Introducción a React.js

### Cómo crear componentes

Esta es la manera más simple y rápida para trabajar con React, solo necesitamos ejecutar el comando: npx create-react-app nombre-del-proyecto

Con npm start inicializa el proyecto

Dentro del index.js siempre tendremos un div con un id, como root que será la raíz de nuestro proyecto, y la usaremos para empezar a construir con JavaScript:

<!-- Aquí es en donde todo nuestro código renderiza los componentes. -->

<div id="root"></div>

Para iniciar el entorno de desarrollo podemos ejecutar el comando npm start, con esto tendremos nuestro servidor corriendo en el puerto 3000

1. En index.js se renderiza todos los componentes del proyecto
2. Luego En APP.js crea los componentes y los llama

import react from "react";

import { TodoCounter } from "./TodoCounter";

import { TodoSearch } from "./TodoSearch.js";

import { TodoList } from "./TodoList.js";

import { TodoItem } from "./TodoItem.js";

import { CreateTodoButtom } from "./CreateTodoButton.js";

const todos=[

{text:'Cortar cebolla', completed:false},

{text:'Tormar el curso de intro a react', completed:false},

{text:'Llorar con la llorona', completed:false}

];

function App() {

return (

<react.Fragment>

<TodoCounter />

<TodoSearch />

<TodoList>

{todos.map(todo => (

<TodoItem text={todo.text} />

))}

</TodoList>

<CreateTodoButtom />

</react.Fragment>

);

}

export default App;

1. y por ultimo se crean cada archivo su index.js en carpetas del nombre del componente donde tiene sus funciones

**TodoCounter**

import React from "react";

function TodoCounter(){

return(

<h2> Has complentado 2 de 3 ToDos</h2>

)

}

export {TodoCounter};

**TodoList**

import react from "react";

function TodoList(props){

return(

<section>

<ul>

{props.children}

</ul>

</section>

);

}

export { TodoList};

Propiedad children

También podemos utilizar los componentes de React como etiquetas abiertas, para pasarle contenido, elementos o incluso otros componentes, la manera de acceder a ellos es con la propiedad especial children.

const Componente = (props) => {

return(

<div className="titulo">

{props.children}

{/\* props.children = <h1>¡Soy un título anidado!h1> \*/}

div>

)

}

En el archivo App.js que contiene todos nuestros componentes se encapsula en <React.Fragment>

</React.Fragment>

Ò

<>

...

</>

Agrupación por tipo de archivo

En esta estructura solo agrupamos los **archivos similares**, es la más recomendada para proyectos grandes, también sirve para tener una excelente organización en proyectos pequeños, por ejemplo:

└── src/

├── assets/

│ ├── img/

│ └── foto.jpg

│ ├── fonts/

│ └──ubuntu.woff2

├── components/

│ ├── CreateTodoButton/

│ ├── index.js

│ └── CreateTodoButton.css

│ ├── TodoCounter/

│ ├── TodoItem/

│ ├── TodoList/

│ ├── TodoSearch/

├── context/

│ ├── TodoContext.js

├── hooks/

│ ├── useLocalStorage.js

├── utils/

│ ├── fetch.js

### Importar css en react

import React from "react";

import './todoCounter.css'

function TodoCounter(){

return(

<h2 className="TodoCounter">

Has complentado 2 de 3 ToDos

</h2>

)

}

export {TodoCounter};

Agregar fuentes

<link rel="icon" href="%PUBLIC\_URL%/favicon.ico" />

### Manejar eventos

Manejar eventos en React es muy similar a manejar eventos en el DOM, solo necesitamos pasarle una propiedad **on + evento**, por ejemplo: onClick, onChange, onMouseOver Manejar el estado

<button onClick={click}></button>

function CreateTodoButton(props) {

const onClickButton = (msg) => {

alert(msg);

};

return (

<>

{/\* ✅ \*/}

<button

className="CreateTodoButton"

onClick={() => onClickButton('Aquí se debería abrir el modal')}

>

+

</button>

{/\* ❌ \*/}

<button

className="CreateTodoButton"

onClick={onClickButton('Esta función se ejecuta al inicio, no al presionar el botón'}

>

+

</button>

</>

);

}

### Manejar efectos

El **hook de efecto** en react nos permite ejecutar un pedazo de código cada vez que necesitemos, a lo largo de la vida de nuestro componente, también cuando se cumplan ciertas condiciones.

