ceviche.js

Capítulo 5: Pruebas

> *Ahora que conocemos un poco más a fondo JavaScript y cómo manejar el DOM, así como algunas APIs del navegador, necesitamos estar completamente seguros de que nuestro código funcione, por lo que es importante realizar pruebas en él.*

Pruebas automatizadas

Cada vez que copiamos un código y refrescamos el navegador para saber si funciona o no, estamos probando. Es un proceso rápido, pero en ocasiones puede ser tedioso y aburrido, así que lo ideal sería dejar que la computadora pruebe por nosotros. Es aquí donde aparecen las pruebas automatizadas.

En términos simples, una prueba automatizada verifica el correcto funcionamiento de un código mediante valores `true` o `false`. Por ejemplo, si deseamos probar una suma, podemos hacer lo siguiente:

```
var resultado = 10 + 15;

console.assert(resultado === 25, 'La suma de 10 y 15 debe ser 25');
```

Usamos `console.assert` para verificar una condición, que debe dar `true`, o, en caso contrario, mostrar un mensaje de error, el cual contiene la descripción de la validación. `console.assert` es un método de la consola disponible en todos los navegadores modernos (desde IE8 en adelante, Firefox, Chrome, Safari y Opera).

Idealmente, nuestro código más importante debe tener pruebas automatizadas que validen su correcto funcionamiento.

Pruebas unitarias

Existen diferentes tipos de pruebas, de acuerdo a la forma cómo se desea probar el código:

- Pruebas unitarias: Buscan validar una parte del código a la vez, sin importar para qué se utilice dicho código.
- Pruebas funcionales: Buscan validar toda acción que un usuario normalmente haría en el sitio o la aplicación web en la que se trabaja.

Las pruebas unitarias pueden ser sencillas de realizar, si se identifican correctamente las partes de la aplicación que deben probarse. En el capítulo 2 creamos un módulo llamado `titleBuilder`, que permite crear un título para la web de La Buena Espina según la sección que estemos visitando:

```
var titleBuilder = (function() {
  var baseTitle = 'La Buena Espina';
  var parts = [baseTitle];

  function getSeparator() {
    if (parts.length == 2) {
      return ' – ';
    }
    else {
      return ' › ';
    }
  };

  return {
    reset: function() {
      parts = [baseTitle];
    },
    addPart: function(part) {
      parts.push(part);
    },
```

```
      toString: function() {
        return parts.join(getSeparator());
      }
    };
  })();
```

Ahora, crearemos algunas validaciones con `console.assert`:

```
console.assert(titleBuilder.toString() === 'La Buena Espina', 'El
título por defecto debe ser "La Buena Espina"');
```

De esta forma validamos que el título sea "La Buena Espina" si no hemos navegado por ninguna sección del sitio. ¿Qué pasaría si una validación falla? Tenemos un mensaje de error en la consola de la siguiente forma:

```
console.assert(titleBuilder.toString() === ' - La Buena Espina - ',
'El título por defecto debe ser "La Buena Espina"');
// Assertion failed: El título por defecto debe ser "La Buena
Espina"
```

El primer parámetro es una condición que debe evaluarse como verdadero, mientras que el segundo parámetro es la descripción de la validación.

Podemos seguir haciendo más validaciones, de la siguiente forma:

```
titleBuilder.addPart('Carta');
titleBuilder.addPart('Pescados');
titleBuilder.addPart('Ceviches');

console.assert(titleBuilder.toString() === 'La Buena Espina › Carta
› Pescados › Ceviches', 'El título ahora debe ser "La Buena Espina
› Carta › Pescados › Ceviches"');
```

```
titleBuilder.reset();
titleBuilder.addPart('Locales');

console.assert(titleBuilder.toString() === 'La Buena Espina —
Locales', 'El título ahora debe ser "La Buena Espina — Locales"');
```

Si las validaciones con `console.assert` pasan correctamente, la descripción de la validación no se mostrará. Este comportamiento puede no ser tan útil: **¿Cómo sabemos cuántas validaciones han pasado correctamente, y cuántas no?** Además, **¿de qué nos sirve tener los mensajes de error en la consola?**. Es aquí donde entran en escena diversos frameworks para pruebas, una de las cuales es QUnit.

