React the Complete Guide

Para crear un proyecto utilizamos los comandos:

**npx create-react-app my-app**

**cd my-app**

**npm start**

**SEC-4: REACT STATE & WORKING with Events**

**WORKING with “STATE”**

Primero importamos el **useState**:

import {useState} from "react";

Luego dentro del **componente**, **pero por fuera del return** lo llamamos. El primer elemento es el valor actual con el que lo inicializamos, y el segundo elemento es una function que actualiza el primer elemento:

function Expenseitem(props) {

  const [title, setTitle] = useState(props.title);

  const clickHandled = () => {

    setTitle("Updated");

  }

  return (

    <Card className="expense-item">

      <ExpenseDate date={props.date}/>

      <div className="expense-item\_\_description">

        <h2>{title}</h2>

        <div className="expense-item\_\_price">${props.amount}</div>

      </div>

      <button onClick={clickHandled}>Change title</button>

    </Card>

  );

}

**ADDING TWO-WAY BINDING**

En la función para la etiqueta **form** seteamos al final los valores nuevamente en un string vacío, así al enviarse el formulario se puede volver a completar sin tener que borrar uno mismo la información puesta con anterioridad. Y en los input le pasamos al value el primer elemento del useState:

const [enteredTitle, setEnteredTitle] = useState("");

const submitHandler = (e) => {

        e.preventDefault();

        const expenseData = {

            title: enteredTitle,

            amount: Number(enteredAmount),

            date: new Date(enteredDate),

        }

        console.log(expenseData);

        setEnteredTitle("");

        setEnteredAmount("");

        setEnteredDate("");

    }

return (

      <form onSubmit={submitHandler}>

        <div className="new-expense\_\_controls">

          <div className="new-expense\_\_control">

              <label>Title:</label>

              <input

                type="text"

                value={enteredTitle}

                onChange={titleChangeHandler}/>

          </div>

        </div>

**SEC-9: WORKING with FRAGMENTS, PORTALS & REFS**

**REACT FRAGMENTS**

Se utiliza como un wrapper, es decir, no renderiza ningún elemento HTML del DOM, solo rellena el requerimiento de JSX para renderizar un solo componente.

Podemos hacerlo de dos formas:

* Poniendo una etiqueta vacía.

return (

    <>

      <AddUser onAddUser={addUserHandler}/>

      <UserList users={usersList}/>

    </>

  );

* O utilizando **React.Fragment**:

return (

    <React.Fragment>

      <AddUser onAddUser={addUserHandler}/>

      <UserList users={usersList}/>

    </React.Fragment>

  );

**WORKING with PORTALS**

Para utilizar los **portals** primero debemos ir a la carpeta **public/index.html** y escribir los div con un id para identificarlos: agregué el div backdrop-root y overlay-root.

<body>

    <noscript>You need to enable JavaScript to run this app.</noscript>

    <div id="backdrop-root"></div>

    <div id="overlay-root"></div>

    <div id="root"></div>

</body>

Luego, en el componente ErrorModal, creamos dos nuevas funciones, una que contiene el backdrop y otro el modal:

function Backdrop(props) {

  return <div className="backdrop" onClick={props.onConfirm} />;

}

function ModalOverlay(props) {

  return (

    <Card className={"modal"}>

      <header className="header">

        <h2>{props.title}</h2>

      </header>

      <div className="content">

        <p>{props.message}</p>

      </div>

      <footer className="actions">

        <Button onClick={props.onConfirm}>Close</Button>

      </footer>

    </Card>

  );

}

A continuación importamos **ReactDOM from “react-dom”** y lo utilizamos para crear un portal en la función de ErrorModal, mediante el método **createPortal()**, el cual recibe dos parámetros, el primero es el React node que queremos renderizar, por ejemplo **<Backdrop/>**, y el segundo parámetro es un apuntador al conteiner en el DOM real donde estos elementos deberían ser renderizados, el cual escribimos en el index.html.

function ErrorModal(props) {

  return (

    <React.Fragment>

      {ReactDOM.createPortal(

        <Backdrop onConfirm={props.onConfirm} />,

        document.getElementById("backdrop-root")

      )}

      {ReactDOM.createPortal(

        <ModalOverlay

          title={props.title}

          message={props.message}

          onConfirm={props.onConfirm}

        />,

        document.getElementById("overlay-root")

      )}

    </React.Fragment>

  );

}

**WORKING with REFS**

Lo utilizamos para conectar un elemento HTML con el prop **ref**. Primero importamos el **useRef**:

import { useRef } from "react";

Luego dentro de la función creamos una constante que va a ser la prop de **ref** que le pasemos y luego dentro del elemento html pasamos esa prop:

function AddUser(props) {

  const nameImputRef = useRef();

  const ageImputRef = useRef();

...

