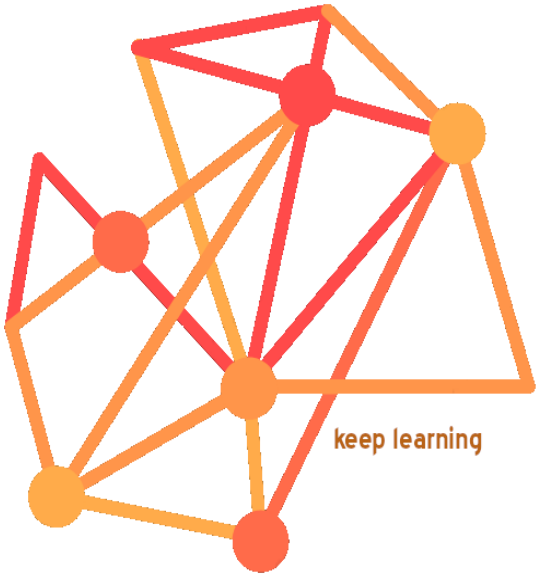
React



Juan Carlos Pérez Rodríguez

Sumario

Introducción

React (también llamada React.js o ReactJS) es una biblioteca Javascript de código abierto diseñada para crear interfaces de usuario con el objetivo de facilitar el desarrollo de aplicaciones en una sola página. Es mantenido por Facebook y la comunidad de software libre. En el proyecto hay más de mil desarrolladores libres. (definición Wikipedia)

React Native es un framework open-source creado por Facebook que se usa para desarrollar aplicaciones Android, IOS, etc Así en lugar de enfocarse al navegador, se hace uso de React en otras plataformas de dispositivos móviles.

React Native llama a las APIs de Android e IOS para hacer el renderizado de la aplicación así hay un “look and feel” de la aplicación React Native como si se hubiera desarrollado en Objective-C (IOS) o Java (Android) Al no usar un webview sino llamar directamente la API nativa es una forma más eficiente que otros sistemas basados en Javascript para desarrollar aplicaciones de dispositivos móviles.

Instalaciones necesarias

visual studio code, node, postman, android studio, git

extensiones para vscode:

Auto Close Tag;

ES7+ React/Redux/React-Native snippets

JSON to TS; Paste JSON as Code ; Typescript importer

(es posible que sea necesario instalar en el sistema operativo xclip)

extensiones de Google Chrome:

React Developer Tools, Redux DevTools

React

Antes de empezar con React Native precisaremos unos conceptos mínimos de React. Procederemos inicialmente a introducirlos:

Para crear y ejecutar una aplicación react llamada: primeraapp (nota: se solicita siempre que se usen minúsculas) podemos usar:

```
npx create-react-app primeraapp

cd primeraapp ; npm start
```

Y si quisiéramos con **typescript**:

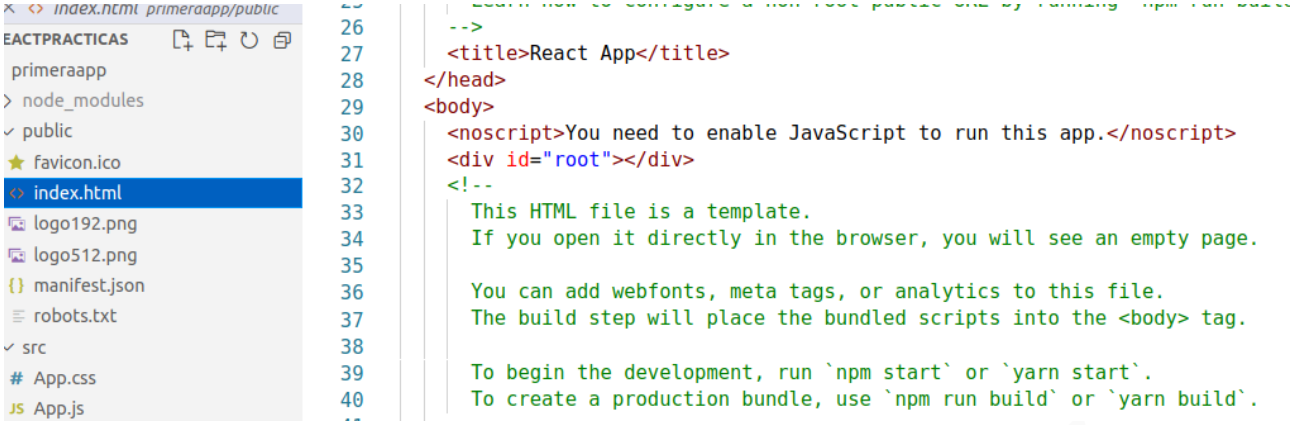
```
npx create-react-app primeraapp --template typescript

cd primeraapp ; npm start
```

Observamos que nos ha creado una estructura que incluye los módulos de node usados, una carpeta pública y una carpera: src Es ahí donde vamos a poner nuestra aplicación realmente

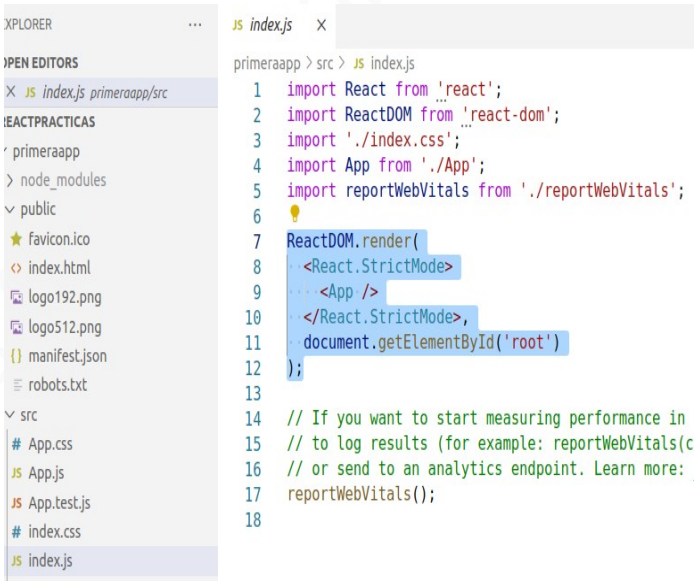
La siguiente imagen ilustra la estructura de carpetas

Sabemos que las aplicaciones suelen empezar en algún index. En este caso miremos el **index.html**:



Si leemos lo que se describe en comentarios XML vemos que nos dice que es una aplicación pura de javascript (no funciona si el navegador no lo tiene habilitado) y realmente estamos ante un fichero vacío. Únicamente tenemos un espacio contenedor: que es el espacio donde se va a renderizar nuestra aplicación

Ahora veamos un fichero más interesante: index.js

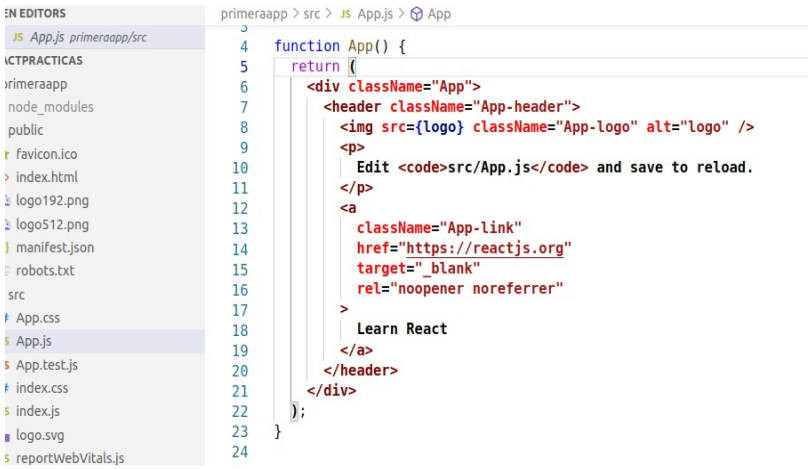


Vemos que hay un document.getElementById(‘root’) que es el que vimos antes en index.html

También vemos que hay una etiqueta: <App />

Esa etiqueta nos está indicando que ahí hay un componente que se llama: App La idea de trabajar por componentes es tener asociado un objeto con parte gráfica (código html y estilos css) y una parte funcional (atributos y funciones en código javascript) No difiere de la idea que tendríamos de un objeto componente en otros lenguajes de programación: textfield en Java Swing, etc

Finalmente miremos el fichero: **App.js** que realmente es el de nuestra aplicación (es el compoente principal: <App /> que vimos en index.js)



Como una primera aproximación vamos a borrar casi todo lo escrito ahí y haremos un hola mundo en el fichero App.js:

```
function App() {
        return (
                <div className="App">
                        <h3> Hola Mundo!</h3>
                </div>
        );
}
export default App;
```

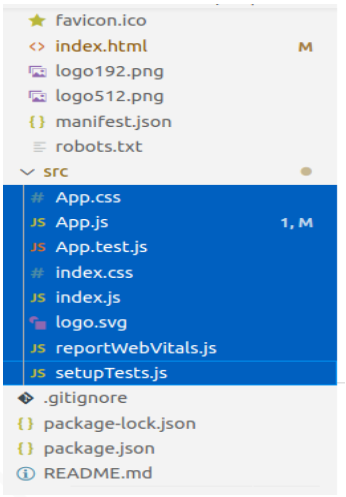
Adicionalmente en index.html vamos a cambiar el title a: <title> hola mundo </title>

Práctica 1: Crear el hola mundo descrito y agrega tu nombre completo al <h3>
(usando npx para crear la app y npm start para arrancarla como se indica en el tema)

Empezando de cero nuestra aplicación react

Vamos a borrar el casi todo el contenido de la carpeta: **src** (recordar que ahí es donde va estar nuestra aplicación) Básicamente dejamos el package.json, .gitignore y el readme.md

En la imagen se muestra seleccionado lo que vamos a borrar:

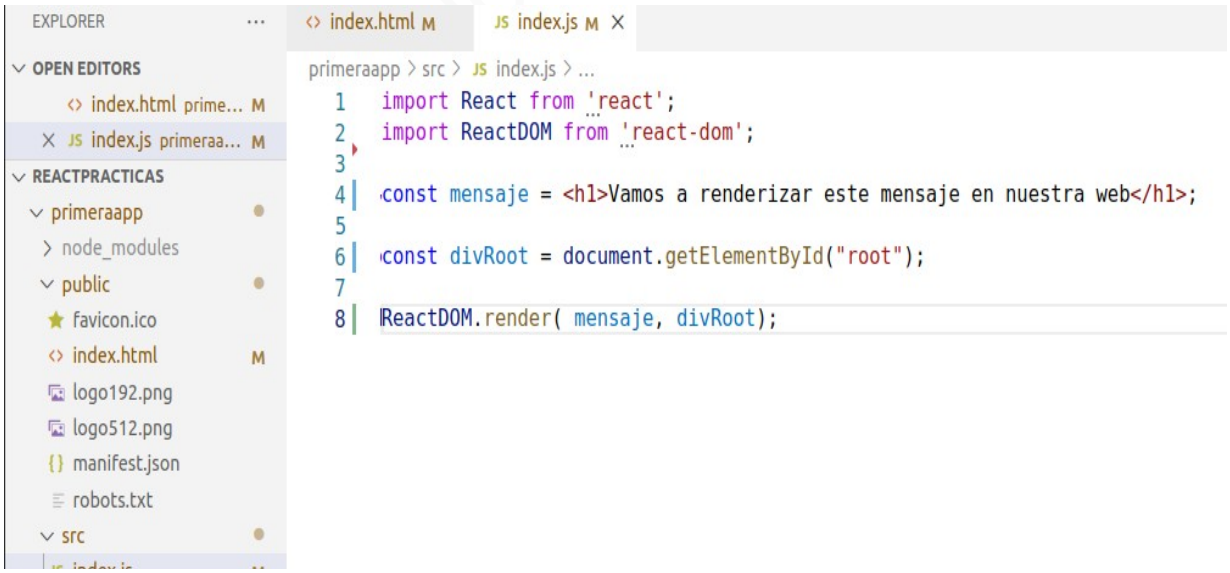


Vamos a crear nuestro propio punto de partida de nuestra app. Para ello hacemos nuestro propio: src/index.js

Ponemos el siguiente texto en index.js: (se muestra en texto copiable y más abajo como imagen)

```
import React from 'react';
import ReactDOM from 'react-dom';// también puede aparecer como: 'react-dom/client'

const mensaje = <h1>Vamos a renderizar este mensaje en nuestra web</h1>;
const divRoot = document.getElementById("root");
ReactDOM.render( mensaje, divRoot);
```



Se han puesto las dos capturas para que observemos que cuando ponemos en index.js:

const divRoot = document.getElementById(“root”)

Estamos cogiendo el elemento con el id=root que está en index.html

Finalmente vemos que se ejecuta: ReactDOM.render(mensaje,divRoot);

Que lo que está diciendo es que nos visualice nuestro “mensaje” en divRoot

Observar que en la constante mensaje no hemos puesto comillas al texto que le asignamos. Queremos que se lo tome como un código html no como un texto

Práctica 2: Realizar lo descrito y tomar captura de pantalla del mensaje en el navegador (recordar que por defecto la web está en el puerto 3000)

Reducir tamaño de nuestra App

Si observamos el tamaño generado en sistema de ficheros de la aplicación observamos que es grande (más de 200MB) Eso es principalmente por los modulos de node (carpeta: node_modules) **Para poder exportar nuestra aplicación podemos eliminar esa carpeta**. Para luego volver a dejarla operativa ejecutamos en un terminal que esté dentro de la carpeta de nuestra app:

```
npm install
```

Creación de componentes y primeros pasos

Vamos a crear un fichero llamado: src/ComponenteApp.js y dentro ponemos:

```
import React from "react";

const ComponenteApp = () => {
        return (
                <h1>Buenos días. Este es nuestro primer componente</h1>
        )
}

export default ComponenteApp;
```

Observar que hemos hecho un import de: React pero no parece que lo estuviéramos usando. Realmente sí, porque es necesario para que nos lo trate como un componente y nos lo renderice

Observar también que en el return NO ponemos lo que devolvemos entre comillas. Queremos que nos lo tome como html (o mejor dicho jsx). De hecho, es mejor que tomemos conciencia del uso de los paréntesis en el return: return ( )

En general, será lo aconsejable siempre, ya que le estamos diciendo que queremos devolver un objeto y nos está agrupando las diferentes líneas que pudiéramos escribir en el return como un único objeto a devolver

Ahora para poder usar el componente recién creado vamos a index.js y lo dejamos así:

```
import React from 'react';
import ReactDOM from 'react-dom';
import ComponenteApp from './ComponenteApp';

const divRoot = document.getElementById("root");

ReactDOM.render( <ComponenteApp />, divRoot);
```

Fijarse en que hemos tenido que importar nuestro componente del otro fichero y que ya cuando estamos renderizando (ReactDOM.render) ya lo estamos usando como si fuera una nueva etiqueta html que tuviera su propio comportamiento: <ComponenteApp />

Fragment: devolución en un return de varios elementos

Cuando queremos que nos devuelva varios elementos nuestro componente (será lo más habitual) , no basta con poner paréntesis en el return: return ( ) sino que adicionalmente los elementos deben estar agrupados en un único objeto.

