



Universidad Politécnica de Madrid

React

Redux

Enrique Barra Arias
Álvaro Alonso González

- Redux se define como un "Contenedor de estado predecible para aplicaciones JavaScript".
- Redux es una librería muy pequeña (2KB aprox.). La API es de apenas 5 funciones (aunque tiene muchas extensiones)
- Está escrita en JavaScript vanilla (estándar), por lo que es agnóstica al framework empleado, por lo que puede ser utilizada con cualquier librería o framework como Angular, Polymer, React, etc.
- Ayuda a escribir aplicaciones que se comportan de manera consistente y son fáciles de probar
- Se encarga de desacoplar el estado global de la aplicación de la parte visual (es decir, de los componentes)

Fuente única de verdad

El estado es de sólo lectura

Los cambios se hacen con funciones puras

- El estado de toda la aplicación se almacena en un objeto dentro de un único store
- Facilita la implementación de algunas funcionalidades como deshacer/rehacer que suelen ser muy difíciles de implementar
- Facilita la comunicación con el servidor y el almacenamiento del estado



Redux

- La única manera de cambiar el estado es emitir una acción, un objeto que describa lo que ha pasado
- Todo es predecible
- No hay condiciones de carrera a tener en cuenta



Redux

- Para especificar cómo el estado se transforma a partir de las acciones, se escriben reducers
- Los reducers son sólo funciones puras que toman el estado anterior y una acción, y devuelven el siguiente estado
- Los reducers devuelven los objetos de nuevo estado, en lugar de mutar el estado anterior
- Hay que recordar que un "reducer" es una función que toma la salida anterior y el siguiente valor y calcula la siguiente salida



Redux

- Operan sólo usando los argumentos que reciben y ningún otro elemento fuera de ellas (funciones, variables)
- De manera más formal:
 - Dados los mismos valores de los argumentos, una función pura devolverá siempre el mismo resultado
 - Una función pura no tiene ningún efecto secundario

```
function pureFoo ( a, b ) {  
    return a + b;  
}  
  
console.info( pureFoo( 2, 4 ) ); // 6  
console.info( pureFoo( 3, 6 ) ); // 9  
console.info( pureFoo( 2, 4 ) ); // 6
```

- Más info: https://en.wikipedia.org/wiki/Pure_function

¿Cuándo debería usar Redux?

- Tienes grandes cantidades de estado de aplicación que se necesitan en muchos lugares de la app
 - Mantener el estado en el componente superior lo hace “engordar” demasiado, se crea un gran componente con mucha lógica
- El estado de la aplicación se actualiza con frecuencia
- La lógica para actualizar ese estado es compleja
- La aplicación tiene una base de código de tamaño medio o grande, y es posible que trabajen en ella muchas personas
- Redux se puede considerar una alternativa (más potente) al Context:
<https://changelog.com/posts/when-and-when-not-to-reach-for-redux>

