Introducción a Programación Funcional Reactiva

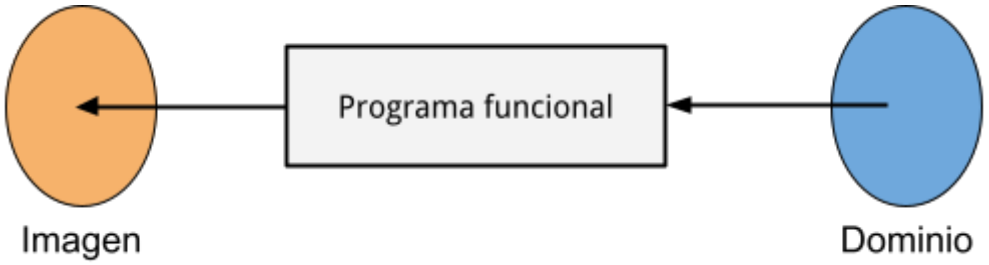
(usando ReactJS)

Introducción

En la construcción de interfaces de usuarios, ya es conocido el patrón MVC …
En este apunte veremos la construcción de UI partiendo de una nueva filosofía que propone la programación funcional.

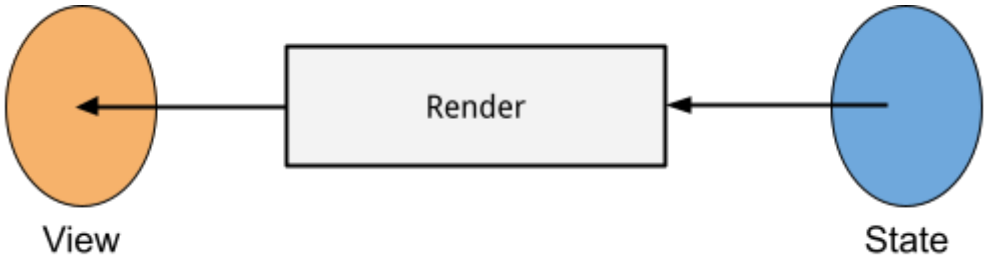
Programación funcional

La programación funcional se trata de un paradigma de programación cuyos componentes principales son funciones, del estilo matemático. Esto significa que tendrá un dominio, una imagen y deberá cumplir los teoremas de unicidad y existencia. Esto les permite representar un programa como una combinación de funciones, que a su vez termina siendo una enorme función:



Lo importante de esta imagen es que **una función siempre va a devolver un solo valor** (perteneciente a la imagen), y que ese valor **depende solamente de los valores ingresados** (pertenecientes al dominio). De esta forma se crea una relación directa entre los elementos del dominio y la imagen.

Para la construcción de interfaces de usuario, dentro del paradigma funcional, necesitaríamos crear una función del siguiente tipo:



En donde el **estado** es la representación abstracta del modelo de la aplicación, y la **vista** se obtiene al *aplicar* nuestra función con dicho estado. Nuestra función es la encargada de crear los elementos visuales a partir de un estado (o modelo), y si queremos *cambiar* la vista de la aplicación deberíamos evaluar dicha función con otro estado. Hablaremos sobre cambios y cómo se evalúa esta función a través del tiempo más adelante.

Bajando a tierra

Para entender esta forma de trabajar es necesario bajar las ideas a código para ver cómo se implementan. Para esto vamos usar una biblioteca creada por Facebook para la construcción de interfaces web (como *Single Page Applications*): ReactJS.

Primero necesitamos crear un proyecto, para facilitar esto usaremos Create React App. Una vez creado el proyecto podremos levantar la app con `npm start`. Eso debería abrir una ventana del navegador, y como dice allí, el archivo que vamos a estar editando es `src/App.js`.

```
class App extends Component {
 render() {
   return (
     <div className="App">
       <header className="App-header">
         <img src={logo} className="App-logo" alt="logo" />
         <h1 className="App-title">Welcome to React</h1>
       </header>
       <p className="App-intro">
         To get started, edit <code>src/App.js</code> and save to reload.
       </p>
     </div>
   )
 }
}
```

Para entender este código es necesario saber qué es un *componente React* y cómo hace para generar la vista correspondiente.

