Curso de React y React Native

Clase 06

Agenda de la clase

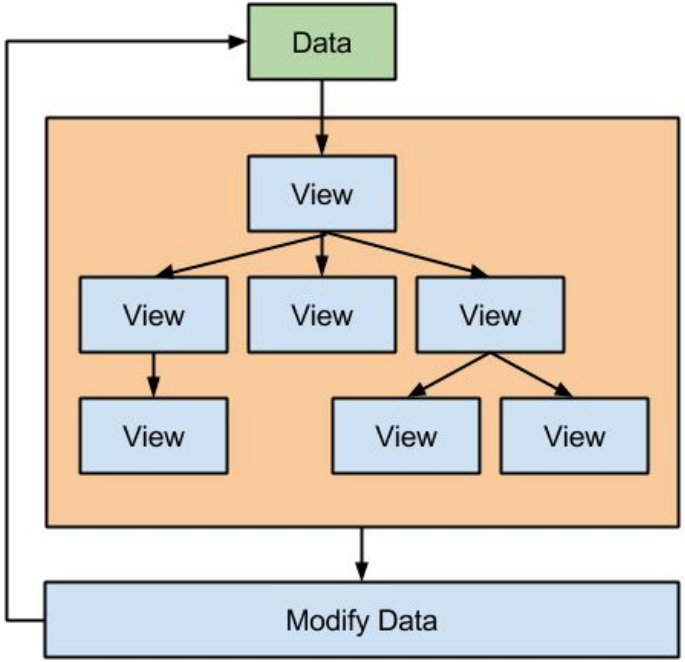
Agenda

- Redux
- Ejercicios.

