BackboneJS

Sesión 8 - Arquitectura Flux para aplicaciones React

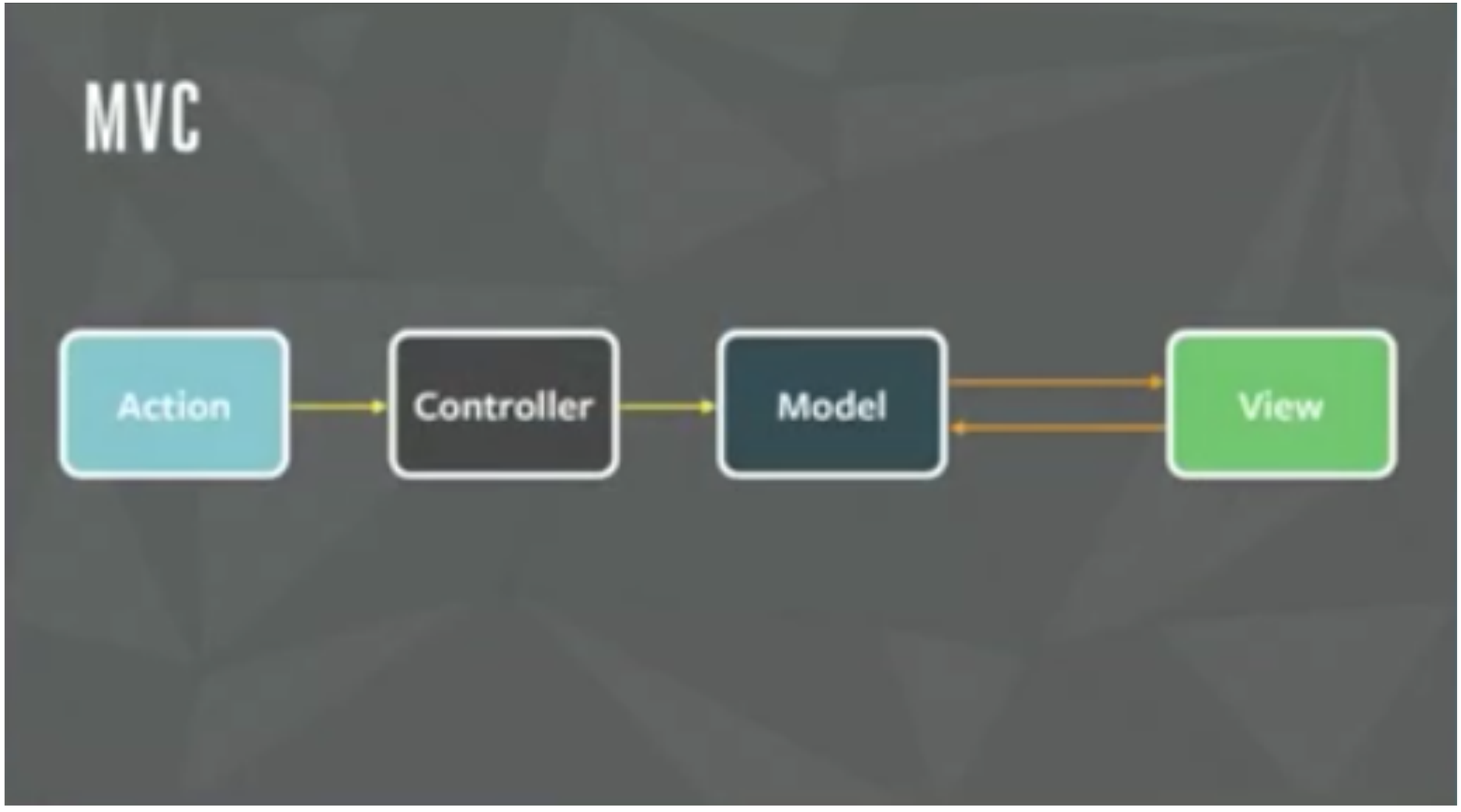
Índice

¿Qué es Flux?

