

Contenido

- 5. Testing
 - Motivos
 - Mocha
 - Aserciones con assert y should
 - Mocking con Sinon
 - Dependencias ocultas y Rewire
 - Infraestructura de test
 - Tests de Componentes
 - Tests de Stores
 - Informes de cobertura de código

Motivos

¿Para qué nos sirven los tests?

- Seguridad para hacer cambios (detectaremos lo que rompamos rápidamente)
- Confianza en el código
- Documentación del módulo que testeamos para otros desarrolladores

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, pero es mucho más cómodo si lo instalamos globalmente con

> npm install -g mocha

mocha

- Los tests se organizan en **suites** y dentro de cada suite tendremos uno o varios **tests unitario**. Un mismo archivo puede tener varias **suites**, e incluso se pueden anidar.
- Podemos pedir a mocha que ejecute las pruebas dentro de un solo archivo, o dentro de una carpeta (por defecto: *test*)

➔ **mocha ./src/tests**

- Ejecutará todos los tests que encuentre dentro de esa carpeta

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

- El flag **-w** (*watch*) hará que mocha vuelva a ejecutar los tests cuando detecte un cambio en los archivos especificados en la ruta (muy útil para TDD)

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

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

```
describe("Mi super suite", function() {  
  
  it("descripción del test", function() {  
  
    //... 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

- Además nos proporciona *hooks* durante la ejecución de los tests para configurar y limpiar el entorno de pruebas:
 - ➔ **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

- 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, especialmente combinado con watch (-w)

mocha

- 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

- 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 necesario para hacer tests unitarios
- Queremos comprobar el código del módulo que estamos testando, no depender del resultado de otros módulos
- Para ello, tenemos que reemplazar las dependencias por *mocks* o imitaciones de esas dependencias

Mocking

- La librería para *mocking* que nosotros utilizamos es **sinon.js**
- <http://sinonjs.org/>
- 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 los argumentos, el valor de retorno, el contexto y las excepciones lanzadas para cada una de las llamadas al espía.
- Puede ser una función anónima o puede envolver una función existente
- Si envuelve una función, al llamar al espía se ejecuta la función original

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`
Nº de llamadas a la función
- ➔ **called** - `spy.called`
Indica si el espía ha sido llamado al menos una vez
- ➔ **args** - `spy.args`
Devuelve un Array con un elemento por cada llamada, que es a su vez un Array con los argumentos
- ➔ **getCall(n)** - `spy.getCall(n)`
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 - 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

- ¿Qué es un stub?
- Un espía con comportamiento predefinido
- Puede ser una 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

- Los stubs son muy útiles para determinar las rutas en el código
- Donde encontremos una llamada a un módulo o función externa al sujeto de la prueba, podemos poner un stub para controlar la salida

Mocking - Stub

```
var otherModule = {  
  doSomething: function (a,b) {  
    if (myModule.multiply(a,b) > 10) {  
      return 10;  
    }  
    else {  
      return 1;  
    }  
  }  
}
```

Mocking - Stub

```
it('otherModule should return 1 if multiply < 10', function() {  
    var stub = sinon.stub(myModule, 'multiply', function() {  
        return 0;  
    });  
    var res = otherModule.doSomething(1, 1);  
    res.should.equal(1);  
    stub.restore();  
});
```


Mocking - Stub

```
it('otherModule should return 1 if multiply < 10', function() {  
  ➔ var stub = sinon.stub(myModule, 'multiply', function() {  
    return 0;  
  });  
  var res = otherModule.doSomething(1, 1);  
  res.should.equal(1);  
  stub.restore();  
});
```

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

Dependencias ocultas

- En los ejemplos que hemos visto, mockear una dependencia era sencillo porque estaban accesibles y la vista.
- Pero ¿y si el módulo que testeamos no exporta sus dependencias?

Dependencias ocultas

```
var _ = require('mori'),
    atom = require('../lib/atom_state'),
    Dispatcher = require('../lib/dispatcher');

var p = {
  page: ['page']
};


var RootStore = {
  getPage: function(state) {
    return _.getIn(state, p.page);
  },

  setPage: Dispatcher.listen("SET:PAGE", function(newPage) {
    atom.assocIn(p.page, newPage);
  })
};


Dispatcher.listen("ORDER:COMPLETE", function() {
  RootStore.setPage('thank-you');
});

module.exports = RootStore;
```

Dependencias ocultas



```
var _ = require('mori'),  
    atom = require('../lib/atom_state'),  
    Dispatcher = require('../lib/dispatcher');
```