React llama componente a esa función que transforma un estado a la vista que queremos dibujar en pantalla. Un componente se puede modelar como una clase que hereda de *Component* y tiene un método *render()* que retorna una vista.

Como ahora la aplicación es muy sencilla, no tiene un estado que la represente, simplemente retorna siempre la misma vista.

Para crear la vista utiliza JSX.

¿Qué es JSX?

No vamos a entrar mucho en detalle con esto, pero vamos a decir que:

- Es una extensión del lenguaje JavaScript para crear componentes web, lo cual hace que se vea similar a HTML, pero no lo es. **Es código JavaScript**.
- En realidad se están creando *elementos React*, que pueden pensarse como objetos que saben cómo manipular el DOM de un HTML para que termine renderizando los elementos visuales como lo haría el HTML que intenta simular.

De esta forma si quisiéramos que la app salude a “Pepito”, nos conviene crear un componente que represente el saludo:

```
class Saludo extends Component {
 render() {
   return <h2>Hola, Pepito</h2>
 }
}
```

Y como todo componente está hecho formado por componentes, lo podemos agregar en nuestra App.

```
class App extends Component {
 render() {
   return (
     <div className="App">
       <header className="App-header">
         <img src={logo} className="App-logo" alt="logo" />
         <h1 className="App-title">Welcome to React</h1>
       </header>
       <p className="App-intro">
         <Saludo />
       </p>
     </div>
   )
 }
}
```

¡Perfecto! Pero si queremos saludar también a “María” deberíamos duplicar el componente. Para eso es mejor que nuestro componente *Saludo* reciba el nombre de la persona a saludar para poder hacer algo como:

```
class App extends Component {
```

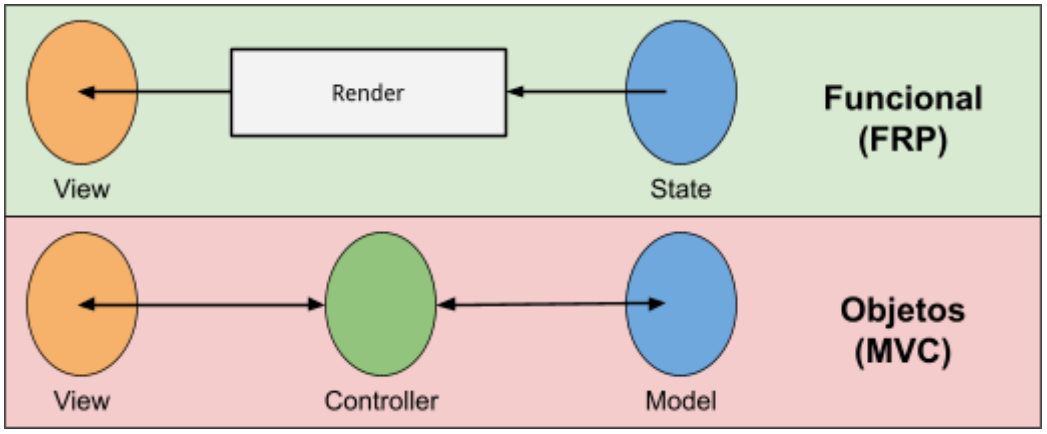
```
 render() {
   return (
     <div className="App">
       <header className="App-header">
         <img src={logo} className="App-logo" alt="logo" />
         <h1 className="App-title">Welcome to React</h1>
       </header>
       <p className="App-intro">
         <Saludo nombre="Pepito"/>
         <Saludo nombre="María"/>
       </p>
     </div>
   )
 }
}
```

Estos *atributos,* que le pasamos al componente Saludo, React lo asignará a una variable ***props*** al instanciar el componente. De esta forma, para generar la vista del saludo, vamos a tener que usar la variable ***this.props.nombre***. Lo grandioso de esto, es que como JSX es código JS y por lo tanto un lenguaje de programación (y no de *presentación* como HTML), podemos programarlo fácilmente.

```
class Saludo extends Component {
 render() {
   return <h2>Hola, {this.props.nombre}</h2>
 }
}
```

Comparación de diseños

Es interesante comparar esta forma de pensar una interfaz de usuario, como una función que dado un estado arma la vista, contra otros patrones conocidos como el de MVC, característicos del **paradigma orientado a objetos**.



