

Contenido

- 4. La lógica de negocio (Stores)
 - Qué es un Store
 - Consultas: extraer información del átomo
 - Comandos: modificar el átomo
 - Dispatcher: recibir comandos
 - Acceso a datos externos
 - Integración en la arquitectura

Qué es un Store

- Un almacén (¡no una tienda!)
- Una **fachada** sobre el átomo que se encarga de un **dominio** concreto

Qué es un Store

- Puesto que el átomo contiene **todo el estado...**
- Los Stores contienen la lógica de negocio para un dominio concreto
- Los distintos subárboles del átomo serán gestionados por distintos Stores
- De nuevo, **SRP** y Separation of Concerns

Qué es un Store

- Un Store es simplemente un módulo Javascript que ofrece funciones para manipular una parte del átomo
- Leer o comprobar parte del átomo
- Modificar parte del átomo cuando se recibe un comando del Dispatcher

Qué es un Store

- **Conceptualmente**, podemos pensar que un Store guarda el estado (aunque en realidad solo gestiona una parte del total)
- Lo que los convierte en una mezcla de Model de MVC
- ...y Repositorio en una arquitectura por capas
- Sólo que **todo el estado** está en el átomo, y por tanto los Stores no guardan nada

Un Store que maneja un único dato: en qué página se encuentra el usuario

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

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

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

  setPage: function(newPage) {
    atom.assocIn(p.page, newPage);
  }
};

Dispatcher.listen("SET:PAGE", RootStore.setPage);

module.exports = RootStore;
```

Un Store que maneja un único dato: en qué página se encuentra el usuario

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

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

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

```
Dispatcher.listen("SET:PAGE", RootStore.setPage);
```

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



Operaciones

Un Store que maneja un único dato: en qué página se encuentra el usuario

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

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

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

  setPage: function(newPage) {
    atom.assocIn(p.page, newPage);
  }
};

Dispatcher.listen("SET:PAGE", RootStore.setPage);

module.exports = RootStore;
```

Atender
mensaje del
Dispatcher



CQS - Command/Query separation

- Cada método en un módulo debe ser:
 - Un **comando** que realiza una acción
 - O una **consulta** que devuelve datos
 - Pero **no ambos!!!**

CQS - Command/Query separation

- Las **consultas no modifican** el estado
- Los **comandos no devuelven** datos

CQS - Command/Query separation

CartStore

Consultas

- getCartProducts()

Comandos

- addProduct()
- removeProduct()
- changeProductQuantity()

Stores - consultas

- Las consultas reciben un estado global como parámetro y casi siempre serán consultas de mori (**getIn** o **get**)
- Normalmente acceden a una clave anidada en ese átomo
- Significa “dame X en **esta** versión que tengo del átomo”

Stores - consultas

```
var _ = require('mori');  
  
//rutas en el átomo (mori)  
var p = {  
  items: ['data', 'cart']  
};  
  
var CartStore = {  
  //Consultas  
  getCartProducts: function(state) {  
    return _.getIn(state, p.items);  
  },  
  ...  
}
```

Stores - consultas

```
var _ = require('mori');
```

```
//rutas en el átomo (mori)  
var p = {  
  items: ['data', 'cart']  
};
```

Atajo: guardamos las rutas
anidadas que utiliza este
Store (DRY)

```
var CartStore = {  
  //Consultas  
  getCartProducts: function(state) {  
    return _.getIn(state, p.items);  
  },  
  ...  
}
```

Stores - consultas

```
var _ = require('mori');  
  
//rutas en el átomo (mori)  
var p = {  
  items: ['data', 'cart']  
};
```

```
var CartStore = {  
  //Consultas  
  getCartProducts: function(state) {  
    return _.getIn(state, p.items);  
  },  
  ...
```

En la consulta, usamos la ruta para acceder al estado (state) que recibimos como parámetro

Stores - comandos

- Los comandos reciben los datos necesarios para su ejecución
- Realizan una operación de modificación en el estado global (**updateIn**, **assocIn**)

Stores - comandos

```
var p = {  
  items: ['data', 'cart']  
};
```

```
//funciones auxiliares
```

```
function productById(id, item) {  
  return _.get(item, 'id') === id;  
}
```

```
var CartStore = {
```

```
//Comandos
```

```
removeProduct: function (product) {  
  var id = _.get(product, 'id');  
  atom.updateIn(p.items, function (items) {  
    return _.remove(_.partial(productById, id), items);  
  });  
},  
...
```

