

Tema 4 - Flux

Flux

APPLICATION ARCHITECTURE FOR BUILDING USER INTERFACES

¿Qué es Flux?

- Es una arquitectura de aplicación lanzada por Facebook poco tiempo después de liberar React
- El mensaje clave: **flujo de datos unidireccional**
- Es un gran complemento a React
- Es más un patrón/marco de referencia que un framework formal
- Múltiples “sabores” o variaciones sobre la idea básica

Elementos de Flux

→ **Action**

Representa “algo que ocurre” en nuestra aplicación. Tienen un identificador único y datos asociados. Ej: “Añadir producto al carrito”

```
{ type: “CART:ADD:PRODUCT”, productId: 1 }
```

→ **Store**

Un almacén de datos para un dominio de nuestra aplicación. Ofrece métodos para consultar estado y datos, y atiende todas las acciones en el sistema. Ej: “Dame los productos que hay en el carrito”

```
CartStore.getCartItems()
```

Elementos de Flux

➔ **Dispatcher**

Mecanismo de comunicación: traslada las acciones a nuestros Stores, es un hub central de la aplicación. Ofrece métodos para despachar acciones y para suscribirse a ellas.

➔ **View**

Nuestros componentes React: desde aquí lanzaremos las acciones que serán atendidas por los Stores, y leeremos de éstos los datos necesarios

Elementos de Flux

→ **Action Creators**

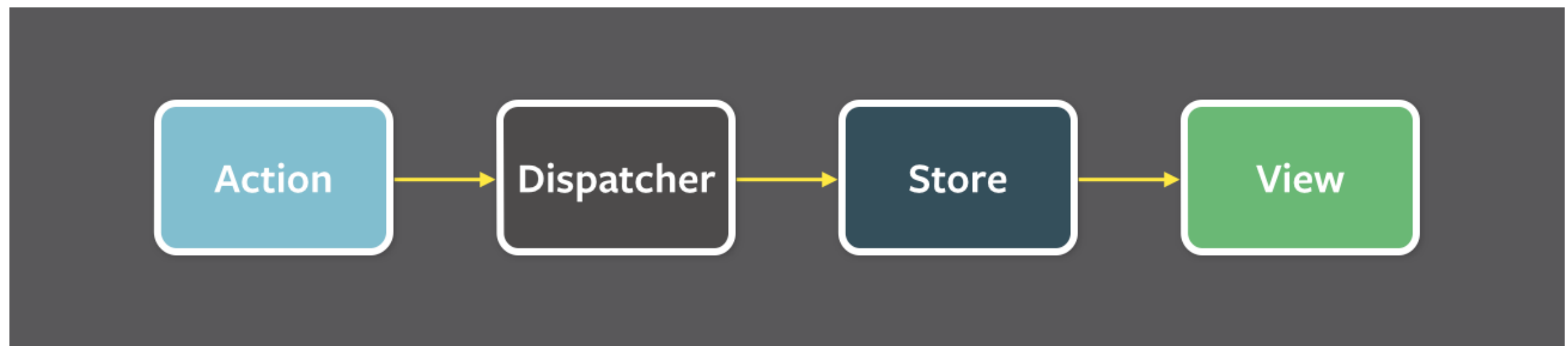
Funciones que crean las acciones y las envían a través del Dispatcher, organizadas en librerías

→ **Services**

Cualquier módulo que acceda a datos externos (ej. API REST) y que lance acciones como resultado de esas operaciones