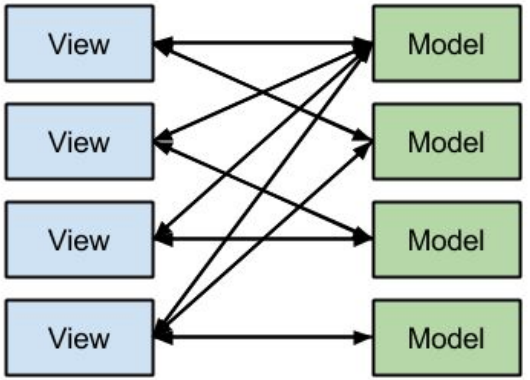
Redux

Introducción

React vs MV*



React



MV* Data Binding

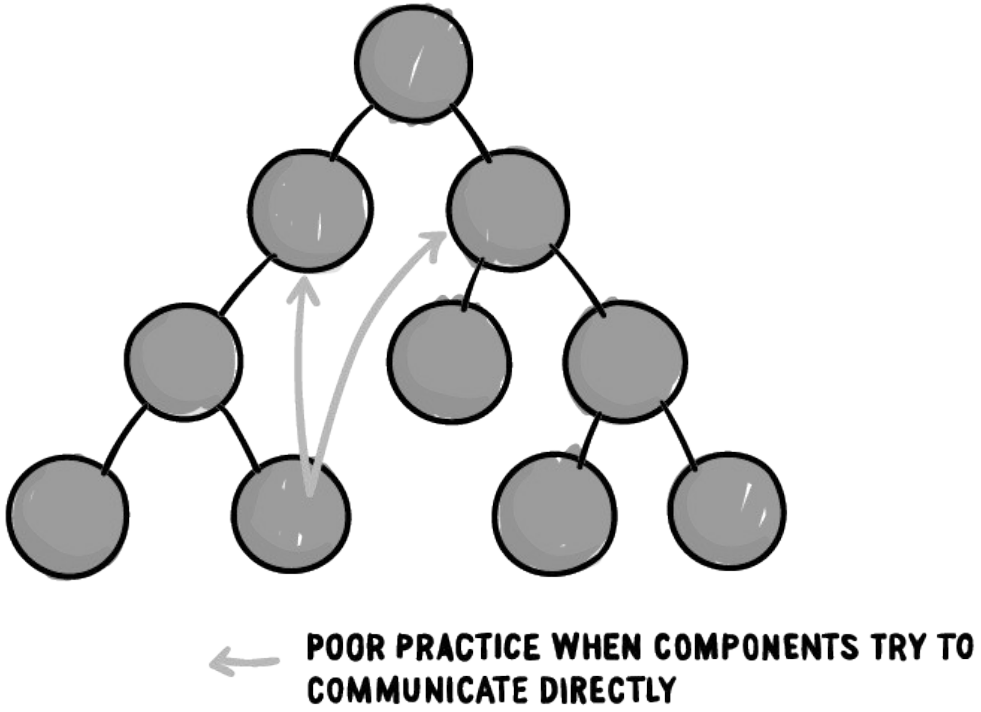
React – Problema con el flujo de datos

En React los componentes tienen dos formas de manejar los "datos" que se muestran al usuario: las `props` que reciben del componente padre o el `state` propio de cada componente (privado).

La forma de pasar información de un componente padre a un hijo es mediante `props` (a partir de sus propias `props` o de su `state`) y de un componente hijo al padre es mediante alguna función que reciba de su padre por `props`.

Esto causa que cuando la jerarquía de componentes crece y existe algún dato en un componente alto en la jerarquía que debe llegar a uno abajo, se genere un "pasamanos" de `props` (componentes reciben `props` solo para pasarla a sus hijos).

