
React4teachers

2022 edition

1. React4teachers: From Zero 2 Hero



2. Contenido del curso

- 1.- Introducció
- 2.- De VanillaJs a React
- 3.- React framework: Motivos de exito
- 4.- React framework: Class components vs Function components
- 5.- React framework: State management - Control de el estado de la aplicacion
- 6.- React framework: Routing
- 7.- Storybook
- 8.- Monorepo
- 9.- TailwindCSS y StyledComponents
- 10.-Despliegue de aplicaciones

3. Ch5 - State management

Hasta ahora hemos visto el estado de los componentes bajo class y hook useState, lo cual nos permite guardar el comportamiento de un componente de forma independiente.

Este concepto es VITAL para los componentes...pero...

- Que sucede si quiero mantener el mismo estado en diversos componentes? Por ejemplo, mantener la opcion marcada, usarla para destacar el menu, y a la vez, mostrar un mensaje en el footer.
- Si quiero mostrar una pagina master/detail con los detalles de los diversos usuarios (otro componente), tengo que cargar todos los datos de la api otra vez?

Este concepto, de gran complejidad, se denomina ESTADO DE LA APLICACION

4. State management

Todos los administradores de estados de aplicacion usan el mismo concepto, similar a BBDD k-V, como Redis, Hazelcast o memcached:

- K: Utilizan una clave unica para almacenar logica.
- V: El valor puede ser una funcion CRUD o un conjunto de datos, normalmente json

La implementacion de React Hooks, incorpora el hook reducer, que puede dar solucion en muchos casos.

Estos conceptos utilizan la siguiente nomenclatura:

- modelo, store, provider: es el conjunto de acciones + persistencia deseada
- key, atom, action: es la funcionalidad a ejecutar
- state: el estado de la aplicacion, que ahora pasa a ser interno del framework

5. State management

El estado de la aplicacion es un concepto que no es nativo en ningun framework y sus aproximaciones suponen una complejidad anadida al desarrollo (p.e., usando localStorage y controlando sus accesos y modificaciones), pero todo ese entorno, esta repetido en todas las aplicaciones.

Por ello, para "evitar" rehacer el mismo trabajo siempre se utilizan librerias denominadas StateManagers, como son (en React):

- Recoil
- Redux: Reducers
- Rematch
- MobX
- Saga: function generators

6. Consejos sobre state manager

- Apuesta por un state manager solido
- Versatil
- Reutilizable en proyectos a traves de monorepo
- De impacto en la comunidad

Una mala eleccion, implica refactorizacion.

Cual usaremos nosotros?

- Veremos una aproximacion a reducir a traves del hook useReducer y useContext
- Utilizaremos la implementacion de redux a traves de modelos de Rematch*
- Veremos como se convierte esa implementacion con react-redux y reducers, que es la opcion mas "fiable"

7. Aproximacion a useContext

- Cuando useState no es suficiente notaras que pasaras a utilizar props en un nodo padre
- El padre pasara el contenido de los estados por propiedad a los hijos
- La aplicacion se volvera sumamente compleja: ref, props (proptypes), state y bindings o callbacks para manejar el estado.
- El siguiente nivel de simplificacion es el hook useReducer
- Para usarlo, realizaremos lo siguiente:
 1. Generaremos un contexto
 2. Englobaremos el componente bajo el contexto

3. Dentro del componente, usaremos el contexto.

8. useContext

```
// Paso 1: Generamos el contexto de aplicacion
// Fuera de componentes -> lo debemos exportar
export const MyDataContext = React.createContext();

// Paso 2: Envolvemos el componente con el contexto
function TheComponentWithState() {
  const [state, setState] = useState('whatever');
  return (
    <MyDataContext.Provider value={state}>
      <ComponentThatNeedsData/>
    </MyDataContext.Provider>
  )
}

// Paso 3: En el componente accedemos al contexto
function ComponentThatNeedsData() {
  const data = useContext(MyDataContext);
  // lo usamos
}
```

9. Aproximacion a useReducer

```
import React, { useReducer } from 'react';
const initialState = {
  count: 0
};
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      return initialState;
  }
}
function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  const handleInc= () => dispatch({type: 'increment'});
```

```

    const handleDec= () => dispatch({type: 'decrement'});
    return (
      <>
        Count: {state.count}
        <button onClick={handleDec}>-</button>
        <button onClick={handleInc}>+</button>
      </>
    );
  }
}