Stores - comandos

```
var p = {  
  items: ['data', 'cart']  
};
```

```
//funciones auxiliares
```

```
function productById(id, item) {  
  return _.get(item, 'id') === id;  
}
```

```
var CartStore = {
```

```
//Comandos
```

```
  removeProduct: function (product) {  
    var id = _.get(product, 'id');  
    atom.updateIn(p.items, function (items) {  
      return _.remove(_.partial(productById, id), items);  
    });  
  },
```

```
  ...
```

En el comando, actualizamos el átomo global usando también la ruta anidada

Stores - estado inicial

Vemos que los Stores leen y escriben en el átomo, pero ¿de dónde sacamos los valores iniciales?

Usamos un archivo externo, **initial_state.js**, que cargamos como valor inicial del átomo usando *atom.swap()*

En este archivo incluimos los valores por defecto para nuestro átomo

Lo podéis ver en acción en el siguiente ejercicio (código incluido en el repositorio)

Stores - estado inicial

Este **initial_state.js** se compila como parte del empaquetado, por lo que si lo cambiamos, tenemos que recompilar.

No nos vale como “configuración” externa.

Pero sí para definir valores por defecto de la aplicación: un vector vacío, un true/false, etc.

Stores - Ejercicio

Vamos a crear los Stores necesarios para nuestra aplicación del carro de la compra: **CatalogStore**, **CartStore**, **OrderStore** y **RootStore**

Como en el tema anterior, podéis arrancar el servidor local en /ejercicios/tema4

> node index.js

Tenemos el esqueleto de los Stores en src/stores

Stores - Ejercicio

- **CatalogStore** - responsable de los productos visibles en el catálogo
- Consultas:
 - ➔ **getProducts**(state) - devuelve todos los productos del catálogo

Stores - Ejercicio

- **CartStore** - responsable de los productos que el usuario tiene en su carrito
- Consultas:
 - ➔ **getProducts**(state)
- Comandos:
 - ➔ **addProduct**(product)
 - ➔ **changeQuantity**(product, *delta*)
 - ➔ **removeProduct**(product),
 - ➔ **emptyCart**()

Stores - Ejercicio

- **OrderStore** - responsable de guardar el pedido (en este ejemplo, los datos personales introducidos en el usuario)
- Consultas:
 - ➔ **getFormErrors**(state) - devuelve un hashMap con los errores de cada campo
 - ➔ **getOrderDetails**(state) - devuelve los datos personales introducidos por el usuario durante el *checkout* una vez validados
- Comandos:
 - ➔ **validateDetails**(details) - implementa la lógica de validación, recibe los datos personales en un hashMap

Stores - Ejercicio

- **RootStore** - responsable de almacenar la página actual en la que se encuentra el usuario
- Consultas:
 - ➔ **getPage**(state) - devuelve la página actual
- Comandos:
 - ➔ **setPage**(page) - almacena la página actual en el átomo

Dispatcher - qué es

- El Dispatcher es simplemente un mecanismo Pub/Sub o, conceptualmente, un bus de mensajes
- Permite que otros componentes se **suscriban o escuchen** a determinados mensajes
- Permite que otros componentes **publiquen o despachen** mensajes a través de él
- Para cada evento **X**, tendrá que llamar a todos los suscriptores interesados con los mismos argumentos recibidos del emisor

Dispatcher - ventajas

- Utilizar el Dispatcher permite desacoplar la interfaz de usuario de la lógica de negocio
- Un componente de React que quiera “hacer algo” sólo tiene que enviar su petición/comando a través del Dispatcher
- Un Store responsable de un dominio determinado **no necesita saber quién quiere qué**, simplemente atiende el mensaje y actualiza el átomo
- Un mismo mensaje puede ser recibido por varios Stores

Dispatcher y Stores

- Un Store se **suscribe** a los mensajes que atiende
- A veces puede **emitir** mensajes cuando necesita que ocurran cosas fuera de su dominio

Dispatcher y Stores

```
Dispatcher.listen("SET:PAGE", RootStore.setPage);
```

```
//En otro lugar...
```

```
Dispatcher.emit("SET:PAGE", "alerts");
```

Dispatcher y Stores

➔ `Dispatcher.listen("SET:PAGE", RootStore.setPage);`

//En otro lugar...

