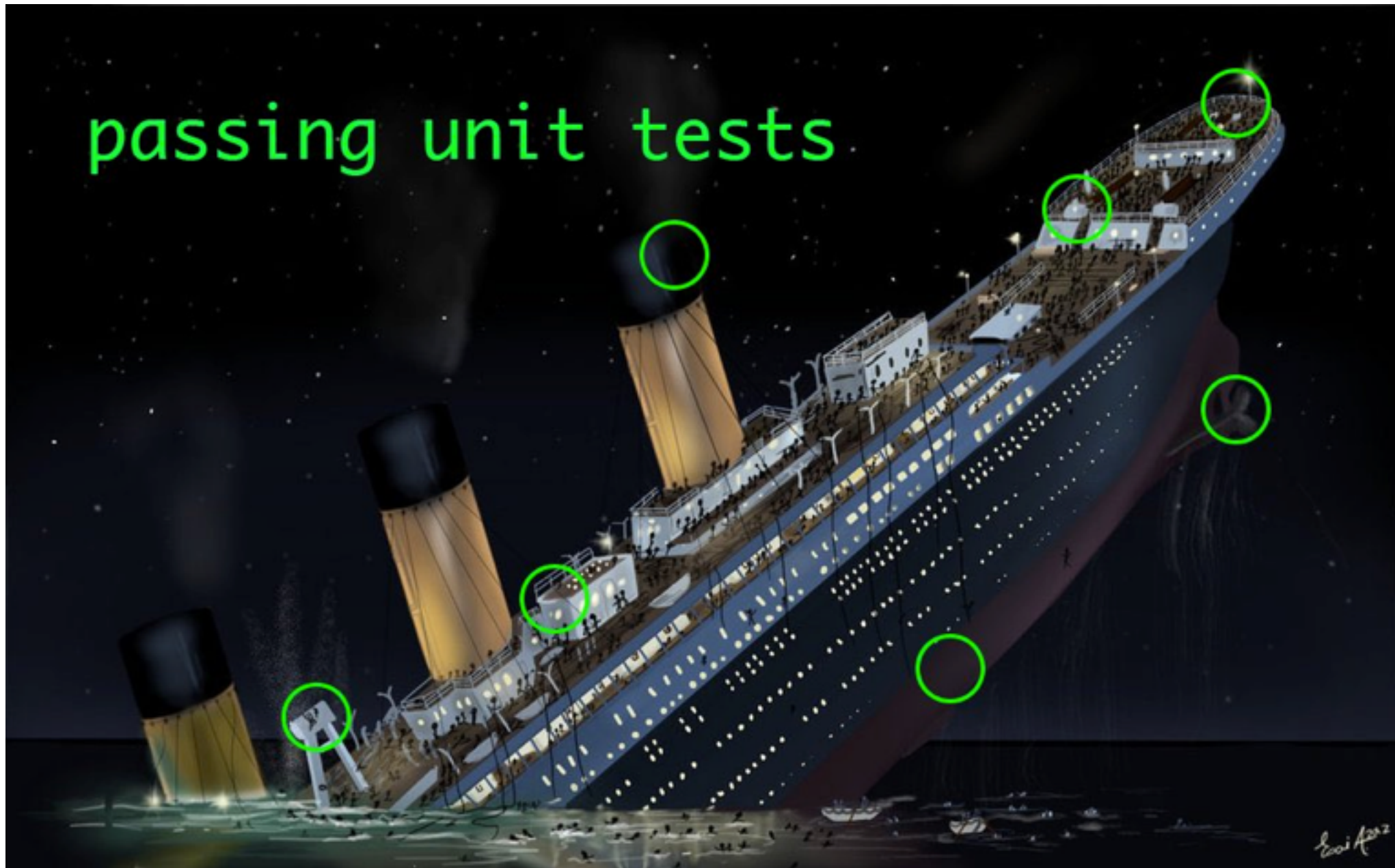


Contenido

6. Testing

- Motivos
- Mocha
- Aserciones con assert y should
- Mocking con Sinon
- Infraestructura de test (ES6, JSX)
- Tests de Componentes
- Tests de Reducers
- Tests de Action Creators
- Cobertura de código

Motivos



¿Para qué nos sirven los tests?

Seguridad (regresiones)