```
var p = {  
  page: ['page']  
};
```

```
var RootStore = {  
  getPage: function(state) {  
    return _.getIn(state, p.page);  
  },  
  
  setPage: Dispatcher.listen("SET:PAGE", function(newPage) {  
    atom.assocIn(p.page, newPage);  
  })  
  
};  
  
Dispatcher.listen("ORDER:COMPLETE", function() {  
  RootStore.setPage('thank-you');  
});
```

```
module.exports = RootStore;
```

Dependencias externas, no visibles
al código que instancie este módulo

Dependencias ocultas

- Utilizamos **Rewire**
- <https://github.com/jhnns/rewire>
- Para inyectar dependencias en módulos de node.js
- Lo utilizamos **en lugar de require(xxx)**
- Nos permite acceder a los componentes **internos** del módulo

Dependencias ocultas

- ➔ **rewire**(modulo) - Nos devuelve el módulo, exactamente igual que **require**, pero le añade dos nuevas funciones
- ➔ modulo.__**set**__(dependencia, valor)
Nos permite sustituir la variable privada **dependencia** por otra.
- ➔ modulo.__**set**__({ dep: valor, dep: valor, dep:valor })
Nos permite inyectar múltiples dependencias de una vez
- ➔ modulo.__**get**__("privado")
Nos permite acceder a una variable privada para obtener su contenido

Dependencias ocultas

- **Importante:** `__set__` nos devuelve una **función**
- Esta función (que por convención se suele llamar **restore**), nos permite reestablecer los valores originales en el módulo que hayamos recableado.
- Es muy importante usar **before()** para recablear...
- Y **after()** para restaurar

Dependencias ocultas

```
var fs = require("fs");

function readSomethingFromFileSystem(path, cb) {
  console.log("Reading from file system ...");
  fs.readFile(path, "utf8", cb);
}

exports.readSomethingFromFileSystem = readSomethingFromFileSystem;
```


Dependencias ocultas

➔ **var** fs = require("fs");

```
function readSomethingFromFileSystem(path, cb) {  
  console.log("Reading from file system ...");  
  fs.readFile(path, "utf8", cb);  
}
```

```
exports.readSomethingFromFileSystem = readSomethingFromFileSystem;
```

Para testar este módulo, necesitamos mockear el módulo nativo de Node **fs**

Dependencias ocultas

```
➔ var rewire = require("rewire");  
➔ var myModule = rewire("../lib/myModule.js");  
➔ var fsMock = {  
  readFile: function (path, encoding, cb) {  
    cb(null, "Success!");  
  }  
};  
➔ revert = myModule.__set__("fs", fsMock);  
  
myModule.readSomethingFromFileSystem(function (err, data) {  
  console.log(data); //Success!  
});
```

Cargamos el módulo con rewire, y le cambiamos **fs** por nuestro **fsMock** cuyo comportamiento controlamos

Dependencias ocultas

- Con **Rewire**, podemos eliminar todos los factores externos que influyen en el código que estamos probando
- Eliminamos accesos a disco, red, datos externos
- Unido a **sinon**, nos permite controlar qué obtiene el módulo testeado del “mundo exterior”

Dependencias ocultas



```
var _ = require('mori'),  
    atom = require('../lib/atom_state'),  
    Dispatcher = require('../lib/dispatcher');
```

```
var p = {  
  page: ['page']  
};
```

```
var RootStore = {  
  getPage: function(state) {  
    return _.getIn(state, p.page);  
  },
```

```
  setPage: Dispatcher.listen("SET:PAGE", function(newPage) {  
    atom.assocIn(p.page, newPage);  
  })
```

```
};
```

```
Dispatcher.listen("ORDER:COMPLETE", function() {  
  RootStore.setPage('thank-you');  
});
```

```
module.exports = RootStore;
```

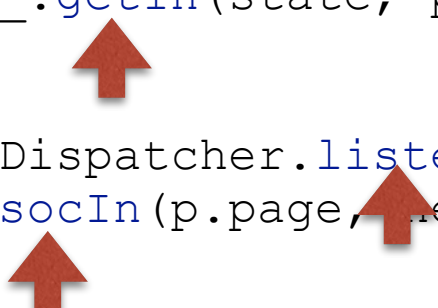
Tendremos que mockear las dependencias que vemos al comienzo del módulo

Dependencias ocultas

```
var _ = require('mori'),  
    atom = require('../lib/atom_state'),  
    Dispatcher = require('../lib/dispatcher');
```