React – Problema con el flujo de datos

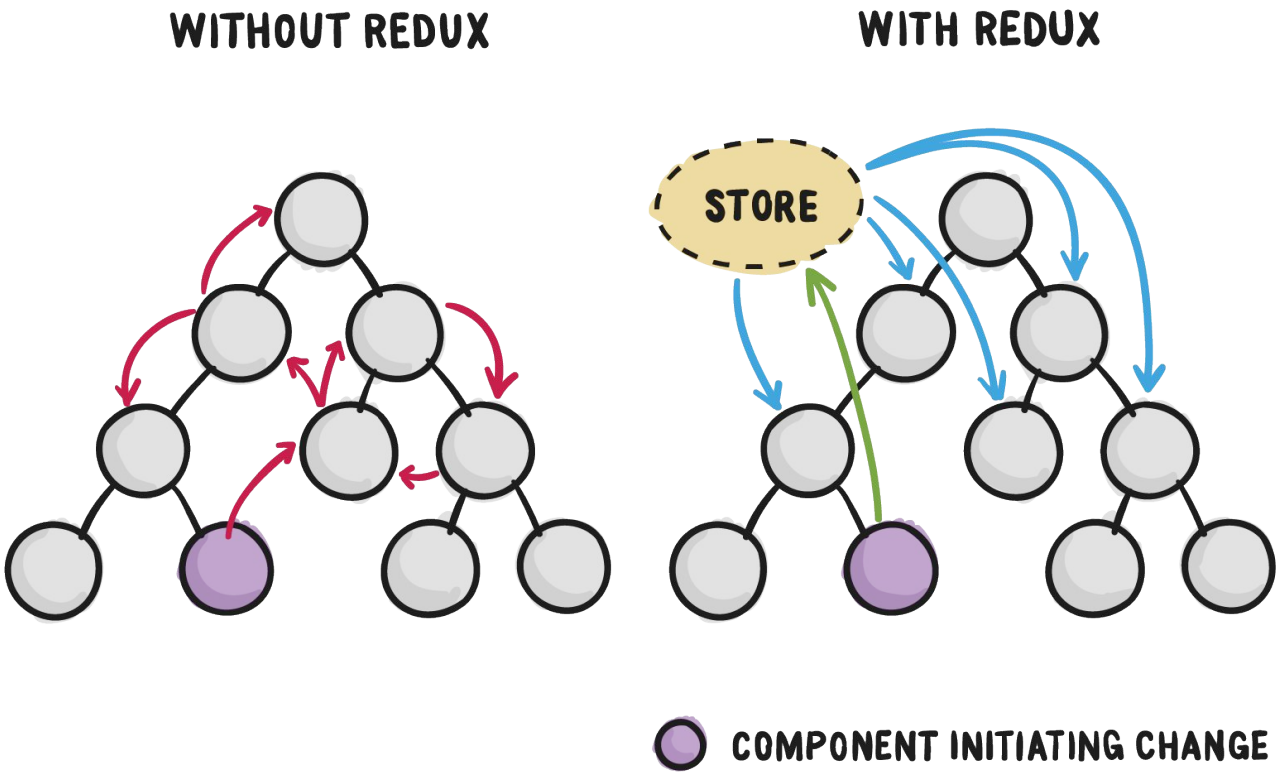


React – Problema con el flujo de datos

La solución que brinda Redux para esto es simple:

Una estructura única que guarda todo el estado de la aplicación en tiempo de ejecución, llamada Store.

React – Redux


WITHOUT REDUX
WITH REDUX
STORE
COMPONENT INITIATING CHANGE

React – Redux

"Redux is a predictable state container for JavaScript apps".

Es una librería que nace como una implementación de Flux (arquitectura propuesta por Facebook). Es muy chica (2KB), su código muy corto y entendible, pero los conceptos que propone son lo que la hacen tan interesante y poderosa.

Define tres conceptos importantes:

- **Store** (donde se guarda el state).
- **Actions.**
- **Reducers.**

Redux – Store's state

El Store es donde se guarda el `state` de la app entera en un objeto.

```
{
  todos: [
   { id: 1, name: 'LearnReact', isComplete: false },
   { id: 2, name: 'Learn Redux', isComplete: true },
   { id: 3, name: 'Learn ReactNative ', isComplete: false },
   { id: 4, name: 'Learn NodeJS', isComplete: false }
  ]
}
```

Un detalle no menor es que el Store es **Read Only**, la forma de modificarlo es emitiendo acciones, llamadas Actions**.**

Redux – Actions

- Las acciones son simples objetos JavaScript.
- *Siempre* tienen un **type**.
- *Puede tener* un **payload**, que puede ser cualquier cosa (opcional).
- Su propósito es describir un evento.
- El evento puede provocar un cambio en el state.

```
const action = {
    type: "Type of the action",
    payload: { data: "Information" }
}
```

```
const addTodo = {
    type: "ADD_TODO",
    payload: "Learn Redux"
}
```

Redux – Action Creators

Normalmente, los mismos tipos de acciones se utilizan en varias partes de una aplicación, por lo que se suele utilizar **Action Creators**, funciones que devuelven una **Action** determinada según los datos que se quieren enviar con la **Action**.

```
const actionCreator = (value) =>
{
   return {
       type: "Type of the action",
       payload: { data: value }
   }
}
const action = actionCreator("information");
```

dispatch es una función de **Redux** para emitir **Actions**, y que lleguen a los **Reducers**

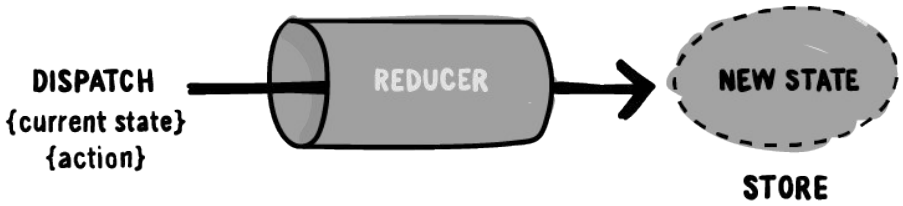
```
const addTodo = (text) => {
   return {
       type: "ADD_TODO",
       payload: text
   };
};
```

```
dispatch(addTodo("Learn React"));
// es equivalente a
dispatch({
      type: "ADD_TODO",
      payload: "Learn React Native"
});
```

Redux – Reducers (1)

Los **Reducers** son funciones JavaScript con la condición de ser funciones **puras.** No modifican los parámetros que reciben y no dependen de nada externo a la función (dado un mismo *input* siempre producen el mismo *output*).