<input

   type="text"

   id="username"

   ref={nameImputRef}

/>

<input

   type="number"

    id="age"

    ref={ageImputRef}

/>

Ahora con esta conexión nosotros podemos acceder al valor que tenga ese input cuando el usuario lo escriba, por lo que podemos eliminar los setState. Todo el código quedaría de la siguiente manera:

import { useState, useRef } from "react";

import "./AddUser.css";

// Components

import Card from "../UI/Card";

import Button from "../UI/Button";

import ErrorModal from "../UI/ErrorModal";

import Wrapper from "../Helpers/Wrapper";

function AddUser(props) {

  const nameImputRef = useRef();

  const ageImputRef = useRef();

  const [error, setError] = useState();

  const addUserHandler = (e) => {

    e.preventDefault();

    const enteredName = nameImputRef.current.value;

    const enteredUserAge = ageImputRef.current.value;

    if (enteredName.trim().length === 0 || enteredUserAge.trim().length === 0) {

      setError({title: "Invalid input", message: "Please enter a valid name and age"})

      return;

    }

    if (Number(enteredUserAge) < 1) {

      setError({title: "Invalid age", message: "Please enter a valid age greater than 0"})

      return;

    }

    props.onAddUser(enteredName, enteredUserAge);

    nameImputRef.current.value = "";

    ageImputRef.current.value = "";

  };

  const errorHandler = () => {

    setError(null);

  }

  return (

    <Wrapper>

      {error && <ErrorModal title={error.title} message={error.message} onConfirm={errorHandler}/> }

      <Card className={"input"}>

        <form onSubmit={addUserHandler}>

          <label htmlFor="username">Username</label>

          <input

            type="text"

            id="username"

            ref={nameImputRef}

          />

          <label htmlFor="age">Age (Years)</label>

          <input

            type="number"

            id="age"

            ref={ageImputRef}

          />

          <Button type="submit">Add User</Button>

        </form>

      </Card>

    </Wrapper>

  );

}

export default AddUser;

**SEC-10: HANDLING SIDE EFFECTS, USING REDUCERS & USING THE CONTEXT API**

**WHAT ARE “SIDE EFFECTS” & INTRODUCING useEffect**

useEffect( () => { … }, [ dependecies ] );

El hook **useEffect()** es llamado con dos argumentos: el primero es una **function**, que debería ser ejecutada después de la evaluación de todo el componente SI las dependencias especificadas cambian.

La **dependencias especificadas** son el segundo argumento, el cual es un array. Cuando una de estas dependencias cambie, la función corre otra vez el programa.

**USING THE useEffect() HOOK**

Primero lo importamos desde react y luego en el siguiente ejemplo es aplicado a un localStorage para mantener iniciada la sesión de usuario:

import React, { useState, useEffect } from "react";

import Login from "./components/Login/Login";

import Home from "./components/Home/Home";

import MainHeader from "./components/MainHeader/MainHeader";

function App() {

  const [isLoggedIn, setIsLoggedIn] = useState(false);

  useEffect(() => {

    const storeUser = localStorage.getItem("isLoggedIn");

    if (storeUser === "1") {

      setIsLoggedIn(true);

    }

  }, []);

  const loginHandler = (email, password) => {

    // We should of course check email and password

    // But it's just a dummy/ demo anyways

    localStorage.setItem("isLoggedIn", "1");

    setIsLoggedIn(true);

  };

  const logoutHandler = () => {

    localStorage.removeItem("isLoggedIn")

    setIsLoggedIn(false);

