# React

¿Qué es React?

React es una librería que nos permite crear aplicaciones.

* + Es una librería declarativa, ósea que hace fácil seguir patrones de diseño y crear interfaces de usuario interactiva. También porque es un código más predecible y fácil de depurar. En resumen, puedes crear tus propias etiquetas.
  + Es eficiente. Ya que, si se hace un cambio en un elemento, no hay que actualizar todo el DOM (Llamado Virtual DOM), sino un solo elemento.
  + Es basado en componentes. Puedes crear componentes encapsulados que manejen su propio estado y puedan ser reusados a lo largo de tus proyectos.
  + Es solo la UI, es una librería, no un framework. Lo que significa que puede combinarse con otros frameworks.

Virtual DOM

Es una representación en memoria del DOM real del navegador. Actúa como intermediario entre el estado de la aplicación y la UI que está viendo el usuario de tu proyecto.

Algoritmo de reconciliación

Es un algoritmo diferencial que se utiliza para actualizar los elementos del árbol de la aplicación.

Fibers

Son objetos internos de React que mantienen la información del árbol de los componentes.

## JSX

Es Javascript + HTML. Ejemplo:  
const name = “Axel”;  
const title = <h1> hi {name} </h1>;

Sintaxis sugar

Son sintaxis diseñados para hacer algunas construcciones más fáciles de leer o expresar. Por ejemplo, los lambda de Java  
const element = (<h1 className = “greeting”> Hello, world! </h1>);

Transpilación

Es la “traducción” de un código a código react para que el navegador lo pueda interpretar. Esto lo realiza Babel.  
const element = React.createElement(‘h1’, {className: ‘greeting’}, ‘Hello, world!’);

Babel

Es un compilador de JS. Transpila el código React a código que sea compatible con el navegador, por ejemplo, el código de “Sintaxis sugar” y el código de ejemplo de “Transpilación”.

## Funciones normales

getFullName (){

*const* {name, lastName} = this.state;

*return* name + " " + lastName;

   }

## Arrow Function

También son llamadas funciones anónimas

getFullName = () =>{

*const* {name, lastName} = this.state;

*return* name + " " + lastName;

   }

render(){

      console.log(this.state);

*return* (

         <div>

            Hola {this.getFullName()}

         </div>

      );

   }

La ventaja de la arrow function es que se puede acortar:

getFullName = () *=> this.*state*.*name + " " + *this.*state.lastName;

## Class component

class Welcome *extends* React.Component{

  render(){

*return* <h1>Hello Axel</h1>

  }

}

## Props

Se pueden pasar las propiedades de las etiquetas a las funciones de JSX para utilizarlas.

Algunas propiedades son:

* Son solo lectura
* Van de padre a hijo
* Puedes enviar cualquier tipo de dato como propiedad

Ejemplo:

HTML:  
    <Welcome name="test"/>

JS:  
function Welcome ({name}){

*return* (<h1>Hello {name}</h1>)

}

## Estado

Es una variable que tiene a cambiar si el componente lo pide. El cambiarlo significa volver a renderizar el componente para actualizarlo.

## this.state

Sirve para definir propiedades de la clase. Se deben hacer en el constructor de la misma.

import React from 'react';

import { createRoot } from 'react-dom/client';

import './index.css';

class MyFirstInput *extends* React.Component {

   constructor(props){

      super(props);

      this.state = {

         name: "Juan",

         lastName : "Ortiz"

      };

   }

   render(){

      console.log(this.state);

*return* (

         <div>

            Hola {this.state.name} {this.state.lastName}

         </div>

      );

   }

}

*const* root = createRoot(document.getElementById('root'));

root.render(

   <MyFirstInput/>);

## useState

Es la equivalencia del this.state, pero para funciones:

import React, { useState } from 'react';

function Example() {

*const* [count, setCount] = useState(0);

*return* (

    <div>

      <p>You clicked {count} times</p>

      <button onClick={() => setCount(count + 1)}>

        Click me

      </button>

    </div>

  );

}

También, no es necesario declarar una propiedad en conjunto con un set:

*const* [, setCount] = useState(0);

*const* [count] = useState(0);

Y en el useState puede ir cualquier dato

*const* [count] = useState([]);

## className

Es una propiedad que sirve para definir la clase css para la etiqueta, el equivalente al class del HTML.