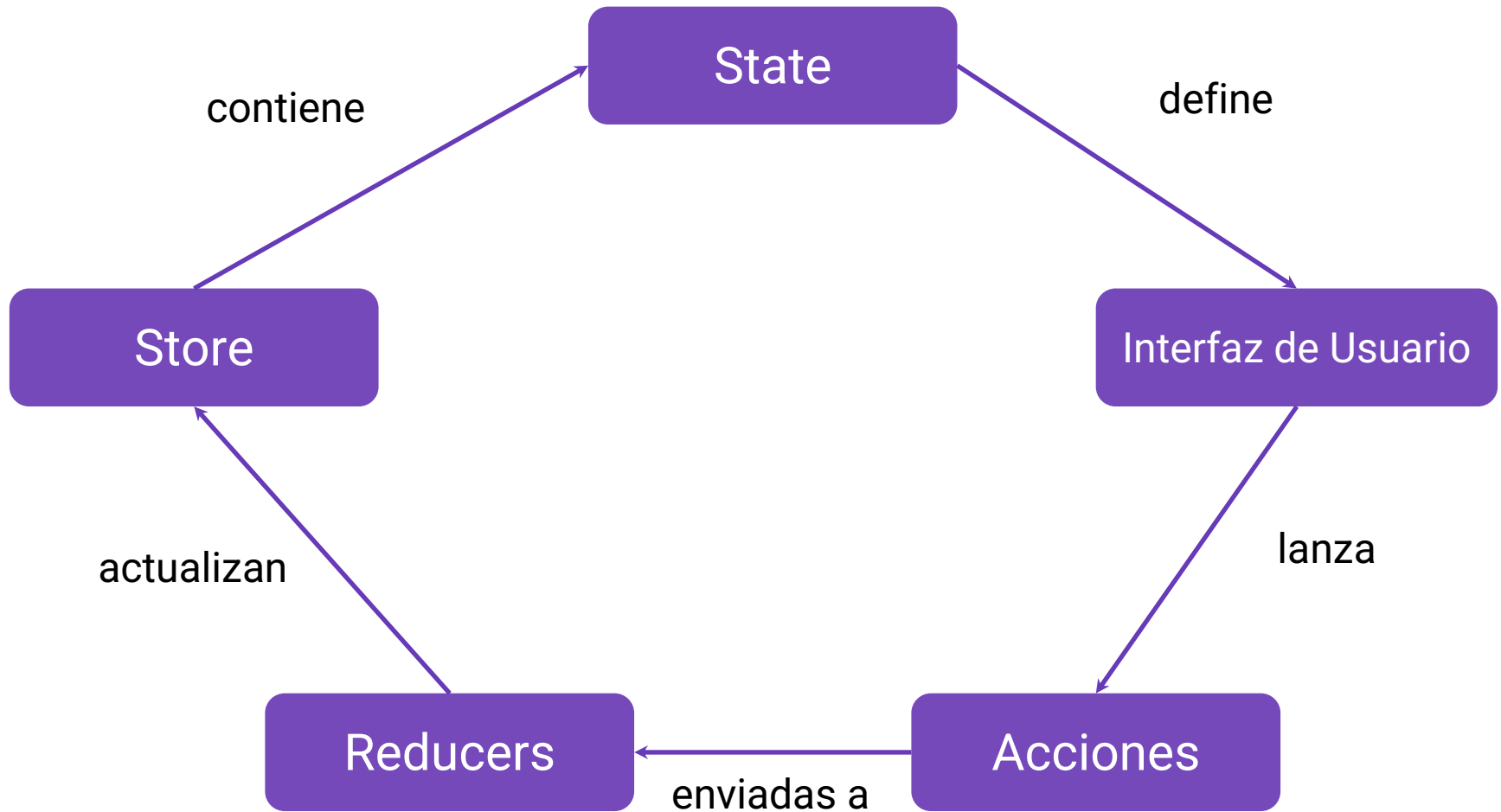


Universidad Politécnica de Madrid

React

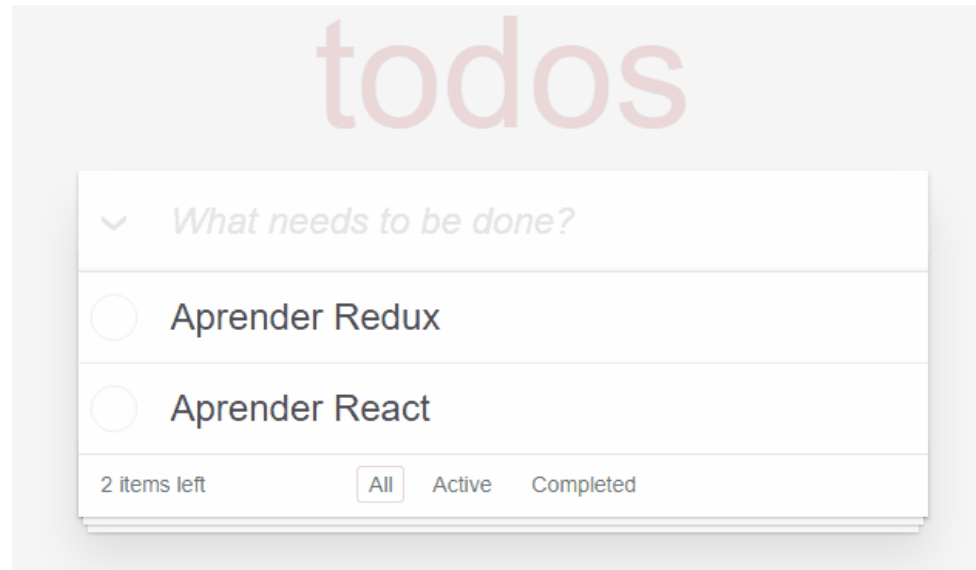
Redux – ciclo de vida

Enrique Barra Arias
Álvaro Alonso González



- El estado de una aplicación es un objeto JS (inmutable, no se modifica directamente)
- Ejemplo: App TO-DO

```
{
  todos: [
    { text: 'Aprender React',
      completed: true },
    { text: 'Aprender Redux',
      completed: false }
  ],
  visibilityFilter: 'SHOW_ALL'
}
```