QUnit

QUnit es un framework para pruebas unitarias creado por jQuery, donde, en lugar de utilizar la consola para mostrar los resultados, crea un reporte en HTML con los resultados de las pruebas realizadas. En QUnit, cada comparación que hacemos se llama *assert*, mientras que el conjunto de *asserts* es llamado *test*.

Para poder utilizar QUnit debemos descargar dos archivos desde su web (o utilizar los archivos vía su CDN):

```
<!DOCTYPE html>
<html>
<head>
  <title>Pruebas unitarias</title>
  <link rel="stylesheet" type="text/css"
href="http://code.jquery.com/qunit/qunit-1.14.0.css">
</head>
<body>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
  <script type="text/javascript" src="http://code.jquery.com/qunit
```

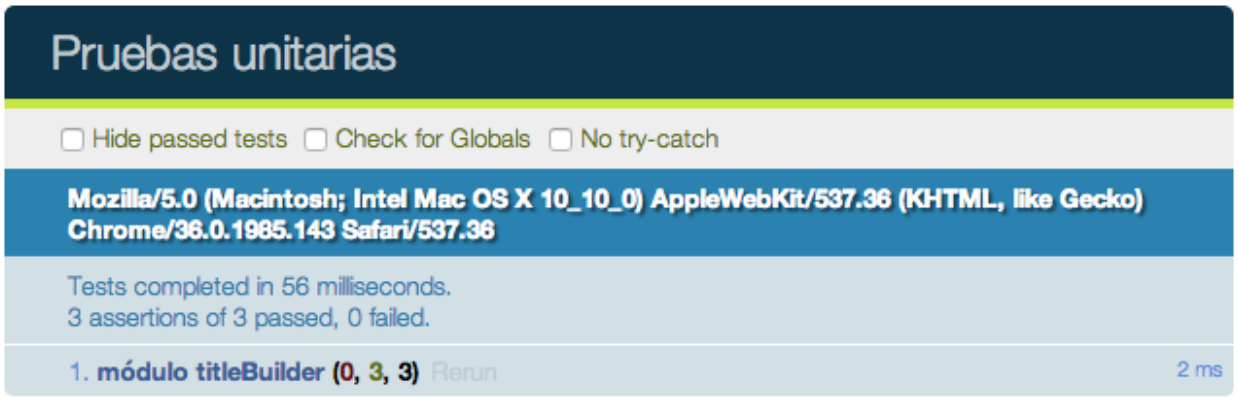
```
/qunit-1.14.0.js"></script>
</body>
</html>
```

Pruebas con QUnit

Con este código tenemos la base necesaria para poder realizar pruebas unitarias con QUnit. Lo siguiente será pasar las validaciones que hicimos con `console.assert` a una prueba unitaria con QUnit:

```
QUnit.test('módulo titleBuilder', function(assert) {
  assert.ok(titleBuilder.toString() === 'La Buena Espina', 'El
título por defecto debe ser "La Buena Espina"');

  titleBuilder.addPart('Carta');
  titleBuilder.addPart('Pescados');
  titleBuilder.addPart('Ceviches');

  assert.ok(titleBuilder.toString() === 'La Buena Espina › Carta ›
Pescados › Ceviches', 'El título ahora debe ser "La Buena Espina ›
Carta › Pescados › Ceviches"');

  titleBuilder.reset();
  titleBuilder.addPart('Locales');

  assert.ok(titleBuilder.toString() === 'La Buena Espina —
Locales', 'El título ahora debe ser "La Buena Espina — Locales"');
});
```

En este caso `assert.ok` toma los mismos valores que `console.assert` (una condición que debe evaluarse como verdadera y la descripción de la validación). Las 3 validaciones o *asserts* son agrupadas en una *prueba* o *test*, definida por el método `QUnit.test`. Al final debe quedar así:



Jasmine

QUnit nos permite realizar pruebas unitarias utilizando un lenguaje un tanto *técnico*, lo que nos permite crear pruebas para un módulo, una función constructora o algún caso similar. Pero, ¿cómo podríamos crear pruebas que sean más legibles? Es aquí donde entra el concepto de Behavior-Driven Development, o BDD.

Behavior-Driven Development es un modo de realizar pruebas donde estas se enfocan en función al *comportamiento* de lo que se va a probar (por ejemplo, qué debería hacer un módulo o una función), y no a, verificar que el código probado devuelva un valor en específico.