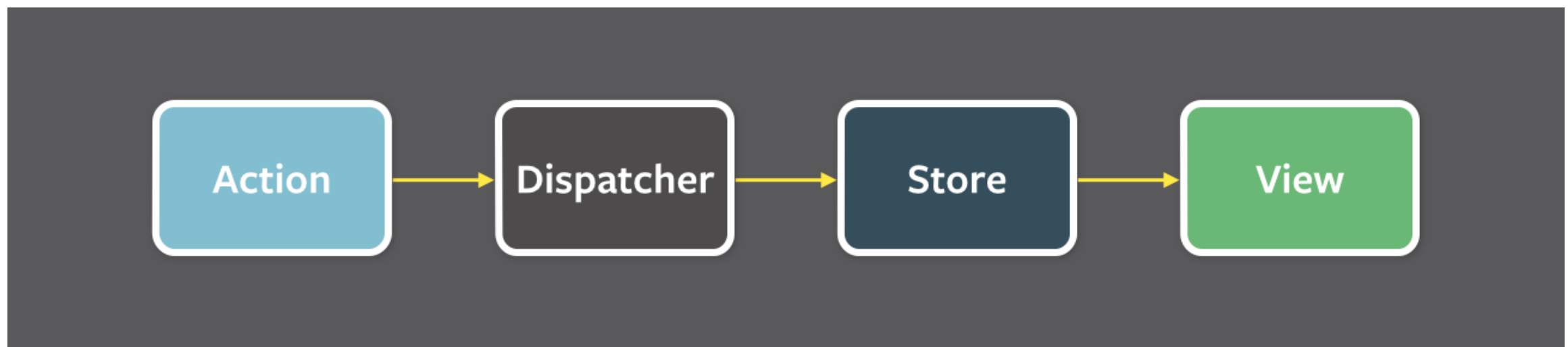
Flujo de datos unidireccional

- Los datos en Flux siempre viajan en **una sola dirección**



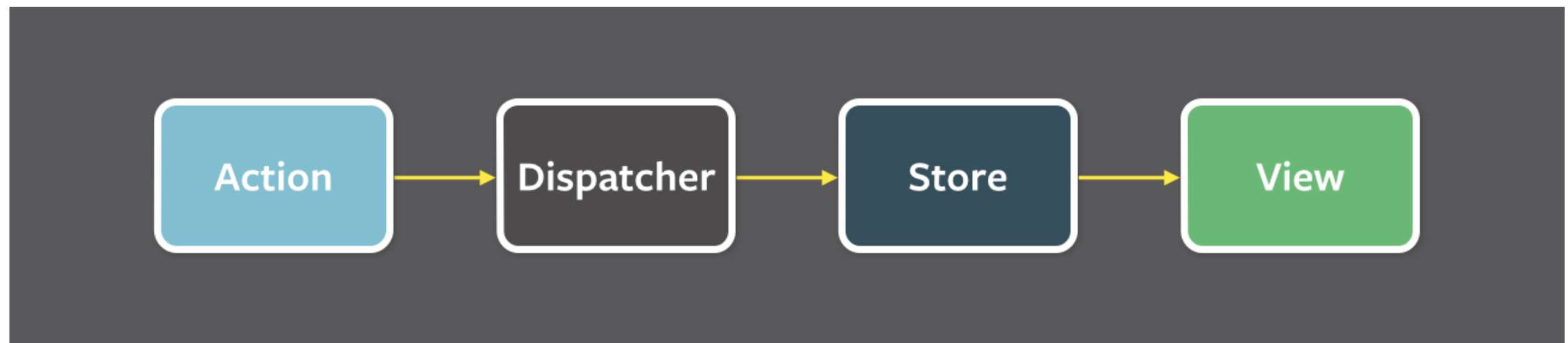
Flujo de datos unidireccional

- Otra forma de verlo: la única forma de provocar un cambio en la UI, es despachando una acción



