón (**por qué**) de existir de este software.

Un ejemplo muy sencillo sería la típica *To Do List*. Voy a usar como *SUT* una aplicación de ejemplo de otros cursos. Proton tasks

```
FEATURE:     the app should allow me to create new tasks
As a:        user with tasks to do
I want to:   create new tasks
In order to: follow up my work
```

```
```

Comportamiento

Cuando hablamos de pruebas web y comportamiento nos referimos a **cómo se comporta el sistema ante las acciones de los usuarios**. Es como una especie de guion esperado. Se plantea un escenario, se simula la interacción del usuario y se comprueba el resultado esperado.

Arrange, Act, Assert

Este guion a fuerza de repetirse acabó por generar un convenio para estructurar el código de las pruebas. Para facilitar su adopción se escogió un acrónimo pegadizo: *AAA o **la triple A***.

Está tomado de las palabras inglesas *Arrange, Act, Assert* que se traducen algo así como: **preparar, actuar y comprobar**.

Para acomodar estos tres eventos a *Cypress* necesitarás algo más de contexto.

Context

Esta función es similar a `describe` pero se usa como un agrupador de nivel inferior. Específicamente yo la uso para definir distintos **escenarios de prueba**. Este sería un esqueleto típico que incluye dichas funciones funciones.

```
describe('Funcionalidad que se pretende probar', () => {
  context('Escenario o situación prevista', () => {
    it('Lo que debería ocurrir', () => {});
  });
});
```

No es un ejemplo de funcionalidad ni de sintaxis. Es simplemente **una guía para que organices el código de pruebas**, al tiempo que lo documentas.

Aceptación

Y hablando de documentar. La *triple AAA* nos ayuda dentro de nuestro código de prueba. Es *muy de programador*. Pero estas pruebas son *tan de caja blanca* que las podía desarrollar o ejecutar alguien que no participa en el desarrollo. Que demonios, **incluso podría ejecutarlas un usuario no técnico**.

Se asemejan a lo que se denomina **pruebas de aceptación**, muy cercanas al usuario. De hecho se deberían definir con sus palabras y con sus criterios. De forma que si se cumplen se puedan considerar como una aceptación implícita de que el software cumple con un requerimiento.

Given When Then

Ya así llegamos al último convenio que quiero presentarte. Es un convenio para **documentar la traza de la prueba**. Un convenio que se aplica a los textos que reciben y escriben las funciones. Un convenio **en lenguaje de usuario**.

GWT es una fórmula fácil de recordar y que nos narra en lenguaje humano lo que sucede por debajo. `Given, when, then` traducido como **dado, cuando, entonces** explica que **dado** un contexto, **cuando** se ejecuta una acción, **entonces** debería haber una consecuencia esperada.

Plantearlo formalmente puede que requiero cierto esfuerzo. No tanto por escribir, más bien porque te obliga a pensar lo que vas a hacer

```
Scenario: add a task
  GIVEN: the form to add tasks
  WHEN: I type task description and click on _Add task_
  THEN: should clear the input box
    AND THEN: should appear on the _Things to do_ list
```

Pero el resultado merece mucho la pena porque después escribirlo en *cypress* es una delicia. Aplicándolo a nuestro ejemplo de la tareas, quedaría algo así:

```
describe(`GIVEN: the form to add tasks`, () => {
  const sutUrl = 'https://labsademy.github.io/ProtonTasks/';
  const selectorFormInput = 'form > input';
  const inputTaskDescription = 'Dummy task one';
  const selectorFormButton = 'form > button';
  const inputButtonText = 'Add task';
  const expectedTaskDescription = 'Dummy task one';
  const selectorIncompleteListLabel = '#incomplete-tasks > li:first-child >
label';
  const selectorIncompleteListButton = '#incomplete-tasks > li:first-child >
button.delete';
  context(`WHEN: I type task description and click on _Add task_ `, () => {
    before(() => {
      cy.visit(sutUrl);
      cy.get(selectorFormInput).type(inputTaskDescription);
      cy.get(selectorFormButton).contains(inputButtonText).click();
    });
    it(`THEN: should clear the input box`, () => {
      cy.get(selectorFormInput).should('not.include.value');
    });
    it(`AND THEN: should appear on the _Things to do_ list`, () => {
      cy.get(selectorIncompleteListLabel).should('contain',
expectedTaskDescription);
    });
    after(() => {
      cy.get(selectorIncompleteListButton).click();
    });
  });
});
```

Yo lo destaco poniéndolo al principio, en mayúsculas y con los dos puntos. Pero lo mejor es que hagas tus pruebas y que lo ajustes de forma **que siempre te deje claro qué es lo que está ocurriendo**.