```
var p = {  
  page: ['page']  
};
```

```
var RootStore = {  
  getPage: function(state) {  
    return _.getIn(state, p.page);  
  },  
  
  setPage: Dispatcher.listen("SET:PAGE", function(newPage) {  
    atom.assocIn(p.page, newPage);  
  })  
};
```



```
Dispatcher.listen("ORDER",  
  RootStore.setPage('th  
));
```

```
module.exports = RootStore;
```

Pero sólo las funciones que nos interesan
(mori.getIn, Dispatcher.listen, atom.assocIn)

Dependencias ocultas

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

```
describe('RootStore', function() {  
    var subject, revert;
```

```
➔ var moriMock = {  
    getIn: function() {}  
};
```

```
➔ var atomMock = {  
    assocIn: function() {}  
}
```

```
➔ var dispatcherMock = {  
    listen: function(message, cb) {  
        return cb;  
    }  
}
```

```
...
```

Creamos mocks para todas las dependencias, sólo con los métodos necesarios

Dependencias ocultas

```
before(function() {  
  subject = rewire('../stores/root');  
  ➔ revert = subject.__set__({  
    _: moriMock,  
    atom: atomMock,  
    Dispatcher: dispatcherMock  
  });  
});
```

Injectamos mocks con
rewire **__set__**

```
after(function() {  
  ➔ revert();  
});
```

Limpiamos llamando a
revert

Dependencias ocultas

...

```
it('Should return the current page in the atom', function() {  
    var getInStub = sinon.stub(moriMock, 'getIn', function() {  
        return 'alerts';  
    });  
    subject.getPage().should.equal('alerts');  
    getInStub.restore();
```

```
});
```

```
it('Should store the new page in the atom', function() {  
    var assocInSpy = sinon.spy(atomMock, 'assocIn');  
    subject.setPage('foo');  
    assocInSpy.calledOnce.should.be.true;  
    assocInSpy.getCall(0).args[1].should.equal('foo');  
    assocInSpy.restore();
```

```
});
```

```
});
```


Dependencias ocultas

...

```
it('Should return the current page in the atom', function() {  
➔ var getInStub = sinon.stub(moriMock, 'getIn', function() {  
    return 'alerts';  
  });  
  subject.getPage().should.equal('alerts');  
➔ getInStub.restore();  
});
```

Usamos **stub** para determinar el comportamiento

```
it('Should store the new page in the atom', function() {  
➔ var assocInSpy = sinon.spy(atomMock, 'assocIn');  
  subject.setPage('foo');  
  assocInSpy.calledOnce.should.be.true;  
  assocInSpy.getCall(0).args[1].should.equal('foo');  
➔ assocInSpy.restore();  
});  
});
```

O un **spy** para comprobar que las llamadas son las esperadas

Infraestructura de tests

- Debemos hacer que el compilador React cubra nuestros tests (mocha —compiler) cuando testeemos componentes
- Usar **mocha.opts** para guardar opciones habituales en un archivo
- Podemos extraer métodos habituales y usar **helpers**
- Podemos extraer mocks comunes y reutilizarlos entre tests.
- Ver configuración en soluciones/tema5/

Tests de Componentes

- ¿Qué probamos en los componentes?
- La **salida**, si depende el estado global o los props del componente
- Que **las interacciones** llaman a los métodos correctos o publican mediante el Dispatcher los mensajes correctos

Tests de Componentes

- Para React, necesitamos un DOM real donde montar nuestros componentes.
- Opción 1: utilizar un browser real para las pruebas, o un browser “programable” como PhantomJS
- Opción 2: emular un entorno browser para que React funcione correctamente, pero podemos ejecutar los tests mucho más rápido

Tests de Componentes - jsdom

- <https://github.com/tmpvar/jsdom>
- “A JavaScript implementation of the WHATWG DOM and HTML standards, for use with node.js”
- Muy fácil de integrar con **mocha**
- Haremos creer a React que se está ejecutando en un navegador real

Tests de Componentes - jsdom

```
//setup jsdom
```

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

```
var doc = jsdom.jsdom('<html><head></head><body></body></html>');
```

```
var win = doc.defaultView;
```

```
global.document = doc;
```

```
global.window = win;
```

Tests de Componentes - jsdom

```
//setup jsdom  
var jsdom = require('jsdom')
```

Creamos un documento HTML con la mínima estructura posible

```
➡ var doc = jsdom.jsdom('<html><head></head><body></body></html>');  
➡ var win = doc.defaultView;
```