Primera opción: usando un ** **:

```
import React from "react";

const ComponenteApp = () => {
	return (
		<div>
			<h1>Componente con varias líneas</h1>
			<h4>(hace falta una etiqueta contenedora como: div )</h4>
		</div>
	);
}

export default ComponenteApp;
```

El problema a lo anterior es que se está agregando al html final etiquetas completamente innecesarias, ya que únicamente lo estamos usando para poder hacer un return que contenga el h1 y el h4 a la vez

La solución viene por la etiqueta: **<Fragment>** que es propia de react y así la librería sabe que únicamente queremos agrupar los elementos en un return

El código del return anterior quedaría así:

```
return (
	<Fragment>
		<h1>Componente con varias líneas</h1>
		<h4>(hace falta una etiqueta contenedora )</h4>
	</Fragment>
);
```

Habitualmente veremos la **forma "acortada"** de escribir una etiqueta fragment. Que es : <>

Ahora el return queda:

```
return (
	<>
		<h1>Componente con varias líneas</h1>
		<h4>(hace falta una etiqueta contenedora )</h4>
	</>
);
```

Incluir variables, expresiones en el return de nuestros componentes jsx

Veamos el siguiente código:

```
const ComponenteApp = () => {
	const primos = [2, 3, 5, 7];
	return (
		<>
			<h1>Primeros números primos:</h1>
			<h4>{JSON.stringify(primos)}</h4>
		</>
	);
}
```

Fijémonos primero en las llaves: "{" y "}" esa va a ser la forma habitual en la que le vamos a decir a react que queremos que nos "interprete" el contenido. Luego simplemente vemos una sentencia Javascript: JSON.stringify(primos) que nos está formateando en JSON nuestro array de primos y así poderlo mostrar bien presentado en nuestra página web

Práctica 3: Reproducir el ejemplo anterior, pero en lugar de mostrar números primos en el <h1> dirá: "mis datos:" y en el h4 le habremos pasado un objeto literal JSON con tu nombre, apellidos y estudios que estás realizando

JSX, TSX

JSX permite escribir elementos HTML en JavaScript y ubicarlos en el DOM sin necesidad de usar las sentencias habituales para tal cometido en javascript: createElement() o appendChild(). Al manejar html no tenemos que tratarlo como una string. Veamos un ejemplo:

```
export const PruebaApp = () => {
  const color = 'Verde';
  const frase = <h4>el color de los árboles es: {color}</h4>;
  return (
    <>
      <h1>Actividad react</h1>
      {frase}
    </>
  )
}
```

La función está devolviendo un trozo de html, de alguna forma estamos “ampliando” las posibilidades de javascript para integrar html en nuestro código javascript

Vemos que JSX admite variables que se les asigne directamente html (no está entrecomillado como si fuera un texto):

```
const frase = <h4>el color de los árboles es: {color}</h4>;
```

Si queremos incluir expresiones javascript dentro de JSX usamos las llaves: {}

```
const color = 'Verde';
const frase = <h1>el color de los árboles es: {color}</h1>;
```

TSX es lo mismo que JSX pero usando Typescript

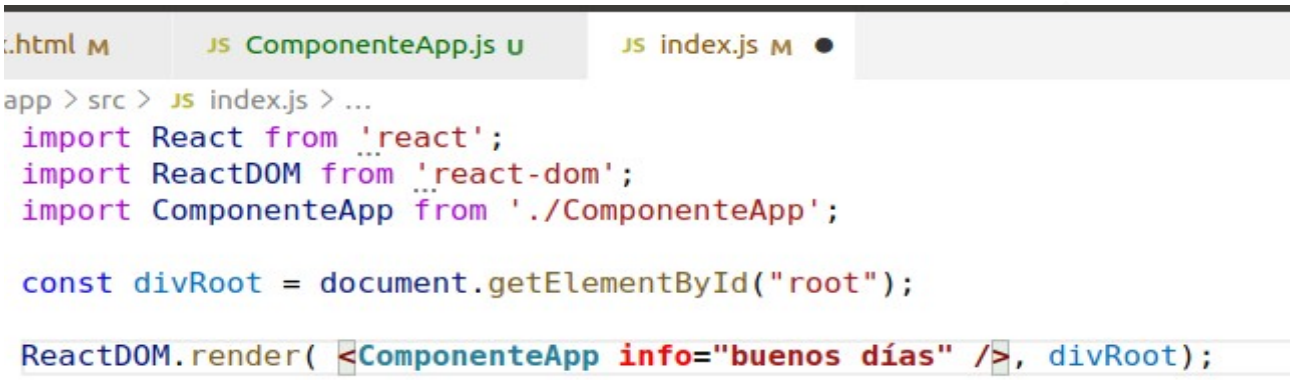
JSX y TSX tienen algunas limitaciones respecto a lo habitual en lenguajes de programación. Por ejemplo, los bucles

Pasar propiedades a un componente y verificarlas con PropTypes

Estamos acostumbrados a pasar información de atributos/propiedades a un elemento html en forma de clave/valor. En el siguiente ejemplo se asocia al atributo type el valor: text

<input type=”text” />

Quisiéramos conseguir un efecto similar en nuestro componente. Pongamos en index.js el paso de una información (pasamos el atributo llamado: info con el valor: “Buenos días”)



Para recibir la información en el componente haremos lo siguiente:

```
import React from "react";
import PropTypes from 'prop-types'
const ComponenteApp = (props) => {
	const primos = [2, 3, 5, 7];
	return (
		<>
			<h1>Primeros números primos:</h1>
			<h4>{JSON.stringify(primos)}</h4>
			<p>Datos recibido en el componente: { props.info}</p>
		</>
	);
}

ComponenteApp.propTypes = {
	info: PropTypes.string.isRequired
}

export default ComponenteApp;
```

Hemos destacado lo nuevo y relevante:

Vemos que se recibe un objeto en la función llamado: props Esto es muy típico. Hace referencia a todo el objeto componente de propiedades que podemos recibir. Vemos que podemos acceder a la información que enviamos (info = “Buenos días”) mediante: props.info

Con lo anterior ya tendríamos suficiente para recibir la información que nos quieran pasar al componente mediante atributos. Pero adicionalmente hemos destacado en el código anterior lo siguiente:

```
ComponenteApp.propTypes = {
        info: PropTypes.string.isRequired
}
```

Como vemos le asignamos un propTypes a ComponenteApp Eso es un validador de la información que nos envíen. Si observamos: info: PropTypes.string.isRequired nos dice que el atributo info debe ser de tipo: string y es exigido que sea enviado: isRequired

Para utilizar PropTypes haremos el import correspondiente:

```
import PropTypes from 'prop-types'
```

Práctica 4: Reproducir el ejemplo anterior, pero cambiando que los atributos que reciba sean: num1 y num2 y lo que muestre es:
La suma de num1 y num2 es: num1 + num2
(donde num1 y num2 serían los datos que recibiera el componente)

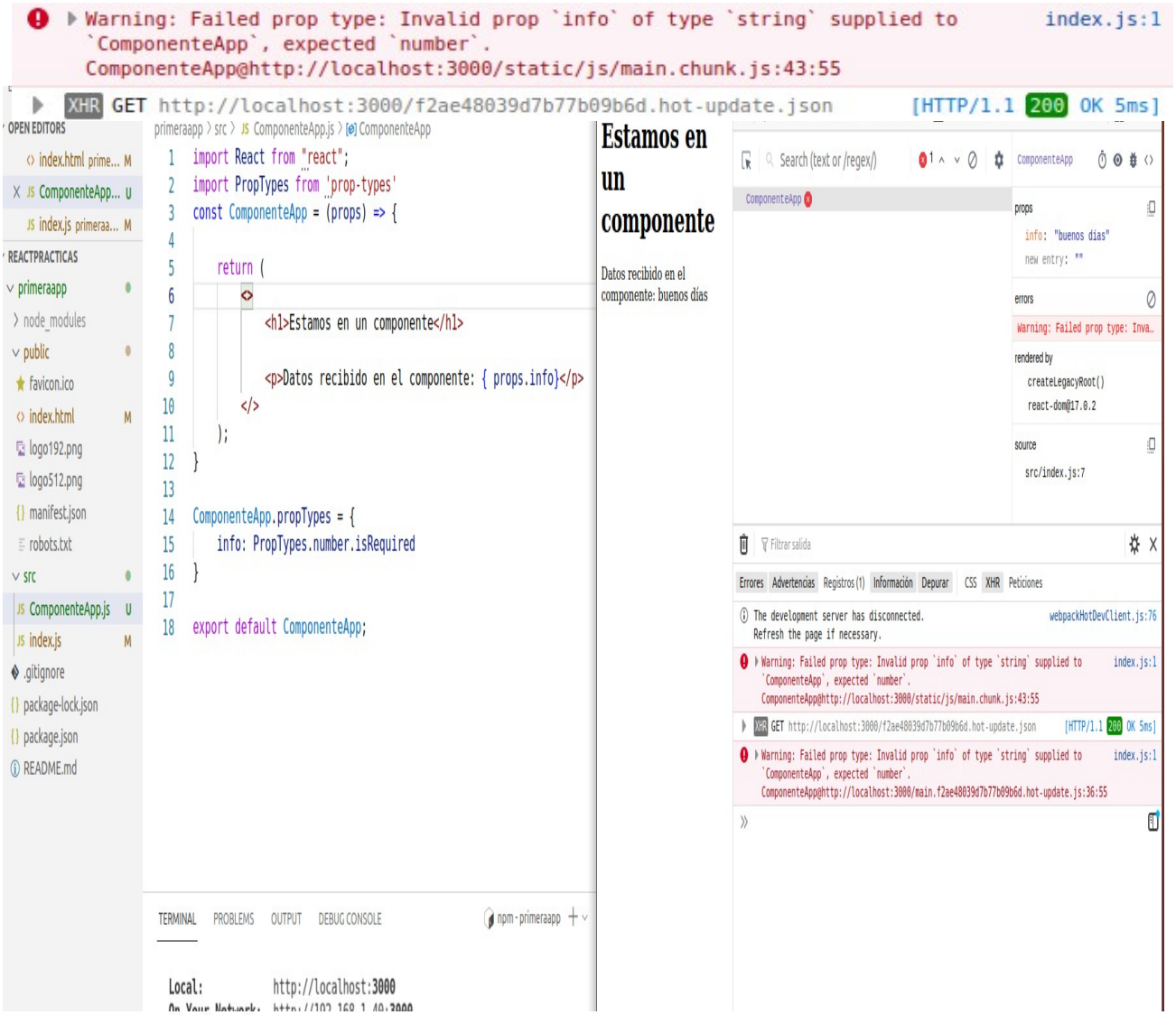
Podemos validar que efectivamente nos lanza un mensaje de aviso mediante la consola del navegador (**debemos tener instaladas las extensiones que hemos dicho al comienzo del tema**)

En la siguiente captura de pantalla se ha establecido que el atributo info debe ser de tipo numérico y obligatorio: **PropTypes.number.isRequired**

Y sin embargo le estamos enviando una string: **<ComponenteApp info=“Buenos días” />**

Así el navegador nos muestra el aviso (observar los warning del navegador):

Veamos el aviso mejor:



Vemos que nos informa que la propiedad info que se ha enviado es de tipo string y lo que se esperaba es de tipo number

Nota de macro para visual studio code

Con las extensiones que hemos declarado al comienzo del tema (snippets) podemos crear un nuevo componente que incluya PropTypes con el snippet: **rafcp**

Pasar propiedades a un componente con typescript

¿ qué diferencia habría si queremos pasar propiedades con typescript ?

Bien, vamos a basarnos en los snippets que hay disponibles y vamos a crear un functional component con typescript: **tsrafce**

Si creamos un fichero llamado: ComponenteTS.tsx (la extensión tsx es para los componentes react en tyepscript) y dentro escribimos: **tsrafce** nos aparece algo parecido a:

```
import React from 'react'
type Props = {
	info?: string
}

const ComponenteTS = (props: Props) => {
	return (
		<div>ComponenteTS se ha recibido por props: {props.info}</div>
	)
}
export default ComponenteTS
```

Lo anterior está pensado para un caso en el que podamos recibir (también puede que no, observar el interrogante: "?") una prop que se llame: info. Ahora imaginemos que hemos puesto esto en index.tsx:

```
root.render(
	<ComponenteTS info="lino"/>
);
```

Vemos que nos ha creado un tipo: Props y lo ha incorporado como un tipo de datos que recibimos. Así que las props funcionan igual que con JSX pero tenemos que definir el tipo de dato que recibimos.