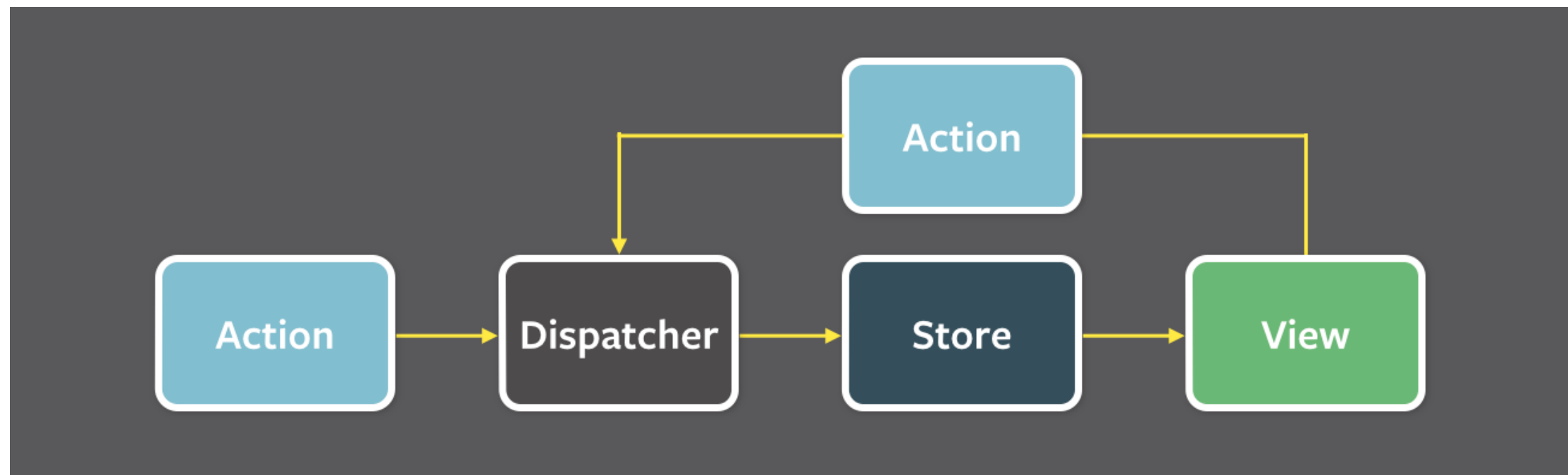
Flujo de datos unidireccional

- Los Stores atienden las acciones y emiten un evento **change** si hay cambios, que las vistas interesadas atenderán para volver a obtener los datos del Store y por tanto **repintarse** (y con ellas sus hijos)



Flujo de datos unidireccional

- Una interacción en una vista provocará que se despache una acción que vuelve a recorrer el ciclo completo



¿Es MVC?

- Aunque lo parezca, no.
- Lo más parecido a un Controlador serán los componentes “grandes” (view-controllers) que se suscribirán a uno o varios Stores y repartirán los datos por la jerarquía de componentes
- Los Stores no son modelos. Si acaso, ocultan dentro modelos, colecciones, etc... Pero son más una **fachada** sobre esos datos.

Acciones

- Son simplemente un objeto Javascript con una propiedad **type** (convención) y el resto de datos relevantes para la acción
- Es una forma de serializar el comportamiento de una aplicación: cualquier cosa que **pueda ocurrir** será modelada como una acción
- El Dispatcher envía cada acción a **todos los Stores**, de forma que una misma acción puede afectar a varios Stores simultáneamente

Acciones - ejemplos

Suceso	Action type	Action data	Ejemplo
Añadir producto al carrito	"CART:ADD:PRODUCT"	productId: Number	<pre>{ type: "CART:ADD:PRODUCT", productId: 1 }</pre>
Cambiar cantidad	"CART:CHANGE:QTY"	productId: Number quantity: Number	<pre>{ type: "CART:CHANGE:QTY", productId: 1, quantity: 1 }</pre>
Navegar a una página	"ROUTE:SET"	page: String	<pre>{ type: "ROUTE:SET", page: "catalog" }</pre>
Almacenar productos	"RECEIVE:PRODUCTS"	products: [...]	<pre>{ type: "RECEIVE:PRODUCTS", products: [{...}, {...}] }</pre>

Creadores de acciones

- Funciones sencillas que abstraen el concepto de acciones y su forma fuera de los componentes
- Normalmente en la misma función lanzamos la acción a través del Dispatcher