```
➡ global.document = doc;  
➡ global.window = win;
```

Propagamos como variables globales **document** y **window**, exactamente igual que hace un browser real

Tests de Componentes - jsdom

- Si guardamos la configuración de **jsdom** y lo incluimos al comienzo de todos nuestros tests, podremos “montar” componentes de React exactamente igual que en el browser

Tests de Componentes

- Además de poder hacer **render** de un componente, necesitaremos:
- Extraer información del componente montado (props, si existe o no, qué componentes hijos tiene, etc.)
- Simular eventos de React (*click*, *change*, etc) para poder testear las interacciones
- React proporciona estas utilidades bajo **React.addons.TestUtils**

React.TestUtils - render




➔ **renderIntoDocument**(ReactElement)

Monta el componente en un nodo no adjuntado al DOM (necesita browser o JSDOM) y devuelve como resultado el componente React.

➔ **React.render**(ReactElement, *domNode*)

Así montamos React en el navegador, pasando como argumento el punto de montaje en el DOM

React.TestUtils

```
 var jsdom = require('jsdom');  
 var React = require('react/addons');  
var TestUtils = React.addons.TestUtils;  
var should = require('should');  
  
//setup jsdom  
var doc = jsdom.jsdom('<html><head></head><body></body></html>');  
var win = doc.defaultView;  
global.document = doc;  
global.window = win;  
  
//our simple test component  
var MyComponent = React.createClass({  
  render: function() {  
    return <div>Hello world!</div>;  
  }  
})  
  
describe('React Render test', function() {  
  it('Should render', function() {  
     var component = TestUtils.renderIntoDocument(<MyComponent />);  
    component.should.be.an.Object;  
  });  
});
```

React.TestUtils - buscar por clase CSS

- ➔ **scryRenderedDOMComponentsWithClass**
(ReactComponent, cssClassName)

Devuelve todas las **instancias** (Array) montadas en el DOM con la clase CSS indicada

- ➔ **findRenderedDOMComponentWithClass**
(ReactComponent, cssClassName)

Como la anterior, pero espera sólo un resultado y lo devuelve. Lanza excepción si hay más de uno

React.TestUtils - buscar por clase CSS

```
//...
var MyComponent = React.createClass({
  render: function() {
    return (
      <div>
        <h1 className='title'>Hello world</h1>
        <p className='summary'>This is good</p>
        <p className='summary'>Very good</p>
      </div>
    );
  }
});

describe('React find-scry with CSS class', function() {
  var component;
  before(function() {
    component = TestUtils.renderIntoDocument(<MyComponent />);
  });

  it('Should render one title heading', function() {
    ➔ var title = TestUtils.findRenderedDOMComponentWithClass(component, 'title');
    title.should.be.an.Object;
    title.props.children.should.equal('Hello world');
  });

  it('Should render two summary paragraphs', function() {
    ➔ var paragraphs = TestUtils.scryRenderedDOMComponentsWithClass(component, 'summary');
    paragraphs.should.have.length(2);
    paragraphs[0].props.children.should.equal('This is good');
    paragraphs[1].props.children.should.equal('Very good');
  });
});
```

findXXX... -> Una instancia
scryXXXX... -> Un Array

React.TestUtils - buscar por tag

→ **scryRenderedDOMComponentsWithTag**

(tree, tagName)

Devuelve un Array con todos los elementos cuya etiqueta sea *tagName*

→ **findRenderedDOMComponentWithTag**

(tree, tagName)

Devuelve el **único** componente cuya etiqueta sea *tagName*

React.TestUtils - buscar por tag

```
var MyComponent = React.createClass({
  render: function() {
    return (
      <div>
        <h1 className='title'>Hello world</h1>
        <p className='summary'>This is good</p>
        <p className='summary'>Very good</p>
      </div>
    );
  }
});

describe('React find-scry with tag name', function() {
  var component;
  before(function() {
    component = TestUtils.renderIntoDocument(<MyComponent />);
  });

  it('Should render one title heading', function() {
    ➡ var title = TestUtils.findRenderedDOMComponentWithTag(component, 'h1');
    title.should.be.an.Object;
    title.props.children.should.equal('Hello world');
  });

  it('Should render two summary paragraphs', function() {
    ➡ var paragraphs = TestUtils.scryRenderedDOMComponentsWithTag(component, 'p');
    paragraphs.should.have.length(2);
    paragraphs[0].props.children.should.equal('This is good');
    paragraphs[1].props.children.should.equal('Very good');
  });
});
```

React.TestUtils - Buscar por tipo

➔ **scryRenderedComponentsWithType**

(tree, *componentClass*)

Devuelve un Array con todos los componentes React del tipo *componentClass*

➔ **findRenderedComponentWithType**

(tree, *componentClass*)

Devuelve el **único** componente del tipo *componentClass*

React.TestUtils - Buscar por tipo