- El estado sólo se puede modificar lanzando acciones
- Las acciones son objetos JavaScript que describen un cambio a realizar en el estado
- Las acciones tienen un atributo **type** (obligatorio) y el resto de los atributos (**payload**) son opcionales y dependen del propósito de la acción
- Ejemplo:

```
{ type: 'ADD_TODO', payload: 'Go to swimming pool' }  
{ type: 'TOGGLE_TODO', payload : 1 }  
{ type: 'SET_VISIBILITY_FILTER', payload : 'SHOW_ACTIVE' }
```

- Se suelen crear mediante funciones llamadas “action creators”

```
export function addTodo(text) {  
  return { type: 'ADD_TODO', payload : text };  
}
```

- Funciones puras que aplican las acciones al estado
- Toman el estado anterior y una acción y devuelven el estado siguiente
 - No modifican el estado anterior (es inmutable) crean una copia y le hacen cambios y devuelven esa copia
- Cosas que no podemos hacer dentro de un reducer
 - Modificar sus argumentos
 - Hacer tareas con efectos laterales como llamadas a APIs o cambios de ruta
 - Llamar a funciones no puras, como `Date.now()` o `Math.random()`
- Se suelen dividir la lógica de la aplicación en varios reducers
 - Cada reducer se encarga de una parte concreta del estado => así es más fácil manejar apps con mucha información
 - De esta manera cada reducer toma como argumento el trozo de estado que le corresponde y la acción a aplicar y devuelve la parte equivalente al nuevo estado

Estructura de un reducer

state es la parte del estado que le corresponde a este reducer, NO el estado global

action contiene toda la información necesaria para modificar el estado, específicamente el tipo de acción y el resto de argumentos (payload)

```
// import {...}
```

```
function myReducer(state = "DEFAULT_STATE", action) {  
  switch (action.type) {  
    case 'ACTION_NAME':  
      let newState = Object.assign([], state);  
      // ... Modify newState  
      return newState; // Return the modified state  
    default:  
      return state;  
  }  
}  
export default myReducer;
```

devuelve el nuevo estado (sólo el trozo correspondiente a este reducer) tras aplicar los cambios necesarios o el estado anterior si nada ha pasado

Ejemplo de Reducer


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

Sólo afecta a la parte del estado correspondiente a "todos" (to-do's)


En el caso por defecto se devuelve el trozo de estado original

Ejemplo de Reducer

```
function visibilityReducer(state = 'ALL', action) {  
  switch (action.type) {  
    case 'SET_VISIBILITY_FILTER':  
      return action.payload;  
    default: return state  
  }  
}
```

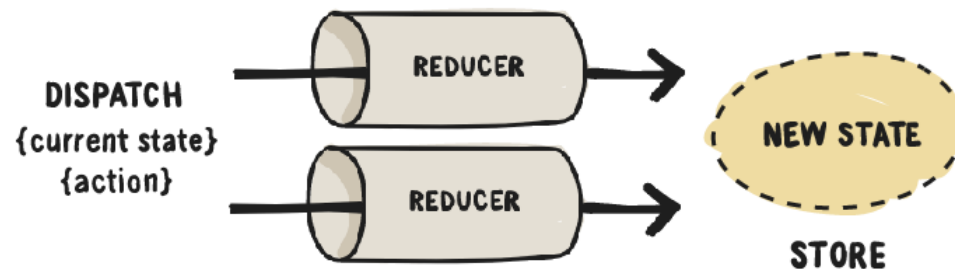


Sólo afecta a la parte del estado correspondiente a "visibilityFilter"



En el caso por defecto se devuelve el trozo de estado original

- El store es el objeto que une state y reducers
- Tiene las siguientes responsabilidades
 - Contiene el **estado completo** de la aplicación
 - Permite el acceso al estado vía **getState()**
 - Permite actualizar el estado vía **dispatch(action)**
 - Registra escuchadores mediante **subscribe(listener)**
 - ...
- Es importante darse cuenta de que sólo hay un store en una aplicación Redux. Cuando queremos dividir la lógica de la aplicación para gestionar los datos utilizamos varios reducers en lugar de varios stores



➤ createStore()

- Esta función crea el store central donde se almacena el estado global
- La función recibe como argumento un reducer y opcionalmente un estado inicial y un *enhancer* que se utiliza para añadir intermediarios. Devuelve el store creado.

```
const store = redux.createStore(reducer, [initialState], [enhancer])
```

➤ Métodos del store:

- **store.getState():** Devuelve el estado actual del store
- **store.dispatch(action):** Emite una acción, que es la única forma de modificar el estado
- **store.subscribe(listener):** Permite suscribirse a cambios que pueden ocurrir en el estado. Se llama al listener cada vez que se emite una acción y el estado puede cambiar.