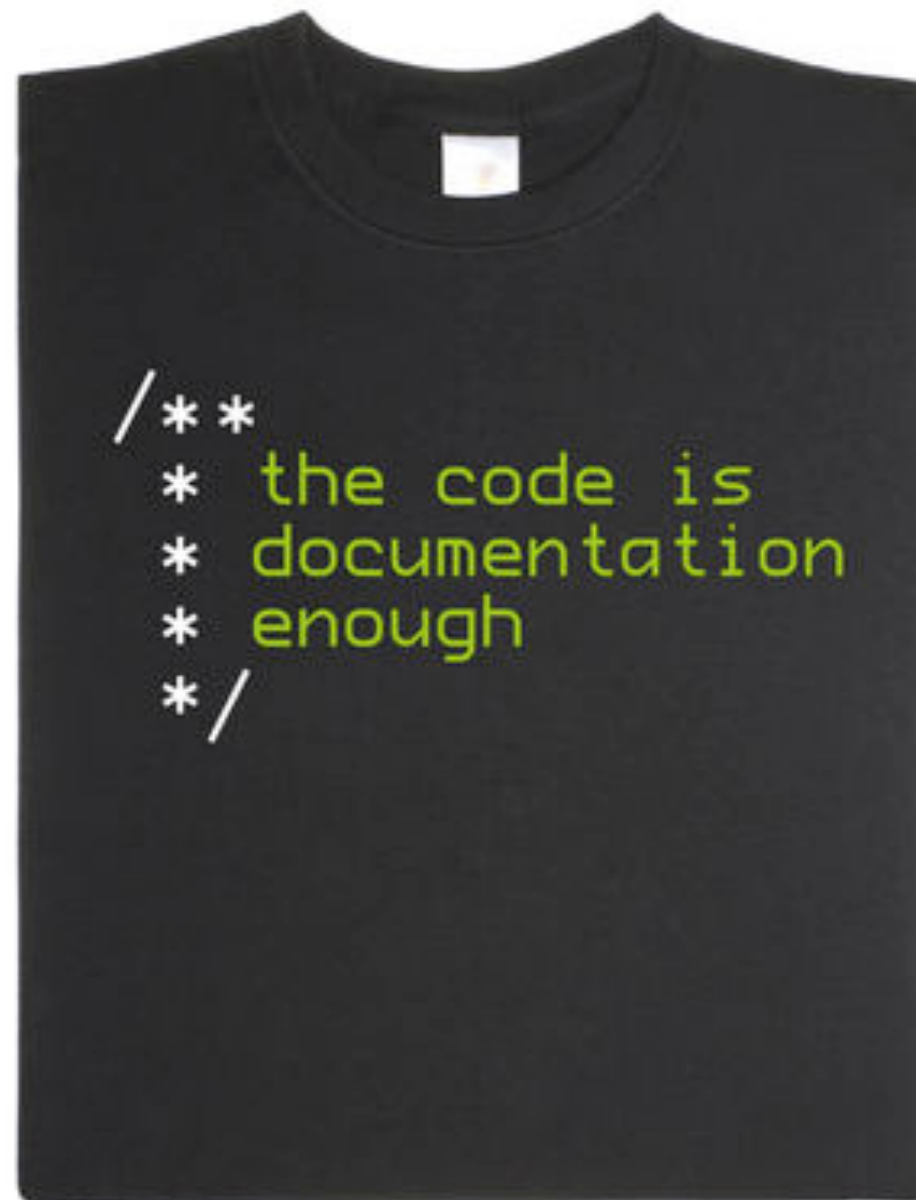


ILLUSTRATION BY SEGUE TECHNOLOGIES

Confianza



Documentación



mocha

- <https://mochajs.org>
- Un framework para ejecutar tests que funciona tanto en **node.js** como en el **browser**
- Muy cómodo para tests asíncronos
- Se puede instalar como dependencia de desarrollo del proyecto, o habitualmente como paquete global:

> npm install -D mocha

> npm install -g mocha

mocha

- Tests -> Suites -> tests unitarios
- Podemos pedir a mocha que ejecute las pruebas dentro de un solo archivo, o dentro de una carpeta (por defecto: *test*)

➔ **mocha ./src/tests**

➔ **mocha ./src/tests/test_concreto.js -w**

➔ **mocha ./src/tests/test_concreto.js --recursive**

mocha

- Tanto las **suites** como los tests unitarios se escriben con funciones anónimas
- Suite:

```
describe ("Mi super suite", function () {  
    //suite anidada o pruebas unitarias..  
})
```


mocha

- Test unitario:

```
describe("Mi super suite", () => {  
  
  it("descripción del test", () => {  
  
    //... código de la prueba  
  
  })  
  
})
```

mocha

- Mocha nos proporciona varias funciones “globales”, no necesitamos importar nada
- ➔ **describe** - describe(“nombre”, *fn*)
Describe una suite, un grupo de pruebas relacionadas
- ➔ **it** - it(“nombre”, *fn*)
Una prueba unitaria

Ejemplo

```
describe('Mi primera suite', function() {  
    it('Mi primer test', function() {  
        //de momento no hago nada  
    });  
});
```

mocha - hooks

- ➔ **before** - `before(fn)`
Se ejecuta una vez al comienzo de la suite dentro de la cual esté incluido
- ➔ **after** - `end(fn)`
Se ejecuta una vez al final de la suite dentro de la cual esté incluido
- ➔ **beforeEach** - `beforeEach(fn)`
Se ejecuta una vez antes de cada test de la suite donde esté incluido
- ➔ **afterEach** - `afterEach(fn)`
Se ejecuta una vez después de cada test de la suite donde esté incluido

Ejemplo

```
describe('Mi primera suite', function() {  
  before(function() {  
    console.log('Hola antes de nada...');  
  });  
  
  after(function() {  
    console.log('Adiós después de todo');  
  });  
  
  beforeEach(function() {  
    console.log('Hola antes de cada prueba');  
  });  
  
  afterEach(function() {  
    console.log('Adiós después de cada prueba');  
  });  
  
  it('Mi primer test', function() {  
    //de momento no hago nada  
  });  
  
});
```

mocha - only

- Durante el desarrollo, podemos ejecutar **solo** una suite/test añadiendo el sufijo **.only**:
 - ➔ `describe.only("Suite", function() { ... })`
 - ➔ `it.only("Test", function() { ... });`
- Muy útil para **depurar** un test concreto

mocha - skip

- Durante el desarrollo, podemos hacer que mocha **no ejecute** una suite/test añadiendo el sufijo **.skip**:

➔ `describe.skip("Suite", function() { ... })`

➔ `it.skip("Test", function() { ... });`

mocha - test asíncrono

- Si queremos ejecutar un test o *hook* asíncrono, o que necesita tiempo para terminar, añadimos un parámetro **done** a la función:

➔ `it("Test asíncrono", function(done)
 { ... });`
- La prueba/hook no termina hasta que llamemos a `done()` dentro del test
- Muy útil para testear **Promesas**, WebSockets, etc.

Ejemplo test asíncrono

```
describe('Mi primera suite', function() {  
  ...  
  
  it('Mi test asincrono', function(done) {  
    setTimeout(function() {  
      console.log('Un segundo después...');  
      done();  
    }, 1000);  
  });  
  
});
```

Aserciones

- Dentro de los tests evidentemente necesitamos realizar aserciones para verificar el comportamiento del sujeto que estamos probando
- node.js proporciona de forma nativa la librería **assert**
- <https://nodejs.org/api/assert.html>

Ejemplo - assert

```
var assert = require('assert');
```

Igualdad ➡

```
describe('Suite con assert', function() {  
  it('Should add 2 + 2', function() {  
    var res = 2 + 2;  
    assert.equal(res, 4);  
  });  
});
```

Desigualdad ➡

```
it('Should add 2 + 2', function() {  
  var res = 2 + 2;  
  assert.notEqual(res, 5);  
});
```

Igualdad profunda (objetos) ➡

```
it('Should compare two objects', function() {  
  var obj1 = { foo: 'bar' };  
  var obj2 = { foo: 'bar' };  
  assert.deepEqual(obj1, obj2);  
});
```

Igualdad profunda (Arrays) ➡

```
it('Should compare two Arrays', function() {  
  var array1 = [1,2,3];  
  var array2 = [1,2,3];  
  assert.deepEqual(array1, array2);  
});
```

Valores “verdaderos” ➡

```
it('Should assert truthy values', function() {  
  var array = [1,2,3];  
  assert(array.length);  
});  
});
```

Aserciones

- Para comprobaciones un poco más complejas (de tipo de datos, de propiedades en objetos, etc) **assert** es un poco *low-level*
- Existen muchas librerías de aserciones para Javascript: **chai**, **expect** o **should**
- Ofrecen una API más rica para realizar evaluaciones

Aserciones - should

- Permite aserciones estilo BDD (*X debería ser/tener*)
- Extiende Object.prototype con lo que se puede llamar directamente sobre las variables que queremos asertar
- Permite encadenar aserciones de forma que el resultado es casi leer lenguaje natural
- Docs: <http://shouldjs.github.io/>