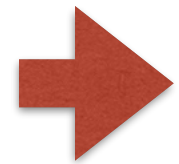
`Dispatcher.emit("SET:PAGE", "alerts");`

SUSCRIPCION - el método **listen** acepta una cadena de un mensaje y una función de callback

Dispatcher y Stores

```
Dispatcher.listen("SET:PAGE", RootStore.setPage);
```

```
//En otro lugar...
```



```
Dispatcher.emit("SET:PAGE", "alerts");
```

PUBLICACION - el método **emit** acepta una cadena que identifica el mensaje, y cualquier otro argumento con datos necesarios, específicos para ese mensaje

Dispatcher y Stores

- El método **listen** del Dispatcher devuelve la misma función que pasamos como callback
- Así que en los Stores, podemos definir los comandos y suscribir esa función a los eventos con una única llamada

Dispatcher y Stores

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

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

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

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

module.exports = RootStore;
```

Dispatcher y Stores

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

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

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



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

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

Dispatcher.listen devuelve la función, que guardamos en el Store como **setPage**

Dispatcher y Stores

```
var _ = require('mori'),  
    atom = require('../lib/atom_state'),  
    Dispatcher = require('../lib/dispatcher');  
  
var p = {  
  page: ['data', 'page']  
};  
  
var RootStore = {  
  getPage: function(state) {  
    return _.getIn(state, p.page);  
  },  
  
  setPage: Dispatcher.listen("SET:PAGE", function(newPage) {  
    atom.assocIn(p.page, newPage);  
  })  
};  
  
module.exports = RootStore;
```

Ojo con este paréntesis, estamos **invocando** Dispatcher.listen!!!

Dispatcher - publicar mensajes

- El método **emit** del Dispatcher recibe un identificador de mensaje y los argumentos necesarios
- Podemos emitir desde un Store para que otro (no sabemos quién) atienda un comando
- O, lo más habitual, llamaremos a **emit** desde los componentes de React para notificar interacciones

Dispatcher - publicar desde React

```
var CartItem = React.createClass({
  propTypes: {
    product: React.PropTypes.object.isRequired
  },
  updateQuantity: function(n, e) {
    e.preventDefault();
    Dispatcher.emit("CART:CHANGE:QTY", this.props.product, n);
  },
  removeItem: function(e) {
    e.preventDefault();
    Dispatcher.emit("CART:REMOVE", this.props.product);
  },
  render: function() {
    //...
```

Dispatcher - publicar desde React

```
var CartItem = React.createClass({
  propTypes: {
    product: React.PropTypes.object.isRequired
  },
  updateQuantity: function(n, e) {
    e.preventDefault();
    ➡ Dispatcher.emit("CART:CHANGE:QTY", this.props.product, n);
  },
  removeItem: function(e) {
    e.preventDefault();
    Dispatcher.emit("CART:REMOVE", this.props.product);
  },
  render: function() {
    //...
```

En lugar de llamar a un callback del padre, emitimos directamente el mensaje

Dispatcher - publicar desde Stores

```
validateDetails: Dispatcher.listen("ORDER:SAVE", function(detailsMap) {  
  var errors = _.hashMap();  
  if(_.get(detailsMap, 'name').trim() === '') {  
    errors = _.assoc(errors, 'name', 'El nombre es obligatorio');  
  }  
  //...  
  
  if(_.count(errors) > 0) {  
    atom.assocIn(p.formErrors, errors);  
  }  
  else {  
    //remove any errors  
    atom.silentAssocIn(p.formErrors, errors);  
    //save the order - the user details in fact  
    atom.assocIn(p.order, detailsMap);  
    //this order is complete  
    Dispatcher.emit("ORDER:COMPLETE");  
  }  
}
```

Dispatcher - publicar desde Stores

```
validateDetails: Dispatcher.listen("ORDER:SAVE", function(detailsMap) {  
  var errors = _.hashMap();  
  if(_.get(detailsMap, 'name').trim() === '') {  
    errors = _.assoc(errors, 'name', 'El nombre es obligatorio');  
  }  
  //...  
  