<input type="text" name="inputOne" className="oneClass"></input>

## Eventos

Los eventos en React se declaran igual que en HTML, pero usando camelCase:

function Form(){

   function handleClick(e){

      e.preventDefault();

      console.log('Clicked')

   }

*return*(

      <div onClick ={handleClick}>

         <label>

            <input type="text" name="inputOne" className="oneClass"></input>

         </label>

      </div>

   )

}

*const* [email, setEmail] = useState("");

*const* handleEmailChange = ({target:{value}}) =>{

    setEmail(e.target.value);

  };

<label>

          Correo

          <input type="email" value={email} onChange={handleEmailChange}/>

</label>

Tener en cuenta que React no admite unos pocos eventos.

## setState

Actualiza una propiedad del state:

class MyFirstInput *extends* React.Component {

   constructor(props){

      super(props);

      this.state = {

         name: "Juan",

         lastName : "Ortiz"

      };

   }

   brother(){

      this.setState({name: "Pedro"});

   }

}

NO HACER EL SETSTATE EN EL CONSTRUCTOR NI EN EL RENDER.

## prevState

Es un callback que viene en el setState, contiene el estrado anterior:

  brother(){

      this.setState(prevState => ({name: prevState.name + " Pedro"}));

   }

Ahí si la propiedad name del estado previo era, por ejemplo, Juan, ahora el nombre será “Juan Pedro”.

## Crear aplicación React y inicializarla

Para usar este comando, es necesario Node >= 8.10 y npm >= 5.6. Npm viene instalado por default con Node JS.

Una vez instalado estos dos, se utilizan estos comandos:

npx créate-react-app {nombre}  
cd {nombre}  
npm start

En el nombre no se utiliza camelCase, si no kebab-case. Esto por buenas practicas.

## e.preventDefault

En un evento, evita que la página entera se actualice.

*const* handleFormSubmit = (e) => {

    e.preventDefault();

  };

## Funciones de ciclo de vida del componente:

* constructor
* componentDidMount: Es invocado después de que un componente es montado (insertado en el árbol)
* componentDidUpdate: Se llama después de actualizar un objeto
* shouldComponentUpdate: Cuando se actualiza el componente en el re renderizado, se llama esta función.
* getDerivedStateFromProps: En caso de que las props se actualicen y tu componente tiene un padre, se usa este método.
* getSnapshotBeforeUpdate: Es invocado justo después de que la salida más reciente es guardada, por ejemplo, en el DOM.

## UseEffect

Es un hook, es parecido a useState. Es una función que se ejecuta cuando cambia un valor del componente, ej:  
 useEffect(() => {}, [siCambiaEstoEntra])

Puede simular componentDidMount(), componentDidUpdate() y componentWillUnmount():

* componentDidMount:

useEffect(() => {}, [])

* componentDidUpdate:  
  useEffect(() => {})
* componentWillUnmount:

useEffect(() => {return () => {/\* \*/} })

Ejemplo:

*const* App = () => {

*const* [name, setName] = useState("");

*const* [age, setAge] = useState("");

*/\*\* componentDidUpdate \*/*

  useEffect(() => {

    console.log("didUpdate");

  })

*/\*\* componentDidMount \*/*

  useEffect(() => {

    console.log("didMount");

  }, [])

*/\*\* cambio de propiedad \*/*

  useEffect(() => {

    console.log("name cambio");

  }, [name])

*/\*\* cambio de propiedades \*/*

  useEffect(() => {

    console.log("name o age cambio");

  }, [name,age])

*return*(

    <div>

      <input value={name} onChange={({target: {value}}) => setName(value)}/>

      <br/>

      <input value={age} onChange={({target: {value}}) => setAge(value)}/>

    </div>

  )

}

## Destructor

El return del componentDidMount se puede usar como destructor.

useEffect(() => {

    console.log("didMount");