Práctica 5: Reproducir el ejemplo anterior de componente con typescript, pero cambiando que los atributos que reciba sean de tipo numérico: num1 y num2 y lo que muestre es:
La suma de num1 y num2 es: num1 + num2
(donde num1 y num2 serían los datos que recibiera el componente)

Llamar de un componente a otro con props

Mirar esta salida:

Podemos obtenerla teniendo una aplicación React que llame a tres componentes reloj:

```
import { Reloj } from './Reloj'

type Props ={
}
export const RelojesMundiales = (props: Props) => {

 return (
   <>
     <h1>Actividad react: Relojes mundiales</h1>
     <Reloj zona="Europe/Madrid" />
     <Reloj zona="America/New_York" />
     <Reloj zona="Europe/London" />
   </>
 )
}
```

Como vemos se le está pasando mediante: props la información de la zona horaria

Así, en el componente: Reloj podemos tener una constante fecha parecida a:

```
const fecha = new Date().toLocaleString( "es-ES",{timeZone: "Europe/Madrid" });
```

Observar que si en lugar de poner nosotros en: const fecha directamente a mano la zona horaria lo recibimos como un parámetro props, obtenemos el date apropiado para cada caso

Aprovecharemos para aprender respecto al operador: ***nullish coalescing: "??"***

Mediante ese operador: **??** podemos ofrecer un valor por defecto cuando un elemento al que necesitamos acceder no existe, es nulo o está sin definir (undefined). Ej con el caso de que la zona no nos la estén enviado como una props. Podemos evitar el error que vendría mediante:

```
const zonaString= props.zona ?? "Europe/Madrid";
```

Práctica 6: Conseguir el renderizado anterior, generando el componente Reloj.ts apropiado. Para ello generaremos el fichero: Reloj.ts y dentro estará el componente TSX

Concepto de State en React. Uso de hooks

A estas alturas ya sabemos que los componente de React son una combinación de HTML + javascript (**JSX**) o typescript (TSX) y que son renderizados apropiadamente por React (ya hemos visto: ReactDOM.render())

Al haber javascript de por medio, es lógico pensar que esos componentes puedan cambiar sus atributos con el tiempo, de forma dinámica. Si eso ocurre sería más que interesante que se renderizara automáticamente el cambio y se mostrara el componente actualizado.

Para visualizar lo anterior imaginemos por ejemplo que queremos pulsar un botón y que nos muestre la hora. Una solución sería la siguiente:

```
import React from "react";

const ComponenteApp = (props:any) => {

        const mostrarHora = ()=>{
                alert(new Date());
        }

        return (
                <>
                        <h1> Pulsar en el botón para ver la hora</h1>
                        <button onClick={mostrarHora}>Pulsar</button>
                </>
        );
}

export default ComponenteApp;
```

Práctica 7: Probar el código anterior. Tomar captura de pantalla del navegador al pulsar el botón

Lo anterior funciona porque le estamos diciendo activamente que emita un alert. ¿ Pero si lo queremos es tener una variable en el interior del componente y que se muestre la hora actualizada al pulsar el botón en lugar de una ventana emergente ?. O todavía más allá ¿ cómo podemos crear un

componente que nos muestre la hora cambiando segundo a segundo sin que el usuario pulse nada ? (pensemos en un timer al estilo de setInterval)

Ahí tendremos un problema porque necesitamos que se ejecute: ReactDOM.Render() sobre el componente cuando pulsemos el botón que tenemos dentro del componente, o en el caso del reloj que se actualice con un timer, que detecte el cambio y ejecute de nuevo el Render. Observar que el problema no está en ejecutar una acción (pulsar un botón, la ejecución de un timer) sino que cuando se haga esa acción o haya un cambio deseado se ejecute de nuevo el renderizado del componente

Para eso React ha creado los: **state** (estados) que son un tipo especial de atributos/propiedades del componente. **Si se modifica el valor de un state** entonces React automáticamente vuelve a renderizar el componente y nos lo muestra actualizado.

Hooks

Si estuviéramos usando React en su forma tradicional de Clases Componente (no con Functional Components) veríamos que los state y las props son atributos especiales en el constructor de la clase: this.props, this.state y así en this.props recibimos los parámetros que nos pasen al componente: <Componente parametro="dato" /> y **en this.state mantenemos aquellas propiedades "especiales" que queremos que si se modifican automáticamente (esto se hace mediante un método llamado: this.setState()) se vuelva a renderizar el componente**. Pero estamos con Functional Components y en este caso necesitamos de los: "Hooks" para poder hacer uso y modificaciones en esos atributos especiales: state sin tener que recurrir al estilo tradicional de clases que extiendan de React.Component

Vamos a ver un componente llamado Contador que tiene una variable número y un botón. La idea es que cada vez que se pulse el botón se actualice el contador y se incremente.

La forma tradicional sería esta:

```
import React, { Component } from 'react'

class Contador extends Component {
	state = { count: 0 } // inicializamos el state a 0

	render () {
		const { count } = this.state // extraemos el count del state

		return (
			<div>
				<p>Has hecho click {count} veces</p>
				{ /* Actualizamos el state usando el método setState */ }
				<button onClick={() => this.setState({ count: count + 1 })}>
					Haz click!
				</button>
			</div>
		)
	}
}

export default Contador;
```

Práctica 8: Crear el código anterior de componente Contador en un fichero nuevo y cargar en index.tsx en la parte de renderizado: ReactDOM.render() ese componente
Probarlo en el navegador y comprobar que efectivamente cambia el contador con los click

Mirando el código anterior vemos que se ha definido un state llamado count y se ha inicializado a cero: `state = { count: 0 }` y luego se hace uso de la función: this.setState() cuando se pulsa el botón, para poder modificar el state y que así se haga el render automáticamente del componente: `this.setState({ count: count + 1 }` .

Vamos a ver el código equivalente con un Hook (el hook en esta ocasión es: useState)

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

type Props = {}

const FCContador = (props: Props) => {
        const [contador, incrementar] = useState(0);

        return (
                <>
                        <p>Has hecho click {contador} veces</p>
                        <button onClick={() => incrementar( contador + 1) }>
                                Haz click!
                        </button>
                </>
        );
}

export default FCContador;
```

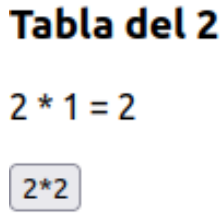
El Hook useState funciona de la siguiente forma. Al escribir:

```
const [contador, incrementar] = useState(0);
```

useState() retorna un array con un state y la función que modifica ese state. En concreto en el ejemplo se crea un nuevo state llamado: contador, y la función que se encarga de modificar ese state contador se llama: incrementar()

useState() recibe como parámetro el valor inicial del state. Así al escribir: useState(0) le estamos diciendo que el state tendrá inicialmente el valor cero. Y así es como se inicializa el state: contador =0

Práctica 9: Realizar con el Hook useState dentro de un functional component un componente que sirva a un usuario para prácticar la tabla del 2. Cada vez que pulse en el botón se le mostrará la solución correcta de la tabla. Así:
la primera vez que haga clic se le mostrará:
2x1 = 2
La segunda vez:
2x2=4
y así sucesivamente.
En definitiva: que vaya mostrando la tabla del 2 a cada click
Observar que después de 2x10 mostrará 2x1



Con lo que hemos visto de como se hacían las cosas en React de la forma tradicional y como funciona el useState() Nos podemos dar cuenta que la palabra: Hook (gancho en inglés) hace referencia a que está "enganchando" las propiedades de estado: state desde los Functional Components que usamos actualmente

Por último dos detalles al respecto del useState():

- **habitualmente la función que modifica el state lleva el mismo nombre con la palabra set delante.** Ejemplo:

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

- Podemos usar la función que modifica el state de dos formas:

1. La que ya hemos usado: Poner el nuevo valor del state como parámetro de la función. Ejemplo:

```
incrementar( contador + 1)
```

2. Definiendo una nueva función en su interior, que recibe el state y retorna el nuevo valor del state:

```
incrementar( (cont) => cont + 1 );
```

Nota: Al trabajar **hooks** con **typescript** es conveniente (en ocasiones obligatorio) darles la información del tipo de dato. Así por ejemplo si estamos guardando un array de personas podríamos poner:

```
const [personas, setpersonas] = useState<Array<Persona>>([]);
```

y si queremos iniciar un objeto vacío (por ejemplo un Usuario) sería:

```
const [user, setuser] = useState<Usuario>({} as Usuario);
```

Práctica 10: Crear un functional component react (usa el snippet: tsrafc) que tenga un botón. Este botón al pulsarlo va agregando un nuevo número aleatorio de 0 a 100 de tal forma que podemos ver gracias al state toda la lista de aleatorios generados (Nota: podemos usar: JSON.stringfy(nombredelarray) para ver el array u otro objeto)

Nota: hay una forma sencilla de crear un nuevo array con un nuevo elemento conservando los datos del anterior. Imaginemos que queremos agregar el número 5:
const arrayanterior: Array<any> = [4, 2, 7];
[...arrayanterior, 5]

Hay muchos Hooks interesantes, el **useEffect** (equivalente en clases React **componentDidMount**).

useReducer() que está inspirado en Redux y muchos más

Entendiendo diferencias entre Functional Components (Stateless) y Componentes tradicionales(Statefull). Atributos estáticos

Los componentes tradicionales tienen todas las características de las clases javascript: podemos crear atributos que sean mutables con el tiempo, localizarlos y leerlos, simplemente llamando a: this.nombreatributo. El estilo de los functional components es Stateless y las cosas que queremos que permanezcan de un render a otro típicamente las guardamos en el State mediante el hook useState. **En otro caso se perderán de un render a otro** ¿ Qué otras alternativas tenemos para que la información perdure entre renders pero no esté almacenado en el State ?

Se puede acudir a atributos estáticos. Los podemos usar así:

```
import React, { useState } from 'react'

const ComponenteConEstatico = () => {
    const [horaactual, sethoraactual] = useState("");

    let dato = 1;

    function actualizar(){
        ComponenteConEstatico.atributoestatico++;
        dato++;
        console.log("Estático: " + ComponenteConEstatico.atributoestatico);
        console.log("dato: " + dato);
        //sethoraactual("" + new Date());
    }
    return (
    <div>
        <h4>ComponenteConEstatico</h4>
        <p>Info en estático: {ComponenteConEstatico.atributoestatico}</p>
        <button onClick={actualizar}>Actualizar</button>

    </div>
  )
}

ComponenteConEstatico.atributoestatico = 2;

export default ComponenteConEstatico
```

Por medio de ellos podemos mantener la propiedad actualizada su información sin usar el state. Pero es estático, afecta a todos los componentes del mismo tipo

Práctica 11: Crear el anterior functional component, ejecútalo y abre la consola ¿ se está actualizando la información del atributo estático ? ¿ y de la variable: dato ? Ahora quita el comentario de la línea: sethoraactual(“” + new Date());
Sabemos que de esa manera al actualizar el state se fuerza un nuevo renderizado ¿ se está actualizando la info del atributo estático ? ¿ y de la variable: dato ?

Nosotros como norma, guardaremos toda la información que queramos mantener entre renderizados en el state

Pasando información (parámetros) en un onClick

Antes hemos visto un botón con un onClick (**observar que los nombres en React para no confundir con su equivalente javascript usan camelcase. Así en lugar de: onclick, decimos: onClick**)

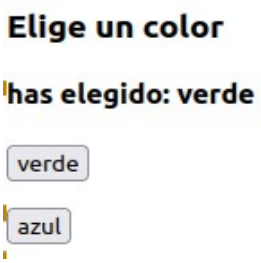
Para los onClick vemos que ponemos el nombre de la función sin paréntesis o le pasamos una lambda completa (una arrow function: ( )=>)

En la opción de únicamente el nombre de la función no podemos pasarle un parámetro al onClick, así que usaremos la arrow function para ese cometido. Ej:

```
function saludoSinParametros(){
        alert("hola Amigo!");
}

function saludoConParametros(mensaje){
        alert(mensaje);
}
return (
<>
  <h3>Realizando saludos:</h3>
  <p>Sin parámetros: <button onClick={saludoSinParametros}>amigo</button></p>
  <p>Con params: <button onClick={()=>saludoConParametros("saludos Ana")}>ana</button></p>
  <p>Con params: <button onClick={()=>saludoConParametros("saludos Marta")}>mara</button></p>
</>
)
```

Imaginemos este renderizado:



La idea es que cuando se pulse en el botón verde diga: has elegido verde

y si se pulsa en el azul diga: has elegido azul

Se creará un método: eligeColor(color) que recibe el color (ya sea verde o azul) según se haya pulsado un botón u otro. Ejemplo para el botón verde:

```
<button onClick={()=>eligeColor("verde")}>verde</button>
```

Práctica 12: Crear la actividad que se acaba de describir. Notar que hay que usar un useState para que muestre un texto u otro según lo que se haya pulsado

Poniendo estilos CSS

El atributo: class **NO** lo podemos usar directamente. Para poder asociar clases a nuestros objetos, React tiene un atributo específico: **className**

