**CRA TEMPLATE**

Las CRA como su nombre lo indica son plantillas para crear Apps de react. Existen muchas en el sitio oficial de CRA, la que nos interesa es la de PWA.

<https://create-react-app.dev/docs/making-a-progressive-web-app/>

*npx create-react-app my-app --template cra-template-pwa*

para esta sección vamos a iniciar una pwa desde el template con el comando

npx create-react-app my-app --template cra-template-pwa

Si se accede al directorio en que se creo la pwa desde el explorador de archivos, podrán verse varios archivos.

Para iniciar, hay que hacer un cd hasta dicho directorio y ejecutar **npm start**

Una vez iniciado, abre un puerto en el localhost donde se corre la PWA.

**ARCHIVOS DE LA PWA DESDE TEMPLATE**

Si abrimos la carpeta de la pwa template en el editor de código, en la carpeta src, además de files como los usados anteriormente, pueden encontrarse archivos de Service Worker y de Register. Como los que se crearon en la primera PWA.

**REGISTRANDO EL SW**

Cuando creamos el template, en el archivo index.js, debería encontrarse una línea de código que ejecuta la funcion

serviceWorkerRegistration.unregister();

con cambiar la funcion por 🡪 register(); estaríamos registrando correctamente el sw

**COMENZANDO PROYECTO CALCULADOR**

Para este proyecto, lo primero que se hace es crear la carpeta components en el directorio **src** y comenzar con el archivo Calculadora.jsx.

Además, se reemplaza todo el archivo de App.js por la construcción de un snippet **rafce** que es un componente simple que retorna un div y se comienzan a agregar elementos desde este.

Inicialmente este es el código de Calculadora.jsx (además de sus import y export):

const Calculadora = () => {

    return (

        <div>

            <label>

                Numero 1: <input type="text" name="" id="" />

            </label>

        </div>

    )

}

Y este el de App.js (sin incluir exports e imports. Recordemos que este elemento se exporta para incluirle en index.js como <App> </App>):

const App = () => {

  return (

    <div>

      <header>

        <h1>Modificando PWA desde templates</h1>

        <p>igual me gustaba el background :(</p>

        <h2>Este proyecto consta de una Calculadora</h2>

        <hr></hr>

        <Calculadora></Calculadora>

      </header>

    </div>

  )

}

El problema o la posibilidad de solución se presenta al tener varios elementos, como el primer y segundo numero, o el span para mostrar la suma, la resta, división, mult,etc.

La idea es manejar un estado, modificarlo desde operaciones y reutilizar componentes, tanto para inputs de numero, como para mostrar resultados.

**REACT DEVTOOLS**

React dev tolos es una extensión para navegadores, que permite visualizar componentes y propiedades en aplicaciones realizadas con react.

Para visualizar las tabs **profiler y Components** hay que inspeccionar y dirigirse a la sección. Estas se insertan como elements, console, etc.

La idea siguiente es generar un componente que controle los <span> </span> que contienen suma, resta, división y multiplicación

**CREANDO EL COMPONENTE RESULTADO**

import React from 'react'

import PropTypes from 'prop-types'

const Resultado = ({operacion, calculo}) => {

    return (

        <div>

            <br />

            <span>{operacion}: {calculo}</span>

        </div>

    )

}

Resultado.propTypes = {

    "operacion": PropTypes.string,

    "calculo": PropTypes.number

}

export default Resultado

este es el código del mismo, consta de un componente con props que presenta un nombre de operación y un resultado

cuando se le invoca desde el componente padre, se le pasa un valor a cada uno de estos props.

Por ejemplo:

<Resultado operacion="suma" calculo= {suma}></Resultado>

**CAPTURANDO VALORES DE UN INPUT:**

Para esto se agrega un estado al componente input. Antes se manejaba un estado en el componente Calculadora.jsx que manejaba el resultado de la operación.

Ahora se agrega uno para controlar cuando el valor en el input cambie, para así manejar el valor para realizar operaciones.

Para esto, es necesario agregar, además del estado y la funcion que setee el estado, un event listener en el input. Esto se hace en el evento **onChange** agregado como escuchador en línea de HTML y se le asocia una funcion.  
Por convención esta funcion comienza con el nombre **handle**, se llamará handleChange

Este es el código que se introduce en NumberInput.jsx como useState

    const [numeros, setNumeros] = useState(0);

Luego, en el componente label se introduce el listener con la funcion que manejará ese cambio

  <label>

      {name}: <input onChange={handleChange} type="text" name="" id="" />

  </label>

La funcion se define de la forma:

    const handleChange = (e) => {

        setNumeros(e.target.value);

    };

Para estas alturas, todo es arrow function, destructuring y demás temas vistos en el repaso.