Ejemplo - should

```
var should = require('should');

describe('Suite con should', function() {
  var user = {
    name: 'Carlos',
    pets: ['Mia', 'Leia', 'Rocky', 'Orco']
  }

  it('Should assert properties', function() {
    user.should.have.property('name', 'Carlos');
  });

  it('Should assert on Arrays', function() {
    user.should.have.property('pets').with.length(4);
  });

  it('Should assert on types', function() {
    user.should.be.an.Object;
    user.pets.should.be.an.Array;
  });

  it('Should allow negations', function() {
    user.should.not.have.property('foo');
  });

  it('Should assert on Booleans', function() {
    (false).should.be.false;
    (true).should.be.true;
    (true).should.be.ok;
  });

  it('Should match with regular expressions', function() {
    var subject = 'hola mundo';
    subject.should.match(/hola/);
    subject.should.not.match(/^mundo$/);
  });
});
```


Mocking

- Mock = imitación, un doble falso
- Muy útil y a veces imprescindible
- **¿Por qué?** Para no testear implícitamente "otro" módulo
- **¿Cómo?** Reemplazar las dependencias por *mocks* o imitaciones de esas dependencias

Mocking

- La librería para *mocking* que nosotros utilizamos es **sinon.js**
- <http://sinonjs.org/>
> **npm install -D sinon**
- Nos ofrece una API rica basada en **spies**, **stubs** y **mocks**.
- Empleamos extensivamente **spies** y **stubs** en nuestros tests.

Mocking - Spy

- ¿Qué es un espía?
- Una función que registra **todo**
- Función anónima o envolver una función existente
- Si envolvemos, la original **se llama**



Mocking - Spy

- ➔ **sinon.spy()** - devuelve un espía anónimo
- ➔ **sinon.spy(obj, 'method')** - devuelve un espía que envuelve un método en un objeto.
- ➔ **spy.restore()** - libera el método envuelto por el espía (deja de monitorizarlo)

Mocking - Spy

- ➔ **callCount** - spy.callCount (Number)
Nº de llamadas a la función
- ➔ **called** - spy.called (Boolean)
Indica si el espía ha sido llamado al menos una vez
- ➔ **calledOnce** - spy.calledOnce (Boolean)
Indica si el espía ha sido llamado exactamente una vez
- ➔ **args** - spy.args (Array)
Devuelve un Array con un elemento por cada llamada, que es a su vez un Array con los argumentos
- ➔ **getCall(n)** - spy.getCall(n) (Object)
Devuelve los datos de una llamada específica
- ➔ **reset()** - Reinicia el espía

Mocking - Spy

```
var should = require('should');  
var sinon = require('sinon');
```

```
//Ejemplo mocking
```

```
var myModule = {  
  add: function(a,b) {  
    return a+b;  
  },  
  multiply: function(a,b) {  
    var res = 0;  
    for(var i=b; i > 0; i--) {  
      res = this.add(res,a);  
    }  
    return res;  
  }  
}
```

Mocking - Spy

```
describe.only('Mocking example', function() {  
    var subject = myModule;  
    var addSpy = sinon.spy(subject, 'add');  
  
    beforeEach(function() {  
        addSpy.reset();  
    });  
  
    it('add() should add two numbers', function() {  
        var res = subject.add(1, 2);  
        addSpy.called.should.be.true;  
        res.should.equal(3);  
    });  
  
    it('multiply() should multiply two numbers', function() {  
        var res = subject.multiply(2, 5);  
        res.should.equal(10);  
    });  
  
    it('multiply(a,b) should call add b times', function() {  
        var res = subject.multiply(5, 4);  
        addSpy.callCount.should.equal(4);  
        var firstCall = addSpy.getCall(0);  
        firstCall.args[0].should.equal(0);  
        firstCall.args[1].should.equal(5);  
    });  
});
```


Mocking - Stub

- Un espía **con comportamiento**
- Función anónima o puede envolver una función existente.
- Si envuelve una función existente, la función original **no será llamada**

Mocking - Stub

- ➔ **sinon.stub**(obj, “method”, *fn*)
Devuelve un stub que envuelve el método indicado del objeto, con comportamiento definido por *fn*
- ➔ *stub.restore*()
Restaura el método original

Mocking - Stub



Controlar ruta del código

Mocking - Stub

```
var module = {  
  doSomething: function (a,b) {  
    API.getJSON(...)  
    .then( ... )  
    .catch( ... )  
  }  
}
```

Mocking - Stub

```
it('API fails it should ... ', function(done) {  
  var stub = sinon.stub(API, 'getJSON', function() {  
    return Promise.reject({ text: 'BOOM' });  
  });  
  module.doSomething(1,1)  
    .catch(err => {  
      err.text.should.equal('BOOM');  
      done();  
    });  
});
```

Mocking - Stub

```
it('API fails it should ... ', function(done) {  
  ➔ var stub = sinon.stub(API, 'getJSON', function() {  
    return Promise.reject({ text: 'BOOM' });  
  });  
  module.doSomething(1,1)  
    .catch(err => {  
      err.text.should.equal('BOOM');  
      done();  
    });  
});
```

Inyectamos directamente el resultado esperado de un módulo externo para controlar la ruta que toma el código

Infraestructura de tests

- Necesitamos **Babel** si queremos:
 - escribir tests con ES6
 - poder importar nuestros archivos ES6 (JSX!)
 - **npm install -D babel-core**
- Podemos indicar a **mocha** que compile con Babel los archivos ***.js** (opción **--compilers**)
- La configuración de Babel la tomará de **.babelrc**

Infraestructura de tests

- package.json (ejercicios/tema6)

```
"scripts": {  
  "test": "cross-env NODE_ENV=test mocha --opts mocha.opts",  
  "tdd": "cross-env NODE_ENV=test mocha --opts mocha.opts -w",  
  "cover": "...",  
  "build": "cross-env NODE_ENV=production webpack -p",  
  "start": "cross-env NODE_ENV=development webpack-dev-server -d --inline --hot"  
}
```

- mocha.opts (opciones para mocha)

```
./src/test  
--compilers js:babel-core/register  
--recursive
```

Tests de Componentes

- ¿Qué probamos en los componentes?
- La **salida**, y las variaciones que deben producir diferentes props
- El **comportamiento**: las interacciones llaman a los métodos correctos (action creators o props recibidas del padre, normalmente)



Tests de Componentes - salida

- Testear React => ¿Testear DOM?
- Opción 1: utilizar un browser real para las pruebas, o un browser “programable” como PhantomJS (con **karma** por ejemplo)
- Opción 2: emular un entorno browser para que React funcione correctamente, pero podemos ejecutar los tests desde node.js (JSDOM)
- Opción 3: utilizar render superficial (**shallow rendering**) y ahorrarnos todo ese setup

Tests de Componentes - salida

- Opción 1: utilizar un browser real para las pruebas, o un browser “programable” como PhantomJS (con **karma** por ejemplo)
- Opción 2: emular un entorno browser para que React funcione correctamente, pero podemos ejecutar los tests desde node.js (JSDOM)
- Opción 3: utilizar render superficial (**shallow rendering**) y ahorrarnos todo ese setup