Comandos custom

Ya que estamos aprendiendo Cypress, veamos un poco más de sintaxis, y alguna de sus utilidades. Por ejemplo la creación de nuevos comandos reutilizables. Para ello disponemos del fichero `/support/commands.js` destinado a contener nuevas definiciones.

```
Cypress.Commands.add('addTask', inputTaskDescription => {
  const selectorFormInput = 'form > input';
  const selectorFormButton = 'form > button';
  const inputButtonText = 'Add task';
  cy.get(selectorFormInput).type(inputTaskDescription);
  cy.get(selectorFormButton).contains(inputButtonText).click();
});
```

Por ejemplo en este caso podemos invocar a este comando cada vez que queramos agregar una nueva tarea durante una prueba. Sería tan sencillo como llamar a cualquier otro método propio de Cypress: `cy.addTask('Mi nueva tarea');`

Tienes más código de muestra en el laboratorio asociado.

Resumen

Quizá no haya una relación directa obligatoria entre todos estos conceptos; tampoco un estándar o convenio universal que lo resuelva todo. Pero **un poco de orden y una guía para no reinventar la rueda** nunca sobran.

Te propongo que te inspires en esta **tabla para organizar y documentar tus pruebas**.

Organización	Documentación	Implementación
Arrange	GIVEN:	`describe()`
Act	WHEN:	`context() before()`
Assert	THEN:	`it()`
After		`after()`

Esto debería ayudarte a demostrar que tu aplicación es aceptable. Es decir, explicar qué hace, para quién lo hace y por qué lo hace.

Pruebas de una SPA y un API

Cypress para probar aplicaciones empresariales.

> *"La calidad requiere hacerlo bien incluso cuando nadie te está mirando"*
>
> -- ✍ **Henry Ford**

Para continuar aprendiendo cypress y además enfrentarnos a situaciones cotidianas voy a cambiar de ejemplo. Ahora será una aplicación desarrollada en *Angular* (podría ser *React* o *Vue*, es lo mismo) y que consume un API rest.

El problema de las SPA

Estas aplicaciones on conocidas porque generan el html dinámicamente ejecutando JavaScript. Aunque responden a muchas rutas, técnicamente son una única página con todo el código necesario para mostrar vistas según se requiera, es decir una *Single Page Application*

El problema es que al solicitar una ruta al servidor, este realmente no la conoce. De hecho devuelve siempre la misma página vacía para que sea el navegador el que agregue contenido. Una solución sencilla es no solicitar la página destino directamente, si no suponer que nos llegará la raíz y proceder a la navegación manual simulando un click por parte del usuario.

El problema de las API

Con las llamadas a API remota nos enfrentamos a un problema distinto: la integración de distintos sistemas. Porque eso es lo que ocurre, en realidad tenemos dos aplicaciones ejecutándose en dos entornos. Por un lado la web en el navegador y por otro el API en un servidor más o menos lejano.

A pesar de que estas pruebas son de integración de alto nivel, es muy recomendable desacoplar estos dos ámbitos. Al menos si lo que queremos es comprobar la funcionalidad web, porque un fallo en el *back* invalidaría nuestras pruebas *front*.

Fixtures

La solución de Cypress es sencillamente elegante. Se trata de usar un doble de pruebas, concretamente un *stub*, que responde con datos fijos las llamadas esperadas.

Ejemplo completo

Te copio aquí el código más significativo que aparece en el proyecto laboratorio.

Primero, como siempre, definimos la funcionalidad que vamos a probar. Recuerda que las pruebas ayuda a documentar funcionalmente la aplicación.

```
FEATURE:     list my current projects
As a:        user with involved in projects
```

```
I want to:   get a list of them
In order to: follow up my work
```

En este caso el SUT es otra *to do list* esta vez desarrollada en *Angular*
https://angularbuilders.github.io/angular-budget

El escenario que quiero probar es otro sencillo *happy path*

```
Scenario: complete a task
  GIVEN: An API with 2 projects
  WHEN: I visit the projects page
  THEN: should show 2 items on the projects list
```

Que se traduce en código *cypress* de manera casi literal

```
describe('GIVEN: An API with 2 projects', () => {
  before(() => {
    const stubbedApiUrl = 'https://api-base.herokuapp.com/api/pub/projects';
    const fixtureData = 'fx:projects';
    cy.server();
    cy.route(stubbedApiUrl, fixtureData);
  });
  context('WHEN: I visit the projects page', () => {
    before(() => {
      const homeSpaUrl = 'https://angularbuilders.github.io/angular-budget';
      cy.visit(homeSpaUrl);
      cy.get('a').contains('Projects').click();
    });
    it('THEN: should show 2 items on the projects list', () => {
      const expectedListItemsLength = 2;
      const selectorListItems = 'section > ul > li';
      cy.get(selectorListItems).should('have.length', expectedListItemsLength);
    });
  });
});
```

Simplemente llamar la atención sobre la navegación forzada por ser una SPA y el uso de la *fixture* para interceptar las llamadas al API devolviendo datos *fake*

```
[
  {
    "name": "dummy one",
    "projectId": 1,
    "_id": "1",
    "owner": "tester"
  },
```

```
  {
    "name": "two dummies",
    "projectId": 2,
    "_id": "2",
    "owner": "tester"
  }
]
```

Con esto te puedes hacer una idea clara de las capacidades de *Cypress* y de la importancia y retorno de inversión de las **pruebas funcionales e2e**.