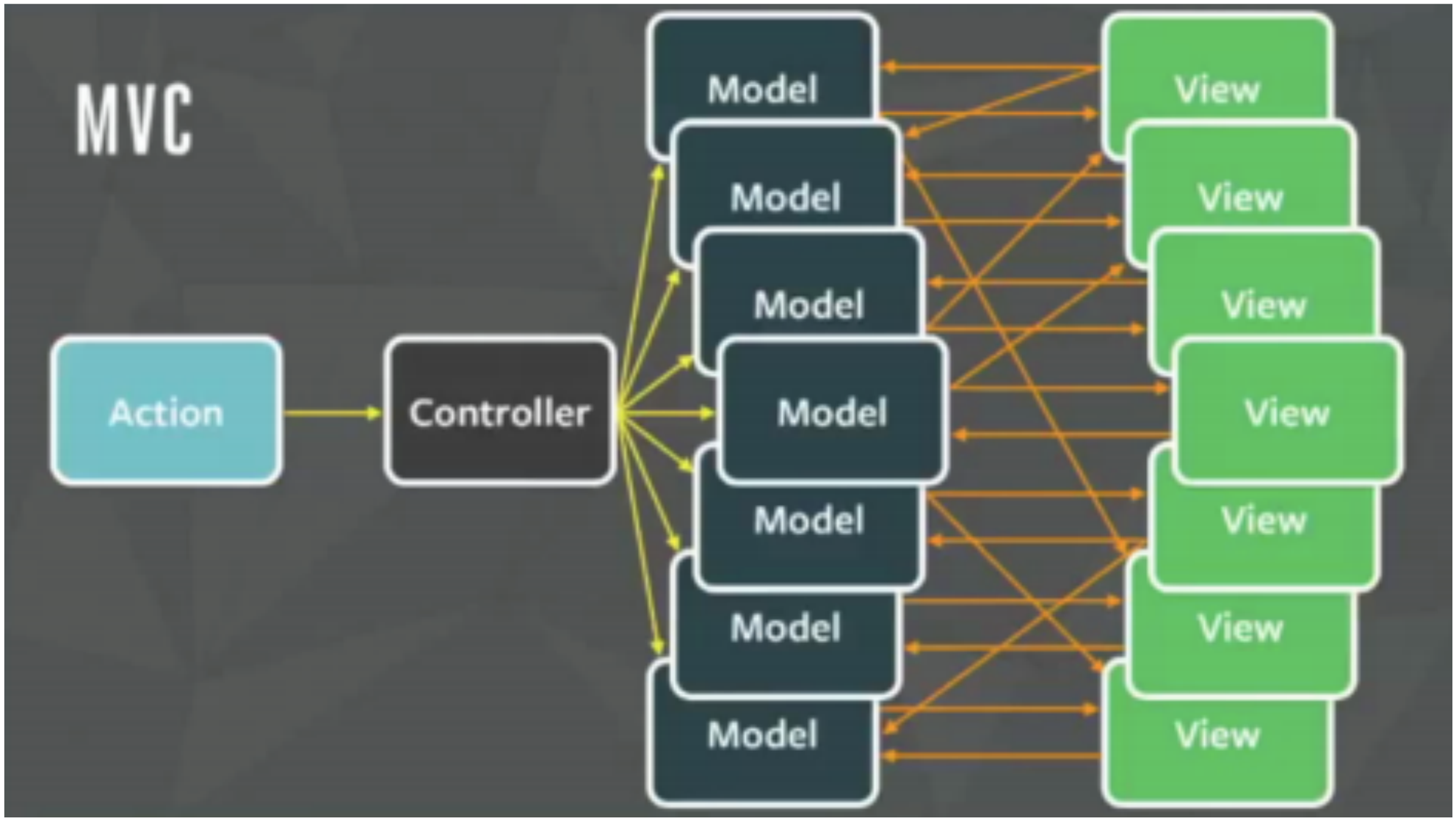
- Flux es una **alternativa a MVC** propuesta por los ingenieros de Facebook y especialmente pensada para aplicaciones de tamaño mediano y grande

- Presentada en la F8, en 2014, en una charla titulada "Hacker Way: Rethinking Web App Development at Facebook". (notas sobre la presentación).