```

Ref¹

10. Hooks State management: app.js

- Generamos un contexto y un reductor, que por ahora pondremos en la cabecera de la aplicacion

```

import React, { useReducer } from 'react';
import ComponentA from './components/ComponentA';
import ComponentB from './components/ComponentB';
import ComponentC from './components/ComponentC';

export const CounterContext = React.createContext();

const initialState = 0;
const reducer = (state, action) => {
  switch (action) {
    case 'increment':
      return state + 1;
    case 'decrement':
      return state - 1;
    case 'reset':
      return initialState;
    default:
      return state;
  }
};

```

11. Hooks State management: app.js

- Utilizamos el reductor, que devolver el estado deseado y una funcion dispatch
- Envolveremos el componente interno, ya no necesitamos usar un callback

¹ <https://medium.com/suyeonme/using-usecontext-and-usereducer-together-lets-create-redux-like-global-state-in-react-87470e3ce7fa>

```
function App() {
  const [count, dispatch] = useReducer(reducer, initialState);

  return (
    <CounterContext.Provider
      value={{ counterCount: count, counterDispatch: dispatch }}
    >
      <div>
        Count - {count}
        <ComponentA />
        <ComponentB />
        <ComponentC />
      </div>
    </CounterContext.Provider>
  );
}

export default App;
```

12. Hooks State management: menu.js

- El componente hijo utiliza useContext para acceder al contexto en el que hemos englobado antes. Así accede al "store"
- A través de dispatch, llama a la acción con un string (k)

```
import React from 'react';
import { useContext } from 'react';
import { CounterContext } from '../App';

function ComponentA() {
  const counterContext = useContext(CounterContext);
  return (
    <div>
      ComponentA
      <button onClick={() => counterContext.counterDispatch('increment')}>
        Increase
      </button>
      <button onClick={() => counterContext.counterDispatch('decrement')}>
        Decrease
      </button>
      <button onClick={() => counterContext.counterDispatch('reset')}>
        Reset
      </button>
    </div>
  );
}

export default ComponentA;
```

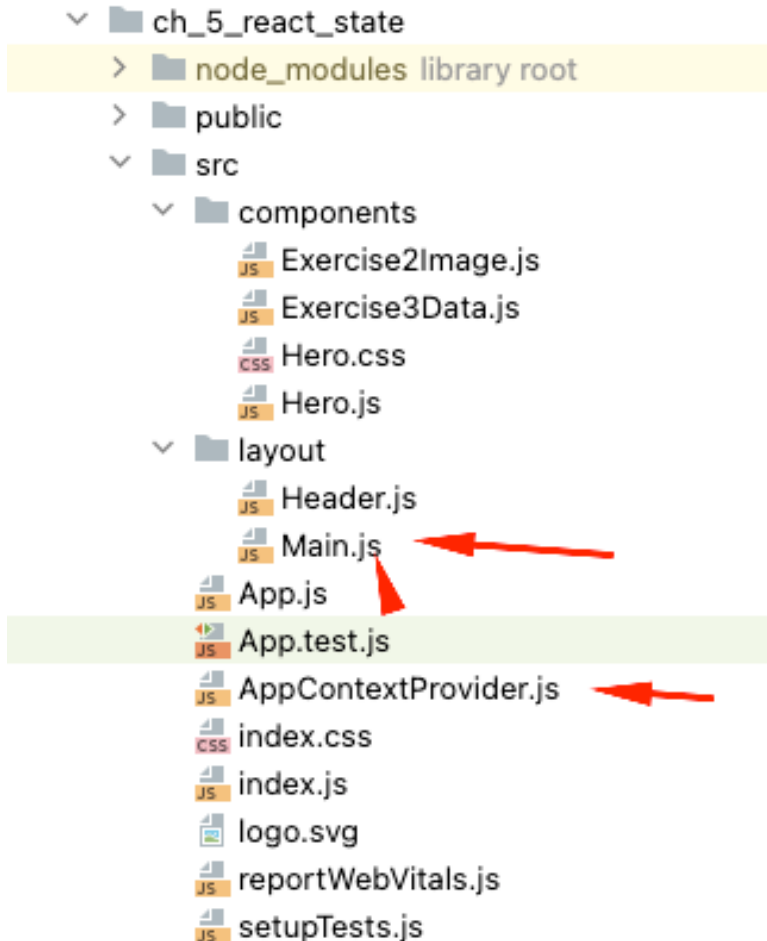
13. Ejercicio guiado

Aplica los reductores con useReducer y useContext a las acciones del menu.