Lento



Rápido

Tests de Componentes - salida

- Además de poder hacer **render** superficial de un componente, necesitaremos:
- Extraer información del componente montado (props, ¿existe o no?, qué componentes hijos tiene, etc.)
- React proporciona estas utilidades bajo **React.addons.TestUtils**

npm: react-addons-test-utils

Test de Componentes - shallow rendering

- El render superficial genera un solo nivel de profundidad del árbol de componentes
- Nos permite restringir nuestros tests al componente que estamos probando, y no entrar en los componentes hijos, nietos, etc.
- No utiliza DOM, ni real ni virtual, y por tanto es muy rápido

Test de Componentes - shallow rendering

```
//src/components/counter.js
import React, { Component, PropTypes } from 'react';
import { connect } from 'react-redux';

export class Counter extends Component {
  render() {
    const { clicks, dispatch } = this.props,
          incrementAction = { type: 'INCREMENT' };

    return (
      <button onClick={ () => dispatch(incrementAction) }>
        Has hecho click { clicks } veces
      </button>
    );
  }
}

const mapStateToProps = state => ({ clicks: state })

export default connect(mapStateToProps)(Counter);
```

Test de Componentes - shallow rendering

```
➡ import React from 'react';
import { createRenderer } from 'react-addons-test-utils';
//node.js basic assertion lib
import assert from 'assert';
import { spy } from 'sinon';
//named import, not connected version!
import { Counter } from '../components/counter';

describe('Counter component', () => {
  let renderer, counter, dispatch = spy();

  beforeEach(() => {
    ➡ renderer = createRenderer();
    ➡ renderer.render(<Counter clicks={ 5 } dispatch={ dispatch } />);
    counter = renderer.getRenderOutput();

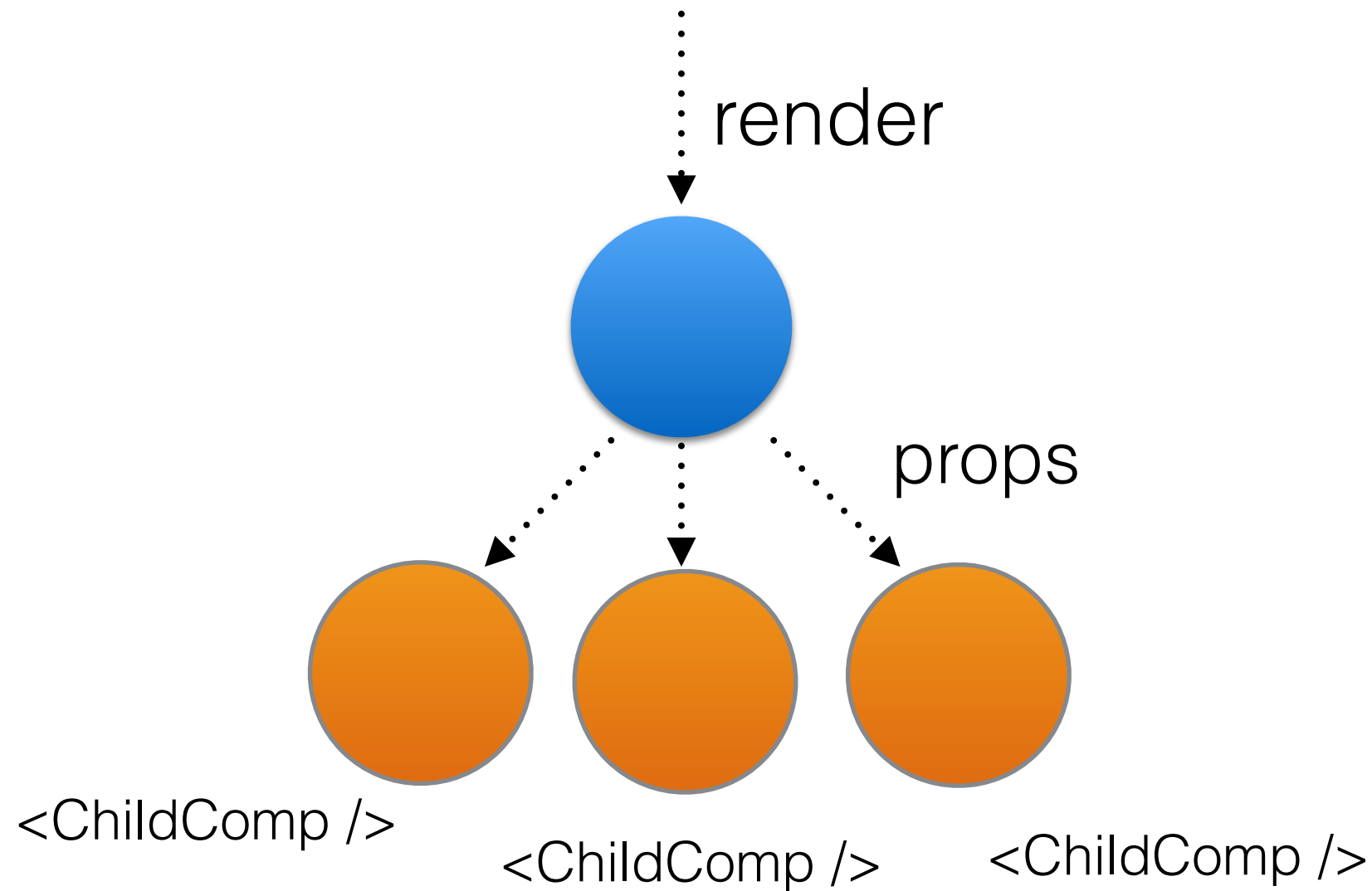
    // resto del test ...
  });
});
```


Test de Componentes - shallow rendering

```
{ '$$typeof': Symbol(react.element),  
  type: 'button',  
  key: null,  
  ref: null,  
  props:  
    { onClick: [Function: onClick],  
      children: [ 'Has hecho click ', 5, ' veces' ] },  
  _owner: null,  
  _store: {} }
```