Creadores de acciones

```
//actions
import ActionTypes from './action_types';
import { Dispatcher } from 'flux';

export function addProductToCart(productId) {
  Dispatcher.dispatch({
    type: ActionTypes.CART_ADD_PRODUCT,
    productId: productId
  });
}

export function changeProductQuantity(productId, newQuantity) {
  Dispatcher.dispatch({
    type: ActionTypes.CART_CHANGE_QTY,
    productId: productId,
    quantity: newQuantity
  })
}

export function receiveCatalogProducts(products) {
  Dispatcher.dispatch({
    type: ActionTypes.RECEIVE_PRODUCTS,
    products: products
  });
}
```

Creadores de acciones

//actions



```
import ActionTypes from './action_types';
```



```
import { Dispatcher } from 'flux';
```

```
export function addProductToCart(productId) {
```

```
  Dispatcher.dispatch({  
    type: ActionTypes.CART_ADD_PRODUCT,  
    productId: productId  
  });  
}
```

```
export function changeProductQuantity(productId, newQuantity) {
```

```
  Dispatcher.dispatch({  
    type: ActionTypes.CART_CHANGE_QTY,  
    productId: productId,  
    quantity: newQuantity  
  })  
}
```

```
export function receiveCatalogProducts(products) {
```

```
  Dispatcher.dispatch({  
    type: ActionTypes.RECEIVE_PRODUCTS,  
    products: products  
  });  
}
```

Creadores de acciones

//actions


import React from *'react'*;

 **import** { addProductToCart } from *'../actions'*;

const CartItem = React.createClass({

//...

handleAddToCartClick: **function**(e) {

 *addProductToCart*(**this**.props.product.id);

},

});

Dispatcher

- El Dispatcher en Flux es un elemento de infraestructura simplemente
- **Todas las acciones enviadas mediante el Dispatcher fluyen a todos los Stores registrados contra él**
- Implementación de referencia:
<https://github.com/facebook/flux>

Dispatcher

- Ofrece sólo **tres métodos** para trabajar con él
- **register**(*storeCallback*)
Registra un callback que será invocado con cada acción despachada. Devuelve un **token**
- **unregister**(*token*)
Elimina un callback registrado previamente usando el token obtenido en *register*

Dispatcher

- ➔ **register**(*storeCallback*)

Registra un callback que será invocado con cada acción despachada. Devuelve un **token**

- ➔ **unregister**(*token*)

Elimina un callback registrado previamente usando el token obtenido en *register*

Dispatcher

- ➔ **waitFor**(*[token, ...]*)

Espera a que terminen los callbacks identificados por el Array de *tokens* antes de continuar la ejecución. Para orquestar diferentes Stores.

- ➔ **dispatch**(*action*)

Despacha la acción a todos los callbacks registrados. Por convención *action* es un objeto JS con una clave *type*.

Stores

- Son la fuente de todos los **datos** y **estado** de nuestra aplicación
- **Datos** pueden ser: modelos traídos de una API REST, configuración de la aplicación, información de sesión del usuario...
- **Estado** puede ser: si está un panel visible, si se debe mostrar una modal con un error, si hay errores al validar un formulario...

Stores

- Normalmente tendremos un Store dedicado a un dominio de nuestra aplicación
- En la tradición Flux son **singletons**
- Tendrán referencias solo a **Dispatcher** y constantes **ActionTypes** y opcionalmente a otros Stores (para **waitFor**)

Stores

- Mantiene sus datos en variables privadas
- Registra un callback en el **Dispatcher** que atiende las acciones
- Ofrece *getters* para acceder a su estado
- Emite un evento **change** para notificar a las Vistas que sus datos han cambiado
- **Sin** *setters*: todos los cambios deben venir por una acción