14. Solucion al ejercicio

Para poder usar los reducers y el contexto adecuadamente debemos realizar lo siguiente:

- El componente padre lo dejaremos muy basico, todo lo pasaremos a componentes hijo, y se encargara solo de administrar el estado y las acciones a la aplicacion
- El contenido que teniamos en el lo pasaremos a un componente main
- Generaremos una clase que englobara la logica de la aplicacion, que denominaremos AppContextProvider.js



15. Solucion al ejercicio

AppContextProvider I

```
import React, { useReducer, createContext } from "react";
//Definimos el contexto de nuestra aplicacion
const AppContext = createContext();
//Definimos un estado inicial de aplicacion, que es interno
const initialState = {
  option: 1,
  data: []
}
//Serializamos las claves de nuestra aplicacion para no tener fallos
const ACTIONS = {
  SET_MENU_OPTION:"setMenuOption",
  LOAD_USERS:"loadUsers"
}
// Aqui esta el turrón! El reducer. Switch entre acciones que modifica
parte del estado
const reducer = (state, action) => {
  switch (action.type) {
    case ACTIONS.SET_MENU_OPTION:
      //return {...initialState.data, option:1}
      return {option: action.payload}
    default:
      return state
  }
}
```

16. Solucion al ejercicio

AppContextProvider II

```
//AppProvider sera nuestro componente magico! Hara todo lo necesario
para las acciones
// En esta clase, tendremos toda la logica de aplicacion
const AppProvider = ({ children }) => {
  //Aqui invocamos el reducer, nos trae el estado y el dispatch para
  lanzar eventos
  const [appState, dispatch] = useReducer(reducer, initialState);
  //Como no queremos usar dispatch fuera de la aplicacion, vamos a
  hacer un hub de funciones
  // Usamos payload como un elemento dinamico -> es un concepto redux
```



```
const appActions = {
  setOption1: () => {
    dispatch({type: ACTIONS.SET_MENU_OPTION, payload: 1})
    console.log("option 1")
  },
  setOption2: () => {
    console.log("option 2")
    dispatch({type: ACTIONS.SET_MENU_OPTION, payload: 2})
  },
  setOption3: () => {
    console.log("option 3")
    dispatch({type: ACTIONS.SET_MENU_OPTION, payload: 3})
  },
}

//Esta function envia como valor un objeto json, con el estado y las
acciones que queremos usar externas
return (
  <AppContext.Provider value={{
    appState: appState,
    appActions: appActions,
  }}>
    {children}
  </AppContext.Provider>
);
};

//Y lo mas importante, solo exportamos el componente y el contexto para
usar useContext
export { AppProvider, AppContext };
```

17. Solucion al ejercicio

App.js

```
import Header from "../layout/Header"
import React from 'react';
import {AppProvider} from "../AppContextProvider";
import {Main} from "../layout/Main";
const App = () => {
  return (
    <AppProvider>
      <div className="text-gray-700 bg-white">
        <Header/>
        <Main/>
      </div>
    </AppProvider>
  );
};
```

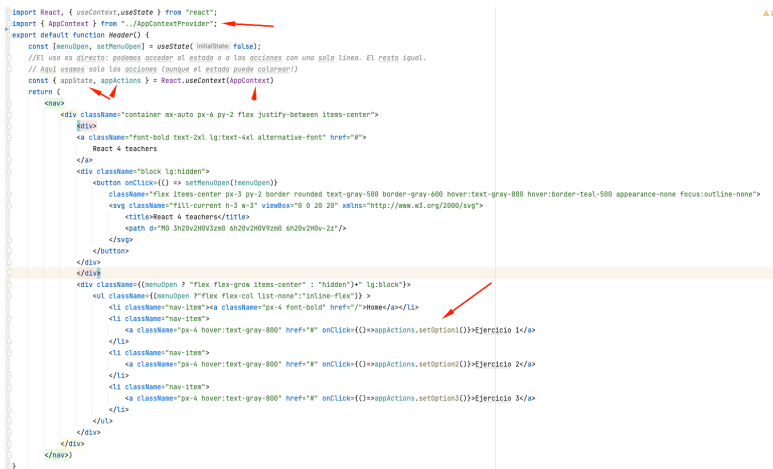
```

    </AppProvider>
  )
}
export default App;

