Polymer *Análisis testing de componentes*

Puntos analizados:

- Cómo se realizan test sobre un componente aislado
    - carga de fixtures
    - uso de stubs, mocks, spies, etc
    - enfoque de pruebas de lógica de presentación (validación apoyada en el DOM?)
    - enfoque de pruebas de lógica de dominio y ubicación de ésta (necesidad de renderizar componente?)
- Lanzado de una suite de tests de varios componentes
- Creación de test que implique la interacción de varios componentes
- Posibilidad de aplicación de enfoques "inside out" e "outlide in"
- Configuración del lanzado sobre diferentes navegadores (y plataformas)
- Obtención de cobertura de pruebas de un componente y de una suite de éstos

Cómo se realizan test sobre un componente aislado

Para esta labor (realmente, para toda la labor de testing sobre componentes polymer), google ha desarrollado una herramienta llamada *web-component-tester*, con la que te permite realizar las pruebas unitarias/integración de uno/varios componentes de una forma sencilla, creando un contexto con todas las dependencias necesarias para la creación de tests. Esta herramienta trae una serie de dependencias, las cuales usaremos para la implementación de nuestros tests, estas dependencias son:

- *Mocha*, será tu framework de pruebas
- *Chai*, lo usarás para definir tus asserts (TDD o BDD)
- *Async*, librería que te proporciona funciones para trabajar la asincronía
- *Lodash* (3.0)
- *Sinon* y *sinon-chai*
- *test-fixture*, componente polymer con el que te permite mantener la integridad de tu html de pruebas entre las diferentes ejecuciones de los tests
- *accessibility-developer-tools*, herramienta que te permite auditar la accesibilidad que tiene implementada tu componente

Nuestro fichero que especifica los test es html, tendremos dos tipos de ficheros html. Uno de ellos se encarga de indicar la suit de tests a ejecutar *(test/index.html)* y el otro son las implementaciones de los tests *(test/*_test.html)*

Ejemplo index.html:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, minimum-scale=1.0, initial-scale=1.0, user-scalable=yes">

    <script src="../../webcomponentsjs/webcomponents-loader.js"></script>
    <script src="../../web-component-tester/browser.js"></script>
  </head>
  <body>
    <script>
      // Load and run all tests (.html, .js):
      WCT.loadSuites([
        'shop-inventory-polymer_test.html'
      ]);
    </script>

  </body>
</html>
```

Lo más destacable es la llamada al método WCT.loadSuites, en el cual, específicas un array de ficheros html que componen tu suit de tests a ejecutar.

Ejemplo de implementación de test:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, minimum-scale=1, initial-scale=1, user-scalable=yes">

    <title>shop-inventory-polymer test</title>

    <script src="../../webcomponentsjs/webcomponents-loader.js"></script>
    <script src="../../web-component-tester/browser.js"></script>
    <link rel="import" href="../src/shop-inventory-polymer/shop-inventory-polymer.html">
    <link rel="import" href="../src/shop-inventory-polymer/shop-inventory-polymer-form.html">
  </head>
  <body>

    <test-fixture id="BasicWear">
      <template>
        <shop-inventory-polymer></shop-inventory-polymer>
      </template>
    </test-fixture>

    <test-fixture id="BasicWearForm">
        <template>
          <shop-inventory-polymer-form></shop-inventory-polymer-form>
        </template>
    </test-fixture>

    <script>

       suite('shop-inventory-polymer-form', () => {
         const wearForm = fixture('BasicWearForm');
         const productNameExpected = 'prueba';
        setup(function() {
          wearForm.product={nombre:productNameExpected};
        });
        test('instantiating the element with default properties works', () => {
          const elementShadowRoot = wearForm.shadowRoot;
          assert.equal(productNameExpected,elementShadowRoot.querySelector('input[name="Nombre"]').value)
        });
      });

      suite('shop-inventory-polymer', () => {
        test('instantiating the element with default properties works', (done) => {
          stub('shop-inventory-polymer-dp', {
            load: function() {
              const products = [{
                    id: 893
                }];
              this.dispatchEvent(new CustomEvent('product-loaded',  {bubbles: true, composed: true, detail: {products:products}}));
            }
          });
          const element = fixture('BasicWear');
          const productSizeExpected = 1;
          const elementShadowRoot = element.shadowRoot;
          flush(function() {
            const productsSize = element.$.wearList.shadowRoot.querySelectorAll('div.product').length
            assert.equal(productSizeExpected,productsSize);
            done();
          });
        });
      });
    </script>

  </body>
</html>
```