Stores

- **flux/utils** ofrece la clase **Store**: base publicada por Facebook.
- Ya incluye:
 - **addListener(callback)**
 - **__emitChange()**
 - **getDispatchToken()**

Stores

- Sólo tenemos que implementar **__onDispatch**(action) para recibir las acciones
- Dentro de **__onDispatch** llamaremos a **this.__emitChange** si una acción modifica los datos del Store
- El constructor recibe como argumento nuestro Dispatcher, para suscribirse automáticamente

Stores - ejemplo

```
import { Store } from 'flux/utils';
import AppDispatcher from '../app_dispatcher';

//private data
let _counter = 0;

function incrementCounter() {
  _counter++;
}

class CounterStore extends Store {
  getCounter() {
    return _counter;
  }

  __onDispatch(action) {
    switch(action.type) {

      case 'COUNTER_INCREMENT':
        incrementCounter();
        this.__emitChange();
        break;

      default:
        //noop

    }
  }
}

export default new CounterStore(AppDispatcher);
```

Stores - Command Query Separation

- La separación comandos / consultas dice que un método en un objeto debería ser siempre uno de los dos tipos
- Los **comandos** modifican el estado
- Las **consultas** devuelven información
- Un comando no devuelve resultados
- Una consulta no tiene efectos secundarios

Uniendo los elementos de Flux

1. Acciones y action creators
2. Stores: estado inicial, atender acciones, notificar cambios
3. Vistas: disparar acciones en respuesta a eventos
4. Vistas-controladores: se suscriben a los cambios en un Store y propagan los datos a sus componentes hijo

Ejercicio - Acciones para nuestro Ecommerce

- Vamos a definir nuestras constantes (*action types*) en un archivo aparte. Ej:

```
export const CART_ADD = 'CART:ADD';
```

- ¿Qué acciones se nos ocurren?

Ejercicio - creadores de Acciones

- Vamos a definir nuestros *action creators* en un archivo que exporte funciones individuales. Ej:

```
export function addProductToCart ...
```

- Nuestros creadores deberán construir la acción (Objeto) a partir de los parámetros y enviarlos a través del Dispatcher
- Tendremos una referencia a las constantes definidas como *action types*

Ejercicio - Stores para nuestro Ecommerce

- Un Store por dominio de la aplicación...
- CatalogStore
- CartStore
- OrderStore
- RouteStore

Conexión Store <-> Contenedores

- Utilizar **componentDidMount** para suscribirse a cambios en Store
- Utilizar **componentWillUnmount** para cancelar esas suscripciones
- Propagar datos de Stores a su propio estado interno, lo que forzará un re-render

Ejemplo - Counter

```
import React from 'react';
import CounterStore from '../stores/counter_store';
import { increment } from '../actions/counter';

const Counter = React.createClass({
  componentDidMount() {
    this._subscription = CounterStore.addListener(this.getState);
    this.getState();
  },
  componentWillUnmount() {
    this._subscription.remove();
  },
  getInitialState() {
    return {
      clicks: CounterStore.getCounter()
    }
  },
  getState() {
    this.setState({
      clicks: CounterStore.getCounter()
    })
  },
  handleClick(e) {
    increment();
  },
  render() {
    return (
      <div>
        <h1>Counter</h1>
        <p>You have clicked { this.state.clicks } times</p>
        <p><button onClick={ this.handleClick }>Click</button></p>
      </div>
    )
  }
});

export default Counter;
```

Ejemplo - Counter

```
import React from 'react';
import CounterStore from '../stores/counter_store';
import { increment } from '../actions/counter';

const Counter = React.createClass({
  ➔ componentDidMount() {
    this._subscription = CounterStore.addListener(this.getState);
    this.getState();
  },
  ➔ componentWillUnmount() {
    this._subscription.remove();
  },
  getInitialState() {
    return {
      clicks: CounterStore.getCounter()
    }
  },
  ➔ getState() {
    this.setState({
      clicks: CounterStore.getCounter()
    })
  },
  handleClick(e) {
    increment();
  },
  render() {
    return (
      <div>
        <h1>Counter</h1>
        <p>You have clicked { this.state.clicks } times</p>
        <p><button onClick={ this.handleClick }>Click</button></p>
      </div>
    )
  }
});

export default Counter;
```