  if(_.count(errors) > 0) {  
    atom.assocIn(p.formErrors, errors);  
  }  
  else {  
    //remove any errors  
    atom.silentAssocIn(p.formErrors, errors);  
    //save the order - the user details in fact  
    atom.assocIn(p.order, detailsMap);  
    //this order is complete  
    Dispatcher.emit("ORDER:COMPLETE");  
  }  
}
```



En este Store validamos los datos del formulario y, si son correctos, tenemos que vaciar el carrito y saltar a la pantalla de agradecimiento

Lo hacemos con un único mensaje que será atendido por los Stores correspondientes

Acceso a datos externos

- Queremos poder acceder a datos sin tener que recompilar la aplicación. Por ejemplo:
 - Archivos de configuración (JSON)
 - Bases de datos mediante una API
 - Información en tiempo real mediante WebSocket
 - ...

Acceso a datos externos

- Este acceso suele tener mucho de infraestructura y poco de lógica de negocio
 - Establecer conexiones
 - Ejecutar petición
 - Notificar éxito o fracaso
 - En el caso de Websockets, notificar datos recibidos
 - ...

Acceso a datos externos

- Para no diseminar este código por todos los Stores que lo necesiten, es mejor crear **servicios** específicos
- Podremos llamar a métodos de estos servicios desde los Stores o mediante comandos vía Dispatcher

Acceso a datos externos

Para operaciones asíncronas, tenemos dos alternativas:

1. Devolver promesas desde el servicio, de forma que el Store actualice el átomo cuando se resuelvan
2. Notificar el resultado mediante a través Dispatcher para que el Store correspondiente los utilice

Acceso a datos externos - Promesas

- Supongamos un servicio **shoppingCartService** que ofrece un método *loadCatalogDataPromise*.
- Este método devuelve una promesa que resuelve con los productos del catálogo.
- Será el **CatalogStore** quien atienda un comando vía Dispatcher para llamar a este método y actualizar el átomo con el resultado

Acceso a datos externos - Promesas

```
var CatalogStore = {  
  getProducts: function(state) {  
    return _.getIn(state, p.products);  
  },  
  
  loadProducts: Dispatcher.listen("CATALOG:LOAD", function() {  
    shoppingCartService.loadCatalogDataPromise()  
      .then(function(products) {  
        atom.assocIn(p.products, _.toClj(products));  
      })  
      .catch(function(err) {  
        alert("Error al cargar los productos", err);  
      });  
  }),  
  //...
```

Acceso a datos externos - Promesas

```
var CatalogStore = {  
  getProducts: function(state) {  
    return _.getIn(state, p.products);  
  },  
  
  loadProducts: Dispatcher.listen("CATALOG:LOAD", function() {  
    ➔ shoppingCartService.loadCatalogDataPromise()  
      .then(function(products) {  
        atom.assocIn(p.products, _.toClj(products));  
      })  
      .catch(function(err) {  
        alert("Error al cargar los productos", err);  
      });  
  }  
},  
//...
```

El propio **CatalogStore** atiende el comando
"CATALOG:LOAD"

Cuando la promesa sea resuelta, se guardan los
datos en el átomo

Acceso a datos externos - servicio con Dispatcher

```
var catalogUrl = '/data/catalog.json';

var ShoppingCartService = {

    //Dispatcher-based version
    loadCatalogData: Dispatcher.listen("CATALOG:SERVICE:LOAD", function() {
        var xhr = new XMLHttpRequest();

        xhr.open('GET', catalogUrl);
        xhr.onload = function() {
            if(xhr.status === 200) {
                Dispatcher.emit("CATALOG:LOAD:COMPLETE", JSON.parse(xhr.response));
            }
            else {
                Dispatcher.emit("CATALOG:LOAD:ERROR", xhr.status, xhr.statusText);
            }
        }

        xhr.send();
    })

}

module.exports = ShoppingCartService;
```


Acceso a datos externos - servicio con Dispatcher

```
var catalogUrl = '/data/catalog.json';
```

```
var ShoppingCartService = {
```

```
    //Dispatcher-based version
```



```
loadCatalogData: Dispatcher.listen("CATALOG:SERVICE:LOAD", function() {
    var xhr = new XMLHttpRequest();

    xhr.open('GET', catalogUrl);
    xhr.onload = function() {
        if(xhr.status === 200) {
            Dispatcher.emit("CATALOG:LOAD:COMPLETE", JSON.parse(xhr.response));
        }
        else {
            Dispatcher.emit("CATALOG:LOAD:ERROR", xhr.status, xhr.statusText);
        }
    }

    xhr.send();
})