MVC en aplicaciones pequeñas

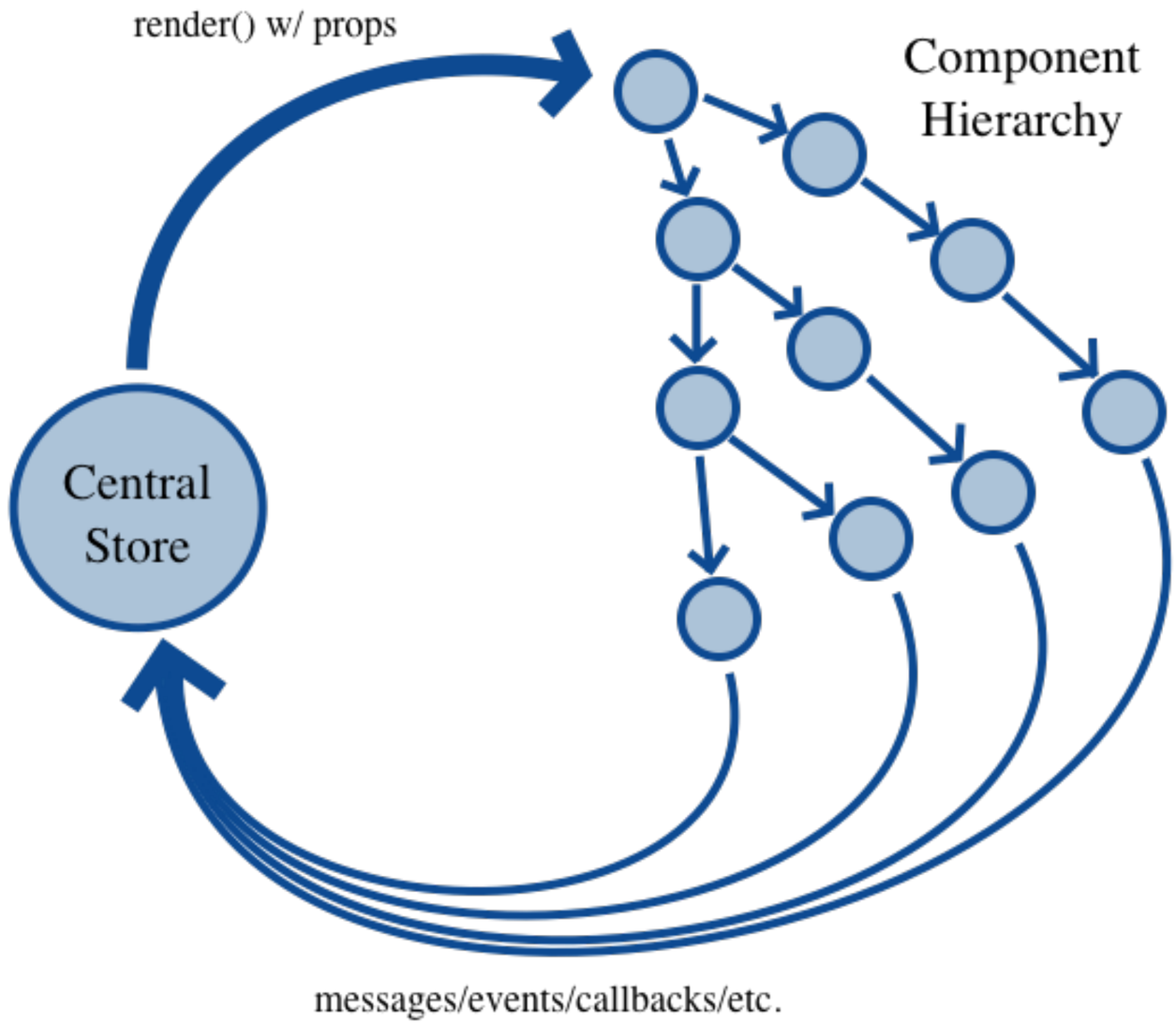


MVC en aplicaciones más grandes



1 idea de Flux: flujo *unidireccional* de datos

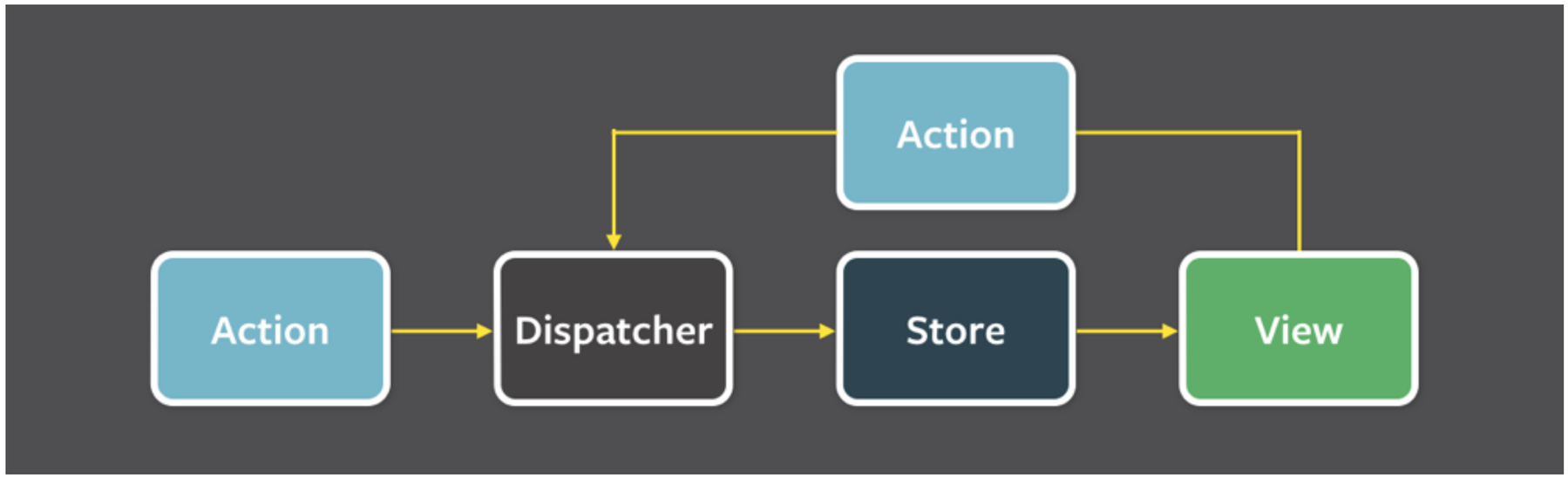
Otra idea de Flux: centralizar el estado en los *stores*



Índice

Elementos de Flux

- **Acciones**: cosas que suceden en la aplicación
- **Dispatcher**: centraliza la acción a ejecutar, se la pasa a los *stores*
- **Store**: lo más parecido al modelo de MVC, encapsula estado y lógica
- **Componente React**: vuelve a disparar una acción



Índice

Librerías para implementar Flux

- Facebook tiene su implementación, que ha hecho *open source*
- Hay muchas alternativas más sofisticadas: Redux, Reflux, Alt, DeLorean, ...

Flux y las herramientas *frontend*

- La mayoría de implementaciones de Flux están pensadas para ser usadas con *npm+webpack* o herramientas similares
- Para simplificar usaremos Flux al estilo "clásico"

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/flux/2.1.1/Flux.js"></script>
```

Ejemplo sencillo de Flux

Un widget que representa el estado de nuestra cuenta bancaria y nos permite retirar y depositar dinero

DISCLAIMER: no funciona de verdad. No modificará vuestra cuenta

El ejemplo completo

http://jsbin.com/kesehunahe/edit?js,output

El dispatcher

- Es la única pieza que nos da implementada directamente la librería de FB
- Es un *singleton*, para centralizar el flujo de ejecución de las acciones
- Instanciar el dispatcher