Mixin para contenedores

- Se repite siempre lo mismo:
- `componentDidMount` - suscribir a evento “change” de Store(s)
- `componentWillUnmount` - eliminar suscripciones
- `onChange` - obtener datos de store y guardarlos en estado

Mixin para contenedores

```
import React from 'react';
```

```
➔ const StoreMixin = function(stores) {  
  var __subs = [];  
  return {  
    ➔ componentDidMount() {  
      stores.forEach(s => {  
        __subs.push(s.addListener(this.__onStoreChange));  
      });  
      this.__onStoreChange();  
    },  
    ➔ componentWillUnmount() {  
      __subs.forEach(s => s.remove());  
    },  
    ➔ __onStoreChange() {  
      if(typeof(this.getState) !== 'function') {  
        throw new Error('StoreMixin expects method `getState`');  
      }  
      this.setState(this.getState());  
    }  
  }  
}  
  
export default StoreMixin;
```

Contenedor con Mixin

```
import React from 'react';
import CounterStore from '../stores/counter_store';
import StoreMixin from './store_mixin';
import { increment } from '../actions/counter';

const Counter = React.createClass({
  mixins: [
    StoreMixin([CounterStore])
  ],
  getState() {
    return {
      clicks: CounterStore.getCounter()
    }
  },
  getInitialState() {
    return { clicks: 0 }
  },
  handleClick(e) {
    increment();
  },
  render() {
    return (
      <div>
        <h1>Counter</h1>
        <p>You have clicked { this.state.clicks } times</p>
        <p><button onClick={ this.handleClick }>Click</button></p>
      </div>
    )
  }
});

export default Counter;
```

Pros/contras StoreMixin

- **PROS**

- Muy sencillo de incorporar

- **CONTRAS**

- Usa estado interno del componente
- Puede que los Mixins desaparezcan en futuras versiones de React

Alternativa: HOC

- ¿Recordáis? Componente de orden superior
- Una función que devuelve un **componente de React** que envuelve y controla otro
- Por ejemplo, **connectToStores**(Component)

connectToStores

```
export function connectToStores (Component) {
  var subs = [];

  var connected = React.createClass({
    displayName: 'Connected(' + (Component.displayName || Component.name) + ')',
    //mixins: [ReactComponentWithPureRenderMixin],
    componentDidMount () {
      for (let store of Component.getStores()) {
        subs.push(store.addListener(this.__onStoreChange));
      }
      this.__onStoreChange();
    },

    componentWillUnmount () {
      subs.forEach(s => s.remove());
    },

    getInitialState () {
      return Component.getState();
    },

    __onStoreChange () {
      this.setState(Component.getState());
    },

    render () {
      return (<Component {...this.state} />);
    }
  });
  return connected;
}
```


connectToStores

```
export function connectToStores (Component) {  
  var subs = [];  
  
  var connected = React.createClass({  
    displayName: 'Connected(' + (Component.displayName || Component.name) + ')',  
    //mixins: [ReactComponentWithPureRenderMixin],  
    componentDidMount() {  
      ➔ for(let store of Component.getStores()) {  
        subs.push(store.addListener(this.__onStoreChange));  
      }  
      this.__onStoreChange();  
    },  
  
    componentWillUnmount() {  
      subs.forEach(s => s.remove());  
    },  
  
    getInitialState() {  
      ➔ return Component.getState();  
    },  
  
    __onStoreChange() {  
      ➔ this.setState(Component.getState());  
    },  
  
    render() {  
      return (<Component {...this.state} />);  
    }  
  });  
  return connected;  
}
```