```
var Title = React.createClass({
  render: function() {
    return <h1 className='title'>{ this.props.message }</h1>;
  }
});

var Summary = React.createClass({
  render: function() {
    return <p className='summary'>{ this.props.text }</p>;
  }
});

var MyComponent = React.createClass({
  render: function() {
    return (
      <div>
        <Title message='Hello Testing World' />
        <Summary text='This is good' />
        <Summary text='Very good' />
      </div>
    );
  }
});
```

React.TestUtils - Buscar por tipo

```
describe('React find-scry with component type', function() {  
  var component;  
  before(function() {  
    component = TestUtils.renderIntoDocument(<MyComponent />);  
  });  
  
  it('Should render one Title component', function() {  
    ➔ var title = TestUtils.findRenderedComponentWithType(component, Title);  
    title.should.be.an.Object;  
    title.props.message.should.equal('Hello Testing World');  
  });  
  
  it('Should render two Summary components', function() {  
    ➔ var paragraphs = TestUtils.scryRenderedComponentsWithType(component, Summary);  
    paragraphs.should.have.length(2);  
    paragraphs[0].props.text.should.equal('This is good');  
    paragraphs[1].props.text.should.equal('Very good');  
  });  
});
```

React TestUtils - simular eventos

- ➔ **Simulate**.*eventName*(DOMNode, [EventData])
Nos permite simular el evento *eventName* sobre el **nodo** (no componente).
- ➔ **React.findNode**(mountedComponent)
Dada una referencia a un componente (React o HTML) montado, nos devuelve el nodo HTML
- ➔ Podemos pasar datos adicionales ya que el evento es ficticio, por ejemplo enviar **keyCode** para simular que se ha pulsado una tecla concreta

React TestUtils - simular eventos

```
var MyComponent = React.createClass({
  getInitialState: function() {
    return {
      clicks: 0
    };
  },
  onClick: function(e) {
    this.setState({ clicks: this.state.clicks+1 });
  },
  render: function() {
    return (
      <div>
        <p>You have clicked { this.state.clicks } times</p>
        ➡ <button onClick={this.onClick}>Click!</button>
      </div>
    );
  }
});
```

React TestUtils - simular eventos

```
describe('React Simulate', function() {
  var component;
  before(function() {
    component = TestUtils.renderIntoDocument(<MyComponent />);
  });

  it('Should render 0 clicks on first mount', function() {
    //we can assert on the component state
    component.state.clicks.should.equal(0);
    //and also on the rendered output
    var paragraph = TestUtils.findRenderedDOMComponentWithTag(component, 'p');
    // 'You have clicked', 0, ' times'
    paragraph.props.children.should.have.length(3);
    //Second is our initial value
    paragraph.props.children[1].should.equal(0);

  });

  it('Should increment click count when button is clicked', function() {
    ➡ var button = TestUtils.findRenderedDOMComponentWithTag(component, 'button');
    //we need the DOM node, not the component
    ➡ var buttonNode = React.findDOMNode(button);
    //simulate click
    ➡ TestUtils.Simulate.click(buttonNode);
    //assert on new state on the parent component
    ➡ component.state.clicks.should.equal(1);
  });
});
```

React TestUtils - resumen

- ➔ Tenemos funciones para buscar por **tipo**, **etiqueta** o **clase CSS**.
- ➔ Las funciones **scry...** devuelven Arrays
- ➔ Las funciones **find...** devuelven 1 sólo elemento, y lanzan una excepción si encuentran más de uno
- ➔ Podemos simular un evento sobre un **nodo** con `TestUtils.Simulate.event`

React TestUtils - resumen

- ➔ Si queremos buscar un elemento anidado... podemos llamar a **findX/scryX** recursivamente.
- ➔ La primera vez obtenemos un elemento intermedio, y la segunda usamos este elemento intermedio como árbol en el que buscar “más profundo”
- ➔ Podemos acceder a las props de un componente
- ➔ Podemos acceder al estado interno de un componente

React TestUtils - resumen

- ➔ En estos ejemplos, la definición del componente estaba en el mismo archivo que el test, y no tenía dependencias.
- ➔ En tests reales, tendremos que utilizar **rewire** para mockear dependencias:
 - ➔ Componentes hijo
 - ➔ Stores
 - ➔ Dispatcher

React TestUtils - mocking

```
➡ var React = require('react'),  
    atom = require('../lib/atom_state');  
  
