Xseed

Curso de React _ Technisys

Xseed

Clase 07 -

Redux

by Diego Cáceres

Repaso

React Router v4 (2)

Se van a utilizar algunos de los componentes que brinda la librería para construir el ruteo de la App. Por ejemplo, `BrowserRouter` se utiliza para envolver toda la App y `Route` para declarar **qué rutas** renderizan **qué componentes**.

```
const BasicExample = () => {
   return (
       <BrowserRouter>
           <div>
               <Route exact path="/" component={Home} />
               <Route path="/about" component={About} />
           </div>
       </BrowserRouter>
   );
}
```

React Router v4 (3)

El componente `Link` sirve para declarar navegación a rutas determinadas. Estos componentes hay que importarlos de la librería antes de poder utilizarlos.

```
import { BrowserRouter, Route, Link } from "react-router-dom";
const BasicExample = () =>
   <BrowserRouter>
       <div>
           <ul>
               <li><Link to="/">Home</Link></li>
               <li><Link to="/about">About</Link></li>
           </ul>
           <hr />
           <Route exact path="/" component={Home} />
           <Route path="/about" component={About} />
       </div>
   </BrowserRouter>;
```

`exact` se utiliza para que no "matchee" `Home` en `About`

React Router v4 (17)

En el caso de tener rutas **ambiguas**, en la que dos pueden matchear, se puede utilizar el `Switch` para envolver las rutas y va a devolver la primera que coincida.

```
const AmbiguousExample = () => <Router>
  <div>
    <ul>
      <li><Link to="/about">About Us (static)</Link></li>
      <li><Link to="/company">Company (static)</Link></li>
      <li><Link to="/kim">Kim (dynamic)</Link></li>
    </ul>
    <Switch>
      <Route path="/about" component={About} />
      <Route path="/company" component={Company} />
      <Route path="/:user" component={User} />
    </Switch>
  </div>
</Router>;
```

En este caso, cuando se navegue a `/about` o `/company` la tercer ruta también "matchea", pero el `Switch` devuelve la primera que coincida.

Xseed

REPASO

High Order Components (1)

Un High-Order Component es una técnica avanzada utilizada para reutilizar la lógica de un componente. No es algo específico de la API de React, sino que es un patrón que surge naturalmente de la posibilidad de composición que hay en React.

Un High-Order Component es una **función** que **acepta un componente** y **retorna un nuevo componente**.

```
const higherOrderComponent = (component) => {
    return componentOnSteroids;
}
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

High Order Components (2)

*Un componente básicamente transforma `Props` en UI, mientras que un **High-Order Component** transforma un Componente en otro Componente, agregando nuevas características.*

High Order Components (3)

Normalmente los High-Order Components son utilizados por las librerías de terceros, como se va a ver en Redux. Pero también se pueden crear HOCs propios. Un ejemplo de esto sería, si en determinado momento quisiera envolver todo un componente con un `div` especial para visualmente identificarlo en pantalla.