Algo curioso e importante de saber es que el código dentro de nuestro **hook de efecto** no se ejecuta como el resto del código, se ejecutará inicialmente justo antes de hacer el renderizado del HTML que resulte de nuestro código de React.

React.useEffect(funcion, [dato1, dato2, datoN])

### Manejo del estado

El manejo del estado en estos componentes es mucho más sencillo, utilizando el hook de estado.

import { useState } from 'react';

function Component() {

const [count, setCount] = useState("");

}

TODOSEARCH

import React from 'react';

import './TodoSearch.css';

function TodoSearch() {

const [searchValue, setSearchValue] = React.useState('');

const onSearchValueChange = (event) => {

console.log(event.target.value);

setSearchValue(event.target.value);

};

return [

<input

className="TodoSearch"

placeholder="Cebolla"

value={searchValue}

onChange={onSearchValueChange}

/>,

<p>{searchValue}</p>

];

}

Completando TODOs

const completeTodo = (text) => {

const todoIndex = todos.findIndex(todo => todo.text === text);

const newTodos = [...todos];

newTodos[todoIndex].completed = true;

setTodos(newTodos);

};

Eliminando TODOs

const deleteTodo = (text) => {

const todoIndex = todos.findIndex(todo => todo.text === text);

const newTodos = [...todos];

newTodos.splice(todoIndex, 1);

setTodos(newTodos);

};

El **levantamiento de estado** es una técnica de React que pone el estado en una localización donde se pueda pasar como **props** a los componentes.

import React from 'react';

import { TodoCounter } from './TodoCounter';

import { TodoSearch } from './TodoSearch';

import { TodoList } from './TodoList';

import { TodoItem } from './TodoItem';

import { CreateTodoButton } from './CreateTodoButton';

// import './App.css';

const defaultTodos = [

{ text: 'Cortar cebolla', completed: true },

{ text: 'Tomar el cursso de intro a React', completed: false },

{ text: 'Llorar con la llorona', completed: true },

{ text: 'LALALALAA', completed: false },

];

function App() {

const [todos, setTodos] = React.useState(defaultTodos);

// El estado de nuestra búsqueda

const [searchValue, setSearchValue] = React.useState('');

const completedTodos = todos.filter(todo => !!todo.completed).length;

const totalTodos = todos.length;

// Creamos una nueva variable en donde guardaremos las coincidencias con la búsqueda

let searchedTodos = [];

// Lógica para filtrar

if (!searchValue.length >= 1) {

searchedTodos = todos;

} else {

searchedTodos = todos.filter(todo => {

const todoText = todo.text.toLowerCase();

const searchText = searchValue.toLowerCase();

return todoText.includes(searchText);

});

}

return (

<React.Fragment>

<TodoCounter

total={totalTodos}

completed={completedTodos}

/>

<TodoSearch

searchValue={searchValue}

setSearchValue={setSearchValue}

/>

<TodoList>

{/\* Regresamos solamente los TODOs buscados \*/}

{searchedTodos.map(todo => (

<TodoItem

key={todo.text}

text={todo.text}

completed={todo.completed}

/>

))}

</TodoList>

<CreateTodoButton />

</React.Fragment>

);

}

export default App;

### Comunicar componentes con props

HTT

### Comunicar componentes con el contexto de React

Es una forma de **tener acceso a datos** a través del árbol de componentes **sin tener que pasar props manualmente** en cada nivel, usando datos globales.

Para esto tenemos un proveedor que envolverá a todos los componentes que serán los consumidores de nuestro contexto.

**Fases:**

1. **Crear** el contexto de nuestra aplicación
2. **Proveer** nuestro contexto con los datos que queremos globales.
3. **Consumir** los datos desde cualquier parte de nuestra aplicación.

Creando el contexto de nuestra aplicación

Primero necesitamos crear nuestra carpeta TodoContext, aunque también en muchos proyectos crean una carpeta context aparte, en donde tendrán todo el contexto de su aplicación.

Es importante crear el contexto con createContext, ya que este nos regresará dos componentes: proveedor y consumidor.

El **hook de contexto** nos ayuda a acceder a datos globales de nuestro contexto, desde cualquier componente hijo, sin tener que pasar estos datos por **props** componente por componente.