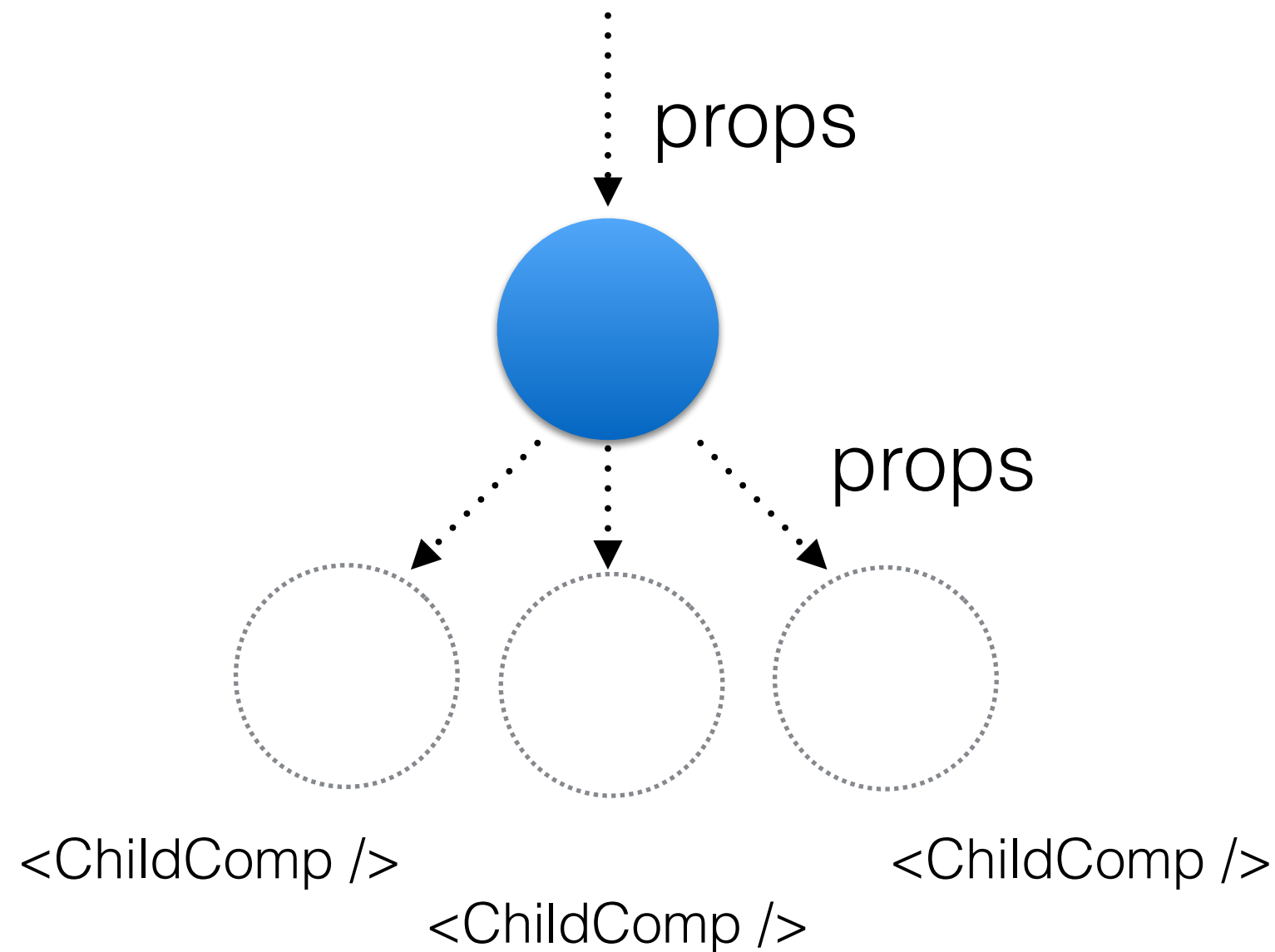
Test de Componentes - shallow rendering

ReactDOM.render(<MyComponent />)



Test de Componentes - shallow rendering

shallow render(<MyComponent a={1} b={2} ... />)



Test de Componentes - shallow renderer

➔ **ReactTestUtils.createRenderer()**

Nos devuelve un "renderer" con la siguiente API

- **renderer.render(Component, props)**
Render superficial de un componente (clase, función) con props
- **renderer.getMountedInstance()**
Nos devuelve la instancia del componente
- **renderer.getRenderOutput()**
La salida del render superficial (= retorno de render())
- **renderer.unmount()**
"Desmonta" el componente

Test de Componentes - shallow renderer

- ➔ Como siempre es igual, podemos usar una función auxiliar común a nuestros tests de componentes

```
// ejercicios/tema6/src/test/helpers.js
import React from 'react';
import { createRenderer } from 'react-addons-test-utils';

export function shallowRender(Component, props = {}){
  let renderer = createRenderer();
  let output = renderer.render(React.createElement(Component, props));
  return {
    //instancia del componente (con los métodos)
    instance: renderer.getMountedInstance(),
    //salida del shallow renderer
    output
  }
}
```

Test de Componentes - inspeccionar el resultado

- Podemos inspeccionar el resultado del render y hacer aserciones, por ejemplo:

```
it('Should render a Header and a catalog list', () => {  
  const { output, instance } = shallowRender(Catalog);  
  assert.equal(output.props.children[0].type.name, 'Header');  
  assert.equal(output.props.children[0].props.text, 'Productos');  
  assert.equal(output.props.children[1].props.className, 'catalog-list');  
});
```


Test de Componentes - inspeccionar con skin-deep

- npm install -D skin-deep

```
import sd from 'skin-deep';  
//...  
it('Should render a Header and a catalog list', () => {  
  const tree = sd.shallowRender(<Catalog />);  
  const Header = tree.subTree('Header'),  
    list = tree.subTree('.catalog-list');  
  //existe un Header  
  assert(Header);  
  //Con el texto correcto configurado  
  assert.equal(Header.props.text, 'Products');  
  //existe un div.catalog-list  
  assert(list);  
});
```

Test de Componentes - buscar por clase / tipo de componente


```
import sd from 'skin-deep';  
//...  
it('Should render a Header and a catalog list', () => {  
  const tree = sd.shallowRender(<Catalog />);  
  const Header = tree.subTree('Header'),  
    list = tree.subTree('.catalog-list');  
  //existe un componente Header  
  assert(Header);  
  //Con el texto correcto configurado  
  assert.equal(Header.props.text, 'Products');  
  //existe un div.catalog-list  
  assert(list);  
});
```



Por la clase del componente
Header

Test de Componentes - buscar por selector CSS

- npm install -D skin-deep

```
import sd from 'skin-deep';  
//...  
it('Should render a Header and a catalog list', () => {  
  const tree = sd.shallowRender(<Catalog />);  
  const Header = tree.subTree('Header'),  
     list = tree.subTree('.catalog-list');  
  //existe un component Header  
  assert(Header);  
  //Con el texto correcto configurado  
  assert.equal(Header.props.text, 'Products');  
  //existe un div.catalog-list  
  assert(list);  
});
```

Por la clase CSS al estilo jQuery:
.catalog-list

Test de Componentes - asertar existencia

- npm install -D skin-deep

```
import sd from 'skin-deep';
//...
it('Should render a Header and a catalog list', () => {
  const tree = sd.shallowRender(<Catalog />),
    output = tree.getRenderOutput();
  const Header = tree.subTree('Header'),
    list = tree.subTree('.catalog-list');
  //existe un componente Header
  ➔ assert(Header);
  //Con el texto correcto configurado
  assert.equal(Header.props.text, 'Products');
  //existe un div.catalog-list
  assert(list);
});
```