Este e.target.value detecta lo que hay en el input al momento de modificarse

**MODIFICANDO EL PROYECTO**

Como se trabajaba hasta el momento, en el componente InputNumber.jsx se declaraba un input, y se llamaba a su etiqueta <NumberInput/> dos veces desde el componente Calculadora.jsx

Ahora, se cambió para que en el componente NumberInput.jsx se incluyan dos inputs distintos, y solo se llame una vez desde Calculadora.jsx

Ahora, el problema sigue apareciendo a la hora de modificar los estados.

Anteriormente se había cambiado para manejar un solo numero, pero probaremos nuevamente con manejar un objeto.

Para que esto funcione correctamente, también es necesario declarar 2 “manejadores” distintos. Uno para el input Numero1 y otro para el input Numero2. De ahí a que sea necesario tener los 2 declarados en el mismo componente.

Este sería el return del NumberInput.jsx

    return (

        <>

            <label>

                Numero 1: <input onChange={handleChange} value={numeros.numero1} type="number" name="" id="" />

            </label>

            <label>

                Numero 2: <input onChange={handleChange2} value={numeros.numero2} type="number" name="" id="" />

            </label>

        </>

    )

Y este el código que maneja cada funcion

    const handleChange = (e) => {

        setNumeros({

            numero1: parseFloat(e.target.value),

            numero2: numeros.numero2

        });

    };

    const handleChange2 = (e) => {

        setNumeros({

            numero1: numeros.numero1,

            numero2: parseFloat(e.target.value)

        });

    };

En la primera se modifica el numero1, obteniendo lo que pase en el evento y se deja como está el numero2

En la segunda funcion, se deja como está el numero 1 y se modifica el numero2

**AUTOMATIZANDO FUNCIONES**

Una de las formas de automatizar las funciones para no tener que manejar una para cada input, es modificando su atributo name. Si en cada etiqueta input insertamos como atributo name el valor de la llave del objeto **números** podremos lograr esto.

Eso debido a que, si por ejemplo, guardo en el atributo name de un input el valor “numero1” y en el otro “numero2” podré declarar una sola funcion, que en lugar de acceder a cada llave explicita, accede a la llave que tenga el nombre que hay en vaor del atributo name.

    const handleChange = (e) => {

        setNumeros({

            ...numeros,

            [e.target.name]: parseFloat(e.target.value),

        });

    };

Este sería el código de la funcion, y en **onChange** se asigna esta funcion, en ambos inputs.

Lo que también puede hacerse para automatizar más todavía el programa es desestructurar el objeto **números** y generar una variable para Numero1 y otra para numero2 para no tener que ir accediendo a números.numero1 o números.numero2 a cada rato.

**MANIPULANDO INFORMACION**

Para la siguiente parte, es necesario “pasar” los componentes Resultado.jsx al componente InputNumber, para así modificar el estado que allí se encuentre y acceder a **numero1** y **numero2.**

Esto se hace cortando las líneas de código

const [suma, setSuma] = useState(15);

            <Resultado operacion="suma" calculo= {suma}></Resultado>

            <Resultado operacion="resta" calculo={suma}></Resultado>

            <Resultado operacion="multiplicacion" calculo={suma}></Resultado>

            <Resultado operacion="division" calculo={suma}></Resultado>

Al archivo Componentes.jsx

**RESOLVIENDO SUMA, RESTA, MULTIPLICACION Y DIVISION**

Para obtener cada uno de los resultados, se procede con una funcion de flecha en la cual se opera con las variables **numero1** y **numero2** (ambas salen del destructuring del estado del InputNumber).

Para poder visualizar correctamente estos resultados, en la prop **calculo** del componente Resultado.jsx se invoca a la funcion correspondiente. Por ejemplo:

<Resultado operacion="suma" calculo= {sumar()}></Resultado>

Se resuelve de esta manera ya que si se intentase modifica el estado desde el evento HanldeChange se presentaría un “delay” al mostrar los números.