Podemos encontrar grandes **similitudes** entre estos dos diseños:

- Ambos proponen modelar los datos necesarios para la vista como una entidad aparte, encargada de manejar la lógica y "desacoplarla" de la vista en sí.
- Ambos poseen un componente entre ese modelo abstracto y los elementos visuales que finalmente llegarán al usuario. Este componente es el responsable de **relacionar** dichas entidades

Y de este último punto se desprende una **diferencia** crucial:

- Para MVC: ese componente intermedio (*controller*) es un objeto encargado de sincronizar la vista y el modelo a través del tiempo.
- Para FRP: ese componente intermedio es una función que se evalúa dado un estado de la aplicación y retorna la vista correspondiente a ese estado.

- Para MVC tanto la vista como el modelo existen independientemente uno del otro y es responsabilidad del controller el tenerlos sincronizados (o *bindeados*).
- En cambio, FRP propone crear la vista a partir de un estado. La vista no existe de por sí, sino que se obtiene a partir de evaluar una función con el estado de la aplicación.

- Una consecuencia de estos últimos puntos: el patrón MVC se basa en el *efecto colateral* (los objetos van mutando a lo largo del tiempo), mientras que en FRP no hay efecto (la vista se obtiene a partir de un estado, y un nuevo estado generará una nueva vista). Esta característica es propia del paradigma funcional y está implícito en la definición de *función*.

Cambios de estado

¡Bien! Esta idea de construir interfaces de usuario que propone FRP parece simple de entender y trabajar.

Sin embargo hay algo que todavía no queda claro:

- La vista se crea al evaluar una función con el estado de la aplicación.
- Si queremos cambiar algo de la vista, debemos **crear un nuevo estado**.
- Un nuevo estado generará una nueva vista, independiente de los estados anteriores y posteriores; esto es porque el paradigma funcional es **atemporal**.
- Entonces,
    - ¿Cómo hace la vista para generar un nuevo estado?
    - ¿Cómo o cuándo se evalúa ese estado para generar la nueva vista?
    - Y si al crear un nuevo estado se genera una vista totalmente nueva, ¿todo el tiempo se estaría re-dibujando toda la vista? ¿cómo afecta a la performance?

Para contestar todas estas preguntas es necesario entender la parte **reactiva** del asunto.

Programación reactiva

La programación reactiva se enfoca en el **flujo de datos** y cómo **propagar sus cambios** entre componentes. Su diseño se basa en el patrón Observer, en donde todo componente observa a aquellos otros de los que depende, y antes cambios en sus dependencias es avisado para actualizar su propio valor.

Un ejemplo sencillo

Si suponemos que el precio final de una compra se calcula:

*precioFinal = precio(carrito) * porcentaje(descuento) + precio(formaDeEnvío)*

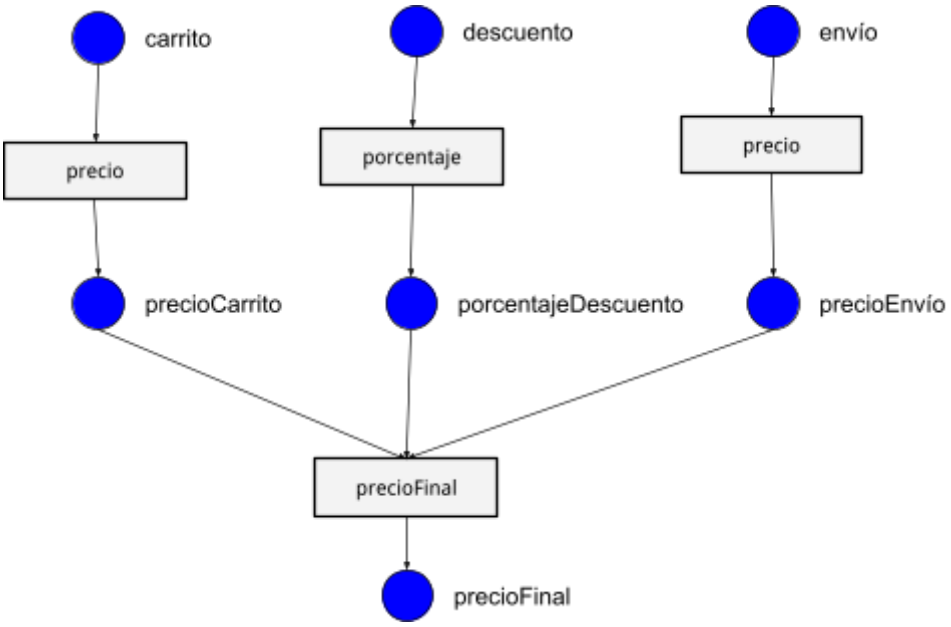
Podemos ver que el valor del precio total depende del valor del precio del carrito, el porcentaje de descuento y el precio de la forma de envío. Por lo tanto, siguiendo la idea reactiva, primero podemos descomponer esto en:

precioCarrito = precio(carrito)
porcentajeDescuento = porcentaje(descuento)
precioEnvío = precio(formaDeEnvío)

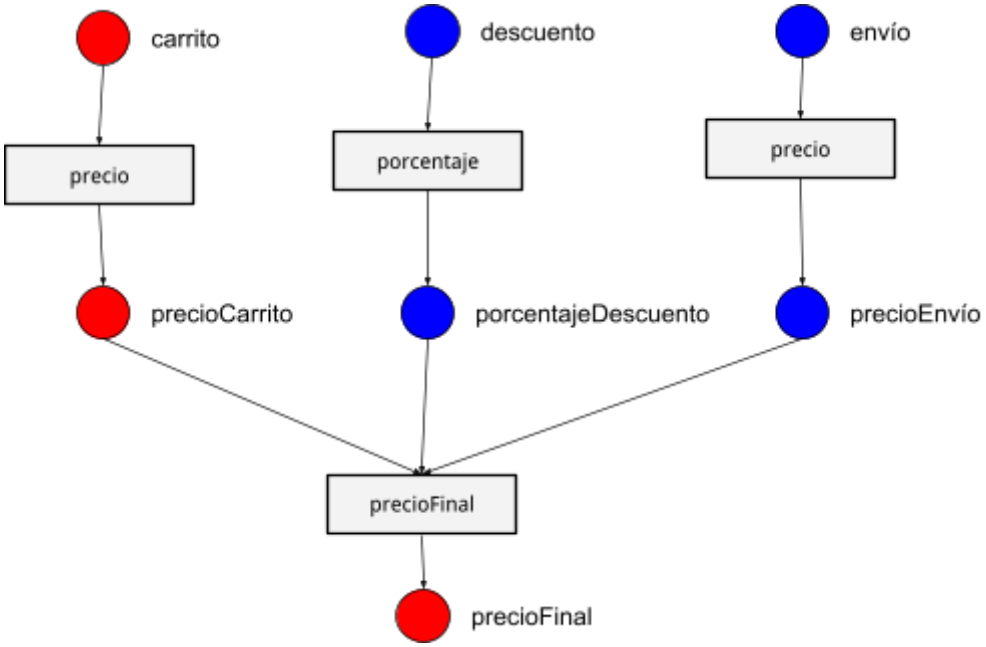
*precioFinal = precioCarrito * porcentajeDescuento + precioEnvío*

El precio final deberá subscribirse a los cambios de dichas dependencias; y a su vez, esas dependencias podrían estar observando otros datos del cual dependa su valor (por ejemplo, el precio del carrito debería estar subscrito a los cambios del propio carrito).

De esta forma, se arma una red de dependencias, que indican la **relación entre observados y observadores**, o mejor dicho, del **flujo de los datos**:



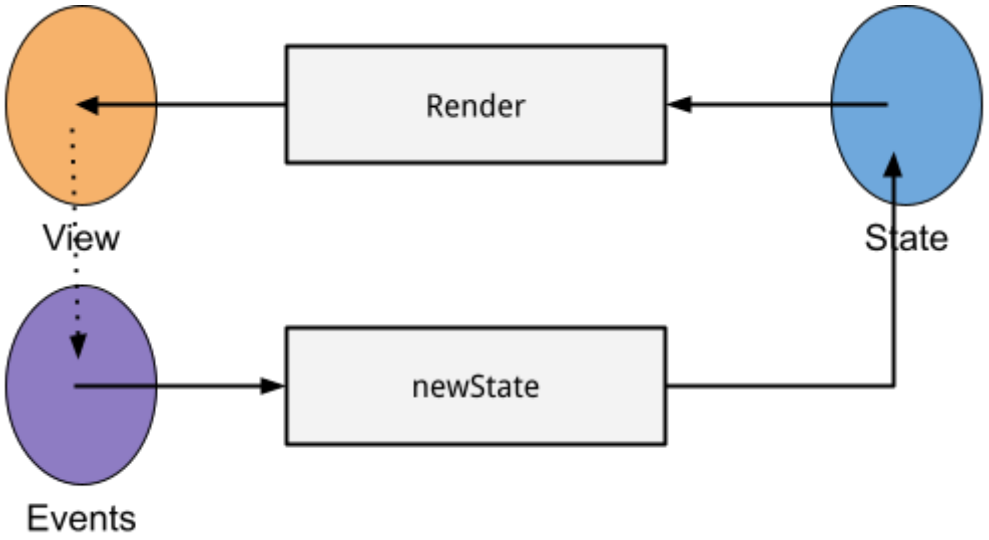
Ante cambios en alguno de estos datos, se avisará por medio *eventos* a los nodos dependientes para que se recalculen sus valores también:



Esto ya nos aclara un poco el panorama que esperamos:

- Se generarán eventos que cambien parte del estado de la aplicación.
- Esos cambios se *propagarán reactivamente* por todos los lugares que dependen de él, incluyendo la vista, para actualizarlos.
- Aquellos componentes que no dependan de dicho cambio no se verán modificados.

Lo importante es entender que cualquier componente podría emitir eventos que cambien el estado, sobre todo la vista:



Estos eventos generarán un nuevo estado que a su vez generarán una nueva vista, pero de manera reactiva, o sea sólo actualizando los valores que realmente cambien, dejando intacto todo el resto.

Un poco más de código

Lo bueno para nosotros es que todo esto es transparente, ya que React se encargará de eso.

Ya vimos cómo crear componentes React encargados de transformar un estado (*props*) en su vista correspondiente, pero por ahora ese estado es estático (no tenemos forma de cambiarlo a través del tiempo).

Para poder agregarle dinamismo vamos a necesitar de otro objeto que represente el estado dinámico (llamado propiamente *state*). Nosotros debemos indicar cuál es el estado inicial, en el ***constructor***; y luego para cambiarlo, llamar a un método especial que tienen los componentes React, ***setState()***, con el cambio de estado.

Algo importante sobre la forma de flujo de una aplicación React es que mientras las *props* son atributos que provienen de un componente padre, el *state* es propio de cada componente en cuestión, y no debería ser conocido por el padre.

Si quisiéramos crear un componente que sea un contador, quedaría:

```
class Contador extends Component {
 constructor(props) {
   super(props)
   this.state = { contador: 0 }
 }

 sumar() {
   this.cambiarContador(this.state.contador + 1)
 }

 restar() {
   this.cambiarContador(this.state.contador - 1)
 }
```