Si subTree() no encuentra resultados devuelve **false**

Test de Componentes - subárboles

➔ **tree.subTree**(selector, [*predicado*])

Devuelve el **primer** componente que cumpla el selector

➔ **tree.everySubTree**(selector, [*predicado*])

Devuelve un Array con todos los componentes/nodos que cumplan el selector

➔ Las llamadas a tree son "chainables"

Test de Componentes - subárboles

➔ tree

.subTree(".catalog")

.subTree(".catalog-list")

.everySubTree("CatalogItem")

Devolverá todos los componentes CatalogItem dentro de div.catalog-list, dentro de div.catalog

Test de Componentes - subárboles con .dive()

➔ tree

```
.dive([".catalog", ".catalog-list"])  
.everySubTree("CatalogItem")
```

Devolverá todos los componentes CatalogItem dentro de div.catalog-list, dentro de .catalog

Test de Componentes - examinar subárbol

- ➔ **tree.type**

El **tipo** del elemento (String / clase React)

- ➔ **tree.props**

El objeto props de React

- ➔ **tree.text()**

Representación de JSX como String "<Header />" si es un componente, o representación HTML/texto si es un control HTML

- ➔ **tree.toString()**

Representación como HTML "<div....>"

Test de Componentes - Comportamiento

- Como no tenemos DOM con shallow render, no podemos simular eventos del DOM
- Pero podemos llamar directamente a los métodos
 - Si es una prop, tenemos acceso a las props (Ej: props.onClick)
 - Si es un método de la clase (ej. "handleClick") podemos llamarlo directamente **accediendo a la instancia del componente**

Test de componentes - llamar a métodos de instancia

```
it('Should dispatch addToCart and push in handleAddToCart', () => {  
  const push = spy(),  
        addToCart = spy();  
  const tree = sd.shallowRender(<Catalog push={ push } addToCart={ addToCart } />)  
  ➡ const instance = tree.getMountedInstance();  
  
  //call the instance method  
  ➡ instance.handleAddToCart(2);  
  
  assert.equal(addToCart.calledOnce, true);  
  assert.equal(addToCart.getCall(0).args[0], 2);  
  assert.equal(push.calledOnce, true);  
  assert.equal(push.getCall(0).args[0], 'cart');  
});
```


Test de componentes - helper skin deep

```
//src/test/helpers.js
import React from 'react';
import sd from 'skin-deep';

export function shallowRender(Component, props = {}){
  let tree = sd.shallowRender(React.createElement(Component, props));
  return {
    //component instance for method testing
    instance: tree.getMountedInstance(),
    //component output tree
    tree
  }
}
```

Ejemplo de uso

```
const { tree, instance } = shallowRender(Catalog, { isFetching: true }),
      loadingMessage = tree.subTree('.catalog-list').textIn();
```

Ejercicio - Carrito de la compra

- Escribe un test unitario para el componente **Cart** de la aplicación del carrito de la compra
- Componente en **ejercicios/tema6/src/components/ecommerce/cart.js**
- Esqueleto del test en **ejercicios/tema6/src/test/components/cart.js**

Ejercicio - Carrito de la compra - ¿qué testamos?

- **Salida del componente**

- Existe un componente **Header** y tiene configurado el texto "Tu compra"
- Existe un componente **CartItem** por cada producto en el carro
- Existe una celda con el precio total del carrito
- Existe un botón para volver al catálogo
- Existe un botón de Checkout, sólo cuando el carrito tiene productos

Ejercicio - Carrito de la compra - ¿qué testamos?

- **Comportamiento del componente**
 - Llama a la función **goToCatalog** (recibida como prop) en el click del botón de volver al catálogo
 - Llama a la función **goToCheckout**(recibida como prop) en el click del botón de finalizar compra
 - Llama a la función **changeQuantity** con los argumentos correctos, al cambiar la cantidad de un producto con los botones

Tests de Reducers

- Más fácil imposible, llamar a una función y hacer aserciones sobre el nuevo estado que nos devuelve
- Jugamos con el primer argumento (state) y el objeto Action para buscar las condiciones específicas de cada test

Tests de Reducers - ejemplo

```
describe('Reducer', () => {
  const fakeAction = { type: '@@INIT' },
        loadAttempted = { type: LOAD_CATALOG_ATTEMPTED },
        loadSucceeded = { type: LOAD_CATALOG_SUCCEEDED, payload: [1, 2, 3] },
        loadFailed = { type: LOAD_CATALOG_FAILED, error:
                        { status: 404, text: 'Not found' } };

  const initialState = {
    data: [],
    isFetching: false,
    error: {}
  };

  it('Should return valid initial state', () => {
    var state = reducer(undefined, fakeAction);
    assert.deepEqual(state, initialState);
  });

  it('Should activate isFetching flag with load attempt', () => {
    var state = reducer(undefined, loadAttempted);
    assert.equal(state.isFetching, true);
  })
  //...
```

Tests de Action Creators

- Lo podemos incluir dentro del test del reducer, y tener una suite del módulo completo, con dos suites anidadas ("Reducer", "Actions").
- Para los básicos no hay mucho que testear (mapeo de argumentos de la función a propiedades en el objeto action)
- Para los thunks, tendremos que llamarlos con espías como **dispatch** y **getState**
- Hacer aserciones sobre las acciones enviadas a **dispatch**

Tests de Action Creators con acceso a API

- El caso de uso más habitual de thunks
- Tenemos que mockear **fetch** para establecer rutas dentro del thunk (éxito, fallo)
- Existe (cómo no) una librería: **fetch-mock**
- `npm install -D fetch-mock`

fetchMock

- `fetchMock.mock(ruta, respuesta)`
Establece la respuesta predefinida para llamadas a esa ruta con `fetch`. Encadenable.
Ruta = String | RegExp | Function(url, options)
- `fetchMock.reset` - reinicia el mock (ej. entre test y test)
- `fetchMock.restore` - desenvuelve `fetch` original

fetchMock - mock()

fetchMock

```
.mock('/api/products', [1,2,3])  
.mock('/api/login', 'POST', { success: true })  
.mock(/products\\/\\d+$/, 'PUT', { id: 1, name: 'Sphero', price: 199.99 })  
.mock('/products\\/\\d+$', 'DELETE', { status: 403, throws: 'Not an admin' })
```

fetchMock - aserciones

- fetchMock.**called**(*ruta*) -> Boolean
Espía sobre llamadas a fetch
- fetchMock.**calls**(*ruta*)
Devuelve un objeto con las rutas capturadas y las no capturadas
{ matches: [], unmatched: [] }
- fetchMock.**lastCall**(*ruta*) -> []
Devuelve los argumentos de la última llamada a fetch
- fetchMock.**lastUrl**(*ruta*) -> String
Devuelve la URL de la última llamada capturada a fetch
- ...