➡ var ShoppingCart = require('../shopping_cart/');  
  
var RootComponent = React.createClass({  
  componentDidMount: function() {  
    atom.addChangeListener(this._onAtomChange);  
  },  
  _onAtomChange: function() {  
    this.forceUpdate();  
  },  
  render: function() {  
    var state = atom.getState();  
    return (<ShoppingCart state={state} />);  
  }  
});  
  
module.exports = RootComponent;
```

React TestUtils - mocking

```
var React = require('react/addons'),  
    TestUtils = React.addons.TestUtils,  
    rewire = require('rewire'),  
    should = require('should');
```

➔

```
var ShoppingCartMock = React.createClass({  
  render: function() {  
    return (<div className='shopping-cart'></div>);  
  }  
});
```

➔

```
var atomMock = {  
  addChangeListener: function() {},  
  getState: function() {  
    return { foo: 'bar' };  
  }  
}
```

```
describe('Root', function() {  
  var RootComponent = rewire('../components/root');
```

➔

```
RootComponent.__set__('atom', atomMock);  
RootComponent.__set__('ShoppingCart', ShoppingCartMock);
```

```
var dom;
```

```
beforeEach(function() {  
  dom = TestUtils.renderIntoDocument(React.createElement(RootComponent));  
});  
...
```

React TestUtils - mocking

```
beforeEach(function() {  
  dom = TestUtils.renderIntoDocument(React.createElement(RootComponent));  
});  
  
it('Should render the ShoppingCart component', function() {  
➡ var shoppingCart = TestUtils.scryRenderedComponentsWithType(dom, ShoppingCartMock);  
  shoppingCart.length.should.equal(1);  
});  
  
it('Should pass the current atom state to ShoppingCart component', function() {  
➡ var shoppingCart = TestUtils.findRenderedComponentWithType(dom, ShoppingCartMock);  
  shoppingCart.props.should.have.property('state').deepEqual({foo: 'bar'});  
});  
  
});
```

Ejercicio - Carrito de la compra

- Escribe un test unitario para el componente **Cart** de la aplicación del carrito de la compra
- Tendremos que crear mocks para todas las dependencias que veamos en el componente, excepto **React** y **mori**.
- Para cada uno de estos mocks, tendremos que mockear sólo los métodos necesarios

Ejercicio - Carrito de la compra

- La salida tiene un componente **CartItem** por cada producto en el carro
- La salida contiene el precio total del carrito (pista: **props.children**)
- La salida contiene un mensaje de “carrito vacío” si no hay productos en el carro
- Se publica el mensaje correcto en el Dispatcher al hacer click en “Seguir comprando”
- Se publica el mensaje correcto en el Dispatcher al hacer click en “Finalizar compra”

Ejercicio - Carrito de la compra

- La carpeta del tema5 ya incluye todos los Stores y componentes de la aplicación ShoppingCart, listos para ser testados.
- Poned vuestro test dentro de la carpeta **test**
- Ejecutar tests con watch (incluye los ejemplos):
 - ➔ `cd /ejercicios/tema5/src`
 - ➔ `mocha`

Tests de Stores

- ¿Qué testeamos en los Stores?
- Las consultas: que devuelven los valores esperados del estado global
- Los comandos: que son llamados por el Dispatcher, y que modifican el átomo correctamente

Tests de Stores

- Al igual que con los componentes, tendremos que mockear dependencias y utilizar **rewire** para inyectarlas durante las pruebas.
- Sin embargo, **no sustituiremos** el **átomo**, ni la librería **mori**, ni el **Dispatcher** normalmente.
- Perdemos un test unitario “puro” pero ganamos un test de integración con el átomo y el Dispatcher.

Tests de Stores - consultas

- Con nuestra arquitectura, testear consultas es sumamente sencillo
- Establecemos el átomo a un valor concreto
- Y llamar los métodos del Store pasándole el valor del átomo.
- Hacemos aserciones sobre el resultado de las consultas

Tests de Stores - consultas

```
describe('Root Store', function() {  
  var subject = require('../stores/root');  
  var fakeState = {  
    page: 'foo'  
  };  
  
  before(function() {  
    atom.silentSwap(_.toClj(fakeState));  
  });  
  
  it('Should return the current page', function() {  
    var page = subject.getPage(atom.getState());  
    page.should.equal(fakeState.page);  
  });  
});
```

Tests de Stores - consultas