```
let DebugComponent = ComponentToDebug => {
    return class extends Component {
        render() {
          return (
            <div className="debug">
              <ComponentToDebug {...this.props}/>
            </div>
          );
        }
    };
}
```

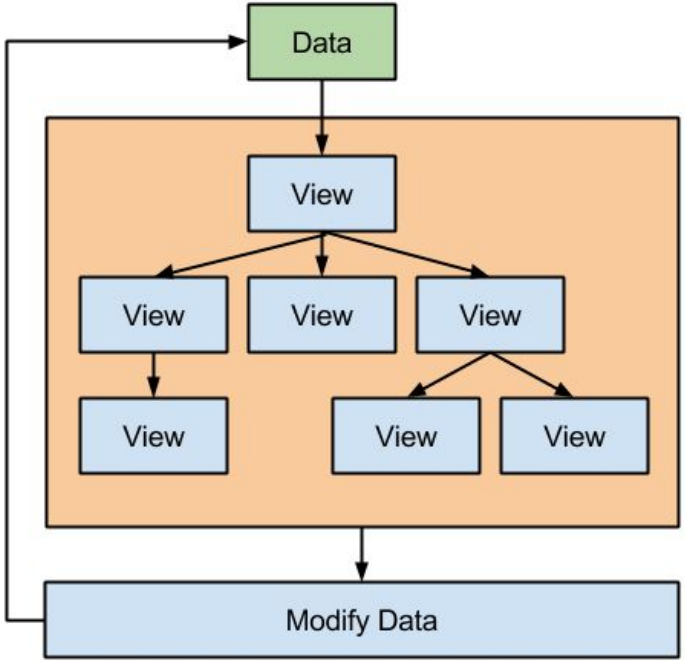
```
<Switch>
    <Route path="/" exact component={Home} />
    <Redirect from="/old-match" to="/will-match" />
    <Route path="/will-match" component={WillMatch} />
    <Route component={DebugComponent(NoMatch)} />