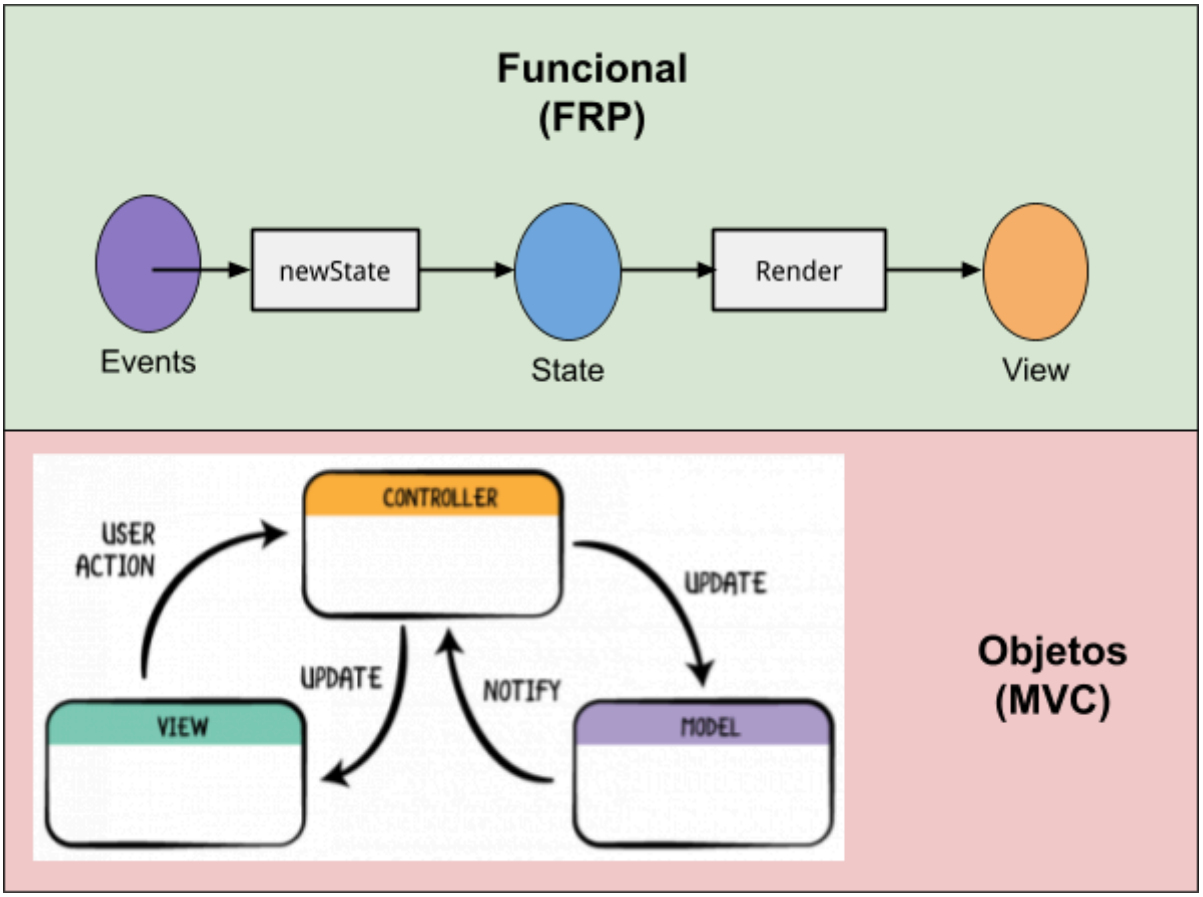
```
  cambiarContador(n) {
    this.setState({contador: n})
  }

  render() {
    return (<div>
      <a href="#" onClick={(event) => { this.restar() }}>-</a>
      {this.state.contador}
      <a href="#" onClick={(event) => { this.sumar() }}>+</a>
    </div>)
  }
}
```

Se ve claramente como:

- En el constructor de la clase definimos el estado inicial: `this.state = { contador: 0 }`
- La vista puede generar 2 tipos de eventos: `sumar()` y `restar()`
- Ambos eventos terminan llamando al método `setState()` que es el encargado de llevar a cabo el proceso reactivo de actualización del estado, que terminará actualizando la vista.

Comparación de cambios

Comparando cómo se producen los cambios de estados en FRP vs MVC es donde se nota la mayor diferencia entre diseños:



Mientras que **para FRP el estado es estático**, lo que significa que ante algún evento se genere todo un estado nuevo que terminará por cambiar la vista respectiva; **MVC trabaja sobre un estado dinámico** que se debe ir actualizando junto con la vista.

Se puede apreciar el **flujo unidireccional de FRP** , que hereda del paradigma funcional, contra el **flujo bidireccional que destaca a MVC**.

Conclusiones

En este apunte vimos los conceptos básicos que plantea **FRP** en la construcción de interfaces de usuario. Además fuimos comparándolo con un patrón muy conocido: **MVC**.

Por lo tanto tenemos dos formas, bastantes distintas, de diseñar nuestras aplicaciones, y la "*mejor opción*" dependerá del tipo de interfaz que queramos crear, o sea, de sus requerimientos. Entonces, la pregunta que nos podríamos hacer es: **¿Cuál es la mejor de las dos propuestas para construir una determinada UI?**

No existe una respuesta absoluta para esta pregunta, pero sí nos podemos guiar por aquellas cualidades que nos beneficia cada diseño. Les dejamos algunos ejemplos:

1. Importancia del flujo de estados

Una característica a tener en cuenta es si **queremos o no llevar un control de los estados** por los que pasa la aplicación.