En este fichero diferenciamos tres bloques funcionales:

- Definición de dependencias (componentes necesarios para las pruebas)

```
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, minimum-scale=1, initial-scale=1, user-scalable=yes">

  <title>shop-inventory-polymer test</title>

  <script src="../../webcomponentsjs/webcomponents-loader.js"></script>
  <script src="../../web-component-tester/browser.js"></script>
  <link rel="import" href="../src/shop-inventory-polymer/shop-inventory-polymer.html">
  <link rel="import" href="../src/shop-inventory-polymer/shop-inventory-polymer-form.html">
</head>
```

- Definición de fixtures (siempre asociandolos un id para su posterior uso)

```
<test-fixture id="BasicWear">
  <template>
    <shop-inventory-polymer></shop-inventory-polymer>
  </template>
</test-fixture>

<test-fixture id="BasicWearForm">
    <template>
      <shop-inventory-polymer-form></shop-inventory-polymer-form>
    </template>
</test-fixture>
```

- Implementación de la suit de tests

```
 suite('shop-inventory-polymer-form', () => {
   const wearForm = fixture('BasicWearForm');
   const productNameExpected = 'prueba';
  setup(function() {
    wearForm.product={nombre:productNameExpected};
  });
  test('instantiating the element with default properties works', () => {
    const elementShadowRoot = wearForm.shadowRoot;
    assert.equal(productNameExpected,elementShadowRoot.querySelector('input[name="Nombre"]').value)
  });
});

suite('shop-inventory-polymer', () => {
  test('instantiating the element with default properties works', (done) => {
    stub('shop-inventory-polymer-dp', {
      load: function() {
        const products = [{
              id: 893
          }];
        this.dispatchEvent(new CustomEvent('product-loaded',  {bubbles: true, composed: true, detail: {products:products}}));
      }
    });
    const element = fixture('BasicWear');
    const productSizeExpected = 1;
    const elementShadowRoot = element.shadowRoot;
    flush(function() {
      const productsSize = element.$.wearList.shadowRoot.querySelectorAll('div.product').length
      assert.equal(productSizeExpected,productsSize);
      done();
    });
  });
});
```

Bien, vayamos paso a paso. En la **definición de las dependencias**, no es necesario realizar un exhaustivo análisis. Simplemente recalcar que, a parte de las dependencias de polymer, habrá que referenciar todos los componentes que vayamos a usar en los **test-fixture**, en este caso, necesitaremos a **shop-inventory-polymer-form** y **shop-inventory-polymer**.

En este ejemplo, hemos definido **dos fixtures**, cada una con su identificador, para su posterior carga en la implementación de cada test. Estas fixtures, contienen la instanciación del componente. Lo potente de emplear el componente **test-fixture** como wrapper del nuestro, es que se encargará de renderizar nuestro componente de nuevo, antes de cada test, por lo que, cualquier manipulación que hagamos sobre este, no se verá reflejada en los siguientes tests.

En la **definición de nuestra suit de tests**, hay varios puntos destacables, vayamos poco a poco. Lo primero es indicar cual es la fixture que vas a necesitar:

```
const wearForm = fixture('BasicWearForm');
```

Mocha te proporciona distintos hooks, uno de ellos, el cual usamos en la suite **shop-inventory-polymer-form**, es en el que estamos modificando el parámetro que espera recibir el componente *(método: setup)*, añadiendo un producto con un nombre asociado. Lo que se esperaría es que el componente, renderize un formulario con el input del nombre relleno con el mismo que se ha inyectado:

```
test('instantiating the element with default properties works', () => {
  const elementShadowRoot = wearForm.shadowRoot;
  assert.equal(productNameExpected,elementShadowRoot.querySelector('input[name="Nombre"]').value)
});
```