</Switch>
```

Xseed

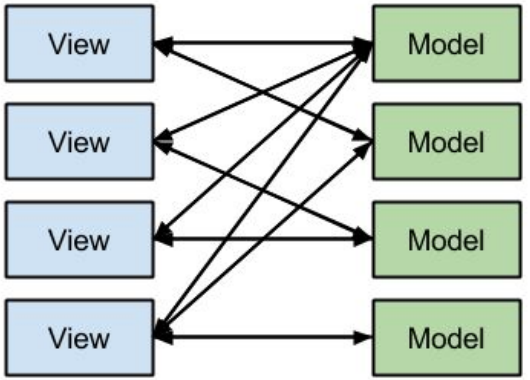
Redux

Introducción

React vs MV*



React



MV* Data Binding

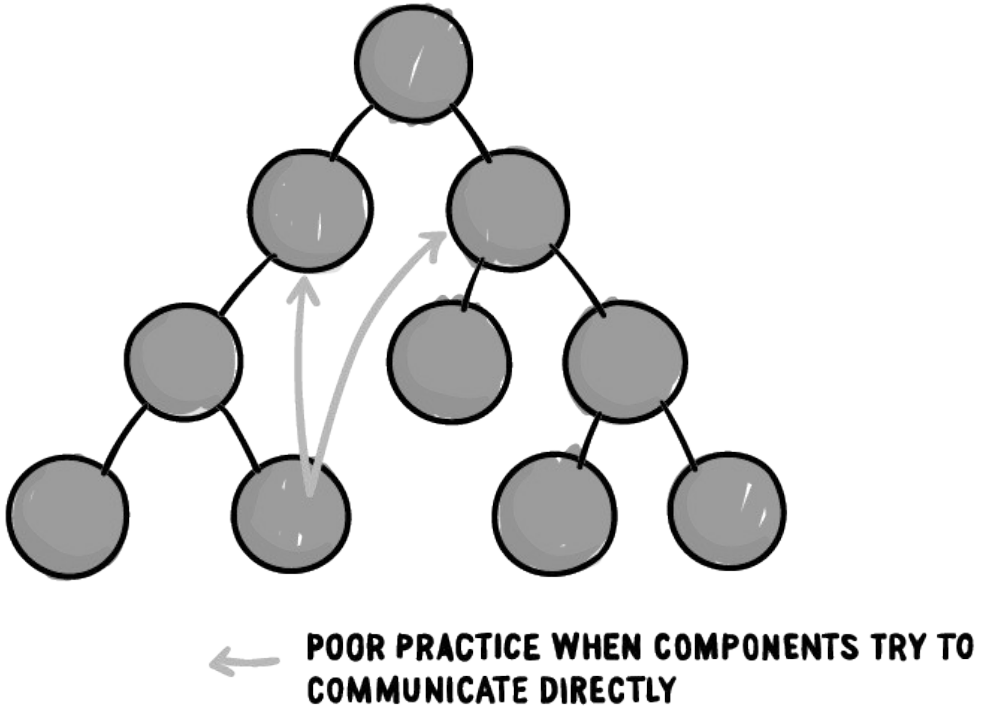
React – Problema con el flujo de datos

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

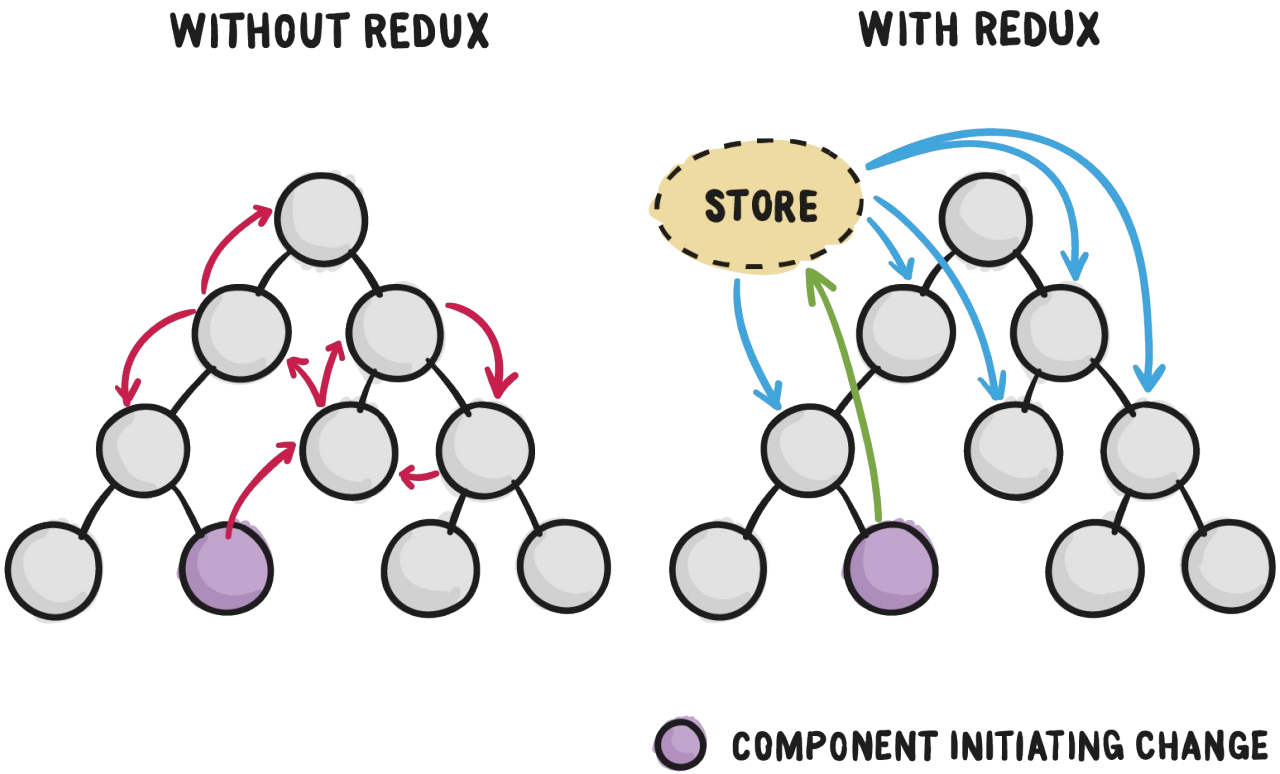


React – Problema con el flujo de datos

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 útil.

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 árbol de objetos.