Tiene la misma funcionalidad que el consumer de nuestro contexto, pero **useContext** también tiene una manera más sencilla de utilizar y una sintaxis mucho más clara.

Context.Consumer a useContext

TodoCounter.js

import React from 'react';

import { TodoContext } from '../TodoContext';

import './TodoCounter.css';

function TodoCounter() {

const { totalTodos, completedTodos } = React.useContext(TodoContext);

return (

<h2 className="TodoCounter">Has completado {completedTodos} de {totalTodos} TODOs</h2>

);

}

export { TodoCounter };

¿Cuándo se recomienda emplear React Context?

* Estado global
* Tema
* Configuración de la app
* Autenticación de usuario
* Configuración de usuario
* Lenguaje preferido
* Colección de servicios

### Crear custom hooks

GHGH

### Persistir nuestro estado con localStorage

También es posible la persistencia de datos sin necesidad del back-end, utilizando la **API de almacenamiento web**, el **localStorage**, que nos permite almacenar datos localmente en el navegador, que persistirán incluso si el usuario recarga la página o cierra el navegador.

Nos permite guardar datos persistentes en el navegador del usuario, que podremos acceder, modificar y hasta eliminar, para esto **localStorage** tiene varios métodos.

* Guardar datos: setItem(nombre, dato)
* Acceder a datos: getItem(nombre)
* Borrar un dato: removeItem(nombre)
* Eliminar todos los datos: clear(nombre)

Es muy importante saber que **localStorage** solamente puede guardar texto, no objetos, arreglos, números, **solo strings** para esto podemos utilizar unos métodos de **JSON**:

* Convertir a texto: JSON.stringify()
* Convertir a JavaScript: JSON.parse()

const localStorageTodos = localStorage.getItem('TODOS\_V1');

let parsedTodos;

if (!localStorageTodos) {

// Si el usuario es nuevo no existe un item en localStorage, por lo tanto guardamos uno con un array vacío

localStorage.setItem('TODOS\_V1', JSON.stringify([]));

parsedTodos = [];

} else {

// Si existen TODOs en el localStorage los regresamos como nuestros todos

parsedTodos = JSON.parse(localStorageTodos);

}

El objetivo de un custom hook es **reutilizar código**, entonces este hook debería poder funcionar para guardar cualquier tipo de dato en el localStorage.

Primero necesitamos analizar que **parámetros** necesita tener nuestro custom hook:

1. Un nombre para el item en nuestro localStorage.
2. Un estado inicial

También tenemos que regresar algunos datos para que nuestro hook sea funcional:

1. Los datos actuales de nuestro ítem en el localStorage.
2. Una función para actualizar los datos de este ítem.

### Simular una petición a una API, para así manejar el estado de carga y también manejar errores en su caso

CVF

### Crear portales para teletransportar componentes

Los portales nos permiten **teletransportar componentes** a otro nodo de HTML, y seguir comunicándose con otros componentes como si estuviera en el mismo nodo.

Se emplean en ocasiones donde los estilos CSS restringen los elementos. Por ejemplo, problemas de apilamiento z-index y desbordamiento overflow.

**¿Para qué podemos usarlos?**

* Modales
* Tooltips
* Menús flotantes
* Widgets

// Es el mismo import para las versiones mayores o iguales a ReactV18

import ReactDOM from "react-dom";

ReactDOM.createPortal(componente, contenedor)

Creando nuestro modal

Lo primero que necesitamos es crear otro div dentro del body en nuestro HTML.

<div id="modal"></div>

Una vez tenemos nuestro div exclusivo para nuestro modal ya podemos trabajar con React:

Modal/index.js

Crearemos un componente Modal:

import React from 'react';

// Necesitamos ReactDOM para renderizar nuestro modal en el DOM

import ReactDOM from 'react-dom';

import './Modal.css';

function Modal({ children }) {

// ReactDom tiene este método para crear portales

return ReactDOM.createPortal(

<div className="ModalBackground">

{children}

</div>,

document.getElementById('modal')

);

}

export { Modal };

### Deploy con GitHub Pages

Para instalar esta dependencia en nuestro proyecto, necesitamos ejecutar este comando en la terminal, dentro de la **carpeta root** en la que está nuestro proyecto: npm install --save-dev gh-pages

Ahora solo necesitamos ejecutar nuestros scripts: npm run deploy.