- Reciben el state actual,
  y el action que ocurrió.
- Siempre devuelven un nuevo state




```
function reducer(state, action) {
   // Nos aseguramos que este definido:
   state = state || initialState;
   if(action.type === "type1"){
       // Calcular nuevo estado.
   }
   else if(action.type === "type2"){
       // Calcular nuevo estado.
   }
   return state;
}
```

Redux – Reducers (2)

A continuación se muestra un ejemplo de una aplicación donde simplemente se lleva un contador:

```
function counterReducer(state = 0, action) {
   switch (action.type) {
       case 'INCREMENT':
           return state + 1
       case 'DECREMENT':
           return state - 1
       default:
           return state
   }
}
```

Redux – Reducers (3)

Normalmente el **state** del **Store** será un objeto con varias partes, por lo que el **Reducer** raíz puede ser una composición de distintas funciones para cada parte del **state**.

```
// Estructura del store, de una App de todos
state = {
   todos: [
       { text: 'Learn React', completed: false },
       { text: 'Learn Redux', completed: true },
       { text: 'Learn NodeJS', completed: false }
   ],
   visibilityFilter: 'SHOW_ALL'
}
```

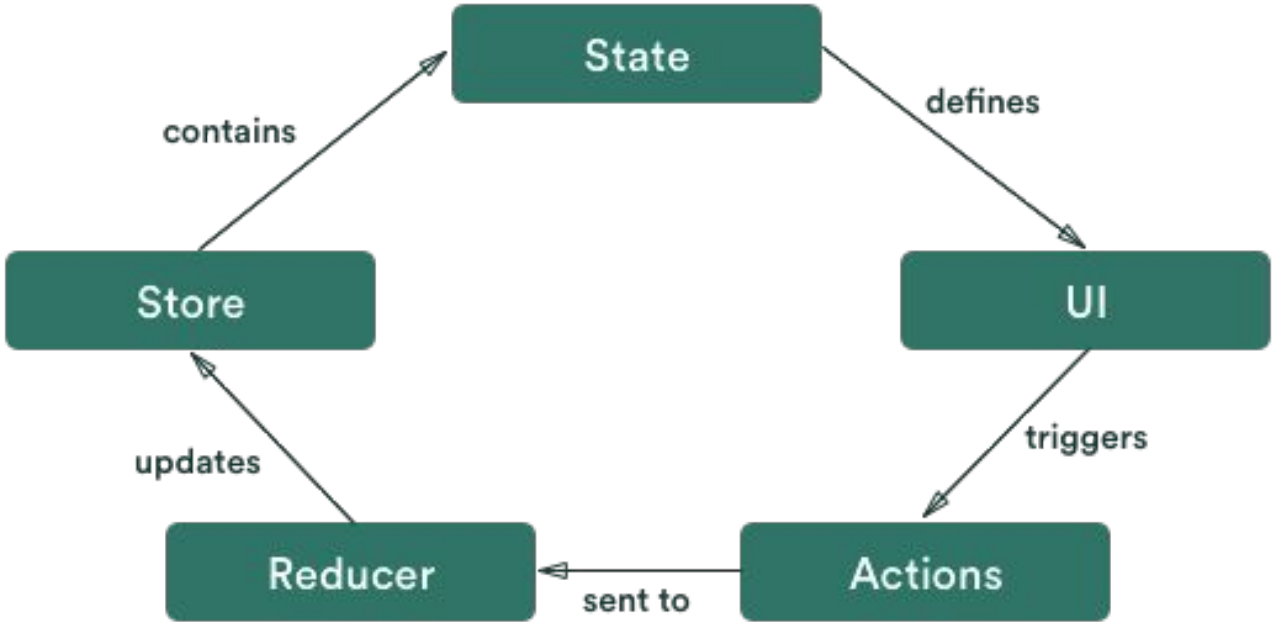
```
// Reducer raiz
function todoApp(state = {}, action) {
   return {
       todos: todosReducer(state.todos, action),
       visibilityFilter: visibilityFilterReducer(state.visibilityFilter, action)
   };
}
```