En el segundo test implementado, se realiza una prueba de integración de los componentes que está empleando **shop-inventory-polymer**. Los componentes que interactúan en este test son los siguientes:

- shop-inventory-polymer-dm -> shop-inventory-polymer-dp
- shop-inventory-polymer-list

En esta prueba, vamos a crear un stub del data provider, para indicar cuántos productos hay en la carga inicial:

```
stub('shop-inventory-polymer-dp', {
  load: function() {
    const products = [{
          id: 893
      }];
    this.dispatchEvent(new CustomEvent('product-loaded',  {bubbles: true, composed: true, detail: {products:products}}));
  }
});
```

Como en este caso, debemos esperar a que realice la carga de los productos, el navegador debe renderizar la vista. Para esta espera, debemos usar el método (*flush*) que nos proporciona **wct** para las interacciones con parámetros que implican renderizado de componentes:

```
flush(function() {
  const productsSize = element.$.wearList.shadowRoot.querySelectorAll('div.product').length
  assert.equal(productSizeExpected,productsSize);
  done();
});
```

Como resumen, este test comprueba que, *mockeando* la respuesta del data-provider para que devuelta solo un producto para el inventario, el componente **shop-inventory-polymer** renderiza correctamente otro componente **shop-inventory-polymer-list** en el cual solo se muestra un elemento de la lista.

Lanzado de una suite de tests de varios componentes

Esto es tan sencillo como añadir tantos componentes como te sean necesario dentro del componente **test-fixture.**

Creación de test que implique la interacción de varios componentes

Está definido en el apartado (*Cómo se realizan test sobre un componente aislado*), para ser exactos, la suit **shop-inventory-polymer**.

Posibilidad de aplicación de enfoques "inside out" e "outlide in"

El framework de pruebas que te proporciona **WCT**, te permite llevar a cabo ambos enfoques de TDD. Si quieres llevar a cabo el enfoque Inside-Out (*Classic school*), puedes crearte un test-fixture con el componente a testear. Recuperarlo e invocar todas las funciones que tenga definidas en su clase implementada, realizando los tests que sean necesarios sobre ese método en especial:

```
... Definición del componente
<link rel="import" href="../../bower_components/polymer/polymer-element.html">
<link rel="import" href="./shop-inventory-polymer-dp.html">

<dom-module id="my-component">
  <script>
    class MyComponent extends Polymer.Element {
      static get is() { return 'my-component'; }
      sum(a,b){
          return a+b;
      }
    }
    window.customElements.define(MyComponent.is, MyComponent);
  </script>
</dom-module>
... Definición del fixture
<test-fixture id="MyComponent">
    <template>
      <my-component></my-component>
    </template>
</test-fixture>
... Definición del test
suite('shop-inventory-polymer-form', () => {
     const myComponent = fixture('MyComponent');
    test('should sum two numbers', () => {
      const a=2,b=2,total=4;
      assert.equal(total,myComponent.$.sum(a,b))
    });
});
```

Si por otra parte, quieres llevar a cabo un TDD Outside-In (London school / mockist TDD), una vez tengas definido todos los servicios con los que se verá nutrido tu componente, puedes crearte stubs tanto de funciones que exponen los componentes como de respuestas de servidores.

Configuración del lanzado sobre diferentes navegadores (y plataformas)

WCT, por defecto, te lanza los tests en los navegadores safari, firefox y chrome. Pero esto es configurable mediante su fichero de configuración (*wct.config.json*) tal que:

```
{
  ...
  "plugins": {
    "local": {
      "browsers": ["firefox", "google-chrome-stable"]
    }
  }
...
}
```

Obtención de cobertura de pruebas de un componente y de una suite de éstos

Tras la ejecucción del comando wct, previamente instalado:

```
npm install -g web-component-tester
```

Existe la posibilidad, mientras la instalación de plugins de wct, modificar el reporte que genera, configurando apropiadamente el fichero **wct.conf.json**.