*return*() => {

      console.log("willUnmount");

    }

  }, [])

## Renderizado condicional

*const* handleButtonClick = () => setLogged(true)

*return*(

    <div>

      <input value={name} onChange={({target: {value}}) => setName(value)}/>

      <br/>

      <input value={age} onChange={({target: {value}}) => setAge(value)}/>

      <br/>

      <button onClick={handleButtonClick}>Presiona</button>

      <br/>

      {isLogged ? <h1>Logeado</h1> : undefined}

      {isLogged && <h1>Logeado</h1>}

    </div>

  )

## Iteración de listas:

Para iterar una lista, se utiliza {nombreLista}.map(({nombreValor},{índice}) => { return () })}. Por ejemplo:

*const* jsonData= [{

  key:"1",

  name: "Juan"

},{

  key:"2",

  name: "Illo"

}]

*const* App = () => {

*return*(

    <div>

      {jsonData.map((value,index) => {

*return*(

          <div key={value.key}>

            <p>{value.name}</p>

          </div>

        )

      })}

    </div>

  )

}

Si a la propiedad que usa map, se le agrega “?”, ignorará las listas indefinidas, por ejemplo:

jsonData?.map()

## Estilos condicionales

Se pueden definir estilos que se usen bajo ciertas condiciones:

<div className={`search ${isAtTop ? "search--top" : "search--center"}`}>

            <SearchBox onSearch={handleSearchClick} onClose={handleCloseClick} />

        </div>

----------

.search--center{

    justify-content: center;

    align-items: center;

}

.search--top{

    justify-content: center;

    align-items: flex-start;

}

## Acortar parámetros

Cuando se tiene un objeto que se pasa por parámetro, en lugar de ir pasando uno a uno con {objeto}.{parámetro}, se puede pasar el objeto y en la definición de la función especificar que propiedades se necesitan:

<SearchResultsItem key={value.id} {**...**value}/>

export default function SearchResultsItem({name, email}){

*return*(

    <div style={{backgroundColor: "#E8E7E7",marginTop: 5, marginBottom: 5, padding: 10, width:"100%"}}>

                    <p>{name}</p>

                    <p>{email}</p>

                    </div>

                    );

}

## React Router

Es una librería para implementar el routing en la aplicación.

Para instalarla es:  
npm i –S react-router-dom

## Utilerias más utilizadas

* BrowserRouter: Es el routeador, es el tag que tiene que englobar a toda la aplicación.
* Switch: Se usa cuando tenemos varias pantallas .
* Route: Ruta, define uno por cada pantalla.
* Link: Es el link, un a del HTML de toda la vida.

## Declarar una ruta

import { BrowserRouter, Routes, Route } from "react-router-dom"

import Home from "../views/Home";

export default function Pages(){

*return*(

        <BrowserRouter>

            <Routes>

                <Route path="/" exact={true} element={<Home/>}/>

            </Routes>

        </BrowserRouter>

    );

}

## Redirect

Etiqueta de redirección, en caso de que se cumplan ciertas condiciones, redirige al usuario a la página especificada:

export default function Pages(){

*const* loggedIn = true;

*return*(

        <BrowserRouter>

            <Route exact path="/">

                {loggedIn ? <Redirect to="/dashboard"/> : <Home/>}

            </Route>

        </BrowserRouter>

    );

}

Si el redirect es más complejo, se puede especificar paths, query params y states, entre otras cosas:

<Redirect to={{pathname: "/login", search: "?utm=your+face", state: {referrer: currentLocation}}}/>

## Ruta 404

                <Route path="\*" exact={true} element={<FourOFour/>}/>

## Fetch

Es una utilería que viene propia del objeto Window, lo que significa que viene ya con Javascript.

Con este método, se pueden realizar peticiones a una API. Por ejemplo:

*const* getUsers = *async* () => {

            fetch("https://jsonplaceholder.typicode.com/users")

            .then(response => response.json())

            .then(data => {

                setData(data);

            })

        };

        getUsers().catch(null);

También se puede evitar la anidación de la siguiente manera:

*const* getUsers = *async* () => {

*const* response = *await* fetch("https://jsonplaceholder.typicode.com/users");

*const* data = *await* response.json();

            setData(data);

        };

        getUsers().catch(null);

## Axios

Es una librería que nos ayuda a realizar requests en node y en el front end.