Redux – Reducers (4)

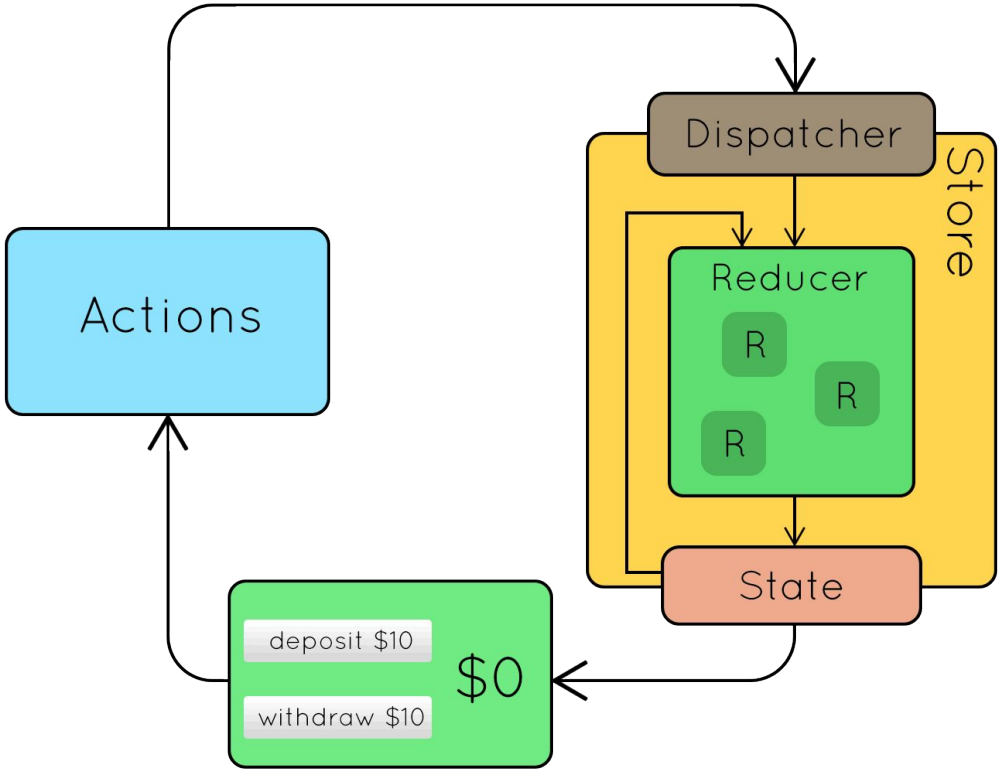
```
function visibilityFilterReducer(state = 'SHOW_ALL', action) {
    if (action.type === 'SET_VISIBILITY_FILTER') {
        return action.filter;
    } else {
        return state;
    }
}
function todosReducer(state = [], action) {
    switch (action.type) {
        case 'ADD_TODO':
            return state.concat([{ text: action.text, completed: false }]);
        case 'TOGGLE_TODO':
            return state.map((todo, index) =>
                        action.index === index ?
                          { text: todo.text, completed: !todo.completed } : todo
            )
        default:
            return state;
    }
}
```

Los **Reducer** tienen que ser funciones puras, por ende no pueden mutar el state que reciben. Los métodos `concat` y `map` respetan esto ya que devuelven un nuevo array.