Nuestro componente deberá
implementar métodos
estáticos

getStores(): Array<Store>
getState(): Object

connectToStores

```
export function connectToStores (Component) {
  var subs = [];

  var connected = React.createClass({
    displayName: 'Connected(' + (Component.displayName || Component.name) + ')',
    //mixins: [ReactComponentWithPureRenderMixin],
    componentDidMount () {
      for (let store of Component.getStores ()) {
        subs.push (store.addListener (this.__onStoreChange));
      }
      this.__onStoreChange ();
    },

    componentWillUnmount () {
      subs.forEach (s => s.remove ());
    },

    getInitialState () {
      return Component.getState ();
    },

    __onStoreChange () {
      this.setState (Component.getState ());
    },

    render () {
      ➔ return (<Component {...this.state} />);
    }
  });
  return connected;
}
```

Pasará al componente una
prop por cada clave que
devuelva
Component.getState

Contenedor con connectToStores

```
import { connectToStores } from './connect';
import { increment } from '../actions/counter';

const Counter = React.createClass({
  statics: {
    getStores() {
      return [CounterStore];
    },
    getState() {
      return {
        clicks: CounterStore.getCounter()
      }
    },
  },
  handleClick(e) {
    increment();
  },
  render() {
    return (
      <div>
        <h1>Counter</h1>
        <p>You have clicked { this.props.clicks } times</p>
        <p><button onClick={ this.handleClick }>Click</button></p>
      </div>
    )
  }
});

export default connectToStores(Counter);
```

Pros/contras connectToStores

- **PROS**

- No usamos estado interno del componente, usamos sólo **props**
- Recuperamos **validación de PropTypes** :)
- Muy fáciles de testar (sin “conectarlos”) - exportando el componente y el conectado desde el mismo archivo
- Compatible con sintaxis **class ES6** (futuro probable)

- **CONTRAS**

- Solución más compleja
- Exigimos métodos estáticos al Componente (aunque sería muy fácil de eliminar... ¿cómo?)

Ejercicio - contenedores en nuestro Ecommerce

- Catalog
- Cart
- Checkout

Vistas y Action Creators

- Obtener datos vía props
- Evitar estado interno (excepto en formularios)
- Responder a interacciones llamando a funciones vía props o mediante creadores de acciones

Ejemplo Action Creators - Counter

```
import Dispatcher from '../app_dispatcher';

export function increment() {
  Dispatcher.dispatch({
    type: 'COUNTER_INCREMENT'
  });
}
```

Ejemplo Action Creators - Counter

```
import { increment } from '../actions/counter';

const Counter = React.createClass({
  //...
  propTypes: {
    clicks: React.PropTypes.number.isRequired
  },
  handleClick(e) {
    increment();
  },
  render() {
    return (
      <div>
        <h1>Counter</h1>
        <p>You have clicked { this.props.clicks } times</p>
        <p><button onClick={ this.handleClick }>Click</button></p>
      </div>
    )
  }
});
```


Ejercicio - acciones en nuestro Ecommerce

- addToCart
- changeQty
- removeFromCart
- setPage
- saveOrder
- validateOrder
- receiveCatalogProducts

Flux - resumen

1. **Flujo unidireccional de datos**

Accion -> Dispatcher -> Store -> Vistas

2. **Stores**

Lógica de negocio, almacén de datos/estado

3. **Acciones**

Todas las “cosas” que puedan ocurrir en la aplicación

4. **Vistas**

Contenedores (enlace con Stores)

Componentes puros (sólo props)

Flux - puntos debatibles

- Flux no es un framework cerrado, absoluto, inmóvil
- Puntos abiertos a debate
 - ¿Action creators o Dispatch(xxx) en las Vistas?
 - ¿Cuánta lógica de negocio va dentro y cuánta va fuera de los Stores (Ej. validar pedido)
 - ¿Cómo integramos datos externos?

¿Qué nos queda?

- Routing
- Acceso a APIs HTTP
- Server-side rendering