}
```

```
module.exports = ShoppingCartService;
```

Este servicio atiende directamente
comandos del Dispatcher

Acceso a datos externos - servicio con Dispatcher

```
var catalogUrl = '/data/catalog.json';

var ShoppingCartService = {

    //Dispatcher-based version
    loadCatalogData: Dispatcher.listen("CATALOG:SERVICE:LOAD", function() {
        var xhr = new XMLHttpRequest();

        xhr.open('GET', catalogUrl);
        xhr.onload = function() {
            if(xhr.status === 200) {
 Dispatcher.emit("CATALOG:LOAD:COMPLETE", JSON.parse(xhr.response));
            }
            else {
                Dispatcher.emit("CATALOG:LOAD:ERROR", xhr.status, xhr.statusText);
            }
        }

        xhr.send();
    })
}

module.exports = ShoppingCartService;
```

En caso de éxito, publica otro mensaje con los datos cargados

Acceso a datos externos - servicio con Dispatcher

```
var catalogUrl = '/data/catalog.json';

var ShoppingCartService = {

    //Dispatcher-based version
    loadCatalogData: Dispatcher.listen("CATALOG:SERVICE:LOAD", function() {
        var xhr = new XMLHttpRequest();

        xhr.open('GET', catalogUrl);
        xhr.onload = function() {
            if(xhr.status === 200) {
                Dispatcher.emit("CATALOG:LOAD:COMPLETE", JSON.parse(xhr.response));
            }
            else {
 Dispatcher.emit("CATALOG:LOAD:ERROR", xhr.status, xhr.statusText);
            }
        }

        xhr.send();
    })

}

module.exports = ShoppingCartService;
```

En caso de error, publica un mensaje diferente para que “alguien” lo atienda

Acceso a datos externos

- Con promesas, un único mensaje nos sirve para todo el proceso, y es el Store quien se encarga de todo.
- Sin promesas, necesitamos varios mensajes: uno para activar el servicio, y otros con las respuestas para los Stores.
- En ambos casos, para lanzar la carga de datos externos utilizamos el Dispatcher.

Acceso a datos externos

```
var React = require('react'),
    _ = require('mori'),
    atom = require('./lib/atom_state'),
    Dispatcher = require('./lib/dispatcher'),
    initialState = require('./config/initial_state'),
    shoppingCartService = require('./services/shopping_cart');

var Root = require('./components/root');

window.onload = function() {
  //cargamos el átomo con un valor inicial
  atom.silentSwap(_.toClj(initialState));
  //montamos el componente en el DOM
  React.render(<Root />, document.body);
  //cargamos los datos externos del catálogo

  // Opcion 1 - el servicio usa el Dispatcher
  // este evento se atiende en el propio shoppingCartService
  // como acabamos de ver
  Dispatcher.emit("CATALOG:SERVICE:LOAD");

  // Opcion 2 - el propio store usará el servicio con promesas
  // este evento se atiende en CatalogStore
  Dispatcher.emit("CATALOG:LOAD");
}
```

Acceso a datos externos

```
var React = require('react'),
    _ = require('mori'),
    atom = require('./lib/atom_state'),
    Dispatcher = require('./lib/dispatcher'),
    initialState = require('./config/initial_state'),
    ➡ shoppingCartService = require('./services/shopping_cart');
```

```
var Root = new React.Component({
  render: function() {
    window.onload = function() {
      //cargamos el servicio
      atom.silent(function() {
        //montamos el servicio
        React.render(
          //cargamos el servicio
          // Opcion 1 - el propio store usará el servicio con promesas
          // este evento se atiende en CatalogStore
          // como acabamos de ver
          Dispatcher.emit("CATALOG:SERVICE:LOAD");

          // Opcion 2 - el propio store usará el servicio con promesas
          // este evento se atiende en CatalogStore
          Dispatcher.emit("CATALOG:LOAD");
        }
      )
    }
  }
});
```

¿Por qué incluimos el servicio si no lo vamos a llamar directamente?

Pista: Dispatcher...

Integración en la arquitectura

- La integración de los Stores en la arquitectura implica:
 1. User las consultas de los Stores en los componentes
 2. Emitir mensajes a través del Dispatcher en los componentes
 3. Suscribir los Stores a esos mensajes como ya hemos hecho con Dispatcher.**listen**

Integración en la arquitectura (1)


- Los **componentes** usarán las **consultas** ofrecidas por los Stores, pasándole su *snapshot* del átomo recibido vía props, para extraer los datos que necesitan

Integración en la arquitectura (1)

Recordemos que pasamos el valor actual del átomo al primer componente “visible” de nuestra aplicación desde el Root