fetchMock - aserciones

```
fetchMock
  .mock('http://domain1', 200)
  .mock('http://domain2', 'PUT', {
    affectedRecords: 1
  });

myModule.onlyCallDomain2()
  .then(() => {
    expect(fetchMock.called('http://domain2')).to.be.true;
    expect(fetchMock.called('http://domain1')).to.be.false;
    expect(fetchMock.calls().unmatched.length).to.equal(0);
    expect(JSON.parse(fetchMock.lastUrl('http://domain2'))
      .to.equal('http://domain2/endpoint');
  })
  .then(fetchMock.restore)
```

Tests de Action Creators con acceso a API

```
import 'isomorphic-fetch';
//fetch mock
import fetchMock from 'fetch-mock';

describe('Actions', () => {
  let dispatch = spy(),
      getState = spy();

  beforeEach(() => {
    dispatch.reset();
    fetchMock.reset();
  });

  it('Should fetch /api/products.json and dispatch LOAD_CATALOG_SUCCEEDED on success', (done) => {
    const thunk = fetchProducts();
    //preparar respuesta
    fetchMock.mock('/api/products.json', [1,2,3,4]);
    //ejecutar thunk
    thunk(dispatch, getState).then(() => {
      //se ejecutó fetch
      assert.equal(true, fetchMock.called('/api/products.json'));
      assert.equal(dispatch.callCount, 2);
      assert.equal(dispatch.getCall(0).args[0].type, LOAD_CATALOG_ATTEMPTED);
      assert.equal(dispatch.getCall(1).args[0].type, LOAD_CATALOG_SUCCEEDED);
      done();
    });
  });
});
```

Ejercicio - Testear reducer Cart

- Vamos a testear el reducer que gestiona el carrito de la compra
- Implementación en `src/modules/cart/index.js`
- Esqueleto en `src/test/modules/cart/index.js`
- Debemos cubrir las siguientes acciones:
ADD_TO_CART
REMOVE_FROM_CART
CHANGE_QUANTITY
EMPTY_CART

Ejecutar suites múltiples

- Hay que intentar evitar contaminar el objeto global
- Todas nuestras variables para cada suite, dentro de **describe(...)**
- Importante: dejar el “entorno” después de cada suite como estuviera antes (before(), after())

Cobertura de código

- Se pueden generar informes de code coverage a partir los tests de mocha
- El informe nos dirá, para los módulos probados, por dónde ha pasado el código y por donde no, dándonos un porcentaje de cobertura

Cobertura de código

- Hay muchas librerías que generan el informe, a partir de un formato estándar compatible
- Necesitamos una que entienda ES6, y que nos muestre las fuentes en ES6 (JSX, no transpilado)
- npm: istanbul, isparta

Cobertura de código

- Lo que hace Isparta / Istanbul es "cubrir" la ejecución de los tests y generar el informe al final
- Curiosamente hay que ejecutarlo "babelizado"
- Instalación local:
`npm install -D babel-cli isparta`

Cobertura de código

```
"scripts": {  
  "test": "cross-env NODE_ENV=test mocha --compilers js:babel-core/register ./src/test --recursive",  
  "tdd": "cross-env NODE_ENV=test mocha --compilers js:babel-core/register ./src/test --recursive -w",  
  "cover": "babel-node ./node_modules/.bin/isparta cover _mocha -- --compilers js:babel-core/register ./src/test --recursive -R spec",  
  "build": "cross-env NODE_ENV=production webpack -p",  
  "start": "cross-env NODE_ENV=development webpack-dev-server -d --inline --hot"  
},
```

Cobertura de código

package.json

```
"scripts": {  
  "test": "cross-env NODE_ENV=test mocha --opts mocha.opts",  
  "tdd": "cross-env NODE_ENV=test mocha --opts mocha.opts -w",  
  "cover": "babel-node ./node_modules/.bin/istanbul cover _mocha -- --opts mocha.opts",  
  "build": "cross-env NODE_ENV=production webpack -p",  
  "start": "cross-env NODE_ENV=development webpack-dev-server -d --inline --hot"  
},
```

mocha.opts

```
./src/test  
--compilers js:babel-core/register  
--recursive
```