  };

  return (

    <React.Fragment>

      <MainHeader isAuthenticated={isLoggedIn} onLogout={logoutHandler} />

      <main>

        {!isLoggedIn && <Login onLogin={loginHandler} />}

        {isLoggedIn && <Home onLogout={logoutHandler} />}

      </main>

    </React.Fragment>

  );

}

export default App;

**useEffect & DEPENDENCIES**

const Login = (props) => {

  const [enteredEmail, setEnteredEmail] = useState('');

  const [emailIsValid, setEmailIsValid] = useState();

  const [enteredPassword, setEnteredPassword] = useState('');

  const [passwordIsValid, setPasswordIsValid] = useState();

  const [formIsValid, setFormIsValid] = useState(false);

  useEffect(() => {

    setFormIsValid(

      enteredEmail.includes('@') && enteredPassword.trim().length > 6

    );

  }, [enteredEmail, enteredPassword])

  const emailChangeHandler = (event) => {

    setEnteredEmail(event.target.value);

  };

  const passwordChangeHandler = (event) => {

    setEnteredPassword(event.target.value);

  };

  const validateEmailHandler = () => {

    setEmailIsValid(enteredEmail.includes('@'));

  };

  const validatePasswordHandler = () => {

    setPasswordIsValid(enteredPassword.trim().length > 6);

  };

  const submitHandler = (event) => {

    event.preventDefault();

    props.onLogin(enteredEmail, enteredPassword);

  };

...

**WHAT NOT TO ADD AS DEPENDENCIES**

No debemos agregar:

* No necesitamos agregar funciones de actualización de estado (como en la ultima sección con setFormatIsValid. React garantiza que esas funciones nunca cambien, por lo tanto, no necesita agregarlas como dependencias (aunque podría).
* Tampoco es necesario agregar APIs o funciones “incorporadas” (built-in), como fetch(), localStorage, etc.
* Tampoco necesita agregar variables o funciones que haya definido fuera sus componentes (por ejemplo, si crea una nueva función auxiliar en un archivo separado): dichas funciones o variables tampoco se crean dentro de una función de componente y, por lo tanto, cambiarlos no afectará a sus componentes (los componentes no se volverán a evaluar si tales variables o funciones cambian y viceversa).

**USING THE useEFFECT CLEANUP FUNCTION**

Esta función se va a ejecutar al ser actualizado el useEffect, no se ejecutará en el primer renderizado:

const Login = (props) => {

  const [enteredEmail, setEnteredEmail] = useState("");

  const [emailIsValid, setEmailIsValid] = useState();

  const [enteredPassword, setEnteredPassword] = useState("");

  const [passwordIsValid, setPasswordIsValid] = useState();

  const [formIsValid, setFormIsValid] = useState(false);

  useEffect(() => {

    const identifier = setTimeout(() => {

      console.log("Checking form validity");

      setFormIsValid(

        enteredEmail.includes("@") && enteredPassword.trim().length > 6

      );

    }, 700);

    return () => {

      console.log("CLEANUP");

      clearTimeout(identifier);

    };

  }, [enteredEmail, enteredPassword]);

**INTRODUCING useReducer & REDUCERS IN GENERAL**

useReducer se utiliza para manejar estados más complejos que con useState. No debemos actualizar un estado dependiendo de otro estado, pero si debemos hacerlo, useReducer es una buena solución.

**USING THE userReducer() HOOK**

Se utiliza de la siguiente forma:

const [state, dispatchFn] = useReducer(reducerFn, initialState, initFn);

* **State:** es el estado instantáneo que tenemos por default.
* **DispatchFn:** es una función que nos permite actualizar el state. Esta función en vez de solo configurar el valor del state, enviará una acción. Esta acción será consumida por el primer argumento que le pasemos al useReducer, que será **reducerFn**.
* **ReducerFn:** esta función contiene el state previo automáticamente y la función la cual se dispara automáticamente una vez que la acción fue enviada. Recibe el estado previo y devuelve el nuevo estado.
* **InitialState:** el state inicial.
* **InitFn:** la función inicial que debería correr el state inicial.

import React, { useState, useEffect, useReducer } from "react";

import Card from "../UI/Card/Card";

import classes from "./Login.module.css";

import Button from "../UI/Button/Button";

const emailReducer = (state, actions) => {

  if (actions.type === "USER\_INPUT") {

    return { value: actions.val, isValid: actions.val.includes("@") };