```
describe('Root Store', function() {  
  var subject = require('../stores/root');  
➔ var fakeState = {  
    page: 'foo'  
  };  
  
  before(function() {  
➔    atom.silentSwap(_.toClj(fakeState));  
  });  
  
  it('Should return the current page', function() {  
➔    var page = subject.getPage(atom.getState());  
    page.should.equal(fakeState.page);  
  });  
});
```

Tests de Stores - comandos

- Testear comandos es muy similar
- Establecemos el átomo a un valor concreto, si es necesario
- Llamamos a `Dispatcher.emit` como si se tratase de un componente u otro Store
- Realizamos aserciones sobre el nuevo estado del átomo resultante usando `mori`

Tests de Stores - comandos

```
it('Should store the new page when receiving SET:PAGE', function() {  
➡ Dispatcher.emit('SET:PAGE', 'changed');  
  atom.getIn(['page']).should.equal('changed');  
});  
  
it('Should store the thank-you page when receiving ORDER:COMPLETE', function() {  
  Dispatcher.emit('ORDER:COMPLETE');  
➡ atom.getIn(['page']).should.equal('thank-you');  
});
```

Ejercicio - Testear Cart Store

- Vamos a testear el Store que gestiona el carrito de la compra
- El Store está implementado en `/src/stores/cart.js`
- Deberemos cubrir las consultas **getCartProducts** y **getCartTotal**
- Deberemos cubrir los comandos **addProduct**, **removeProduct**, **changeQuantity** y **emptyCart**

Ejercicio - Testear Cart Store

- Trucos al testear Stores: donde veamos en la ejecución de un método una bifurcación condicional...
- Es señal casi segura de que deberemos hacer **dos** pruebas para testar ese método, con dos estados iniciales diferentes.
- Si usamos **rewire**, podemos extraer los atajos de rutas en el átomo que usamos en los Store y reutilizarlo en el test (“p”).

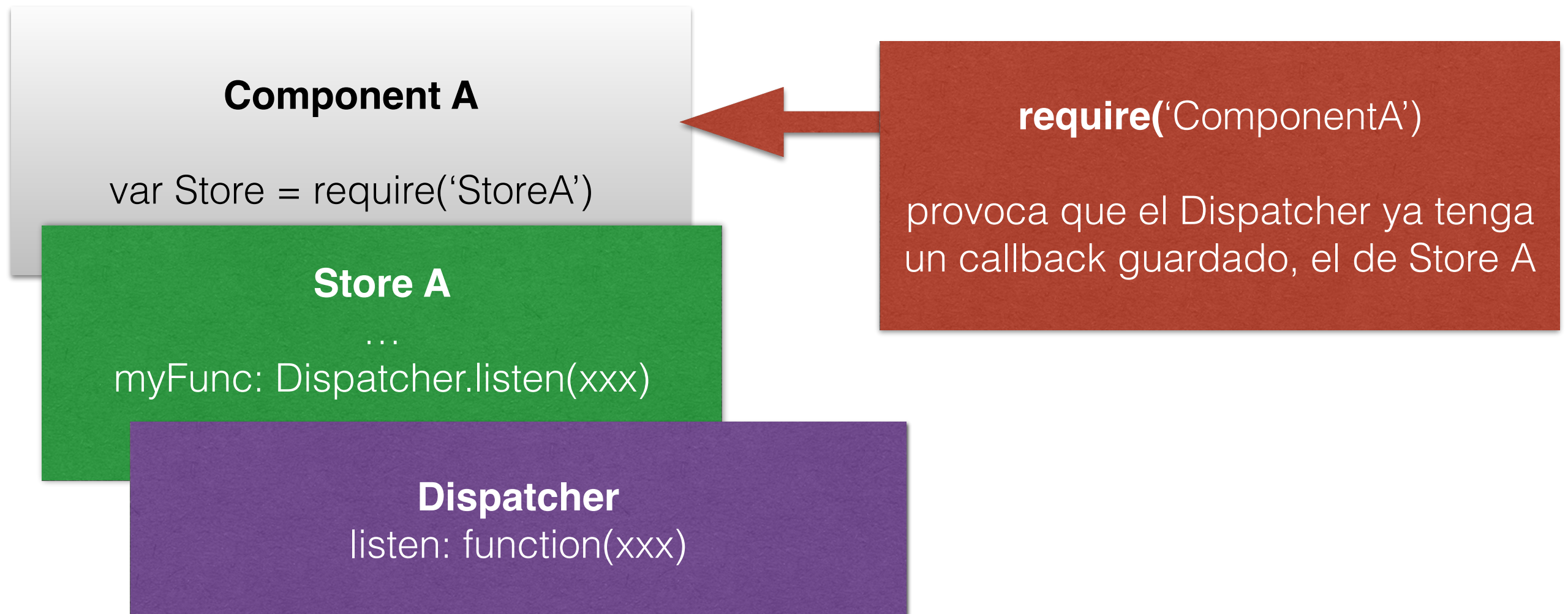
Ejecutar suites múltiples

- 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
- No olvidar `restore()` de **sinon**
- No olvidar `revert()` de **rewire**

Ejecutar suites múltiples

- Al testar Stores sin mockear Dispatcher, podemos tener problemas con que se queden *listeners* colgando.
- ¿Por qué?
- Por que casi todos los Stores usan **Dispatcher.listen** para definir sus comandos

Ejecutar suites múltiples



Ejecutar suites múltiples

- Lo que significa que en cuando ese módulo sea cargado, registrarán los *listeners*
- Si ejecutamos un test de un componente que depende un Store, ese Store será cargado y sus *listeners* conectados, lo cual nos puede provocar problemas en los tests de Stores que se ejecuten después.
- Podemos solucionarlo limpiando todos los listeners al comienzo de un test con:

➔ **Dispatcher.removeAllListeners();**

Ejecutar suites múltiples