Jasmine es una biblioteca que permite realizar pruebas unitarias utilizando BDD, lo que nos da la opción de crear pruebas más interesantes que solo hacer *El título por defecto debe ser "La Buena Espina"*. Así mismo, nos da métodos para realizar validaciones en un lenguaje más natural y no tan técnico, como verificar si un número es mayor o menor que otro, si una cadena es parte de otra cadena, si algún valor puede ser **considerado** como `true` o `false` (denominados *truthy* o *falsy*, respectivamente), entre otros.

Type coercion y valores *truthy* y *falsy*

Cuando comparamos un valor dentro de una condicional pueden pasar dos

cosas: O el valor que se compara es un booleano (es `true` o `false`), o no lo es. Si es booleano, la condicional se ejecuta directamente:

```
if (10 + 15 === 25) {
  console.log('10 + 15 es igual a 25');
}

// 10 + 15 es igual a 25
```

Pero si el valor que se compara no es un booleano, ocurre un *type coercion*, o conversión implícita. JavaScript es un lenguaje con tipado dinámico, lo que significa que una variable, o propiedad, pueden tener cualquier tipo de valor, sin necesidad de hacer una conversión explícita, o *casting*. Esto puede ocurrir en dos casos:

1. Al usar `==`, o `!=`:

```
if (10 + 15 == '25') {
  console.log('10 + 15 es igual a 25, aunque sea una cadena');
}

// 10 + 15 es igual a 25, aunque sea una cadena
```

En este caso, `==` (y `!=`) compara valores y no tipos de datos (es decir: `10 + 15` es igual a 25`, y` '25' `tiene el mismo valor que` 25`).

1. O al pasar un valor a una condicional:

```
if (10) {
  console.log('10 es convertido implícitamente a true');
}

// 10 es convertido implícitamente a true
```

En JavaScript, un número diferente a `0` es *igual* a `true`, mientras que `0` es igual a `false`. De igual forma, una cadena vacía es *igual* a `false`, mientras que, si tuviera algún caracter (incluyendo espacios), sería *igual* a `true`.

Cada vez que nos referimos a que un valor es *igual* a `true` (así, en cursiva), decimos que ese valor es *truthy*. Por otro lado, si decimos que un valor es *igual* a `false` (de nuevo, en cursiva), decimos que ese valor es *falsy*.

Algunos ejemplos más sobre *type coercion* se pueden encontrar en este link.

Para poder usar Jasmine debemos descargarlo desde la cuenta del proyecto en GitHub. En este caso, trabajaremos con la versión 2.0.2. El zip descargado contiene una estructura de archivos y carpetas que utiliza Jasmine:

- Carpeta `lib`: Aquí se encuentran todos los archivos que componen Jasmine, incluyendo hojas de estilo y la biblioteca en sí.
- Carpeta `spec`: En esta carpeta deben estar todas las pruebas que haremos a nuestro código.
- Archivo `SpecRunner.html`: Este archivo servirá de reporte para las pruebas que haremos. Similar a la página que armamos para QUnit.
- Carpeta `src`: Aquí debería ir el código que queremos probar.

Por defecto, Jasmine viene con código de ejemplo dentro de las carpetas `spec` y `src`, e indicando el orden en el que debe ir nuestro código en `SpecRunner.html` (en este caso, usando el ejemplo que el mismo Jasmine nos da):