```
import './MiComponente.css'
export const MiComponente = () =>{
return (
<div className="contenedor">
...
</div>
)
}
```

Ese atributo es el equivalente de class cuando trabajamos directamente con html y css. Luego como es habitual, ponemos nuestros estilos en ficheros .css Podemos llamar directamente a ese fichero con el import correspondiente: `import './MiComponente.css'`

Si queremos agregar bootstrap, podemos por ejemplo, agregar en public/index.html el cdn:

```
<head>
<!-- aquí el resto de nuestro head …  -->
<link
  rel="stylesheet"
  href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
                                                    integrity="sha384-
1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"
  crossorigin="anonymous"
/>

</head>
```

Podemos también usar módulos npm como alternativa:

```
npm install bootstrap@5.1.3
```

y luego en src/index.jsx o src/index.tsx:

```
import 'bootstrap/dist/css/bootstrap.css';
```

Bucles y condiciones en JSX/TSX

En angular disponemos de ngFor, Y en los diferentes lenguajes de servidor suele haber una forma específica para los for, if y demás circunstancias de código en plantillas de presentación

El abordaje de React es usar los medios que nos da de forma natural javascript. Y el formato de trabajo recuerda al de xquery (mediante llaves: “{ }“ informas de la parte de código javascript y hacemos uso de return para devolver los elementos)

Ejemplo con la app monedas (una lista de monedas donde se conoce el nombre de la moneda y el país. También se conoce el histórico de tipos de cambio respecto al euro de cada moneda). Método render() modificado:

```
type Props = {}

type Moneda ={
  nombre: string,
  pais: string
}

const ListaMonedas = (props: Props) => {

    const [monedas, setmonedas] = useState<Array<Moneda>>([]);
    function addMoneda(){
       const moneda: Moneda ={
          nombre: "libra",
          pais: "uk"
       }
       setmonedas([...monedas,moneda]);
    }

    return (
      <>
        <h3>Cliente de  monedas</h3>
        <div>
            <button onClick={addMoneda}>
              agregar moneda
            </button>

            <h4>Monedas: </h4>
            <ul>
            {
              monedas.map(  (m:Moneda) => {
                return (
                  <li> {m.nombre} </li>
                );
              })
            }
            </ul>
        </div>
      </>
    );
  }
```

Vamos a centrarnos en el trozo de código que aporta algo nuevo:

```
            <ul>
            {
              monedas.map(  (m:Moneda) => {
                return (
                  <li> {m.nombre}  </li>
                );
              })
            }
            </ul>
```

El método: map() que en javascript existe para los arrays, permite recibir una función que transforme cada elemento del array (en el código anterior cada elemento del array es una moneda que asignamos a la variable: m) Luego en el return incluimos cada elemento

Práctica 13: Reproducir el ejemplo anterior en la aplicación monedas. Hacer que los <li> no muestren únicamente el nombre de la moneda sino también el país. Ej:
 libra de UK

Si queremos usar la forma tradicional de bucles en JSX/TSX, debemos hacerlo fuera del return. Acumulamos un array y luego lo mostramos en el return:

```
let nums=new Array();
for (let i = 0; i < 10; i++) {
        nums.push(<p>{i}</p>);
}
return (
        <div className="contenedor">
                { nums}
        </div>
)
```

Bucles e identificador único en Componentes

React muestra un warning con los códigos anteriores. Esto es porque quiere tener un identificador único para cada componente (evita problemas de renderizado) . Al hacerlo desde un bucle en tiempo de ejecución, no sirve la asociación interna que hace, y nosotros debemos ponerle un identificador. Una alternativa es usar números aleatorios, otra es apoyarnos en que el método: map() de array aparte de recorrernos todos los elementos nos da el índice que ocupa cada uno en el array. Así el siguiente código es válido para generar componentes con id:

```
let arr:Array<number> = [2,3,4,5,6,7,8,9,10];
        return(
                <div className="grid">
                {
                        arr.map( (elemento,index)=>{
                    return <PracticarTabla key={index} tabla={elemento} />;
                        })
                }
                </div>
);
```

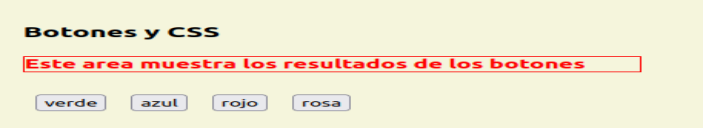
Lo anterior usa un map para hacer un bucle. Veamos un ejemplo de ese tipo de “bucle”

```
let arr = [5,7,9,11];
return (
  arr.map( (valor,indice)=>{
  return <div>Número: {valor} en la posición del array: {indice}</div>
})

)
```

Práctica 14: Crear un componente: TodasLasTablas que use el componente ya creado Así muestra las tablas del 2 al 10 (mirar imagen ejemplo)
Se usarán las pros: <PracticarTabla tabla={5} /> → Esto genera la tabla del 5. Usar un map para un array [2,3,..,10] y establece para cada componente PracticarTabla el prop para su tabla

Sea el renderizado:



Práctica 15:
Crear el renderizado anterior. Al pulsar en botón rojo el área tiene color fuente rojo y borde rojo. Si se pulsa en verde, pues en verde, y así con todos. Se recomienda crear las 4 clases CSS y luego que se establezcan mediante:
`<h4 className={claseaplicada}>Este area muestra los resultados de los botones </h4>`

Práctica 16: En la práctica de los relojes de zonas horarias, crear un array con 5 zonas horarias, entre ellas: Londres, Madrid y usando array.map generar los 5 componentes Reloj con su respectiva propiedad timezone, dándole estilos CSS a los componentes

Condiciones en JSX/TSX

Para las condiciones pasa más o menos similar. Si en angular hay una instrucción específica: ngIf en React nos apoyamos en los **if** de javascript o en los operadores: **&&, ?**

Habitualmente hacemos uso del operador ternario ? cuando queremos un comportamiento if-else y usamos el operador && cuando es únicamente un if

Ej.

```
render() {
  let dato: number = 5;
  return (
      <div>
      {
        ( dato > 8 )? <span>dato es mayor que 8</span> : <span>dato menor</span>

      }
      </div>
  );
}
```

Para los casos que únicamente se quiera el if sin el else:

```
render() {
  let dato: number = 5;
  return (
      <div>
        Dato es: {dato}
        {
          ( dato > 8 ) && <span> y es mayor que 8</span>
        }
      </div>
  );
}
```

En el siguiente ejemplo renderizamos un componente u otro según el aleatorio obtenido:

```
  public render() {
    let dato = Math.random();
    return(
    <>
      {
        (dato>=0.5)?<ComponenteA />:<BComponent />
      }
    </>
    );
  }
}

type Props = {}

const ComponenteA = (props: Props) => {
  return (
    <div>Este es el componente a </div>
  )
}

const BComponent = (props: Props) => {
  return (
    <p>this is BComponent!! </p>
  )
}
```

Práctica 17: Crear un componente que tenga dos botones. Cuando se pulse en el primer botón se cargará un componente que mostrará 10 números aleatorios de 0 a 100 a pulsar un botón llamado “generar” que esté dentro del componente
Si se pulsa en el otro botón se carga otro componente que reemplaza el anterior que muestra un saludo y la fecha actual (la fecha se enviará mediante props)

hook useEffect()

useEffect() es un hook que se dispara de forma equivalente a los siguientes métodos del ciclo de de vida de un componente clásico en React:

- componentDidMount; - componentDidUpdate; - componentWillUnmount

¿Cuál es la idea de este hook ? Bien, pensemos en esa información externa que necesitamos para el funcionamiento de un componente. Si estamos en un componente de una red social querremos que nos aparezcan los amigo conectados. Esa información nos la dará una api externa y desde que montamos nuestro componente (componentDidMount) querremos estar pendientes del cambio de estado de nuestros amigos. Así, cuando el componente ya no esté en uso (componentWillUnmount) ya no tendremos necesidad de indagar respecto a si nuestros amigos están conectados. Si creamos un "efecto" que se dispare en el momento en el que montamos el componente, que esté pendiente de las actualizaciones y está vigente hasta que lo desmontamos habremos alcanzado nuestro objetivo

Este hook se disparará cuando lo haga cualquiera de los 3 métodos que acabamos de nombrar. Digamos que es oportuno cuando hace falta: "un efecto secundario" a las diferentes situaciones por las que pase el componente. Observar que esto es que **se dispara en cada renderizado**. Eso puede ser excesivo y se puede restringir. Siendo un caso muy habitual dejarlo que se ejecute únicamente cuando se monta el componente y luego cuando se desmonta.

Vamos a poner el ejemplo que hay en la documentación oficial. Se muestra como se actualiza el título de la página web cada vez que hacemos click (y así aumenta el contador) Primero lo vemos con useEffect() y luego con React.Component que tendrá que hacer uso de dos de los métodos: componentDidMount() y componentDidUpdate()

Versión con FuntionalComponent y el hook useEffect():

```
import React, { useState, useEffect } from 'react';
function Example() {
  const [count, setCount] = useState(0);

  // De forma similar a componentDidMount y componentDidUpdate
  // Actualiza el título del documento usando
  //la API del navegador
  useEffect(() => {     document.title = `You clicked ${count} times`;  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
```

```
        Click me
      </button>
    </div>
  );
}
```

Hemos destacado los dos hook que utilizamos: El ya conocido: useState() y el nuevo: useEffect()

Veamos el equivalente en componentes tradicionales:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {    document.title = `You clicked ${this.state.count} times`;  }
  componentDidUpdate() {  document.title = `You clicked ${this.state.count} times`; }
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

Práctica 18: Realizar los dos ejemplos anteriores (FuntionalComponent con useEffect() y React.Component con los métodos componentDidMount() componentDidUpdate(). Adaptarlos a React con Typescript

Para no hacer múltiples llamadas innecesarias a la api, puede ser un buen momento para obtener los datos y tenerlos en nuestra aplicación. Así, llamaremos a la api al montar el componente .

Para entender mejor lo anterior veamos un ejemplo:

```
const [data, setData] = useState<PokemonCardData>();


useEffect(() => {
```

```
        const miEfecto = async() => {
                const newData = await consultaAsincronaApi();
                setData(newData);
        }

        miEfecto();
}, []);
```

El código anterior establece un efecto, que crea la función asíncrona: miEfecto() y la lanza.

Sabemos por la documentación que esa función: miEfecto() se crearía y se lanzaría en cada renderizado. Pero no será así porque le hemos dicho que únicamente lo haga en el momento del montaje ¿ cómo ?. Observar que le estamos pasando un array vacío: `[]` a useEffect:

```
}, []);
```

En ese array se especifican las variables que debe vigilar useEffect para dispararse si cambian. Si NO ponemos ninguna variable entonces no se dispara en ningún momento adicional al del montaje

Vamos a ejecutar el siguiente ejemplo para entenderlo mejor:

```
import React, { useEffect, useState } from 'react'
type Props = {}
const EjemploUseEffect = (props: Props) => {
        const [contador, setcontador] = useState<number>(100);
        useEffect(() => {
                const efecto = () =>{
                        let fecha = new Date();
                        console.log(fecha);
                        //setcontador(-1);
                }
                efecto();
        }, /*[] */ )
        return (
          <div>
                <h3>info en state: {contador}</h3>
                <button onClick={() => setcontador(contador + 1)}>Actualizar state</button>
          </div>
        )
}

export default EjemploUseEffect
```

Práctica 19: Abriendo la consola para ver los mensajes de log, ejecutar el código anterior. ¿ se muestra la fecha cada vez que se renderiza (modifica el estado) ? ¿ el contador empieza en qué número ?
Ahora modifica el código anterior quitando los comentarios en la línea: //setContador(-1) ¿ qué ocurre ahora ? ¿ En el primer renderizado (antes de pulsar el botón) qué muestra el contador? ¿Y después de ejecutar el botón?
Sigue modificando el código quitando los comentarios en el array de useEffect quedando la línea final del useEffect() así:

```
}, [] )
```

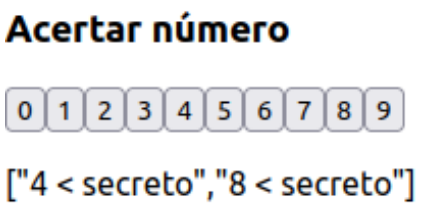
¿ se ejecuta es useEffect() en cada renderizado ? ¿ se ejecuta en el momento del montaje ?

Finalmente vamos a dejar nuestro useEffect así:

```
useEffect(() => {
  const efecto = () =>{
    let fecha = new Date();
    console.log(fecha);
    setcontador(-1);
  }
 efecto();
}, [contador>10] )
```

Ahora ¿cuándo se ejecuta el useEffect?

Práctica 20: Realizar un componente para el juego de Acertar número secreto (de 0 a 9). Tendremos 10 botones siguiendo el patrón:
<button onClick="{()=>apostar(7)}" > 7 </button>
Al montarse el componente se genera el número aleatorio secreto, que permanecerá sin modificación hasta que el usuario acierte el número. Cuando se pulsa en los botones de apuesta se informa al usuario de si ha acertado, si el número es menor o mayor que secreto



Vamos a entender mejor el uso de useEffect y de paso trabajamos con las tareas periódicas mediante setInterval() Para ello, nos basaremos en un ejemplo inspirado en la documentación oficial.