Hace lo mismo que el fetch utilizando XMLHttpRequests desde el navegador, desde el backend hace llamadas http, entre otras características.

Para instalarlo se usa:

npm i –S axios

Algunos ejemplos de llamadas son:

*const* axios = require('axios').default;

*// Make a request for a user with a given ID*

    axios.get('/user?ID=12345')

      .then(function (response) {

*// handle success*

        console.log(response);

      })

      .catch(function (error) {

*// handle error*

        console.log(error);

      })

      .finally(function () {

*// always executed*

      });

*// Optionally the request above could also be done as*

    axios.get('/user', {

        params: {

          ID: 12345

        }

      })

      .then(function (response) {

        console.log(response);

      })

      .catch(function (error) {

        console.log(error);

      })

      .finally(function () {

*// always executed*

      });

*// Want to use async/await? Add the `async` keyword to your outer function/method.*

*async* function getUser() {

      try {

*const* response = *await* axios.get('/user?ID=12345');

        console.log(response);

      } catch (error) {

        console.error(error);

      }

    }

## Context.API

Context provee una forma de pasar datos a través del árbol de componentes sin tener que pasar props manualmente en cada nivel

## Provider

Componente que contiene las propiedades y funciones que utilizarán todos los componentes de la aplicación utilizando Context y un hook.

Los provider solo van desde un componente para abajo en el árbol genealógico.

Se puede tener 1 a N providers, pero es recomendable 1 provider para cada cosa (e.g. transacciones, cuentas, clientes, etc.).

Lo malo es que también si se utiliza, todo el árbol hacia debajo de donde está el provider se re renderiza.

Es recomendable usarlo en proyectos pequeños y medianos como mucho.

Ejemplo:

Carpeta context/pokemons:

* Index.js:

import { createContext } from "react";

*const* PokemonContext = createContext("pokemons");

export default PokemonContext;

* Provider.js:

export default function PokemonProvider({ children }){

*const* [pokemons, setPokemons] = useState([]);

*const* getPokemons = *async* () => {

        try{

*const* pokemonsResult = *await* apiCall({url: "https://pokeapi.co/api/v2/pokemon?limit=100"})

            setPokemons(pokemonsResult.results);

        }catch(error){

            setPokemons([]);

        }

    };

*return* (

        <PokemonContext.Provider value={{ getPokemons, pokemons }}>

            {children}

        </PokemonContext.Provider>

    );

}

Y en el archivo principal (App.js en este caso):

import './App.css';

import Routes from "./routes"

import PokemonProvider from './context/pokemons/Provider';

function App() {

*return* (

    <PokemonProvider>

      <Routes/>

    </PokemonProvider>

  );

}

## Path param en React Router

El path param en react router se hace utilizando los dos puntos (“:”), por ejemplo:

export default function Pages(){

*return*(

        <BrowserRouter>

            <Routes>

                <Route path="/pokemon/:id" exact={true} element={<PokeDetail/>}/>

            </Routes>

        </BrowserRouter>

    );

}

## useParams

Devuelve un objecto de parejas llave/valor de los parámetros URL (Path). Úsalo para acceder a match.params de la ruta actual.

## Link

Sería el anchor tag (<a>) en js, por ejemplo:

export default function PokemonListItem({name, url}){

*const* getId = () => url.split("/")[6];

*return*(

        <>

            <p>{name}</p>

            <button>

                <Link to={`/pokemon/${getId()}`}>Ver detalle</Link>

            </button>

        </>

    )

}

## useLocation

Utilería que contiene todo el objeto de window.location()

## Scroll restoration

Para restaurar el scroll a arriba de todo, se puede crear un componente:

import { useEffect } from "react";

import { useLocation } from "react-router-dom";

export default function ScrollToTop(){

*const* { pathname } = useLocation();

    useEffect(() => {

        window.scrollTo(0,0);

    }, [pathname])

*return* null;

}

E implementarlo en un return:

export default function Pages(){

*return*(

        <BrowserRouter>

            <ScrollToTop/>

            <Routes>

                <Route path="/" exact={true}

            </Routes>

        </BrowserRouter>

    );

}

Zustand

Un state manegement diferente a Context.API y Redux. Funciona como Arbol Binario. Los providers de Context API son llamados Store aquí.