➤ **combineReducers()**

- En Redux sólo tenemos un store para manejar el estado global.
- Es buena práctica tener varios reducers (uno para cada parte/trozo de estado)
- Con esta función podemos combinarlos en un único reducer que se pasará como argumento a la función createStore

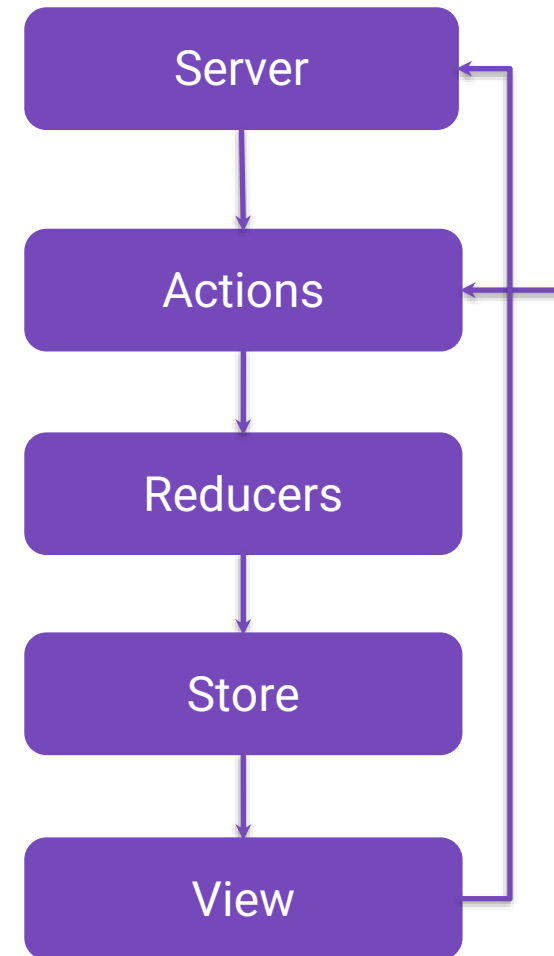
- **¿Por qué debemos tener varios reducers?** Porque de esta manera podemos dividir nuestros problemas en varias partes y es más simple modularizar la aplicación

➤ Hay más métodos que no usaremos en esta asignatura:

- **applyMiddleware()**
- **bindActionCreators()**
- **compose()**

➤ Más info: <https://redux.js.org/api/api-reference>

1. Alguien llama a **store.dispatch(action)**
 - Desde cualquier parte de la app, un componente, una llamada AJAX, un timeout, ...
2. El store de Redux llama al **reducer** pasándole la acción y el estado actual
 - Este reducer puede ser uno sólo o la combinación de varios tras llamar a `combineReducers()`
3. Los reducers se ejecutan generando un **nuevo estado** (cada uno la parte que le corresponde)
4. El **store** guarda el nuevo estado y notifica a cada escuchador registrado con `store.subscribe(listener)`



➤ Redux tutorials

➤ <https://redux.js.org/introduction/getting-started>

➤ Redux Core Concept

➤ <http://redux.js.org/docs/introduction/CoreConcepts.html>

➤ Redux Style Guide

➤ <https://redux.js.org/style-guide>

➤ Getting started with Redux (by Dan Abramov)

➤ <https://egghead.io/courses/getting-started-with-redux>

- Ejemplo de un contador:

- https://github.com/REACT-UPM/ejemplos_redux/blob/main/counter.html

- Ejemplo de una webapp con MVC (modelo-vista-controlador) en html/js básico:

- https://github.com/REACT-UPM/ejemplos_redux/blob/main/mvcquiz.html

- Ejemplo webapp anterior añadiéndole Redux (sin React):