```
import React, { useEffect, useState } from 'react'

type Props = {}

const EjemploRelojActivo = (props: Props) => {
    const [fechaactual, setfechaactual] = useState<string>("");
    useEffect(() => {
      const timerID = setInterval(
        tick,
        1000
      );

    }, [])

    function tick(){
        const newfecha = ""+ new Date();
        setfechaactual(newfecha);
    }

    return (
        <div>
            <h3>Ejemplo Reloj Dinámico</h3>
            {fechaactual}
        </div>
    )
}

export default EjemploRelojActivo
```

Observar que el useEffect() se ejecuta únicamente cuando se monta el componente (se ha puesto un array vacío como parámetro: [ ]) Y es en ese momento cuando se ejecuta el setInterval()

Práctica 21: Copiar y ejecutar el ejemplo anterior. Buscar información sobre setInterval() ¿ qué significa el 1000 que le pasamos como parámetro ? ¿ para qué vale el valor devuelto timerID ?. Comentar la línea: setfechaactual(newfecha) de la función tick() y escribir en su lugar: console.log(newfecha); ¿ qué ocurre con el renderizado ? Mirar en la consola que información está mostrando y explicar lo que ocurre

Práctica 22: Ahora que ya sabemos usar setInterval() y combinarlo con useEffect() modificar la actividad de los relojes mundiales de tal forma que se muestren con la información de la hora actualizada cada segundo

Hook useRef(). Accediendo al DOM directamente con referencias

Generalmente en react hacemos uso de funciones que se disparan para leer y escribir información en el DOM. Pero es habitual en otros estilos de programación acceder a elementos del DOM directamente, ya sea por id o de otra forma

Tenemos una opción para acceder directamente al DOM (**saltándose el DOM virtual de react**) mediante el hook useRef():

Este hook se hizo principalmente como reemplazo al equivalente con componentes tradicionales por clases: React.createRef() que permite acceder directamente al DOM (**pudiendo así, establecer el foco en un input por ejemplo**) Pero adicionalmente **te permite mantener un atributo mutable y que mantenga la información entre renders sin hacer uso del State**

Ejemplo de uso de useRef() para acceder a un objeto del DOM:

```
import React, { useRef, useState } from 'react'
  const EjemploUseRef = () => {

  const inputnumero = useRef<HTMLInputElement>({} as HTMLInputElement);
  const divresultado = useRef<HTMLDivElement>({} as HTMLDivElement);

  function multiplicar(){
    let htmlinput = inputnumero.current;
    let numero = Number(htmlinput.value);

    let htmldiv = divresultado.current;
    htmldiv.innerText = "" + ( 2 * numero);
  }
  return (
  <div>
    <h4>Componente Ejemplo useRef</h4>
    <input type="text" ref={inputnumero} />
    <button onClick={multiplicar}>Multiplicar por 2</button>
    <div ref={divresultado}>
    </div>
  </div>
  )
}
export default EjemploUseRef
```

Las dos líneas anteriores marcadas muestran como trabajar con useRef() Observar que al trabajar con typescript tenemos que decirle el tipo de objeto: en este caso HTMLInputElement.

Vemos que mediante useRef() creamos una etiqueta llamada en este caso: inputnumero que se establece en el atributo: ref= en el elemento que queremos tener vinculado del DOM

Práctica 23: Usando useRef(), crear un componente con 2 input y un párrafo (etiqueta: <p>) donde uno de los inputs sea para el nombre y el otro input para los apellidos. Al pulsar en el botón tomará la información de los dos inputs y lo mostrará en el párrafo concatenados y dirá cuántas letras tiene el nombre completo

Nota: Uso del operador: ? Para establecer que un objeto puede ser null

Cuando definimos el hook useRef si elegimos iniciarlo a null: useRef<HTMLInputElement>(**null**)

Implica que typescript reconozca que puede ser un objeto nulo y por tanto la siguiente sentencia: **let** numero = inputnumero.current.value;

genere problemas al llamar al atributo: value de un objeto nulo. La forma de solucionar es (sabiendo que es seguro que el objeto no es nulo) hacer uso del operador: "?"

```
let numero = inputnumero.current?.value;
```

Práctica 24: Modificar el ejercicio de acertar número. Ahora en lugar de 10 botones, habrá un único input y un único botón. Al pulsar el botón en la acción que desencadene se usará useRef() para tomar la información que haya en el input y así realizar la apuesta

Práctica 25: Crear un functional component con dos botones uno dice: aleatorio que cada vez que se pulsa, agrega un aleatorio a un array apuntado por useRef() y otro botón que dice: mostrar este último botón copia el array almacenado en la referencia y lo pone en el state. Mostrándose así el array de números generados

Práctica 26: Crear un componente que tenga un cuadro de texto y un botón. Cuando se pulse en el botón se cargará otro componente debajo del botón que será la tabla de multiplicar (del 1 al 10) si es un número lo introducido. Si en lugar de un número fuera una palabra, entonces se cargará otro componente que nos dirá la cantidad de letras de la palabra y la cantidad de mayúsculas y minúsculas. Pasar la información a esos dos componentes mediante props

useRef() para atributos sin DOM. Crear y parar un timer setInterval()

En el siguiente ejemplo podemos ver el caso de un useRef() en acción. Toma el dato del setInterval() para poder detenerlo al pulsar en el botón.

```
const EjemploUseRef = () => {

    const [stateFecha, setstateFecha] = useState<string>("");
    const [stateButton, setstateButton] = useState<boolean>(false);
    const refTimer = useRef<ReturnType<typeof setInterval>>();


    function actualizarHora(){
      setstateFecha( "" + new Date());
    }

    function iniciarParar(){
      if( !stateButton){
        refTimer.current = setInterval( actualizarHora, 1000);
        setstateButton(true);
      }else{
        clearInterval(refTimer.current);
        setstateButton(false);
      }
    }

    return (
    <div>
      <h3>prueba timer</h3>
      <button onClick={iniciarParar}>{stateButton?"parar":"iniciar"}</button>
      <p>{stateFecha}</p>
    </div>
    )
}
export default EjemploUseRef
```

Nota: observar que para obtener el tipo de dato en typescript que devuelva una función (en este caso el tipo de dato que devuelve setInterval) hacemos uso de: ReturnType. Veamos un ejemplo sencillo:

```
function numeroComplejo(a:number, b:number){
  return {
    real: a,
    imag: b
  }
}
const complejo: ReturnType<typeof numeroComplejo> = numeroComplejo(2,3);
```

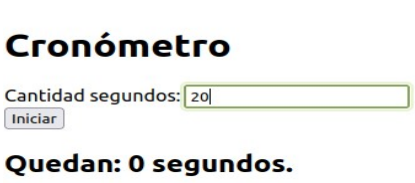
Para el manejo de fechas haremos uso de: new Date() y así tenemos la hora actual.

Los objetos de tipo Date tienen un método que se convierten en tiempo unix epoch (milisegundos desde 01-01-1970) mediante: Date.getTime()

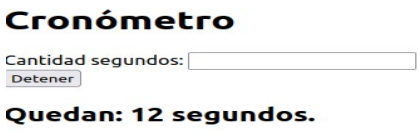
si sumamos a la hora actual los segundos que ha introducido el usuario multiplicado por mil:

```
this.horaFinalizacion = new Date(ahora.getTime() + segundos*1000);
```

Imaginemos que ahora queremos hacer una cuenta atrás:



Antes de empezar



funcionando

finalizado

Observar que hay un botón para iniciar y detener el cronómetro.

Práctica 27: Realizar un componente react: Cronometro.tsx De tal forma que el usuario introduzca la cantidad de segundos y al pulsar iniciar vaya mostrando la cuenta atrás

Acceder información del DOM mediante eventos, en lugar de referencias

Si la información a la que queremos acceder está en un objeto que es susceptible de disparar un evento, podemos prescindir de las referencias.

Por ejemplo, para acceder a la información de texto para botones o inputs:

información almacenada en un botón:

```
<button onClick={enviarDato}> 8 </button>
```

podemos recoger el objeto que dispara el evento (event.target) y por tanto tener acceso a todo el objeto (incluyendo el número 8 del ejemplo)

Veamos como tomarlo para el caso anterior del botón (método enviarDato)

```
const enviarDato = (evento: React.MouseEvent<HTMLButtonElement>) => {
        evento.preventDefault();
        console.log(evento.currentTarget.innerText);
}
```

Como estamos usando typescript debemos decirle que el evento es del tipo MouseEvent y (para restringirlo mejor) le especificamos que lo dispara un botón: `React.MouseEvent<HTMLButtonElement>`

La siguiente sentencia que vemos: evento.preventDefault() es para impedir el comportamiento normal de un elemento html ante un evento. Imaginemos por ejemplo un formulario, por defecto el submit implica un renderizado completo de la página (con el gasto que implica y además es problemático en react). En el caso de un button no es tan necesario pero se pone como ejemplo en general.

Finalmente vemos como acceder al objeto que ha disparado el evento: evento.currentTarget de esta forma obtenemos todo el objeto botón, y por tanto, podemos acceder a su contenido en texto mediante: **innerText**

Nota: observar que **MouseEvent** tiene que importarse de react

información almacenada en un input: `<input type="text" onChange={handleChange} />`

Vemos el uso de un evento muy habitual en los input: **onChange** . En el ejemplo del botón hicimos uso de: **onClick** y ahora usaremos onChange

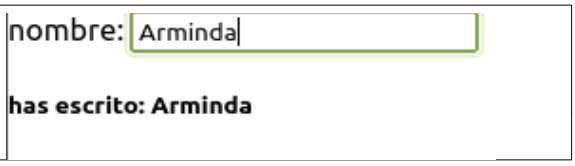
Y ahora el método que gestiona el evento:

```
function handleChange(event:ChangeEvent<HTMLInputElement>){
        event.preventDefault();
        console.log(event.currentTarget.value);
}
```

En este caso le hemos especificado a typescript que es un evento del tipo ChangeEvent que lo dispara un input: HTMLInputElement

Como ya sabemos los input tienen un atributo value. Así que accedemos ese elemento una vez tomamos el objeto input igual que hicimos antes con el button: event.currentTarget

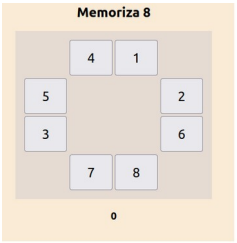
Práctica 28: Realizar un componente llamado: MostrarInput que se introduzca el texto en el input y se vaya mostrando en un h5



Práctica 29: Realizar un componente llamado: OperarBotones que al pulsar el botón de la izquierda divida al valor actual entre 2 y si se pulsa el de la derecha multiplique. Se debe hacer mediante un único método que responda a onClick (el mismo método para los dos botones)

Práctica 30: Implementaremos el juego de acertar número secreto. Pero en esta ocasión, habrá un input y un botón. Al pulsar el botón el programa evalúa la apuesta del input. NO se usarán referencias. Se hará el uso de manejo de eventos para acceso al DOM

Práctica 31: Realizar un componente react: Memoria8.tsx que realice el juego de memorizar de forma ordenada 8 números. Durante 3 segundos se le muestra al usuario los 8 números y luego se ocultan (vale mostrar cualquier otra cosa) Luego el usuario irá pulsando en los botones. Primero debe pulsar el botón que incluye el 1, si pulsa en la casilla que lo tiene se le muestra y ya queda para siempre, en otro caso no muestra nada. Luego lo mismo con el que incluye el 2, etc. Cada pulsación de botón aumenta un contador.



Veamos el siguiente componente que trabaja con productos:

```
type Producto = {
    nombre: string,
    precio: number,
    cantidad: number
}

export default function FormularioProductos(props: Props){
    const [listaproductos, setlistaproductos] = useState<Array<Producto>>([]);

    function procesarformulario(e: React.FormEvent<HTMLFormElement>){
        e.preventDefault();
        let formulario = e.currentTarget;
        const nombre = formulario.nombre.value ?? "";
        const precio = Number(formulario.precioid.value ?? 0);
        const cantidad = Number(formulario.cantidad.value ?? 0);
        const producto: Producto = {
            nombre: nombre,
            precio: precio,
            cantidad: cantidad
        };
        setlistaproductos([...listaproductos,producto]);
    }
    return (
    <div>
        <h3>Info de Productos</h3>
        <form onSubmit={procesarformulario}>