```
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.
- Tienen un **type** (obligatorio).
- Tienen un **payload**, que puede ser cualquier cosa (opcional).
- Su propósito es describir un evento.
- El evento puede provocar un cambio en el state -> Mediante un **Reducer.**

```
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**

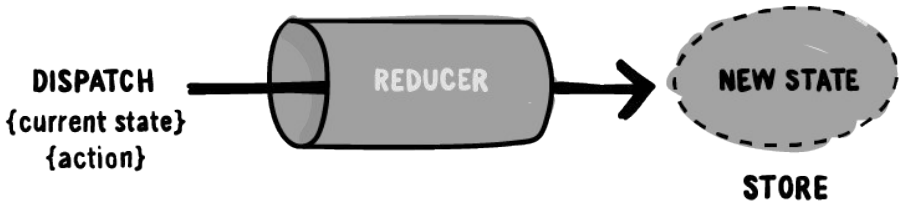
```
const addTodo = (text) => {
   return {
       type: "ADD_TODO",
       payload: text
   };
};
```

```
dispatch(addTodo("Learn React"));
dispatch({ type: “ADD_TODO”, payload: “Tarea 2”});
```

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 counter(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'
}
```

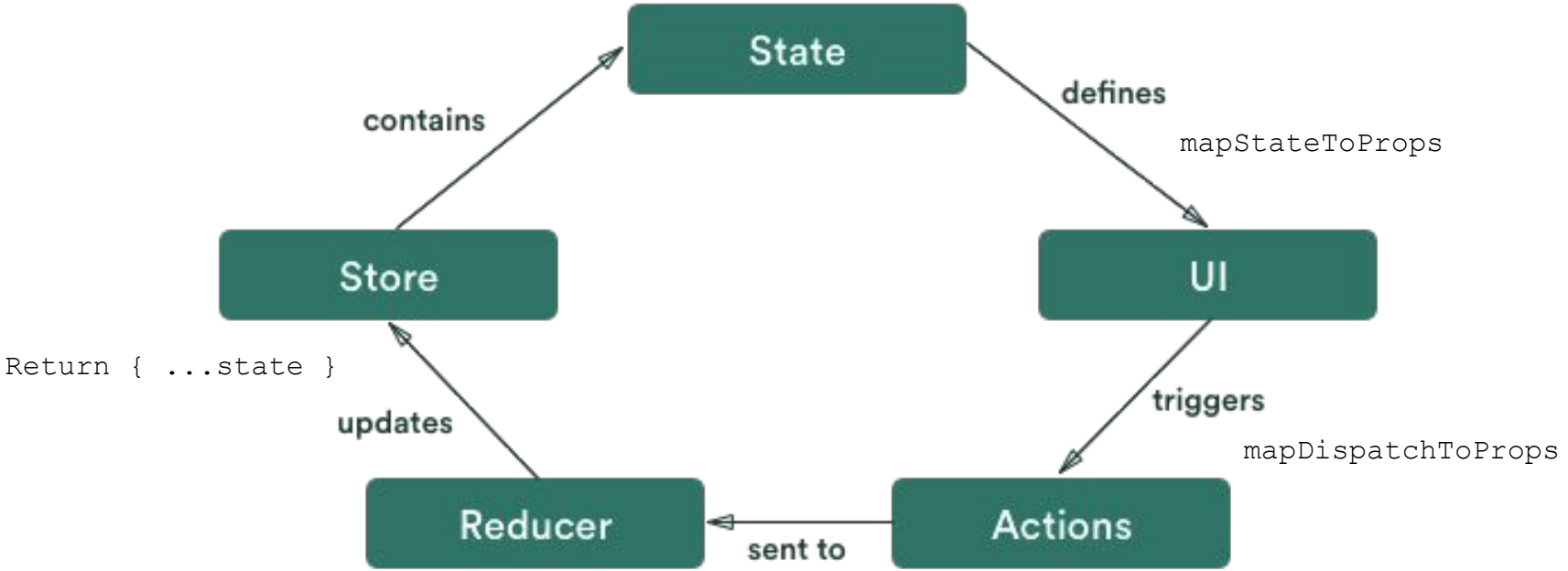
```
// Reducer raiz
function todoApp(state = {}, action) {
   return {
       todos: todos(state.todos, action),
       visibilityFilter: visibilityFilter(state.visibilityFilter, action)
   };
}
```

Redux – Reducers (4)