Por ejemplo, para una aplicación que tiene un formulario: muchos campos para completar y un botón de confirmación al final, no es importante los cambios de estado: no importa si primero completa el nombre y después el apellido, o en cualquier otro orden. Lo que importa es cómo está el estado al confirmar.

En estos casos, trabajar con el patrón MVC que *bindean* cada campo con atributos de un objeto, o sea que el objeto va mutando a medida que se llena el formulario, nos viene bárbaro. Lo que seguramente vamos a querer es hacer algo con ese objeto al presionar alguno de los botones de confirmación, sin importarnos cómo llegó a construirse ese objeto, o cómo llegó a

tener el estado apropiado.

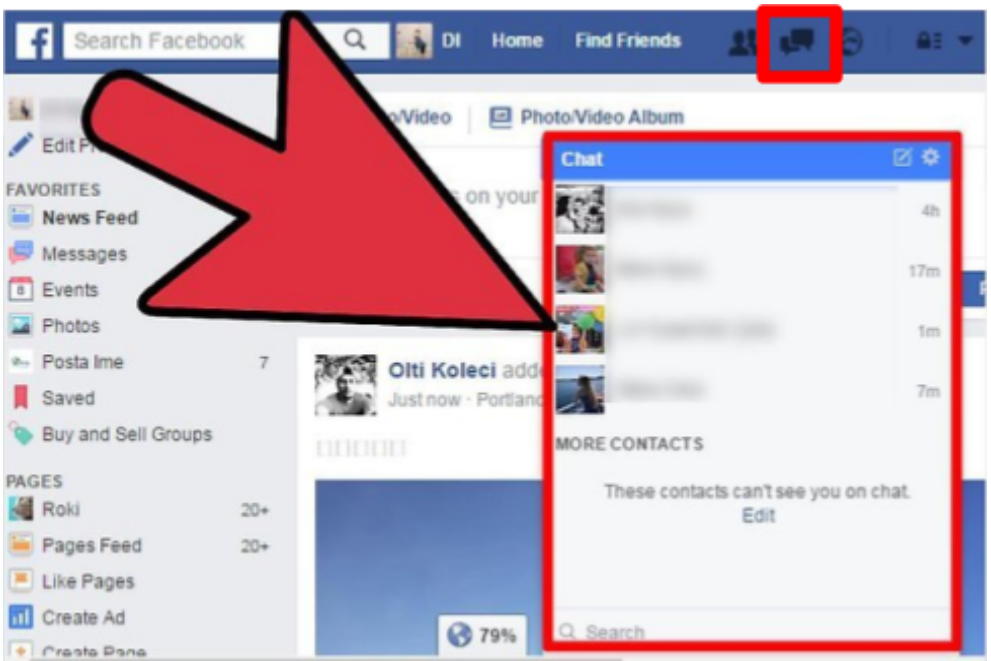
En cambio, si construyéramos una aplicación que lleva un control de los gastos en una casa, es probable que nos interese cada vez que cambie el estado (agregar / modificar / eliminar un gasto) para, por ejemplo, actualizar los saldos parciales.

En este caso pensar del modo que plantea FRP sería beneficioso para llevar un control del cambio de estado. De esa manera, cada vez que cambien los gastos de la casa se podrá actualizar la lista, recalcular los gastos parciales, y guardar la "actividad".

2. Interdependencia de eventos

Otro factor a tener en cuenta es la interdependencia de eventos entre distintos componentes de la pantalla. **Cuando existen componentes que pueden afectarse mutuamente, de distintas maneras, el control de los eventos puede complejizarse.**

Este es el ejemplo del chat de Facebook, una de las razones por la cual crearon ReactJS, el cual tiene varias vías de acceso, o sea, varios componentes involucrados.



En este ejemplo, tanto el clickear sobre el ícono de mensajes como abrir una conversación tiene repercusiones uno sobre el otro. La gente de Facebook se vió en problemas al querer evitar que ambas acciones colisionen y/o evitar loops.

Comprobaron que al propagar los eventos de forma reactiva (evitando loops) y creando las pantallas en forma funcional (cada nuevo estado genera una nueva vista) pudieron evitar estos problemas, como consecuencia del flujo unidireccional que vimos anteriormente.