Redux Flow


State
defines
UI
triggers
Actions
sent to
Reducer
updates
Store
contains

Redux Flow


Actions
Dispatcher
Store
Reducer
R
R
R
State
deposit $10
withdraw $10
$0

Redux – API

La librería brinda la función `createStore` para crear el Store (recibe la función Reducer). El Store es quien permite **escuchar** (`subscribe`) y **emitir** (`dispatch`) los cambios.

```
import { createStore } from 'redux'
import { addTodo } from './actions'
import mainReducer from './reducers'

let store = createStore(mainReducer)
store.subscribe(() =>
      console.log(store.getState())
)
store.dispatch(addTodo('Learn about actions'))
```

React-Redux

React-redux es una librería que provee bindings para facilitar el uso de Redux en React.

- `Provider` => “inyecta” el store en React: Lo hace “disponible” a todo el arbol de componentes.
- `connect` => es una función que permite “conectar” cualquier componente en la jerarquía al Provider.

```
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import reducer from './reducer'

let store = createStore(reducer)
ReactDOM.render(
    <Provider store={store}>
        <App />
    </Provider>,
    document.getElementById('root')
);
```

React-Redux

`connect` es una función que recibe 4 parámetros, todos opcionales, para aplicarle a los componentes que se quieran conectar al Store de Redux.

```
connect(
   [mapStateToProps],
   [mapDispatchToProps],
   [mergeProps],
   [options]
 )
```

```
const ConnectedBoard = connect(...)(Board);
```

```
export default connect(...)(Board);
```

`mapStateToProps` se encarga de definir qué parte del state del state queremos “leer” en el componente y `mapDispatchToProps` para definir qué acciones se podrán "dispatchear" desde el componente.

React-Redux

`mapStateToProps` es una función que recibe el *state* del Store. Las propiedades del objeto que retorna a partir del *state* llegarán al componente que se conecte como `props` (`TodoList` en el ejemplo). Algo similar sucede con `mapDispatchToProps`, pero aquí se definen las funciones que se quiere que el componente que se conecte reciba por props, de forma de poder "dispatchear" acciones a los Reducers.

```
const mapStateToProps = (state, [ownProps]) => ({
    todos: state.todos
})
const mapDispatchToProps = (dispatch, [ownProps]) => ({
    addTodo: text => dispatch(addTodo(text))
})
export default connect(
    mapStateToProps,
    mapDispatchToProps
)(TodoList);
```

Redux - Beneficios

- Comportamiento más predecible, una única forma de alterar el estado.
- Reproducir (o deshacer) cambios de estado.
- "Rehidratar" estados desde una representación serializada.
- Renderizar el mismo estado desde el servidor en el primera request.
- Undo/Redo triviales.
- Múltiples UI reutilizando lógica del negocio.
- Tooling avanzado.
    - Instalar Redux Dev Tools Extension
    - Para usarlo, hay que agregar un parámetro en el `createStore`.

1.1 Basic store

For a basic Redux store simply add:

```
  const store = createStore(
    reducer, /* preloadedState, */
+   window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()
  );
```

Ejercicios

Ejercicio

1. Crear proyecto de React
2. Crear un contador, el cual:
    a. Muestra el valor del contador
    b. Tiene un botón que incrementa el valor contador
    c. Tiene un botón que decrementa el valor del contador
    d. Reinicia el contador a 0

Para ello:

1. Crear una Store, la cual es “provista” a toda la aplicación haciendo uso del `Provider`
2. Crear un componente `Count` el cual estará suscrito al valor del contador, el cual vive en el estado que vive en la Store
3. Crear los botones correspondientes, los cuales “dispatchearán” las acciones necesarias.
4. Experimentar aplicando los diferentes conceptos aprendidos en la clase, de nada sirve si hacemos todo con `setState` (que podríamos), justamente la idea es **usar** Redux.
5. Recordar! Copiar código de las slides no te va a ayudar a aprender.
6. Una buena práctica es tener action creators, reducers, creación de store y demás; en archivos independientes.