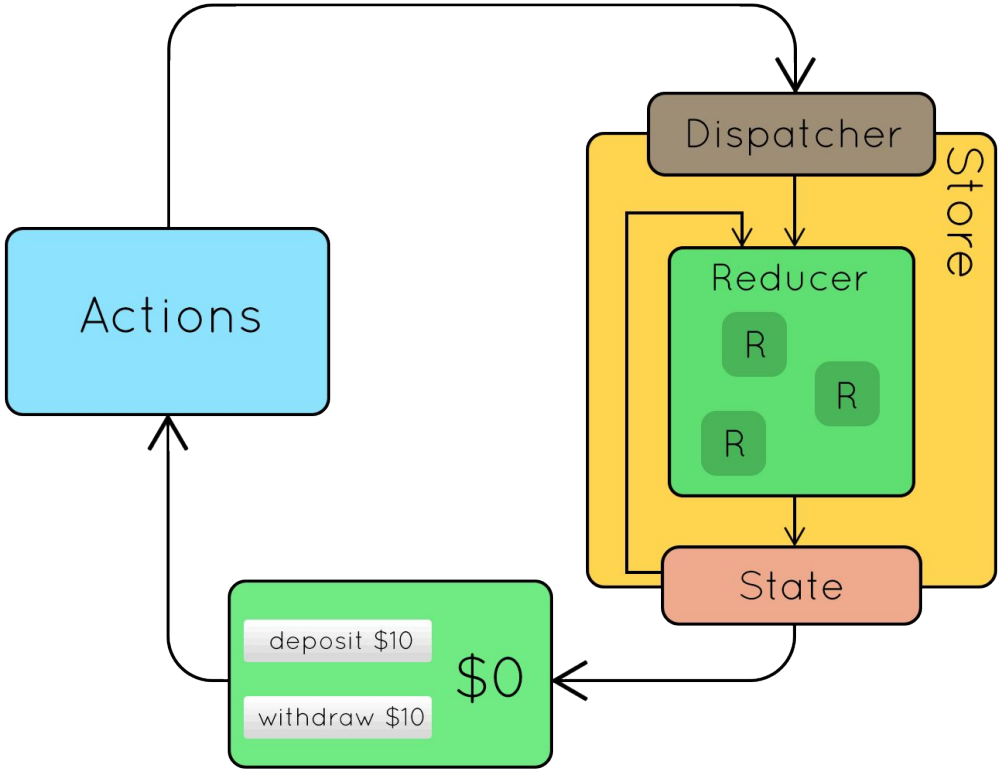
```
function visibilityFilter(state = 'SHOW_ALL', action) {
    if (action.type === 'SET_VISIBILITY_FILTER') {
        return action.filter;
    } else {
        return state;
    }
}
function todos(state = [], action) {
    switch (action.type) {
        case 'ADD_TODO':
            return state.concat([{ 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
contains
defines
mapStateToProps
Store
UI
Return { ...state }
updates
triggers
mapDispatchToProps
Reducer
sent to
Actions

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 tiene un método para subscribirse a los cambios. También la función `dispatch` que es la que envía las **Actions** al **Reducer**.

```
import { createStore, dispatch } from 'redux'
import { addTodo } from './actions'
import todoApp from './reducers'

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

React-Redux

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

- **`Provider`** => inyecta el store en React.
- **`connect`** => funcion que retorna un un HOC que permite conectar cualquier componente en la jerarquía con el Store de Redux (siempre y cuando esté dentro de Provider).

```
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 y retorna un HOC para aplicarle a los componentes que se quieran conectar al Store de Redux.

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

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

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

Normalmente, se utiliza sólo **`mapStateToProps`** para definir qué parte del state del Store se necesitará 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 mutar datos.
- 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

Ejercicios

Descargar el zip `EjerciciosClase08`. Cada carpeta dentro corresponde a un ejercicio y es un proyecto creado con Create React App, por lo que para ejecutarlo es necesario:

- Si usan yarn:
  - `$ yarn install`
  - `$ yarn start`
- Si usan npm:
  - `$ npm install`
  - `$ npm start`

Se debería abrir automáticamente el explorador con el ejercicio corriendo en http://localhost:3000/

Ejercicio 1 – Primera parte

- Este ejercicio es una aplicación con una lista de ToDos (tareas pendientes). Ya tiene creados componentes y el comportamiento para poder seguir agregando mas ToDos.
- Implementar la funcionalidad `ToogleTodo`, para que cuando se haga *tick* a un ToDo se actualice en el `state`.
    - Implementar el método que actualice el state en `App.js` y luego pasarlo por `props` hasta el `onChange` del input de tipo checkbox en `TodoItem.js`.
- Implementar de forma similar la funcionalidad `RemoveTodo`.

Xseed

Ejercicio 1 – Segunda Parte

Partiendo de la solución de la parte 1, implementar en esta aplicación el manejo del estado con Redux.

- Instalar las librerías `redux` y `react-redux`.
- Crear un nuevo archivo en `/src` llamado `reducer.js` con nuestro `reducer`.