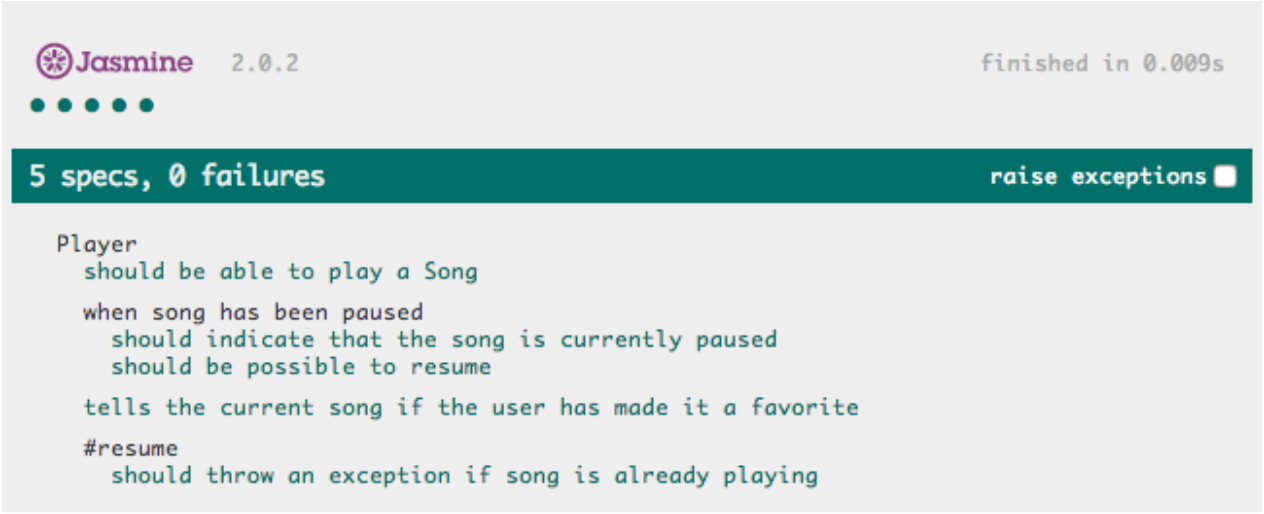
```
<!DOCTYPE HTML>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
  <title>Jasmine Spec Runner v2.0.2</title>

  <link rel="shortcut icon" type="image/png" href="lib/jasmine-
2.0.2/jasmine_favicon.png">
```

```
  <link rel="stylesheet" type="text/css" href="lib/jasmine-
2.0.2/jasmine.css">

  <script type="text/javascript" src="lib/jasmine-
2.0.2/jasmine.js"></script>
  <script type="text/javascript" src="lib/jasmine-2.0.2/jasmine-
html.js"></script>
  <script type="text/javascript" src="lib/jasmine-2.0.2/boot.js">
</script>

  <!-- include source files here... -->
  <script type="text/javascript" src="src/Player.js"></script>
  <script type="text/javascript" src="src/Song.js"></script>

  <!-- include spec files here... -->
  <script type="text/javascript" src="spec/SpecHelper.js"></script>
  <script type="text/javascript" src="spec/PlayerSpec.js"></script>

</head>

<body>
</body>
</html>
```

El *Spec Runner* de Jasmine carga los siguientes archivos:

- `jasmine.js`: La biblioteca que contiene el código de Jasmine
- `jasmine-html.js`: Contiene el código necesario para mostrar los resultados en forma de HTML, dentro del *Spec Runner*.
- `boot.js`: Este archivo se encargar de cargar todo el entorno de pruebas de Jasmine y activar el reporte en HTML (`jasmine-html.js`)

Luego de esto, se debe cargar el código que deseamos probar (los que se encuentran en la carpeta `src`), y luego las pruebas en sí (carpeta `spec`). Al final debe quedar así:



Pruebas con Jasmine

Jasmine tiene una forma de organizar las pruebas, según el enfoque de BDD. De acuerdo a Jasmine, el código debe ser legible como si fuera un texto *en inglés*. Esto se logra utilizando ciertos métodos como `describe` e `it`:

```
describe("Player", function() {
  it("should be able to play a Song", function() {});

  describe("when song has been paused", function() {
    it("should indicate that the song is currently paused",
function() {});

    it("should be possible to resume", function() {});
  });

  it("tells the current song if the user has made it a favorite",
function() {});

  describe("#resume", function() {
    it("should throw an exception if song is already playing",
function() {});
  });
```

```
});
```

En Jasmine, cada método `describe` crea una suite de pruebas (una suite de pruebas es un conjunto de pruebas), y cada método `it` permite definir una prueba (aquí son llamados *specs*). Según la imagen de arriba, el código anterior se debería leer así:

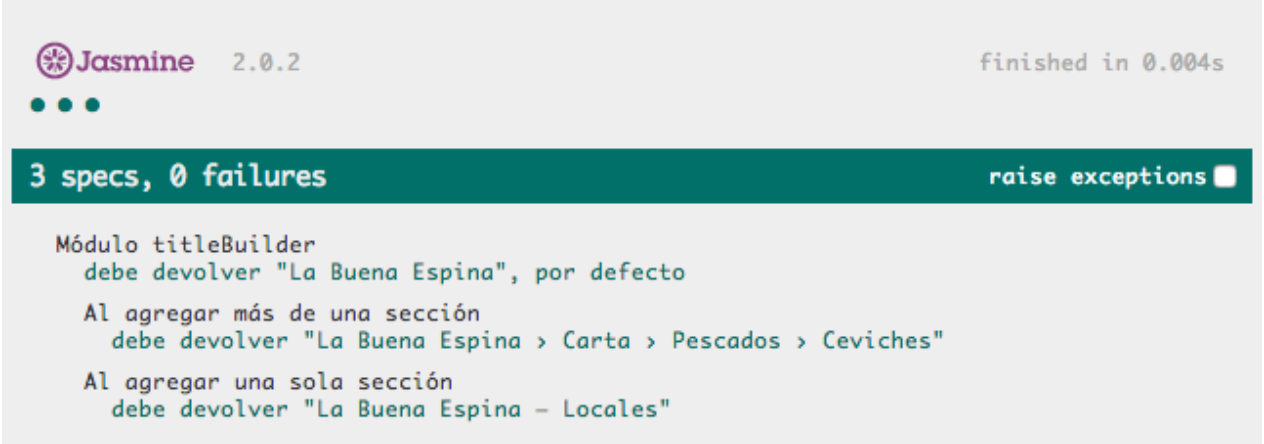
```
Player should be able to play a Song

Player, when song has been paused, should indicate that the song is
currently paused
Player, when song has been paused, should be possible to resume

Player tells the current song if the user has made it a favorite

Player#resume should throw an exception if song is already playing
```

Entonces, nuestras pruebas deben ser escritas de manera similar:

```
describe('Módulo titleBuilder', function() {
  beforeEach(function() {
    titleBuilder.reset();
  });

  it('debe devolver "La Buena Espina", por defecto', function() {
    expect(titleBuilder.toString()).toEqual('La Buena Espina');
  });

  describe('Al agregar más de una sección', function() {
    it('debe devolver "La Buena Espina › Carta › Pescados ›
Ceviches"', function() {
      titleBuilder.addPart('Carta');
      titleBuilder.addPart('Pescados');
      titleBuilder.addPart('Ceviches');
```

```
      expect(titleBuilder.toString()).toEqual('La Buena Espina ›
Carta › Pescados › Ceviches');
    });
  });

  describe('Al agregar una sola sección', function() {
    it('debe devolver "La Buena Espina – Locales"', function() {
      titleBuilder.addPart('Locales');

      expect(titleBuilder.toString()).toEqual('La Buena Espina –
Locales');
    });
  });
});
```

Y el resultado sería el siguiente:



En Jasmine, cada prueba (definida con el método `it`) puede tener una o más validaciones o *asserts*, que, en este caso, son definidas con el método `expect`.

El método `expect` permite utilizar lo que en Jasmine se llaman *matchers*. Estos *matchers* permiten validar a un nivel más complejo que simplemente comparar dos valores con `===` o `!==`. Los *matchers* que vienen por defecto son:

- `toBe`: Igual a utilizar `===`.
- `toEqual`: Similar a `toBe` pero permite comparar objetos literales.

- `toMatch`: Compara cadenas con expresiones regulares o con otras cadenas.
- `toBeDefined`: Valida si una propiedad está definida (que no sea `undefined`)
- `toBeUndefined`: Lo opuesto a `toBeDefined`
- `toBeNull`: Valida si una variable o propiedad tiene valor `null`.
- `toBeTruthy`: Permite saber si un valor es *truthy*.
- `toBeFalsy`: Permite saber si un valor es *falsy*.
- `toContain`: Valida si un elemento está dentro de un array.
- `toBeLessThan`: Valida si un número es menor a otro.
- `toBeGreaterThan`: Valida si un número es mayor a otro.
- `toBeCloseTo`: Valida si un número decimal es cercano a un número entero.
- `toThrow`: Valida si una función lanzará una excepción al ser ejecutada.

Así mismo, el valor devuelto por `expect` tiene una propiedad llamada `not`, el cual permite invertir el valor de cada matcher (recordemos que, a fin de cuentas, una validación debe devolver `true` o `false`).

← 4. Capítulo 4: APIs del navegador 6. Capítulo 6: Peticiones asíncronas →

Acerca de | Disponible en Amazon.com