  }

  if (actions.type = "INPUT\_BLUR") {

    return { value: state.value, isValid: state.value.includes("@") };

  }

  return { value: "", isValid: false };

};

const passwordReducer = (state, actions) => {

  if (actions.type === "PASSWORD\_INPUT") {

    return { value: actions.val, isValid: actions.val.trim().length > 6};

  }

  if (actions.type === "PASSWORD\_BLUR") {

    return { value: state.value, isValid: state.value.trim().length > 6};

  }

  return { value: ""};

}

const Login = (props) => {

  // const [enteredEmail, setEnteredEmail] = useState("");

  // const [emailIsValid, setEmailIsValid] = useState();

  // const [enteredPassword, setEnteredPassword] = useState("");

  // const [passwordIsValid, setPasswordIsValid] = useState();

  const [formIsValid, setFormIsValid] = useState(false);

  const [emailState, dispatchEmail] = useReducer(emailReducer, { value: "", isValid: null });

  const [passwordState, dispatchPassword] = useReducer(passwordReducer, { value: "", isValid: null})

  // useEffect(() => {

  //   console.log('EFFECT RUNNING');

  //   return () => {

  //     console.log('EFFECT CLEANUP');

  //   };

  // }, []);

  const {isValid: emailIsValid} = emailState;

  const {isValid: passwordIsValid} = passwordState;

  useEffect(() => {

    const identifier = setTimeout(() => {

      console.log('Checking form validity!');

      setFormIsValid(

        emailIsValid && passwordIsValid

      );

    }, 500);

    return () => {

      console.log('CLEANUP');

      clearTimeout(identifier);

    };

  }, [emailIsValid, passwordIsValid]);

  const emailChangeHandler = (event) => {

    dispatchEmail({type: "USER\_INPUT", val: event.target.value});

    // setFormIsValid(

    //   event.target.value.includes("@") && passwordState.isValid

    // );

  };

  const passwordChangeHandler = (event) => {

    // setEnteredPassword(event.target.value);

    dispatchPassword({type: "PASSWORD\_INPUT", val: event.target.value})

    // setFormIsValid(

    //   emailState.isValid && event.target.value.trim().length > 6

    // );

  };

  const validateEmailHandler = () => {

    // setEmailIsValid(emailState.isValid);

    dispatchEmail({type: "INPUT\_BLUR"});

  };

  const validatePasswordHandler = () => {

    // setPasswordIsValid(passwordState.value.trim().length > 6);

    dispatchPassword({type: "PASSWORD\_BLUR"})

  };

  const submitHandler = (event) => {

    event.preventDefault();

    props.onLogin(emailState.value, passwordState.value);

  };

  return (

    <Card className={classes.login}>

      <form onSubmit={submitHandler}>

        <div

          className={`${classes.control} ${

            emailState.isValid === false ? classes.invalid : ""

          }`}

        >

          <label htmlFor="email">E-Mail</label>

          <input

            type="email"

            id="email"

            value={emailState.value}

            onChange={emailChangeHandler}

            onBlur={validateEmailHandler}

          />

        </div>

        <div

          className={`${classes.control} ${

            passwordState.isValid === false ? classes.invalid : ""

          }`}

        >

          <label htmlFor="password">Password</label>

          <input

            type="password"

            id="password"

            value={passwordState.value}

            onChange={passwordChangeHandler}

            onBlur={validatePasswordHandler}

          />

        </div>

        <div className={classes.actions}>

          <Button type="submit" className={classes.btn} disabled={!formIsValid}>

            Login

          </Button>

        </div>

      </form>

    </Card>

  );

};

export default Login;

**REACT CONTEXT**

Lo utilizamos para crear una especie de componente que contiene información y no producir una **chain props**.