```

18. Solucion al ejercicio

Header.js



```

import React, { useContext, useState } from "react";
import { AppContext } from "../AppContextProvider";
export default function Header() {
  const [menuOpen, setMenuOpen] = useState({ hidden: false });
  // Ojo con el estado: podemos acceder al estado o a las acciones con una sola línea. El resto igual.
  // Aquí usamos solo las acciones (porque el estado puede colapsar)
  const { appState, appActions } = React.useContext(AppContext);
  return (
    <div>
      <div className="container mx-auto px-6 py-2 flex justify-between items-center">
        <div>
          <h1 className="font-bold text-2xl lg:text-4xl alternative-font" href="#">
            React 4 teachers
          </h1>
        </div>
        <div className="block lg:hidden">
          <button onClick={() => setMenuOpen(!menuOpen)}>
            <div className="flex items-center px-3 py-2 border rounded text-gray-500 border-gray-600 hover:text-gray-800 hover:border-text-500 appearance-none focus:outline-none">
              <svg className="fill-current h-3 w-3" viewBox="0 0 20 20" xmlns="http://www.w3.org/2000/svg">
                <title>React 4 teachers toggle</title>
                <path d="M0 0 20 20" />
              </svg>
            </button>
          </div>
        </div>
        <div className={menuOpen ? "flex flex-grow items-center" : "hidden"} lg:block>
          <ol className={menuOpen ? "flex flex-col list-none" : "list-none flex">
            <li className="nav-item">
              <a className="px-4 font-bold" href="#">Home</a></li>
            </li>
            <li className="nav-item">
              <a className="px-4 hover:text-gray-800" href="#">Ejercicio 1</a>
            </li>
            <li className="nav-item">
              <a className="px-4 hover:text-gray-800" href="#">Ejercicio 2</a>
            </li>
            <li className="nav-item">
              <a className="px-4 hover:text-gray-800" href="#">Ejercicio 3</a>
            </li>
          </ol>
        </div>
      </div>
    </div>
  );
}

```

19. Solucion al ejercicio

Main.js

```
import {Hero} from "../components/Hero";
import Exercise2Image from "../components/Exercise2Image";
import Exercise3Data from "../components/Exercise3Data";
import React, {useContext} from "react";
import {AppContext} from "../AppContextProvider";
//Main es el nuevo componente para renderizar el cuerpo de la aplicacion, dependiendo del estado
//Si quisieramos preservar el estado, utilizaríamos localStorage en el context provider
export const Main = () => {
  const {appState} = useContext(AppContext);
  const {option} = appState;
  console.log('Current state value is: ${option}')
  console.log(appState)
  let renderingPart = <div>Empty component</div>;
  switch (option){
    case 1:
      renderingPart = <Hero/>
      break;
    case 2:
      renderingPart = <Exercise2Image/>
      break;
    case 3:
      renderingPart = <Exercise3Data/>
      break;
    default:
      renderingPart = "Profe: No valid state associated"
      break;
  }
  return ( <div>{renderingPart}</div> )
}
```

20. State management with hooks

Beneficios:

- No requiere librerías y necesita poco código de inicio
- Una vez diseñado el provider, el código es muy sencillo
- Es similar a Redux (por no decir idéntico)

Cons:

- Solo es aplicable a hooks
- No podemos mezclar lógicas, y solo aplicable a proyectos nuevos
- El estado desaparece al refrescar, tenemos que instaurar nuestra persistencia
- Barrera inicial, aunque pequeña

21. State management

Una vez comprendido como Redux ha perfilado el uso de reducers hasta que han sido incorporados en React Hooks, veamos como son las herramientas externas