```
const initialState = { todos: [] };
const reducer = (state = initialState, action) => {
    return state;
};
export default reducer;
```

- Modificar `index.js`, importar lo necesario de `redux` y `react-redux`. Importar también el reducer para crear el Store, envolver `App` en `Provider` y darle el Store al componente `Provider`.

Xseed

Ejercicio 1 (2) – Segunda Parte

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import App from "./App";
import registerServiceWorker from "./registerServiceWorker";

import { createStore } from "redux";
import { Provider } from "react-redux";
import reducer from "./reducer";
let store = createStore(
    reducer,
    window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()
); // Esto es para habilitar Redux Dev Tools.

ReactDOM.render( <Provider store={store}>
        <App />
    </Provider>,
    document.getElementById("root")
);
registerServiceWorker();
```

Xseed

Ejercicio 1 (3) – Segunda Parte

Ahora mover lo que está en el state inicial de `App.js` al `initialState` del reducer.

Modificar `TodoList` para que se conecte con Redux, y lea los todos del Store.

- Se debe importar `connect` de `react-redux`.
- Definir `mapStateToProps` y `mapDispatchToProps` para pasarle al `connect`.
- Exportar una versión conectada de `TodoList`.

```
const mapStateToProps = state => {
    return {
        todos: state.todos
    };
};
const mapDispatchToProps = dispatch => {
    return {};
};
export default connect(mapStateToProps, mapDispatchToProps)(TodoList);
```

Ejercicio 1 (4) – Segunda Parte

Desde `App.js` no es necesario pasar por *`props`* todos a `TodoList`.

Crear un nuevo archivo en `src` llamado `actions.js` para definir los action creators.

- Es buena práctica definir los types en un objeto para no tener errores de tipeo.
- Crear el action creator para `addTodo`.

```
export const types = {
    ADD_TODO: 'ADD_TODO',
    TOOGLE_TODO: 'TOOGLE_TODO',
    REMOVE_TODO: 'REMOVE_TODO',
}
export const addTodo = (text) => {
    return {
        type: types.ADD_TODO,
        payload: text
    };
};
```

Xseed

Ejercicio 1 (5) – Segunda Parte

En `Reducer.js` implementar el caso para la acción de type `ADD_TODO` (importar `generateId` y `types` de donde corresponde).

```
const reducer = (state = initialState, action) => {
    if (action.type === types.ADD_TODO) {
        const newId = generateId();
        const newTodo = { id: newId, name: action.payload, isComplete: false };
        const updatedTodos = [...state.todos, newTodo];
        return {
            ...state,
            todos: updatedTodos
        };
    }
    return state;
};
```

Xseed

Ejercicio 1 (6) – Segunda Parte

Ahora se debe conectar `TodoForm` a Redux de forma que cuando se haga submit del formulario, no se llame a una función que se recibe por *props*, sino que se haga dispatch de la acción correspondiente.

```
const mapStateToProps = state => {
    return {};
};
const mapDispatchToProps = dispatch => {
    return {
        addTodoLocal: text => {
            return dispatch(addTodo(text));
        }
    };
};
export default connect(mapStateToProps, mapDispatchToProps)(TodoForm);
```

Ejercicio 1 (7) – Segunda Parte

- Limpiar `App.js` para que no pase más la función `handleSubmit` ni el array de todos.
- Ahora es necesario implementar las otras dos acciones correctamente: `ToogleTodo` y `RemoveTodo`.
  - Primero crear los action creators.
  - Implementar el reducer.
  - Conectar los componentes correspondientes.

Ejercicio 2

Este ejercicio es la solución de un ejercicio de una clase anterior. Hay que agregar Redux como manejador del state en esta App, y que todo siga funcionando.

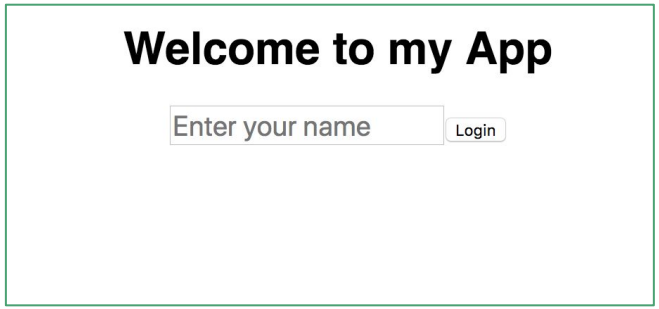
Sólo se quiere manejar en el state de Redux lo relacionado al Login.