En la carpeta **src** creamos una carpeta llamada **context**, en la cual contendrá el archivo **.js** que queramos que sea nuestro context:

import React from "react";

const AuthContext = React.createContext({

    isLoggedIn: false

});

export default AuthContext;

Luego en el archivo **App.js** debemos wrappear todo el return con este context, utilizando **.Provider**. Esto si es que necesitamos la información en todos los componentes (en este caso si):

return (

    <AuthContext.Provider

      value={{

        isLoggedIn: isLoggedIn,

      }}

    >

      <MainHeader onLogout={logoutHandler} />

      <main>

        {!isLoggedIn && <Login onLogin={loginHandler} />}

        {isLoggedIn && <Home onLogout={logoutHandler} />}

      </main>

    </AuthContext.Provider>

  );

A continuación, en el componente al cual le pasamos props a través del **Mainheader**, es decir, el componente **Navigation**, es aquí donde vamos a utilizar la información del **context**, a traves del hook **useContext**:

import React, {useContext} from "react";

import classes from "./Navigation.module.css";

import AuthContext from "../../context/auth-context";

const Navigation = (props) => {

  const ctx = useContext(AuthContext);

  return (

    <nav className={classes.nav}>

      <ul>

        {ctx.isLoggedIn && (

          <li>

            <a href="/">Users</a>

          </li>

        )}

        {ctx.isLoggedIn && (

          <li>

            <a href="/">Admin</a>

          </li>

        )}

        {ctx.isLoggedIn && (

          <li>

            <button onClick={props.onLogout}>Logout</button>

          </li>

        )}

      </ul>

    </nav>

  );

};

export default Navigation;

**SEC-12: A LOOK BEHIND THE SCENES OF REACT**

**REACT.MEMO()**

Se utiliza en el export de un componente para evitar que se re-evalue. Solo se revalúa si las props cambiaron. Pero solo funciona con valores primitivos, es decir si las props son objetos, arrays, valores no primitivos, no funciona.

Para corregir esto utilizamos el hook **useCallback()**.

**USECALLBACK()**

Este hook nos permite almacenar una función en las ejecuciones del componente, la cual no debe ser recreada con cada ejecución.

Lo importamos y utilizamos en la función que queramos, envolviendo todo el resto de la función dentro del useCallback, y como segundo parámetro va un array de dependencias al igual que en el useEffect:

**const miFuncion = useCallback(() => {}, [])**

**USEMEMO()**

Nos permite memorizar cualquier tipo de dato que queramos. El **primer parámetro** que utiliza es una función la cual retorna lo que queramos almacenar, y el **segundo** **parámetro** es un array de dependencias.

**SEC-14: SENDING HTTP REQUEST**

**SENDING A POST REQUEST**

Creamos una base de datos con **Firebase**.

Luego en el **fetch** de nuestro código pasamos como primer parámetro la **URL** y como segundo parámetro un objeto con las keys **method, body y headers:**

async function addMovieHandler(movie) {

    const response = await fetch('https://http-requestfb-default-rtdb.firebaseio.com/movies.json', {

      method: "POST",

      body: JSON.stringify(movie),

      headers: {

        "Content-Type": "application/json"

      }

    });

    const data = await response.json();

    console.log(data);

}

**SEC-15: BUILDING CUSTOM REACT HOOKS**

Los custom hooks se utilizan para no repetir código similar que están en distintos componentes.

**CREATING A CUSTOM HOOK**

Primero creamos dentro de la carpeta src una carpeta llamada **hooks**, luego le damos el nombre que queramos al archivo, PERO a la hora de escribir la función si o si el nombre debe empezar con **use**, luego escribimos el código que se repite en los componentes que son parecidos:

import { useState, useEffect } from 'react';

const useCounter = () => {

  const [counter, setCounter] = useState(0);

  useEffect(() => {

    const interval = setInterval(() => {

      setCounter((prevCounter) => prevCounter + 1);

    }, 1000);

    return () => clearInterval(interval);

  }, []);

  return counter;

};

export default useCounter;

**USING CUSTOM HOOKS**

Luego debemos importar este hook en los componentes como cualquier otro hook, y borramos toda la lógica de ese componente ya que ahora la manejamos desde el custom hook:

import useCounter from '../hooks/use-counter';

const ForwardCounter = () => {

  const counter = useCounter();

  return <Card>{counter}</Card>;

};

Mientras tanto en el custom hook debemos hacer un **return counter**, que es la variable que importamos en el ForwarCounter (mirar la segunda imagen de arriba).