        <label htmlFor="nombreid">Nombre</label>
        <input type="text" name="nombre" id="nombreid"/><br/>
        <label htmlFor="precioid">Precio</label>
        <input type="text" name="precio" id="precioid"/><br/>
        <label htmlFor="cantidadid">Cantidad</label>
        <input type="text" name="cantidad" id="cantidadid"/><br/>
        <button type="submit">Agregar</button>
        </form>
        <textarea value={JSON.stringify(listaproductos,null,2)} cols={100} rows={30}/>
    </div>
    )
}
```

Observar que al pulsar el botón se envía el formulario completo. Para acceder al formulario recibido hacemos uso de: **event.currentTarget** . Esto es importante porque en typescript tenemos que acceder a currenteTarget en lugar de target.

Los objetos del formulario recibido son accesibles tanto por su name como por su id (observar formulario.precioid y formulario.nombre)

Práctica 32: Reproducir el componente anterior y ejecutarlo. Darle algo de CSS.
Agregar (fuera del formulario) un input que mediante el evento onChange permita filtrar el array de productos por nombre (por ejemplo, si escribe queso aparecen todos los productos con nombre queso: "queso rochefort", "queso edam",…)

Práctica 33: Crear un componente con un formulario que contenga dos input numéricos y un submit Al enviar el formulario, se muestran los números primos entre los dos dados en los input. Ejemplo: primos mayores que: 10 primos menores que: 18
mostrará: 11, 13, 17

Edad real del perro	Perro pequeño	Perro mediano	Perro grande
6 meses	15 años	10 años	8 años
1 año	20 años	18 años	16 años
2 años	28 años	27 años	22 años
3 años	32 años	33 años	31 años
4 años	36 años	39 años	40 años
5 años	40 años	45 años	49 años
6 años	44 años	51 años	58 años
7 años	48 años	57 años	67 años
8 años	52 años	63 años	76 años
9 años	56 años	69 años	85 años
10 años	60 años	75 años	96 años
11 años	64 años	80 años	105 años
12 años	68 años	85 años	112 años
13 años	72 años	90 años	120 años
14 años	76 años	96 años	/
15 años	80 años	102 años	/
16 años	84 años	110 años	/

Práctica 34: La tabla anterior refleja la edad real de un perro y su equivalente si fuera humano. Crear un componente con un formulario que contenga un input para poner la edad del perro y tres radio button para elegir el tamaño del perro: pequeño, mediano, grande. Al pulsar el botón de calcular se mostrará la edad "humana" del perro
Nota: recordar que los radio button llevan todos el mismo name y es en el campo: value donde aparece la información que envían y es recibida

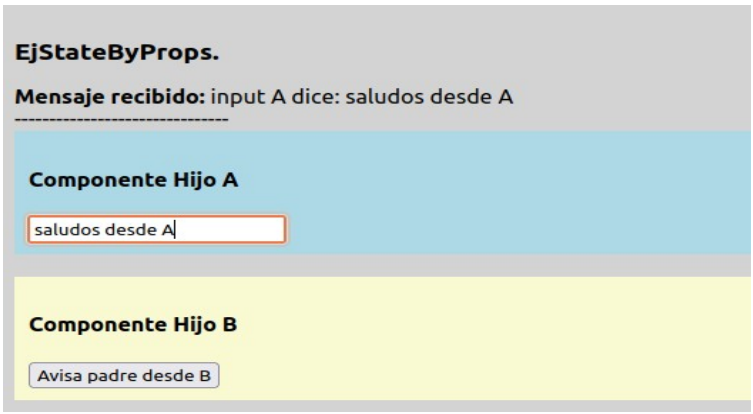
Pasando información entre componentes hijos y padres

Ya hemos visto que podemos pasar información entre un componente padre y los hijos mediante el atributo especial: Props

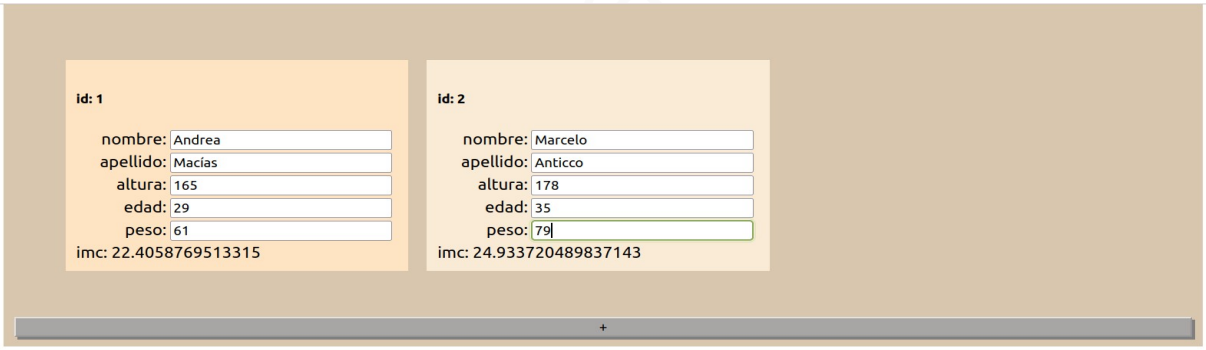
Si nosotros le pasamos una función en el props al componente hijo, en concreto una función que modifique el state del padre, entonces cuando el hijo haga uso de esa función estará modificando el state y por tanto volviendo a renderizar. Se puede decir que le estamos pasando información al padre

Veamos el siguiente ejemplo. El componente envía un método que incorpora el setstate al componente hijo El componente hijo genera un número aleatorio en cada onClick y lo establece en el state del padre mediante la función recibida

```
const PadreModificadoPorHijo = () => {
const [state, setstate] = useState<number>(0);
function modificarState(dato:number){ setstate(dato); }
 return (
  <div>
  <HijoModificaPadre  modificarstatepadre={modificarState}/>
  <h4>dato recibido de hijo: {state}</h4>
  </div>
 )
}
interface Iprops{
 modificarstatepadre: Function;
}
export const HijoModificaPadre = (props:Iprops) => {
 function enviarinfo(){
  const {modificarstatepadre } = props;
  let num = Math.random();
  modificarstatepadre(num);
 }
 return (
  <div>
   <button onClick={enviarinfo}>modificar padre</button>
  </div>
 )
}
```



Práctica 35: Realizar 3 componentes: EjStateByProps, A, B
El componente EjStateByProps contiene al componente A y al componente B
El componente A lo vemos en color azul (observar que tiene un input) Y el componente B está en amarillo (tiene un botón)
Si se escribe en el input del componente A (evento onChange) el texto aparece en el state del padre: "input A dice: " + mensaje escrito en el input
Si se pulsa en el botón del componente B el mensaje recibido en el state del padre es: "pulsado botón en B"



Práctica 36: Generar dos componentes. El componente padre tendrá un array de personas (hay que hacer la clase Persona también) . Mediante ese array personas se generan tantos componentes hijo: PersonaCard como personas tiene el array.
PersonaCard permite ir agregando y/o modificando datos en una ficha persona
Al inicio, en el componente raíz tenemos inicialmente únicamente el botón del más: "+" al pulsarlo se crea una persona en el array y por tanto un componente hijo: PersonaCard que nos permitirá editar los datos de Persona.
Nota: para localizar mejor el objeto (aunque aquí no hay problema ya que la posición del array coincide con id de persona) es importante que al crear cada nueva persona se genera un nuevo id que luego no pueda ser modificado

Veamos el problema de recibir información actualizada mediante props en el input de un componente hijo y poder modificar ese input desde el componente hijo:

La forma de obtener la información de un input con React es mediante el método onChange(): `<input type="text" onChange={handleChange} />`

Pero vamos a tener problemas con lo anterior si queremos enviarle a ese componente hijo información que rellene en el input (imaginemos que en el padre tenemos una lista de personas y en el comoponente hijo tiene un formularios con inputs para la información de una persona. El componente padre es quién le envía la información al hijo para que se rellene en los input. Y que así se pueda editar)

Este es un caso en el que tendremos que vigilar en el: useEffect() del componente hijo las props que recibimos del padre: `useEffect(() => {... }, [props])` y se debe :

llevar la información de las props al input anterior. Quedando algo así:

```
<input type="text" onChange={handleChange} value={props.nombre} />
```

Pero **cuidado!!** si lo dejamos así es un elemento input inmutable ya que el value viene definido por el padre que es una constante y nosotros no modificamos el contenido del input. La forma que nos permite recibir la información de props y editarla después es si tenemos un useState vinculado. Nos quedaría algo así:

```
const [nombre, setNombre] = useState("");
useEffect(() => {
    setNombre(props.nombre);
}, [props.nombre])

return (
  <div>

    <input type="text" onChange={(e)=>setNombre(e.target.value)} value={nombre} />
```

Con lo anterior recibimos la información actualizada recibida por props (si el componente padre decide que estemos editando a una diferente persona, por ejemplo) Y podemos seguir modificando la información del input ya que tenemos puesto un state vinculado

En el siguiente ejercicio tenemos un componente padre con un botón que genera números aleatorios y se los pasa por props a un componente hijo. Ese componente hijo muestra el número en un input y hace la descomposición del número mostrándola en un textarea:

Práctica 37: Realizar la actividad descrita. Conseguir que al pulsar el botón el componente hijo actualice la descomposición. En el componente padre se mostrará el número generado bajo el botón.
Nota: observar que se debe permitir que el componente hijo pueda introducir un valor en el input (si pones en el value del input: <input value={props.numero}> no puedes)

Práctica 38: Crear dos componentes. Uno es el padre que tiene un state de un array de Usuarios (objetos con atributo id y nombre únicamente) definido de los nombres iniciales : [{id: 1, nombre: "Ana"}, {id: 2,nombre:"Aristarco"}] que permitirá generar 2 botones que dicen: <button type=text >Modificar {nombre} … >
Al hacer click en alguno de los botones hace que el componente hijo reciba en un <input> el nombre del Usuario. Se debe poder modificar el nombre y que al pulsar en un botón: Terminar Edición que está dentro del componente hijo hace que el botón del padre donde aparecía el nombre se haya reemplazado por el nuevo nombre

Con el paso de información bidireccional entre padre-hijo podemos generar nuestros propios componentes personalizados eficaces. Pongamos el caso de que queramos un <input> que siempre que escribamos sea el texto en mayúsculas. Podríamos usarlo en un caso como el siguiente (lo hemos llamado: InputToUpper)

```
import React, { useState } from 'react'
import InputToUpper from './InputToUpper'

type Props = {}

const PracticaHijoInformaPadre = (props: Props) => {

    const [datosRecibidos, setDatosRecibidos] = useState("");
    function tomarDatos(datos: string){
        setDatosRecibidos(datos);
    }
  return (
    <div>
        <p>Práctica componente hijo informa a Padre</p>

        <InputToUpper onNewText={tomarDatos}/>
        <p>
        Recibido: {datosRecibidos}
        </p>
    </div>
  )
}

export default PracticaHijoInformaPadre
```

En ese componente: InputToUpper vemos que se ha declarado un props: onNewText que lo que pretende es hacer el equivalente de un: onChange()

El onNewText tiene un prototipo: (texto: string) => void así que el props de InputToUpper debe ser algo así: {

onNewText: (texto:string) => void

}

En el interior de ese componente se debe crear una etiqueta <input> y conseguir que el texto que el usuario introduzca se convierta en mayúsculas. Para ello se propone utilizar el método onChange() que sabemos que soportan los input en React y hacer uso del método: toUpperCase()

Práctica 39: Realizar la actividad descrita con el componente: InputToUpper

Multimedia

Vamos a usar una librería muy sencilla:

```
npm install react-player

npm install @types/react-player
```

Con lo siguiente tomamos una radio, le decimos que nos ponga los controles originales y le estblecemos el tamaño del reproductor:

```
<ReactPlayer
    url="http://allzic06.ice.infomaniak.ch/allzic06"
    controls
    width="400px"
    height="20px"
/>
```

Si quisiéramos ver un video de youtbe basta con cabiar la url y establecer un tamaño de reproductor más acorde:

```
<ReactPlayer
    url="<https://www.youtube.com/watch?v=dQw4w9WgXcQ>"
    controls
    width="400px"
    height="20px"
/>
```

Práctica 40: Crear un componente que se visualice un reproductor y una lista de reproducción (busca urls de mp3 o radios) Cuándo el usuario puse en uno de los elementos de la lista y se de al play en el reproductor sonará la canción.

Enrutado en React

Cuando estamos en una aplicación, por ejemplo de Android nativo (java/kotlin con Android studio), sabemos que tenemos una pantalla (una Activity) y pasamos a otra pantalla (otra Activity) De igual forma, en las aplicaciones web de servidor pasar de una pantalla a otra es tan sencillo como cambiar mediante un enlace de una página web a otra. En definitiva, según el uso que tengamos en nuestra aplicación, puede ser interesante pasar a otro renderizado completamente distinto con controles completamente distintos.

En el mundo de las aplicaciones web del lado cliente empezó a hablarse de las aplicaciones SPA:

SPA son las siglas de Single Page Application. Es un tipo de aplicación web donde todas las pantallas las muestra en la misma página, sin recargar el navegador

Pero veamos una definición más amplia que da wikipedia:

Una **single-page application (SPA),** o aplicación de página única, es una aplicación web o es un sitio web que cabe en una sola página con el propósito de dar una experiencia más fluida a los usuarios, como si fuera una aplicación de escritorio. En un SPA todos los códigos de HTML, JavaScript, y CSS se cargan una sola vez1 o los recursos necesarios se cargan dinámicamente cuando lo requiera la página, normalmente como respuesta a las acciones del usuario. La página no tiene que cargarse de nuevo en ningún punto del proceso y tampoco es necesario transferir a otra página, aunque las tecnologías modernas (como el pushState() API del HTML5) permiten la navegabilidad en páginas lógicas dentro de la aplicación.

En definitiva una SPA permite imitar el comportamiento de “cambio de página” (y por tanto de pantalla)

Un framework plenamente desarrollado y que permite hacer aplicaciones SPA es Angular. Eso dio lugar a que luego otros frameworks como Ionic tuvieran muy fácilmente la posibilidad de realizar aplicaciones para dispositivos móviles con múltiples pantallas con facilidad

React tiene un módulo de enrutado y permite hacer una aplicación SPA, y mediante React-Native incluso la posibilidad de generar aplicaciones para dispositivos móviles

Instalación del router

El proceso de instalación (ya sea con npm u otro medio) se encuentra descrito en la página oficial: https://reactrouter.com

Para la versión 6 en adelante el comando npm sería:

```
npm install react-router-dom
```

Haciendo nuestro primer router

Al seguir las actividades hemos realizado un componente Cronometro y relojes mundiales

Quizás quisiéramos poder tener accesibles los múltiples componentes a la vez en al aplicación React y poder pasar de uno a otro mediante enlaces, al estilo de una App SPA

El router se suele poner a nivel de la aplicación (componente App) porque queremos navegar de un componente a otro. Un posible App.tsx sería:

```
function App() {
  return (
    <div className="App">
      <BrowserRouter>
            <!-- en esta zona ponemos lo que queremos aparezca siempre -->
        <h3>Mi primer router</h3>
        <Navbar />
            <!-- dentro de <Routes> definimos las rutas de nuestra app -->
        <Routes>
          <Route path="/" element={<Cronometro />} />
          <Route path="/relojesmundiales" element={<RelojesMundiales />} />
          <Route path="/cronometro" element={<Cronometro />} />
        </Routes>
      </BrowserRouter>
    </div>
  );
}


function Navbar(){
  return(
    <nav className="Minavbar">
        <Link to="/"> Inicio </Link>
```