```
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;
```

Integración en la arquitectura (1)

Y cablearemos esa referencia por nuestra jerarquía:

```
var ShoppingCart = React.createClass({
  propTypes: {
    state: React.PropTypes.object.isRequired
  },
  getPageComponent: function(page) {
    switch(page) {
      case 'catalog':
      return <Catalog state={this.props.state} />;
      ...
    }
  },

  render: function() {
    var currentPage = RootStore.getPage(this.props.state);

    return (
      <div className="shopping-cart">
        { this.getPageComponent(currentPage) }
      </div>
    );
  }
});
```

Integración en la arquitectura (1)

De forma que un componente consulte al Store sobre “su” átomo

```
render: function() {  
  ➔ var products = CatalogStore.getProducts(this.props.state);  
  
  return (  
    <div className="catalog">  
      <div className="catalog-header">  
        <h2>Productos</h2>  
      </div>  
      <div className="catalog-list">  
        { this.renderItems(products) }  
      </div>  
      <div className="footer"></div>  
    </div>  
  )  
}
```

Integración en la arquitectura (2)

- Los componentes deben emitir mensajes vía Dispatcher para notificar interacciones

Integración en la arquitectura (2)

```
var CartItem = React.createClass({
  propTypes: {
    product: React.PropTypes.object.isRequired
  },
  → updateQuantity: function(n, e){
    e.preventDefault();
    Dispatcher.emit("CART:CHANGE:QTY", this.props.product, n);
  },
  removeItem: function(e){
    e.preventDefault();
    Dispatcher.emit("CART:REMOVE", this.props.product);
  },
  render: function() {
    var p = this.props.product;
    return (
      <tr>
        <td className="qty">{ _.get(p, 'qty') }</td>
        <td className="description">
          <h3>{ _.get(p, 'name') }</h3>
          <p>{ _.get(p, 'description') }</p>
        </td>
        <td className="unit-price">{ _.get(p, 'price') } €</td>
        <td className="subtotal">{ (_.get(p, 'price')*_get(p, 'qty')).toFixed(2) } €</td>
        <td className="actions">
          → <a className="button" onClick={ _.partial(this.updateQuantity, 1) }> + </a>
            <a className="button" onClick={ _.partial(this.updateQuantity, -1) }> - </a>
            <a className="button" onClick={ this.removeItem }>Eliminar</a>
        </td>
      </tr>
    );
  }
});
```

Integración en la arquitectura

- ¿Pasamos siempre el átomo entero (**this.props.state**) de componentes padre a componentes hijo?
- No necesariamente, los componentes tipo “item” que representan un elemento en una lista, probablemente sólo necesitan “su” objeto concreto dentro de la lista (con **key**, recordad)

Integración en la arquitectura

```
var _ = require('mori');
var Catalog = React.createClass({
  propTypes: {
    state: React.PropTypes.object.isRequired
  },
  renderItems: function(products) {
    return _.intoArray(_.map(function(product) {
      return <CatalogItem key={_.get(product, 'id')} product={product} />;
    }, products));
  },

```



Este componente solo necesita un producto

```
render: function() {
  var products = CatalogStore.getProducts(this.props.state);

  return (
    <div className="catalog">
      <div className="catalog-header">
        <h2>Productos</h2>
      </div>
      <div className="catalog-list">
        { this.renderItems(products) }
      </div>
      <div className="footer"></div>
    </div>
  )
}
});
```

Integración en la arquitectura

- Ahora bien, si el componente necesita otros datos, entonces le “cableamos” el átomo y que utilice los Stores necesarios
- Los componentes que reciban la prop **state** tendrán forzosamente que depender de un Store para obtener los datos concretos que necesita
- Los componentes que reciben directamente una estructura inmutable (casi siempre un mapa) podrán acceder directamente a los datos con