**CONFIGURING CUSTOM HOOKS**

Ahora para poder utilizar está lógica en dos componentes distintos pero que comparten la misma lógica, nada más que uno cuenta para arriba y otro para abajo, hacemos lo siguiente:

Como parámetro en el custom hook pasamos un identificador, parecido a un estado, para identificar si hacer cierta lógica u otra:

import { useState, useEffect } from 'react';

const useCounter = (forwards = true) => {

  const [counter, setCounter] = useState(0);

  useEffect(() => {

    const interval = setInterval(() => {

      if (forwards) {

        setCounter((prevCounter) => prevCounter + 1);

      } else {

        setCounter((prevCounter) => prevCounter - 1);

      }

    }, 1000);

    return () => clearInterval(interval);

  }, [forwards]);

  return counter;

};

export default useCounter;

En este caso pasamos como identificador **forwards** en **true**, y luego hacemos un **if statement**. Si forwards es true hacemos que los números cuenten hacia arriba y si es false, cuentan hacia abajo.

Hay que tener en cuenta que ahora este parámetro **forwards** hay que pasarlo como una **dependencia** ya que depende su valor se va a realizar cierta lógica.

Ahora para decirle a React que la lógica que queremos cuando **forwards** este en false se ejecute tenemos que pasar en el componente que cuenta los números hacia atrás, la variable **counter**, con el parámetro **false**:

import Card from './Card';

import useCounter from '../hooks/use-counter';

const BackwardCounter = () => {

  const counter = useCounter(false);

  return <Card>{counter}</Card>;

};

export default BackwardCounter;

**SEC-16: WORKING WITH FORMS & USER INPUT**

import { useState } from "react";

const SimpleInput = (props) => {

const [enteredName, setEnteredName] = useState("");

const [enteredNameTouched, setEnteredNameTouched] = useState(false);

const [enteredEmail, setEnteredEmail] = useState('');

const [enteredEmailTouched, setEnteredEmailTouched] = useState(false);

const enteredNameIsValid = enteredName.trim() !== '';

const nameInputIsInvalid = !enteredNameIsValid && enteredNameTouched;

const enteredEmailIsValid = enteredEmail.trim() !== '' && enteredEmail.includes('@');

const emailInputIsInvalid = !enteredEmailIsValid && enteredEmailTouched;

let formIsValid = false

if (enteredNameIsValid && enteredEmailIsValid) {

formIsValid=true;

}

const nameInputChangeHandler = (e) => {

setEnteredName(e.target.value);

};

const emailInputChangeHandler = (e) => {

setEnteredEmail(e.target.value);

}

const nameInputBlurHandler = (e) => {

setEnteredNameTouched(true);

};

const emailInputBlurHandler = (e) => {

setEnteredEmailTouched(true);

}

const formSubbmited = (e) => {

e.preventDefault();

setEnteredNameTouched(true);

setEnteredEmailTouched(true);

if (!enteredNameIsValid || !enteredEmailIsValid) {

return;

}

console.log(enteredName, "State");

console.log(enteredEmail, "State");

setEnteredName("");

setEnteredEmail("");

setEnteredNameTouched(false);

setEnteredEmailTouched(false);

};

const inputClasses = nameInputIsInvalid || emailInputIsInvalid

? "form-control invalid"

: "form-control";

return (

<form onSubmit={formSubbmited}>

<div className={inputClasses}>

<label htmlFor="name">Your Name</label>

<input

type="text"

id="name"

onChange={nameInputChangeHandler}

onBlur={nameInputBlurHandler}

value={enteredName}

/>

{nameInputIsInvalid && (

<p className="error-text">Name must not be empty</p>

)}

</div>

<div className={inputClasses}>

<label htmlFor="email">Your Email</label>

<input

type="email"

id="email"

onChange={emailInputChangeHandler}

onBlur={emailInputBlurHandler}

value={enteredEmail}

/>

{emailInputIsInvalid && (

<p className="error-text">Email must not be empty and must contain an "@"</p>

)}

</div>

<div className="form-actions">

<button disabled={!formIsValid}>Submit</button>

</div>

</form>

);

};

export default SimpleInput;