```
var MiDispatcher = new Flux.Dispatcher();
```

- Ahora veremos cómo despacha las acciones

Acciones

- Objetos que representan los sucesos de nuestra aplicación
- No contienen código, solo datos
- Normalmente constan de un tipo y un *payload*

```
var Ctes = Ctes || {};
Ctes.Acciones = {
  CUENTA_CREADA: 1,
  ABONO: 2,
  CARGO: 3,
}
```

Tipos de acciones en nuestro caso

```
{
  tipo: Ctes.Acciones.ABONO,
  cantidad: 100
}
```

Ejemplo de acción

Creadores de acciones

- En algún sitio se tienen que instanciar las acciones y pasárselas al dispatcher
- No contiene lógica, solo ensambla el objeto Accion

```
var CreadorAcciones = {
  crearCuenta: function() {
    MiDispatcher.dispatch({
      tipo: Ctes.Acciones.CUENTA_CREADA,
      cantidad: 0
    });
  },
  depositarEnCuenta: function(cant) {
    MiDispatcher.dispatch({
      tipo: Ctes.Acciones.ABONO,
      cantidad: cant
    });
  },
…
```

Stores

- Contienen datos + lógica

```
var CuentaStore = {
  balance: 0,
  getBalance: function() {
    return this.balance;
  },
  manejarAccion: function(accion) {
    switch(accion.tipo) {
      case Ctes.Acciones.CUENTA_CREADA:
         this.balance = 0;
         break;
      case Ctes.Acciones.ABONO:
         this.balance += accion.cantidad;
         break;
      case Ctes.Acciones.CARGO:
         this.balance -= accion.cantidad;
         break;
    }
  }
}
```

Stores

- En nuestra aplicación tenemos solo uno, pero normalmente habrá varios
- El *store* contiene el estado, pero no tiene *setters*, la única forma de cambiar el estado es a través de una acción
- El *dispatcher* envía todas las acciones a todos los stores registrados con él, pero cada store decide si le "hace caso" o no
- Si el store quiere procesar la acción ejecutará una lógica y cambiará el estado en función de ella

Registrar un store en el dispatcher

- Relacionamos el dispatcher con el *callback* de manejo de las acciones

```
MiDispatcher.register(CuentaStore.manejarAccion.bind(CuentaStore))
```

Del store a los componentes

- A los componentes React les interesa el cambio de estado de los stores, para repintarse
- No a todos los componentes les interesarán todos los stores. Solución: eventos (como en Backbone)
  - Cuando cambia su estado el store emite un evento
  - Cada componente escucha al store y el evento que le interesa

Emitir y escuchar eventos

- Los navegadores ofrecen soporte para los eventos nativos del DOM, pero no para eventos “a medida”
- Hay que usar una librería adicional. Facebook tiene la suya propia, fbemitter, pero solo se puede instalar con npm. Usaremos aquí una muy similar, EventEmitter

```
//Hacer que un objeto pueda emitir eventos/registrar listeners
var CuentaStore = Object.assign({}, EventEmitter.prototype, {
   …
});
//CuentaStore puede emitir un evento
CuentaStore.emitEvent(‘evento:saludo’, ‘hola soy yo);
//Cualquier objeto puede hacer esto
CuentaStore.addListener(‘evento:saludo’, this.miListener)
```

Del store a los componentes

- Un componente React escuchará los eventos que le interesen sobre los *store* que le interesen.

```
componentDidMount: function() {
    CuentaStore.addListener(Ctes.Eventos.CAMBIO_CUENTA, this.cambiaStoreListener)
},
```

- Queremos que el componente se repinte al recibir el evento
  - Recordemos que los componentes se repintan automáticamente si cambia su estado

Repintar los componentes

- Cada jerarquía de componentes React copiará en su componente de nivel superior la parte de estado que le interese del store,

```
cambiaStoreListener: function() {
    this.setState({balance: CuentaStore.getBalance()});
},
```

- Es posible que el componente de nivel superior no tenga nada de interfaz y se limite a cumplir la misión de almacenar el estado y disparar el repintado

¿Preguntas?