- https://github.com/REACT-UPM/ejemplos_redux/blob/main/mvcquiz_con_redux.html



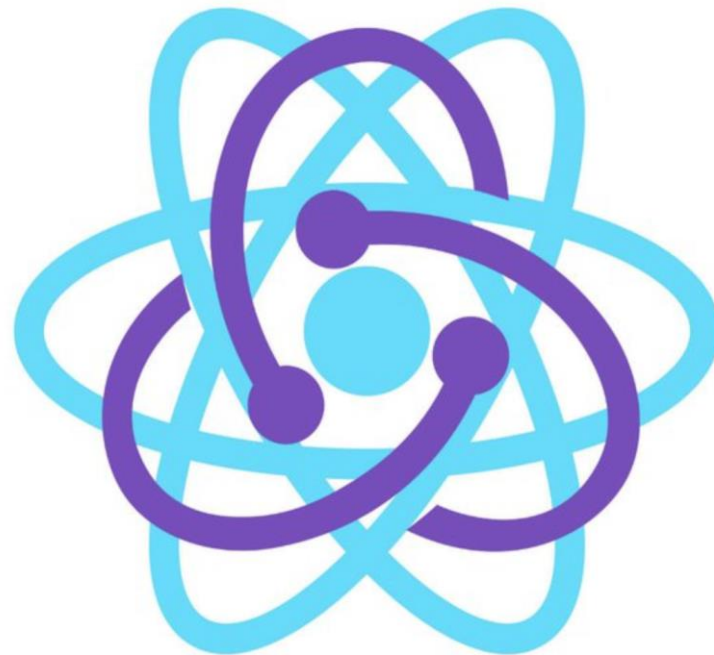
Universidad Politécnica de Madrid

React

React y Redux

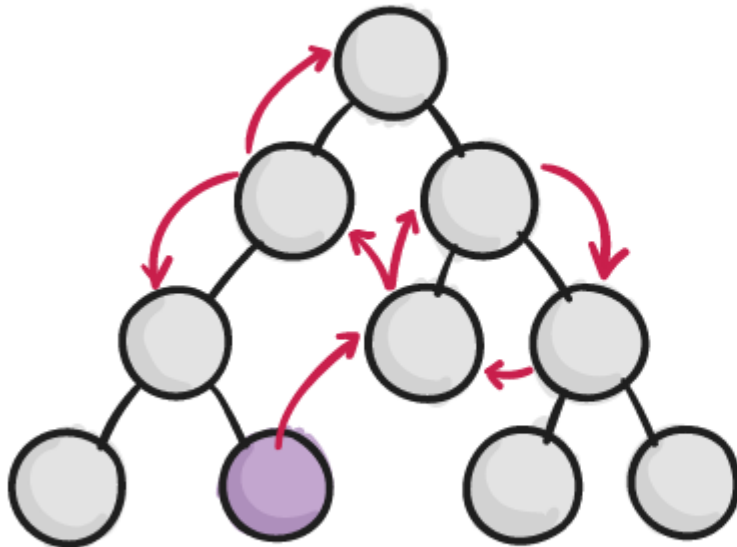
Enrique Barra Arias
Álvaro Alonso González