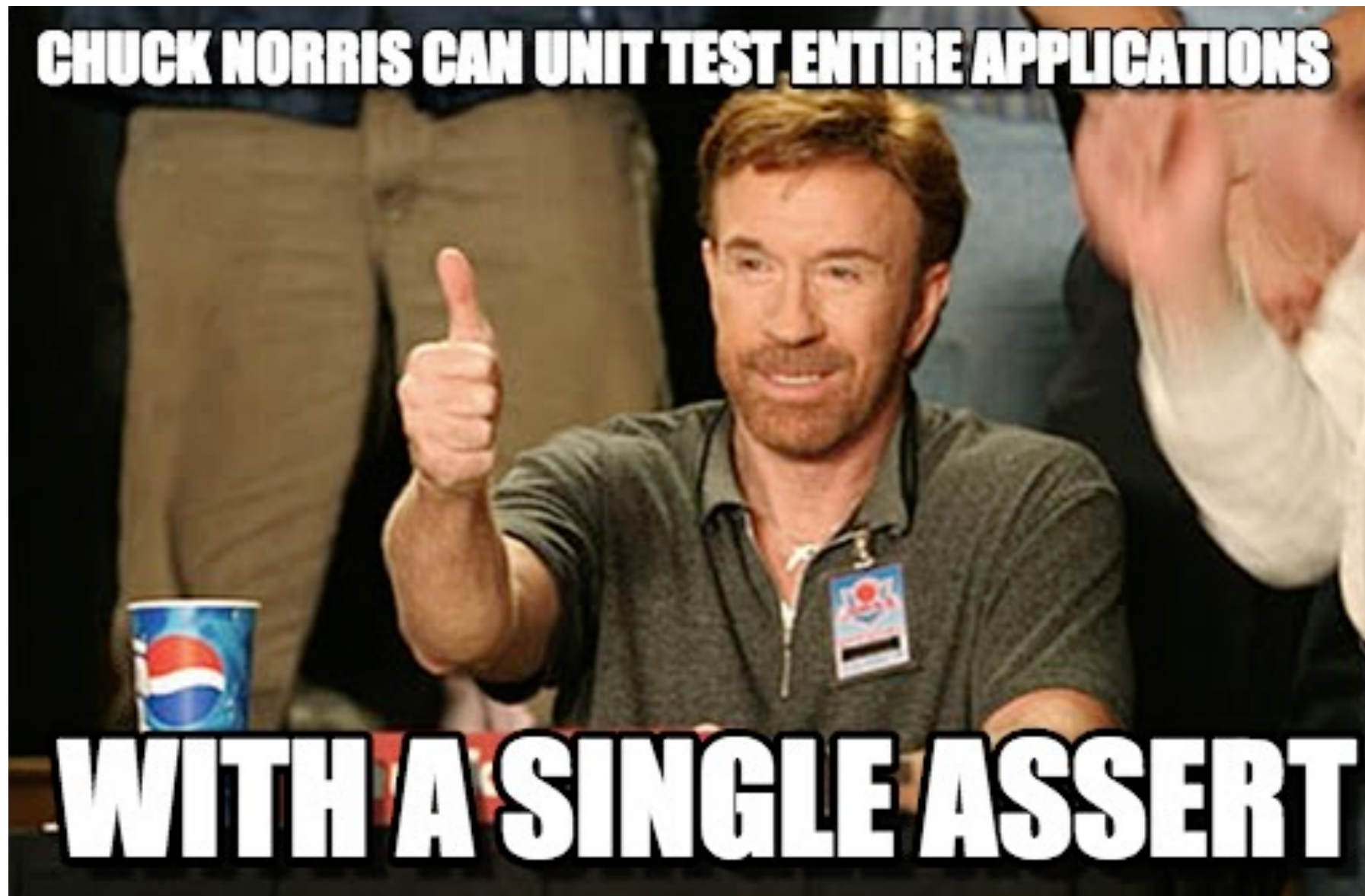
Cobertura de código

all files / components/ecommerce/ catalog.js

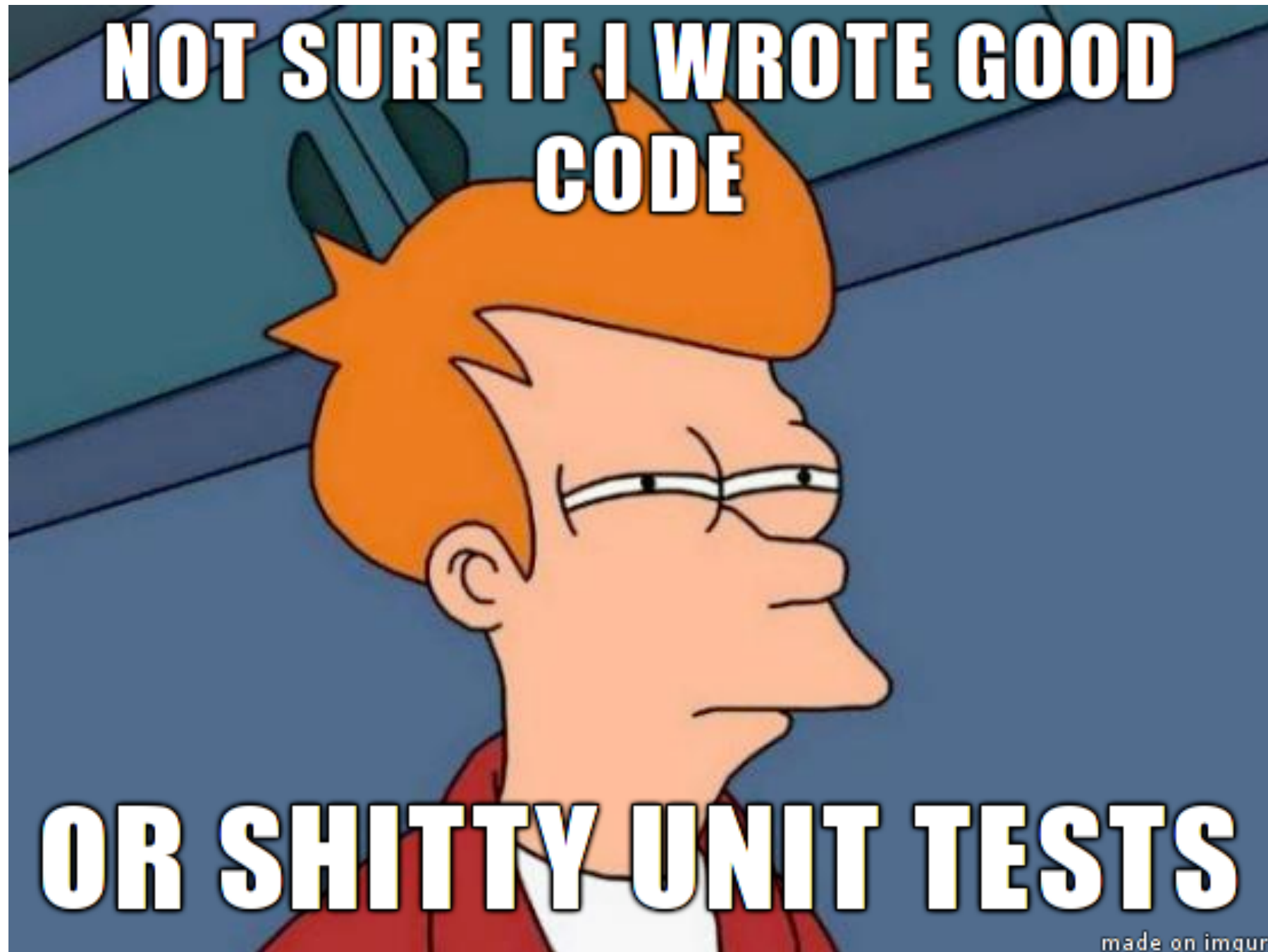
98.51% Statements 66/67 100% Branches 39/39 76.47% Functions 13/17 95.65% Lines 22/23

```
1 1x import React, { Component, PropTypes } from 'react';
2 1x import { connect } from 'react-redux';
3 1x import { get } from '../../lib/api';
4 1x import { fetchProducts } from '../../modules/catalog';
5 1x import { addToCart } from '../../modules/cart';
6 1x import { push } from 'react-router-redux';
7 1x import catalogData from '../../data/shopping_cart';
8 1x import Header from './header';
9 1x import CatalogItem from './catalog_item';
10
11 // Listado de productos de la tienda
12 export class Catalog extends Component {
13 7x   constructor(props){
14     super(props);
15 7x   this.handleAddToCart = this.handleAddToCart.bind(this);
16   }
17
18   //componentDidMount() NO se ejecuta con react shallow render!
19   componentDidMount(){
20     //Simulamos un delay
21 2x   if(this.props.items.length === 0){
22 1x     this.props.fetchProducts();
23   }
24 }
25
26 handleAddToCart(id){
27   //añadir producto al carrito
28 1x   this.props.addToCart(id);
29   //y luego navegar a la página del carrito
30 1x   this.props.push('cart');
31 }
```

Resumen



Resumen



Resumen - unit testing React y Redux

- Test de UI sencillos y rápidos con shallow rendering
- Tests de reducers sencillos por definición
- Tests de action creators asíncronos: jugar con promesas y espías
- Tests de middlewares (ej. en soluciones), relativamente sencillo
- Tests de selectores (ej. en soluciones)

Resumen

- Sólo hemos cubierto tests unitarios
- Tests de integración (no mocks, no shallow render)
- Tests End to End (nightwatch / Selenium)
- Recomendable un servidor de integración continua

¡Gracias!