```
describe('Catalog Store', function() {
  var serviceMock = {
    loadCatalogDataPromise: function() {}
  }

  var fakeProducts = [
    { id: 1, name: 'foo1', price: 10 },
    { id: 2, name: 'foo2', price: 20 },
    { id: 3, name: 'foo3', price: 30 }
  ];

  var subject, p, revert;

  before(function() {
    ➡ Dispatcher.removeAllListeners();
    subject = rewire('../stores/catalog');
    revert = subject.__set__('shoppingCartService', serviceMock);
    p = subject.__get__('p');
  });

  after(function() {
    revert();
  });

  beforeEach(function() {
    //empty the atom
    atom.swap(_.toClj({}));
  });
```

Ejecutar suites múltiples

```
describe('Catalog Store', function() {  
  var serviceMock = {  
    loadCatalogDataPromise: function() {}  
  }  
  
  var fakeProducts = [  
    { id: 1, name: 'foo1', price: 10 },  
    { id: 2, name: 'foo2', price: 20 },  
    { id: 3, name: 'foo3', price: 30 }  
  ];  
  
  var subject, p, revert;  
  
  before(function() {  
    Dispatcher.removeAllListeners();  
    ➔ subject = rewire('../stores/catalog');  
    revert = subject.__set__('shoppingCartService', serviceMock);  
    p = subject.__get__('p');  
  });  
  
  after(function() {  
    revert();  
  });  
  
  beforeEach(function() {  
    //empty the atom  
    atom.swap(_.toClj({}));  
  });  
});
```

En el momento de cargar vía **rewire** el Store, el Dispatcher estará “limpio”.

Suite completa de Shopping Cart

- Tenéis los tests de todos los componentes y todos los Stores de la aplicación en la carpeta **soluciones**
- Podéis ejecutar todos los tests con:
 - ➔ `cd soluciones/tema5/src`
 - ➔ `mocha`

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
- Por ejemplo, **istanbul**
- <https://gotwarlost.github.io/istanbul/>

Cobertura de código

- Istanbul tiene una función para ejecutar un comando de node con la cobertura “activada” (por ejemplo, **mocha**)
- Instalación global:
`npm install -g istanbul`
- Instalación local:
`npm install —save-dev istanbul`

Cobertura de código

- Si ejecutamos nuestros tests con:
`cd ./src/test && mocha`
- Deberemos ejecutar istanbul cover con:
➔ `istanbul cover _mocha`
- El resultado estará disponible en la carpeta **coverage** en la ruta actual. Dentro estará el informe en **lcov-report/index.html**

Cobertura de código

Code coverage report for **components/root.js**

Statements: **87.5%** (7 / 8) Branches: **100%** (0 / 0) Functions: **66.67%** (2 / 3) Lines: **87.5%** (7 / 8) Ignored: none

[All files](#) » [components/](#) » root.js

```
1  'use strict';
2
3  1 var React = require('react'),
4    atom = require('../lib/atom_state');
5
6  1 var ShoppingCart = require('./shopping_cart/');
7
8  1 var RootComponent = React.createClass({
9    displayName: 'RootComponent',
10
11    componentDidMount: function componentDidMount() {
12      2    atom.addChangeListener(this._onAtomChange);
13    },
14    _onAtomChange: function _onAtomChange() {
15      this.forceUpdate();
16    },
17    render: function render() {
18      2    var state = atom.getState();
19      2    return React.createElement(ShoppingCart, { state: state });
20    }
21  });
22
23  1 module.exports = RootComponent;
```