```
        <Link to="/cronometro"> Cronómetro </Link>
        <Link to="/relojesmundiales"> Relojes mundiales </Link>
    </nav>
  )
}

export default App;
```

Hemos señalado lo más relevante. Observar que ahora hay una etiqueta: <BrowserRouter> esa etiqueta se usa para definir el router (se una una única vez) **y lo que se ponga dentro aparecerá en todas las páginas**. Así en todas las páginas aparecerá la etiqueta h3 y también aparecerá el componente: Navbar que hemos definido unas líneas más abajo

Aparece la etiqueta <Routes> esa etiqueta define las url que van a existir para cada componente-página. Veamos un poco más de detalle:

```
        <Routes>
          <Route path="/" element={<Cronometro />} />
          <Route path="/relojesmundiales" element={<RelojesMundiales />} />
          <Route path="/cronometro" element={<Cronometro />} />
        </Routes>
```

Vemos que por defecto (el path raíz: "/") se activará el componente: Cronometro. Que si queremos navegar a: /cronometro nos mostrará el componente Cronometro y así con todos los otros componentes.

Es normal definir dentro de <BrowserRouter >las rutas principales (podemos tener otros archivos de rutas) y poner las partes "fijas" de nuestra aplicación aunque navegues de una página a otra. Así como antes dijimos en este caso va a aparecer siempre el h3 y el componente: <Navbar> Esas partes "fijas" debemos ponerlas justo al principio de BrowserRouter o al final

Ahora echemos un vistazo al componente Navbar que hemos creado:

```
    <nav className="Minavbar">
        <Link to="/"> Inicio </Link>
        <Link to="/cronometro"> Cronómetro </Link>
        <Link to="/relojesmundiales">Relojes mundiales </Link>
    </nav>
```

Aparece la etiqueta: <Link> que nos genera un enlace hacia otra página del router (al estilo de un <a href> pero mejor, ya que no obliga a recargar la página). Si queremos establecer estilos condicionales y más potencial en general usaremos la etiqueta: <NavLink> pero de momento con lo anterior bastará

Práctica 41: Reproducir el código anterior de tal forma que tengamos cargada en nuestra app 3 componentes que ya hemos hecho: Cronometro y RelojesMundiales y PersonasIMC Agregar un componente About. Ese componente lo que mostrará es nuestros datos: nombre, apellido, curso. Se debe poder navegar mediante el Navbar a todos los componentes. La ruta inicial: "/" lo que debe mostrar es el componente About

Práctica 42: Crear otra aplicación React donde el componente App tendrá un router, en esta ocasión, en lugar de "mi primer router" debe informar que es una aplicación de juegos y deben estar cargados varios de los componentes de juegos que hemos hecho: el memoriza8, acertarnumero Pudiendo pasar de un juego a otro gracias a nuestro router

Comunicando con una Api: Axios

Una parte que se da en muchas aplicaciones, es la comunicación con otros, almacenar y recuperar información de forma centralizada. Típicamente eso lo hacemos accediendo a una api

Para la gestión de los accesos a la api mediante consultas http vamos a utilizar axios:

npm install axios

Vamos a ver un ejemplo de petición Get a la api de pokemon y lo explicamos:

```
export default function PokemonCard(props: IProps) {
    const [cardData, setCardData] = useState<PokemonCardData>({} as PokemonCardData);
    const uri: string = "https://pokeapi.co/api/v2/pokemon/3/";

  useEffect(() => {
    async function getPokemonCard(direccion:string){
      const response = await axios.get(direccion);
      const newCard: PokemonCardData = {
        name: response.data.name,
        image: response.data.sprites.front_default
      }
      setCardData(newCard);
    }

    getPokemonCard(uri);
  }, []);

  return(
    <div className="PokemonCard">
        <h3>{cardData.name}</h3>
        <img src={cardData.image} alt={cardData.name} />
    </div>
  );
}
```

El código anterior, consulta por los datos de un pokemon (el 3), pero lo hace dentro de useEffect de forma asíncrona ¿ por qué ?

El useEffect() que se ha usado, se ejecuta únicamente cuando se monta el componente (observar el array vacío: [] que le pasamos a useEffect()). Al ser un componente que pretende mostrar los datos de un pokemon, lo lógico es que desde el inicio se carguen los datos de ese pokemon para mostrarlos. El motivo de hacer la consulta asíncrona es evidente: las apis pueden devolver con retrasos las consultas.

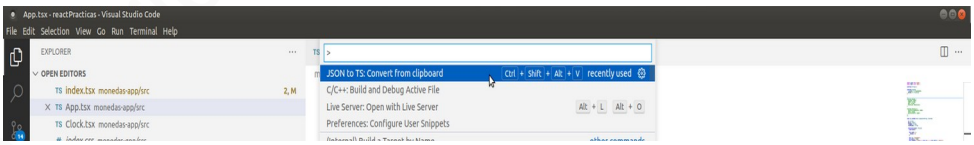
La información que tomamos de la api, típicamente se guarda en el state (cosiguiendo así que sea perdurable entre renderizados y a la vez obligar a que se redibuje el componente desde que estén los datos de la api disponibles)

Para evitar problemas, se recomienda que las funciones asíncronas ejecutadas por useEffect() se definan dentro del propio useEffect (si la función que le pasamos la tratamos de anotar como async() veremos un warning. Debemos resolver la asincronía y su saneamiento nosotros dentro del propio useEffect) . Así que la sintaxis que usaremos habitualmente será:

```
useEffect(() => {
   async function funcionasincrona(){ ...  /* consultar api y  guardar en state  lo obtenido de la api*/ }
   funcionasincrona();
}, [] );
```

Práctica 43: Crear otra aplicación React para trabajar con los pokemon.
Hacer dos componentes: PokemonListCard y PokemonCard
La lista obtiene el JSON de la url: https://pokeapi.co/api/v2/pokemon?offset=20&limit=20 que en el array: results aparecen 20 url de la api pokemon. Esas url se les pasarán como props a PokemonCard, sustituyendo la constante uri del ejemplo anterior por la información venida por props. Mostrando así de cada pokemon su nombre y su imagen. Agregar también su altura y peso (response.data.weight y response.data.height) con el correspondiente sufijo (el peso está en: kg y la altura en: m)

Hay una extensión que nos ayuda a obtener los tipos/interfaces de los datos typescript que corresponden al JSON que devuelva una API. Para desencadenarla copiar el texto en el portapapeles luego CTRL+SHIFT+P y elegimos json to ts:



Práctica 44: Crear otra aplicación React para trabajar con los datos de población de las capitales de provincia.
El INE publica en: https://servicios.ine.es/wstempus/js/es/DATOS_TABLA/2911?tip=AM
Pero habrá que adjuntar imagen de cada provincia (usar json-server con los datos ya preparados)
Hacer 3 componentes: CapitalesList, CapitalCard
CapitalesList toma la lista de las capitales y pasa como props a CapitalCard En CapitalCard aparecerá la imagen y el nombre de la capital de provincia

Hook useParams

En programación web es habitual enviar información de párametros en las url. Observemos el siguiente código con pokemon:

```
<ul>

  <li><a href="displaypokemon/21">Spearow</a> </li>
  <li><a href="displaypokemon/22">Fearow</a> </li>
  <li><a href="displaypokemon/37">Vulpix</a> </li>
</ul>
```

Al pulsar sobre cualquiera de los enlaces nos abra una página que tiene el mismo maquetado pero que difiere únicamente en los datos del pokemon que muestran

Si estuviéramos programando del lado del servidor, para que lo anterior funcione, se debe hacer que todas las rutas anteriores lleven a la misma página y se tome la información del id del pokemon (que en este caso es el número que aparece en la ruta) y al cargar la página se obtengan los dato de ese pokemon para ese id

Con lo anterior, también podríamos acceder directamente a cada pokemon por id, poniendo directamente en el navegador la ruta pertinente: Ej. /displaypokemon/23 → nos lleva a la página que gestiona el pokemon ekans

Sabemos que con React router podemos hacer una aplicación SPA y así poder hacer lo anterior. Para ello podemos crear un componente genérico: DisplayPokemon que viendo que recibe el número 23 como parámetro nos muestra la información específica del pokemon 23 Observar que esto NO es recibir la información por props. Al componente se accede directamente con la ruta establecida: /displaypokemon/23

Este hook prácticamente obliga al uso de Funtional Components y se evidencia que React camina hacia ahí. A partir de ahora veremos los ejemplos con Funtional Components

Para una api de tipos de cambio de diferentes monedas respecto al euro, el router quedaría:

```
import React from 'react';
import {  Link, Route, BrowserRouter, Routes } from 'react-router-dom';
```

```
import Monedas from './Monedas';
import ManageMoneda from './ManageMoneda';

export default function App() {
  return (
    <BrowserRouter>
      <h1>Aplicación Monedas</h1>
      <Navbar />

      <Routes>
        <Route path="/" element={<About />} />
        <Route path="/about" element={<About />} />
        <Route path="/monedas" element={<Monedas />} />
        <Route path="/moneda/:idmoneda" element={<ManageMoneda  />} />
      </Routes>
    </BrowserRouter>
  )
}

function Navbar() {
  return (
    <nav>
      <Link to="/"> Inicio </Link>  
      <Link to="/about"> Acerca de </Link>  
      <Link to="/monedas"> Monedas </Link>
    </nav>
  );
}
```

Lo único que se muestra nuevo es que hay una ruta que responde a parámetros de path. Así: /moneda/14 nos lleva al componente: ManageMoneda y se le pasa un parámetro llamado: idmoneda al componente-página. Cuando establecemos las rutas le decimos como es el parámetro mediante dos puntos: ":" Observar :idmoneda en el ejemplo: <Route path="/moneda/:idmoneda"

Veamos el nuevo componente: ManageMoneda

```
import axios from 'axios';
import { useEffect, useState } from 'react';
import { useParams } from 'react-router-dom';
import { Moneda } from './Monedas';

interface IState{  id: string, nombre: string, pais: string }

export default function ManageMoneda() {

  const [stmoneda, setstmoneda] = useState<IState>({} as Moneda);

  const { idmoneda }= useParams();

  let rutaDeMoneda = "http://localhost:8080/api/v1/monedas/";

  useEffect(
    () => {
      const getMoneda = async (monedaid) =>{

            let  response = await axios.get(rutaDeMoneda + monedaid);
            let moneda:Moneda = response.data;
            console.log(moneda);
            setStmoneda(moneda);
      }
      getMoneda(idmoneda);
    },
    [idmoneda]
  )



  return (
      <div>
        {JSON.stringify(stmoneda)}
      </div>
  )
}
```

La parte nueva del código es:

```
  const { idmoneda }= useParams();
```

vemos como se comporta el hook. Simplemente tomamos de la ruta el parámetro idmoneda que nos han pasado (recordar que en el router escribimos Route path="/moneda/:**idmoneda**")

También tenemos ejemplo de un useEffect que trabaja con una función asíncrona (getMoneda()) y que recibe un segundo parámetro aparte de la función

```
  useEffect(
    () => {
      const getMoneda = async (monedaid: string|undefined) =>{
            let rutaDeMoneda = "http://localhost:8080/api/v1/monedas/";
            let { data } = await axios.get(rutaDeMoneda + monedaid);
            let moneda:Moneda = data;
            console.log(moneda);
            setStmoneda({moneda});
      }
      getMoneda(idmoneda);
    }, [idmoneda]
  )
```

El segundo parámetro le está diciendo cuándo debe desencadenarse: Si recibe un array vacío: [ ] únicamente se desencadenará useEffect al cargar el componente

En general lo que se pasa entre corchetes es que variables tienen que cambiar para que se ejecute useEffect. Por ejemplo si apareciera: [count] le está diciendo a useEffect() que se ejecute si la variable count ha cambiado

Nota: se recomienda que cuando hay que hacer acciones asíncronas dentro de useEffect() La función que se utlice (en este ejemplo getMoneda()) se defina dentro de useEffect()

Práctica 45: Crear en la aplicación de pokemon, las rutas con parámetro: /pokemon/id y que envíe a un componente que cargue el pokemon correspondiente.
Hacer lo mismo con las capitales de provincia y que lo que se muestre e el componente con más información de la capital (componente Capital, no el componente CapitalCard)
En Capital.tsx tenemos un componente con la imagen, el nombre y los datos de población de los últimos años.

axios POST

En esta parte vamos a ver como crear nuevos objetos en la api. Primero un trozo de código de ejemplo de un componente que permitiría crear una nueva moneda. En este caso se llamará: CreateMoneda