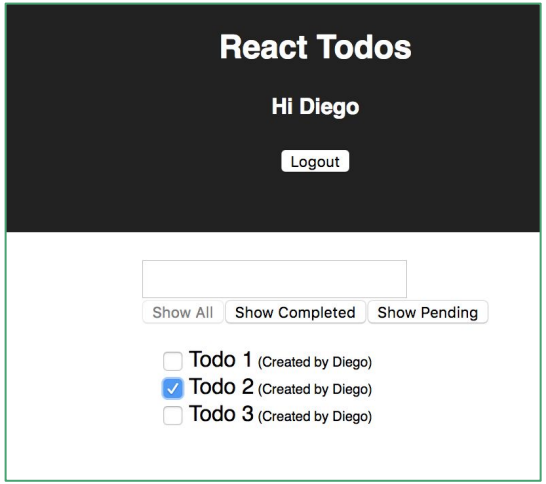
Ejercicio 3

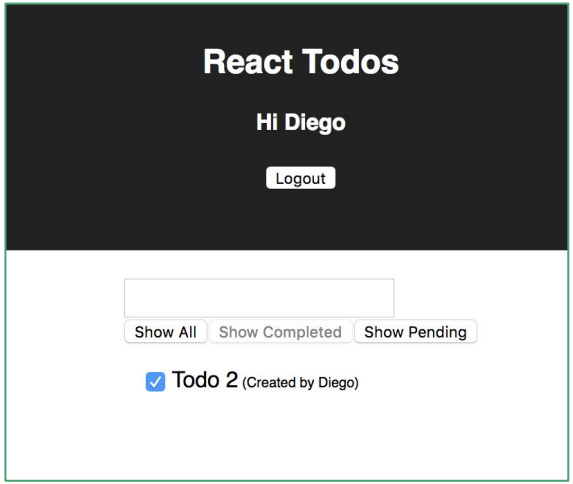
- Este ejercicio parte de la solución del ejercicio de Todos.
- Deben agregar un Login, que pida el nombre de usuario y esto se guarde en el Store de Redux. Luego de hacer Login, se ve la lista de Todos. Esto se puede implementar dentro de App.js, no es necesario crear otro componente.
- También debe existir la posibilidad de hacer Logout.
- Cuando se crea un ToDo, se debe guardar el ‘owner’ que lo creó, es decir, el usuario Logueado. Mostrar esta info en TodoItem también.
- Al hacer Logout, no se borran los ToDos, sino que puedo hacer Login con otro usuario y verlos.
- Agregar tambien el manejo del filtro de visibilidad, puedo filtrar entre:
  - Todos
  - Completados
  - Pendientes

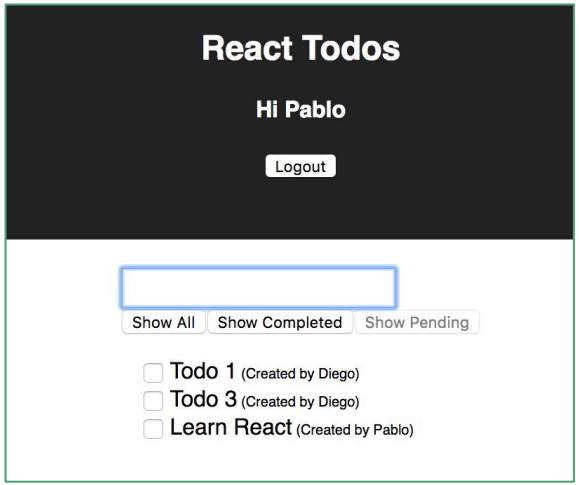
Xseed

Ejercicio 3









Ejercicio 3 – Segunda Parte

Vamos a utilizar el patrón Presentational - Container, y partir el reducer en distintos archivos.

- Los componentes `TodoForm`, `TodoItem`, y `TodoList` se deben partir en dos archivos, un container con la lógica de Redux, y uno de presentación que solo renderiza `JSX`.
- Crear 3 reducers, uno para `Auth`, otro para `Todos`, y otro para `VisibilityFilter`.
    - Utilizar la función `combineReducers` de Redux para crear el reducer raiz: https://redux.js.org/api-reference/combinereducers