- Recoil
- Redux: Reducers
- Rematch
- MobX
- Saga: function generators

22. Recoil overview

- Compatible con hooks y class
- Utiliza el concepto de atom y selectores
- Bajo impacto sobre hooks useState a useRecoilState

```
import React from 'react';
import {
  RecoilRoot,
  atom,
  selector,
  useRecoilState,
  useRecoilValue,
} from 'recoil';

function App() {
  return (
    <RecoilRoot>
      <CharacterCounter />
    </RecoilRoot>
  );
}

const textState = atom({
  key: 'textState', // unique ID (with respect to other atoms/selectors)
  default: '', // default value (aka initial value)
});
```

23. Recoil

Beneficios:

- Simple, similar a sintaxis de React
- Usado por facebook en herramientas internas

- Rendimiento

Inconvenientes:

- Demasiado nueva

24. Redux overview

- Muy utilizado
- Compatible con class y hooks
- Logica de aplicacion separada
- Requiere preparacion
- Reutilizable en diversos proyectos (base vanilla)

```
// Usamos una funcion de reduccion que controla estados
// Esta funcion se llama con una accion
function counterReducer(state = initialState, action) {
  // Segun la accion que se ejecute, modificamos
  // la logica y devolvemos el state
  switch (action.type) {
    case 'counter/incremented':
      return { ...state, value: state.value + 1 }
    case 'counter/decremented':
      return { ...state, value: state.value - 1 }
    default:
      // Si no hay cambios, devolvemos el mismo estado
      // (acciones incorrectas?)
      return state
  }
}
```

25. Redux

Beneficios:

- Usado desde 2015
- Soporte en herramientas de desarrollo
- Permite ver el historico de cambios de estado
- Pequeno: redux + react-redux = 3kb

- Muy funcional
- El ecosistema sincroniza localStorage y callbacks

Problemas:

- Cuesta un bloqueo mental
- Requiere definir cada aspecto, por lo que su implementacion no es muy rapida.

26. Rematch overview

- Se utiliza con redux para simplificar su implementacion
- Utiliza el mismo concepto de Store
- Incorpora models para evitar los reducers

```
import React from 'react'
import ReactDOM from 'react-dom'
import { Provider } from 'react-redux'
import { init } from '@rematch/core'
import createUpdatedPlugin from '@rematch/updated'

import * as models from './models'
import App from './App'

const store = init({
  models,
  // create plugin
  plugins: [createUpdatedPlugin()], // add to plugin list
})

ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
)
```

- [Rematch website](https://rematchjs.org/)²

² <https://rematchjs.org/>

- [Rematch example](#)³

27. MobX overview

```
import {observer} from 'mobx-react-lite'
import {createContext, useContext} from "react"

const TimerContext = createContext<Timer>()

const TimerView = observer(() => {
  // Grab the timer from the context.
  const timer = useContext(TimerContext) // See the Timer definition
  above.
  return (
    <span>Seconds passed: {timer.secondsPassed}</span>
  )
})

ReactDOM.render(
  <TimerContext.Provider value={new Timer()}>
    <TimerView />
  </TimerContext.Provider>,
  document.body
)
```

28. Mobx

Beneficios:

- Es reactivo, si se modifica un valor, renderiza todos los componentes que lo usan
- No hay acciones ni reductores
- Menos código

Problemas:

- No tan usado como redux
- Menos código > magia!

³ <https://github.com/rematch/rematch/blob/main/examples/updated-react>

- Requiere proxies ES6, por lo que no es compatible con ECMA5 (IE11)

29. Saga overview

Utiliza el concepto de function generators de vanillaJs Su uso es altamente complejo

```
import { takeEvery } from 'redux-saga/effects'
import Api from './path/to/api'

function* watchFetchProducts() {
  yield takeEvery('PRODUCTS_REQUESTED', fetchProducts)
}

function* fetchProducts() {
  const products = yield Api.fetch('/products')
  console.log(products)
}
```

[Saga](#)⁴

30. Referencias

[React context api + reducer = redux](#)⁵

⁴ <https://redux-saga.js.org/docs/basics/DeclarativeEffects>

⁵ <https://www.burhanuday.com/blog/react-context-api-usereducer-redux-ogo>