Este sería el código final de las operaciones:

    const sumar = () => numero1 + numero2;

    const restar = () => numero1 - numero2;

    const multiplicar = () => numero1 \* numero2;

    const dividir = () => numero1 / numero2;

    <Resultado operacion="suma" calculo= {sumar()}></Resultado>

    <Resultado operacion="resta" calculo={restar()} ></Resultado>

    <Resultado operacion="multiplicacion"  calculo={multiplicar()}></Resultado>

    <Resultado operacion="division" calculo={dividir()}></Resultado>

**HELPERS**

La idea de los helpers, es crear un archivo externo que contenga las funciones, estados o demás recursos de un componente, para dejar código limpio en el archivo del mismo.

Cuando digo código limpio, me refiero solo al código necesario para su funcionamiento, las llamadas a funciones, a props, lo esencial.

Para este proyecto se crea una carpeta llamda helpers y un archivo llamado operaciones.js

En este archivo, se crea una constante llamada operaciones (donde se definen todas ellas) y por ultimo se exporta.

El principio de funcionamiento es retornar los valores en forma de objeto, para asi desestructurarle en el archivo InputNumber.jsx

También es importante destacar que la funcion a importar operaciones debe recibir como parámetros el objeto números, que es el que maneja el estado de ambos inputs, y la funcion setNumeros, para así modificar el estado de los mismos.

Este es el código final del archivo InputNumber.jsx

import React from 'react'

import { useState } from 'react'

import Resultado from './Resultado'

import { operaciones } from '../helpers/funciones'

const NumberInput = ({ valor }) => {

    const [numeros, setNumeros] = useState({

        numero1: 0,

        numero2: 0

    });

    const { handleChange, sumar, restar, multiplicar, dividir, numero1, numero2} = operaciones(numeros, setNumeros);

    return (

        <>

            <label>

                Numero 1: <input onChange={handleChange} value={numero1} type="number" name="numero1" id="" />

            </label>

            <label>

                Numero 2: <input onChange={handleChange} value={numero2} type="number" name="numero2" id="" />

            </label>

            <Resultado operacion="suma" calculo= {sumar()}></Resultado>

            <Resultado operacion="resta" calculo={restar()} ></Resultado>

            <Resultado operacion="multiplicacion"  calculo={multiplicar()}></Resultado>

            <Resultado operacion="division" calculo={dividir()}></Resultado>

        </>

    )

}

export default NumberInput

y este el código final de **operaciones.js**

export const operaciones = (numeros, setNumeros) => {

    const { numero1, numero2 } = numeros;

    const handleChange = (e) => {

        setNumeros({

            ...numeros,

            [e.target.name]: parseFloat(e.target.value),

        });

    };

    const sumar = () => numero1 + numero2;

    const restar = () => numero1 - numero2;

    const multiplicar = () => numero1 \* numero2;

    const dividir = () => numero1 / numero2;

    return { handleChange, sumar, restar, multiplicar, dividir, numero1, numero2 };

}

En síntesis, esto es como si lo que hay dentro de la función estaría declarado dentro del mismo archivo InputNumber.js y se le invocase en forma de funcion para obtener sus funciones y variables.

Es el mismo tipo de modularizado que se usa en los proyectos php realizados anteriormente

**DESPLIEGUE**

Para desplegar la aplicación, puede utilizarse netlify.

Este es el link de la app: <https://wonderful-murdock-5dbb4a.netlify.app/>

Este es mi link de netlify: <https://app.netlify.com/teams/agustinstringa/overview>

En la pestaña **team overview**, en la sección Sites, se puede arrastrar la carpeta de archivos del proyecto.

Para esta app, es necesario ejecutar **npm run build** en el terminal de vs para crear la carpeta “build” en el proyecto. Esa carpeta se “arrastra” a netlify

**CONFIGURANDO LA PWA**

Continuando con el despliegue en netlify, lo siguiente será modificar otras características como el icono.

Para crear los iconos, vuelve a usarse PWA builder iamge generator, y se incorporan al **manifest.json**. Además pueden realizarse otros cambios como el color de theme o de fondo.

Una vez realizadas todas estas, es necesario ejecutar en el terminal **npm run-script build** para volver a generar la carpeta con el proyecto.

Una vez hechas las configuraciones necesarias, es necesario actualizar el proyecto subido a netlify. Para esto, una vez en “la pagina principal” del sitio, hay que dirigirse a la pestaña “deploy” y allí habrá disponible un “drag and drop” como el que se utilizó para subir el sitio inicialmente.