React + Redux

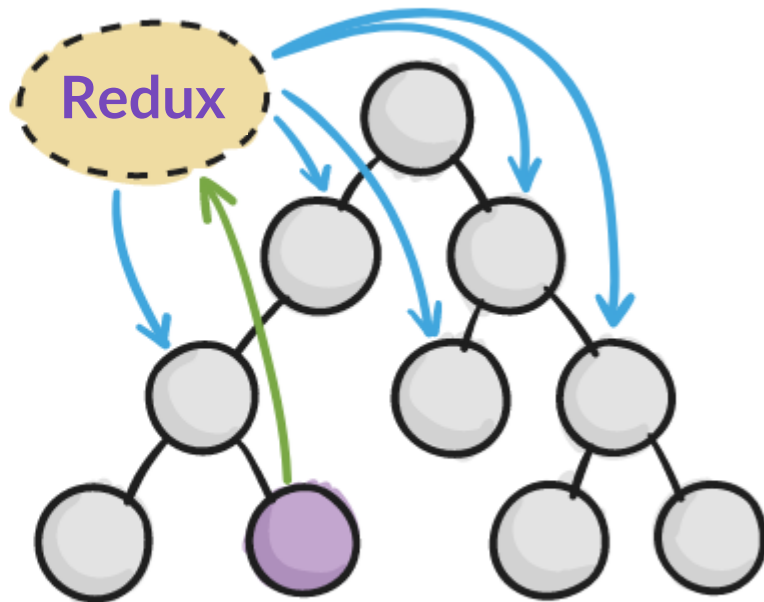


- Necesitamos pasar de **Redux** a **React** lo siguiente:
 - El **state**
 - Las **actions** que modifican el state
- También queremos mantener la inmutabilidad del estado y las funciones que lo modifican
- ¿Qué es lo más parecido que hay en React? => las props
- **Solución:** Creamos un componente `<ReduxProvider/>` que le pasa el estado como props al componente raíz de la app
- Usaremos para ello una librería llamada **react-redux**
 - que se instala vía npm

WITHOUT REDUX



WITH REDUX



 **COMPONENT INITIATING CHANGE**

- Podemos tener todo el estado en el Redux, pero también podemos dejar parte del estado en los componentes de React.
- **React state:** estado de un componente. Cada componente tiene acceso a su estado y puede pasárselo a sus hijos como props. Suele contener información relevante sólo a la vista
 - Por ejemplo, si el componente está activo o tiene un desplegable abierto. No queremos que este estado sea persistente. No se guardará en la base de datos y cuando vuelva a abrir mi aplicación no lo necesitaré: Tengo un estado por defecto para ello
- **Redux state:** estado de la aplicación al que todos los componentes deberían poder acceder. Suele contener información que guardamos entre sesiones
 - Por ejemplo, cada parte del estado que queremos que sea persistente, como los datos, etc. Lo necesitamos cuando abrimos la aplicación de nuevo

<https://github.com/reactjs/react-redux>

- Redux es agnóstico del framework (se puede usar con Angular, Polymer, ...). Para unirlo con React se necesita una librería
- Esta librería cuenta con una API que provee:
 - Un componente **provider**
 - Un método **connect**

- `<Provider store={store}/>`
- Componente de React que contendrá nuestra aplicación (el componente raíz) pasándole el estado de Redux como props
- Las props de Provider
 - Store (el store de la app)
 - Children (el componente raíz de nuestra app)

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

- **connect**([mapStateToProps], [mapDispatchToProps], [mergeProps], [options])
- Conecta un componente de React con el store
- No modifica el componente que se le pasa sino que devuelve uno nuevo (exportaremos este nuevo para usarlo en la aplicación en lugar del anterior)
- Params (todos opcionales):
 - **mapStateToProps**: recibe el estado y devuelve el objeto de props que será pasado al componente
 - Si especificamos un componente, se suscribirá al store. Cada vez que se modifique el estado, se llamará a mapStateToProps y recibirá nuevos props

- <https://react-redux.js.org/api/hooks>
- React-redux provee varios hooks
- useDispatch
- useSelector
 - Los “selectors” son unas funciones especiales para derivar datos del estado
- useSelector (este no se debe utilizar, mejor usar selectors)

- Creamos un nuevo componente **ReduxProvider**, que actúa como intermediario entre index.js y App.jsx
 - Renderiza el **Provider** de React-redux con su store
- En vez de renderizar App en index.js, renderizaremos ReduxProvider
- En App.js usaremos el método **connect** de React-redux para conectar App con el store usando el método mapStateToProps

1. Descargar dependencias
 - `npm install --save-dev react-redux redux`
2. Definir el estado de la app y ponerlo en `src/constants/constants.js`
3. Crear el fichero `src/redux/actions.js` con las acciones
4. Create the file `src/redux/reducers.js` con los reducers
5. Crear el componente `src/redux/ReduxProvider.js`
6. Modificar `src/index.js` para renderizar `ReduxProvider` en vez de `App`
7. Modificar `src/components/App.js` para conectarse con Redux

- Redux ha evolucionado mucho y ahora se recomienda usar redux toolkit (RTK)
- Leed las razones de su uso:
 - <https://redux.js.org/introduction/why-rtk-is-redux-today>
 - Mejora el trato de la inmutabilidad
 - Nos da muchas ayudas encapsuladas
 - Evita malos usos y bugs recurrentes
- Pero a cambio “oculta” cosas que parecen “magia”
- <https://redux.js.org/tutorials/fundamentals/part-8-modern-redux>

- Algunos recursos de aprendizaje adicionales:
 - <https://redux.js.org/introduction/learning-resources>
- Nuestro Tres en raya con redux:
 - <https://github.com/REACT-UPM/ejemplo tres en raya/tree/redux>