Para instalarlo se usa el comando “npm install zustand”.

## Store – Zustand

Dentro de los stores de Zustand podemos almacenar cualquier dato y consumirlo desde cualquier componente. Datos como:

* Funciones
* Arreglos
* Enteros
* Booleanos

## create()

Método que crea una Store de Zustand.

*const* usePokemonsStore = create((set, get) => ({

    getPokemons: *async* () => {

        try{

            set({ isLoading: false, errorMessage: "", hasError: false });

*const* pokemonsResult = *await* apiCall({url: "https://pokeapi.co/api/v2/pokemon?limit=100"});

            set({pokemons: pokemonsResult.results });

        } catch(error) {

            set({ pokemons: [], errorMessage: "Algo ha pasado, verifica tu conexión", hasError: true });

        } finally {

            set({ isLoading: false });

        }

    },

    pokemons: [],

    isLoading: false,

    hasError: ""

}));

## Obtener datos del store de Zustand

*const* { getPokemons, pokemons, isLoading, hasError } = usePokemonsStore(

        state => ({getPokemons: state.getPokemons, pokemons: state.pokemons, isLoading: state.isLoading, hasError: state.isLoading }));

## Shallow

Parametro de la función que obtiene datos del store de zustand, si se usa, la página solo se re-renderizará cuando cambie un dato del store.

*const* { getPokemonDetails, pokemonDetails, isLoading, hasError, errorMessage } = usePokemonsStore

    (state => ({getPokemonDetails: state.getPokemonDetails, pokemonDetails: state.pokemonDetails,

        isLoading: state.isLoading, hasError: state.hasError, errorMessage: state.errorMessage}), shallow);

## getState()

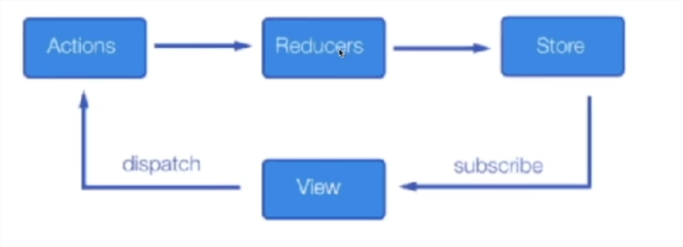
Método que obtiene todo el objeto de un store:

*const* test = usePokemonsStore.getState();

Es recomendable usarlo cuando solo se tiene que llamar una o dos veces.

## Redux

## Redux – Como viaja la información



Vista: Es la pantalla/componentes, lo que ve el usuario. Mediante el dispatch (disparadores) ejecuta acciones para que actualice la store. Mediante la store muestra la información.

Acciones: Las acciones piden la información que necesita la vista.

Reducers: Son funciones puras que se encargan de actualizar la Store.

Store: es un mega objeto.

## Principios

* Única fuente de la verdad (Single source of truth): Solo hay un lugar de donde se va a traer la información.
* El estado es de solo lectura (read only).
* Los cambios se realizan mediante funciones puras.

## Acciones

Son objetos planos de JavaScript. Una ascción debe tener una propiedad type que indica el tipo de acción a realizar. Las acciones son normalmente definidos como string constantes. Ejemplo:

## Reducers

Las acciones describen que algo pasó, pero no especifican cómo cambió el estado de la aplicación en respuesta. Esto es trabajo de los reducers.

## Store

* Contiene el estado de la aplicación
* Permite el acceso al estado vía getState();
* Permite que el estado sea actualizado via dispatch(action).
* Registra los listeners vía subscribe(listener);
* Maneja la anulación del registro de los listeners vía el retorno de la función de subscribe (listener).