```
mori.get(this.props.XXX, "clave")
```


Integración en la arquitectura

```
var _ = require("mori");
var CatalogItem = React.createClass({
  propTypes: {
    product: React.PropTypes.object.isRequired
  },
  onAddClick: function() {
    Dispatcher.emit("CART:ADD", this.props.product);
    Dispatcher.emit("SET:PAGE", "cart");
  },
  render: function() {
    var product = this.props.product;
    return (
      <div className="product row">
        <div className="product-summary col three-fourths">
          <h2 className="product-title">{_.get(product, 'name')}</h2>
          <div className="product-details">
            <div className="product-image col one-fourth">
              
            </div>
            <div className="product-summary col three-fourths">
              <p>{_.get(product, 'description')}</p>
            </div>
          </div>
          <div className="product-add-to-cart col one-fourth">
            <div className="product-price">{_.get(product, 'price')}</div>
            <div className="add-to-cart">
              <a className="button" onClick={this.onAddClick}>Comprar</a>
            </div>
          </div>
        </div>
      </div>
    );
  }
});
```

Integración en la arquitectura

- La mayoría de componentes ahora tendrán que importar (*require*) el Dispatcher, mori y los Stores relevantes.
- A cambio sólo usaremos el estado interno en los formularios, y no tendremos que configurar callbacks vía props.

Ejercicio

- Vamos a terminar la aplicación del carrito de la compra con nuestra arquitectura completa
- Adaptamos los componentes del tema anterior para que usen Stores con **this.props.state**, o mori directamente sobre una prop de tipo “item” para “mostrar” datos, y el Dispatcher para notificar interacciones
- Añadimos los Dispatcher.**listen()** a los Stores
- Usamos el servicio shoppingCart para cargar los datos del catálogo de forma externa

Routing

- Nuestro carrito de la compra del tema anterior tiene un problema
- El usuario **no puede utilizar el botón Atrás** del navegador
- Queremos una experiencia de usuario que no rompa sus hábitos
- Hacer click en Atrás para volver a la página anterior es un hábito **muy arraigado**

Routing

- Para solucionar esto, tenemos que utilizar un router en el cliente
- Su función es detectar cambios en la URL y poder ejecutar código como respuesta
- Hay muchísimas librerías, todas muy similares, por ejemplo:
- <http://visionmedia.github.io/page.js/>

Routing

- ➔ **page**("path", callback, [,callback...])
Configura el router para que atienda la URL **path**, llamando a los callbacks definidos
- ➔ **page**(options) o **page**.start(options)
Inicia el router para que capture los cambios de URL

Routing - configuración Page

```
function configureRoutes () {  
  //page routes  
➔ page ('/', setPage ('catalog')) ;  
  page ('/cart', setPage ('cart')) ;  
  page ('/checkout', setPage ('checkout')) ;  
  
  //start routing  
➔ page ({  
    hashbang: true  
  }) ;  
}
```

Routing - configuración Page

```
//page router "middleware"  
function setPage(page) {  
  return function(ctx) {  
    Dispatcher.emit('SET:PAGE', page);  
    ctx.handled = true;  
  }  
}
```


Routing - configuración Page

```
//page router "middleware"  
function setPage(page) {  
➔ return function(ctx) {  
    Dispatcher.emit('SET:PAGE', page);  
    ctx.handled = true;  
}  
}
```

setPage devuelve una función, que será el callback que ejecute el router, para establecer la página actual

Routing - configuración Page

```
//page router "middleware"  
function setPage(page) {  
  return function (ctx) {  
    ➡ Dispatcher.emit('SET:PAGE', page);  
    ➡ ctx.handled = true;  
  }  
}
```

Usaremos nuestro Dispatcher para establecer la página, y le diremos a page que **hemos atendido la route** (ctx.handled = true)

Routing

- Si incluimos nuestra configuración de rutas en la aplicación y llamamos a *configureRoutes()* antes de montar nuestro componente Root...
- Podemos generar mensajes del Dispatcher simplemente con los cambios de URL
- Además, **Page** captura nuestros enlaces HTML automáticamente, por lo que seguir la url “**/cart**” hará que la página actual cambie al resumen del carrito.

Ejercicio

- Modifica nuestros componentes del tema anterior para usar **page** y permitir al usuario navegar adelante/atrás.
- El botón de añadir al carrito desde el catálogo puede ser un enlace a “/cart”
- El botón de seguir comprando, enlace a “/”
- El botón de finalizar la compra, enlace a “/checkout”
- Nuestros eventos *onClick* se seguirán ejecutando pero además **Page** cambiará la ruta por nosotros