```
import React from 'react'
import axios from 'axios';

export default function CreateMoneda() {
    function agregarMonedaApi(event:React.FormEvent<HTMLFormElement>){
        event.preventDefault();
        let formulario: HTMLFormElement = event.currentTarget;

        let inputnombremoneda: HTMLInputElement = formulario.nombremoneda;
        let inputpaismoneda: HTMLInputElement = formulario.paismoneda;

        let nombre:string = inputnombremoneda.value;
        let pais:string = inputpaismoneda.value;
        const newmoneda = {
            "nombre": nombre,
            "pais": pais
        }
        let ruta = "http://localhost:8080/api/v1/monedas";

        const axiospost = async(rutaDeMoneda:string)=>{
            try{
                const response = await axios.post(rutaDeMoneda, newmoneda )
                console.log(response.data);
            }catch(error){
                console.log(error);
            }
        }
        axiospost(ruta);
    }
    return (
        <>
           <form onSubmit={agregarMonedaApi}>
      Nombre: <input type="text"  name="nombremoneda"  /><br />
      País: <input type="text"   id="paismoneda" /> <br />
                <button  type="submit">Crear </button>
            </form>
        </>
    )
}
```

Vemos que **axios.post** recibe dos parámetros: la ruta para hacer el post, y los datos que se van a enviar para crear la moneda

Práctica 46: Reutilizar el ejemplo anterior para nuestra aplicación de capitales de provincia y mejorarlo de tal forma que se pueda crear una capital de provincia nueva . Observar que hay que agregar la ruta pertinente en el router. Se propone: /crearcapital y poner el link pertinente en el <nav>
Nota: no vamos a subir imágenes nuevas. Las imágenes ya estarán cargadas en json-server lo único que hacemos en el axios.post es decirle la ruta de la imagen

Práctica 47: Las opciones de: modificar capital y borrar capital son muy sencillas una vez dominado lo anterior. Buscar el funcionamiento específico de axios y crear los componentes de borrado y modificación pertinentes para realizar esas acciones

Práctica 48: En la actividad que hicimos del cálculo del IMC, donde mostrábamos una lista de personas donde cada una se representaba en un componente PersonaCard, hacer uso de json-server para ir agregando los objetos persona a la api y leerlos desde allí. Comprobar que quedan correctamente creados (el fichero json queda modificado. Y además la próxima vez que se arranque la aplicación tomará los datos actualizados)

Ponerle un router y tener soporte para rutas parametrizadas. Habilitar también el borrado de personas en la api

Hook useNavigate

En algunas ocasiones podemos querer cambiar de página/pantalla de forma programática (sin que el usuario pulse un enlace)

Un ejemplo de lo anterior es cuando hacemos un submit de un formulario. Quizás queramos que en lugar de seguir mostrando la misma página una vez pulsado el botón nos lleve a o otro sitio

Para lo anterior existe: useNavigate()

El funcionamiento es muy sencillo:

- declaración:

```
import { useNavigate } from 'react-router-dom';

export default function CreateMoneda() {
	let navigate = useNavigate();
```

- utilización:

```
function irainicio(){
	navigate("/");
}
return (
	<>
		<button onClick={irainicio}>Llévame al inicio</button>
	</>
)
```

Como vemos, basta usar la instrucción: **navigate**() para que nos lleve a donde queramos

Práctica 49: En nuestras apps de capitales y personas imc, al hacer la edición y pulsar en el botón que ejecuta el cambio en la api, usar el hook para que la app cargue directamente el componente raíz (se entiende que una vez se ha terminado de editar, no hay ningún interés en quedarse en el componente de edición)

Contexto: Datos compartidos entre componentes

En la siguiente sección abordaremos el login y la securización. **Un caso clásico de datos que deben estar disponibles para toda la aplicación es si hay un usuario válido logueado y su rol**

La forma básica de pasar información entre componentes (la mayor parte de las veces, el state y su setter) es mediante props si es de un componente padre a un hijo. Tal acción no tiene sentido (o se dificulta mucho) si estamos trabajando con un router en apps SPA.

También si la información no la necesita un hijo, ni un nieto, sino un bisnieto, nos vemos en la situación de estar pasando por las props información innecesaria para los componentes intermedios: hijo y nieto.

Un abordaje especialmente interesante es **Redux** que tiene la filosofía de guardar el state de todos los componentes en un único almacén disponible por toda la aplicación. Pero Redux implica un grado de complicación, que no es la mejor opción para muchas aplicaciones que son sencillas. Aparte que Redux es una librería de terceros, que no pertenece a React

¿Cómo establecemos un contexto ?

El contexto consiste en un componente que establecemos en el árbol de componentes a la altura que queremos que tenga alcance. Por ejemplo, si queremos que tenga como alcance todas las rutas de la aplicación, debemos establecer el contexto, como un ancestro de nuestras rutas:

```
function App() {
  return (
    <div className="App">
      <BrowserRouter>
        <AppContextProvider>
          <NavBar />

          <Routes>
            <Route path="/" element={<About />} />
            <Route path="/about" element={<About />} />
            <Route path="/relojesmundiales" element={<RelojesMundiales />} />
            <Route path="/cronometro" element={<Cronometro />} />
          </Routes>
        </AppContextProvider>
      </BrowserRouter>

    </div>
  );
}
```

En el ejemplo se muestra un componente llamado: AppContextProvider que envuelve todas las rutas de nuestra aplicación (y también el NavBar)

Una vez establecido el componente de contexto, si queremos usar el contexto en cualquier componente, se hace uso de un hook específico: **useContext**()

Bien, si queremos acceder a la información de usuario y su setstate correspondiente entre componentes padres e hijos haríamos uso de props. Cuando haces uso del contexto se hace algo similar, pero en lugar de tomar la información de props lo hacemos del hook useContext()

Así, supongamos que tenemos en nuestro contexto un state llamado user para el usuario logueado y su setstate correspondiente: setUser. Entonces en cualquier componente que precisemos usarlos ejecutaríamos:

```
const {user, setUser}=useContext(NOMBREDELCONTEXTO);
```

Ya con eso estaría. Ya sabemos que con useContext() podemos obtener lo que necesitemos del contexto. Pero nos falta saber como establecer que datos pertenecen al contexto

Veamos un ejemplo y luego explicamos:

```
import React, { createContext, useContext, useState, Dispatch, SetStateAction } from "react";
import { Usuario } from "./Usuario";


export interface AppContextType {
  user: Usuario;
  setUser: Dispatch<SetStateAction<Usuario>>
}

export const AppContext = createContext<AppContextType>({} as AppContextType);

export const AppContextProvider = (props: any) => {
  const [user, setuser] = useState<Usuario>({} as Usuario);
  const contextValues: AppContextType = {
    user: user,
    setUser: setuser
  };
  return (
    <AppContext.Provider value={contextValues}>
      {props.children}
    </AppContext.Provider>
  );
};

export const useAppContext = () => {
  return useContext(AppContext); // as AppContextType;
};

export default AppContextProvider;
```

La “receta” para crear un contexto y establecer los datos, consiste en primero crear el contexto. Para eso existe el hook: createContext()

```
export const AppContext = createContext<AppContextType>({} as AppContextType);
```

Vemos que se crea un contexto , que se llamará: AppContext y que los datos que va a servir (lo que se va a almacenar en el contexto) son del tipo: AppContextType Ese es un tipo personalizado que hemos creado nosotros con la información que precisamos guardar. Como nosotros lo que queremos guardar es la información del usuario (el state del usuario y su setstate) el tipo personalizado AppContextType es:

```
export interface AppContextType {
  user: Usuario;
  setUser: Dispatch<SetStateAction<Usuario>>
}
```

Observar el setUser (que es el setstate de usuario) Los setstate siguen el prototipo: Dispatch<SetStateAction> Nosotros le tenemos que decir a qué objeto se le está haciendo el state. En este caso es a un Usuario: Dispatch<SetStateAction<Usuario>>

Una vez creado el contexto con createContext() hay que crear el componente que se encargará de “proveer” el contexto que hemos creado. un Provider es exactamente eso. Veamos ejemplo:

```
export const AppContextProvider = (props: any) => {
  const [user, setuser] = useState<Usuario>({} as Usuario);
  const contextValues: AppContextType = {
    user: user,
    setUser: setuser
  };
  return (
    <AppContext.Provider value={contextValues}>
      {props.children}
    </AppContext.Provider>
  );
};
```

Lo primero que observamos en este componente es que se crea un useState() de Usuario y lo guardamos en la constante que hemos llamado: contextValues. Si ese objeto: contextValues que contien el state de usuario pudiéramos pasarlo a los componentes que lo precisen, todos ellos podrían manejar el state del usuario.

Para conseguir eso está el código que aparece en el return:

```
  return (
    <AppContext.Provider value={contextValues}>
      {props.children}
    </AppContext.Provider>
  );
```

Observar que los objetos que se crean con el hook createContext:

AppContext = createContext()

tienen un atributo llamado Provider: AppContext.Provider

Vemos que estamos poniendo en ese Provider los datos que queremos que estén disponibles en el contexto: `AppContext.Provider value={contextValues}`

El Provider es un envoltorio para todos los componentes que queramos que estén en el contexto. En React hay un atributo nativo en todo componente que viene con las props llamado: children Ese atributo hace referencia a todos los objetos hijos del componente que nos ha llamado

El último trozo de código, no es necesario, pero nos ayuda tener un hook personalizado que permita acceder al contexto:

```
export const useAppContext = () => {
  return useContext(AppContext);
};
```

Como vemos, lo único que hace es que nos ahorremos decir el nombre del contexto que ejecutará useContext: useContext(AppContext). Así, cuando un componente tenga que llamar al contexto lo que ejecutará es: useAppContext() en lugar de: useContext(AppContext)

Práctica 50: Cear un contexto en la aplicación de pokemon La idea es que haya un botón en cada PokemonCard que diga: “establecer favorito” de tal forma que si el usuario pulsa el botón, elige como su pokemon favorito el de la actual PokemonCard. Debe mostrarse los datos de ese pokemon en todo momento (Para ello se propone crear un componente llamado PokemonFavorite igual que el Navbar aparece en todo momento en el router, mostramos los datos de ese pokemon justo dentro de <Browserrouter> pero por fuera de <Routes>

Práctica 51: Cear un contexto en la aplicación de capitales y un componente <Login> accesible en el navbar del router. Este login únicamente guardará el nombre del usuario en el contexto (nada de contraseña ni roles) y en todos los componentes de la aplicación debe decir: “hola nombreusuario! “

Seguridad y persistencia en React

Supongamos que queremos tener una aplicación con recursos protegidos no accesibles a los no autenticados.

Si tenemos una API securizada tendrá un login que nos devolverá un token para poderlo usar en nuestras consultas.

Vamos a suponer que estamos en el caso de una api así que recibe JSON para el login. El siguiente componente: Login.tsx es válido:

```
export const Login = () => {
        function handleform(event:FormEvent){
                event.preventDefault();
                let formulario = event.currentTarget as HTMLFormElement;
                let inputNombre = formulario.nombre as HTMLInputElement;
                let inputPassword = formulario.password as HTMLInputElement;
                let nombre:string = inputNombre.value as string;
                let password:string = inputPassword.value;
                let login ={
                        name: nombre,
                        password: password
                }
                const axiospost = async(rutaDeMoneda:string)=>{
                        try{
                                const { data } = await axios.post(rutaDeMoneda, login )
                                localStorage.clear();
                                localStorage.setItem("token",data);
                        }catch(error){
                                console.log(error);
                        }
                }
                axiospost("http://localhost:8080/api/login");
        }
        return (
        <form onSubmit={handleform}>
            <input type="text" name="nombre" id="nombre" placeholder="nombre" /><br/>
 <input type="password" name="password" id="password" placeholder="password" /><br/>
            <button type="submit">Enviar</button>
         </form>
       )
}
```

La parte de código nueva es poca. Hemos destacado que hemos generado un literal json para pasar la información de nombre de usuario y password (habrá que poner lo esperado por la api). Por otro lado vemos el uso del **localStorage**

localStorage: persistencia en React

Cuando recargamos nuestra página/aplicación toda la información que tenemos en RAM puede "perderse" (probar con una de nuestras aplicaciones pulsar F5) Los navegadores tienen una forma de persistencia basada en clave/valor para almacenar información **Esta información siempre son string** (así que los objetos javascript habrá que hacerles un stringify). **localStorage** es un recurso de acceso global y no tenemos que importarlo para su uso

Veamos las dos sentencias que hemos marcado:

```
localStorage.clear();
localStorage.setItem("token",data);
```

La primera sentencia limpia todo lo que pueda estar almacenado en el localStorage. La segunda sentencia guarda: data (la información devuelta por la api, que era el token) en el localStorage asociado a una clave llamada: "token"

En este caso se ha hecho la presunción que la api nos ha devuelto texto plano, no json, en otro caso habría que hacer: **JSON.stringify(data)** para poder guardar en localStorage

Imaginemos ahora que queremos tomar la información del token almacenado en el localStorage. La forma de hacerlo es:

```
let token:string = localStorage.getItem("token") as string;
```

Volvemos a insistir que localStorage es un recurso global. La sentencia anterior **localStorage.getItem()** se puede ejecutar en cualquier parte de la aplicación y acceder al storage

Práctica 52: En la aplicación de pokemon guardar los datos del pokemon favorito, cuando se establezca o modifique en localstorage. De tal forma que cuando se inicie la App cargue la información del pokemon de local storage y lo ponga en el contexto.

Enviar token en las cabeceras de las peticiones axios

Vamos a ponernos en el caso de la aplicación monedas. Teníamos en la api un recurso usuarios que era accesible únicamente mediante token. Veamos un componente que solicita la información de toda la tabla usuarios:

```
interface IUsuario {
  idusuario: number;
  nombre: string;
}
export const Usuarios = () => {
  const [state, setState] = useState<IUsuario[]>([]);
  useEffect(() => {
    async function getData(){
      let token:string = localStorage.getItem("token") as string;
      let ip: string = "localhost";
      let puerto: number = 8080;
      let rutaBase: string ="http://" + ip + ":" + puerto + "/api/v3";
      let rutaMonedas: string = rutaBase + "/usuarios";
      let ruta = rutaMonedas;
      console.log(ruta);
      const headers = {
        headers: { Authorization: token }
      };
      let respuesta = await axios.get( ruta,headers);
      console.log(respuesta.data);
      setState( respuesta.data );
    }
    getData();
  }, [])

  return (
    <ul>
    {
       state.map(usu =>{
         return <li> {JSON.stringify(usu)} </li>
       })
    }
  </ul>
  )
}
```

La forma de enviar la información en la cabecera es pasando un parámetro adicional a la consulta axios:

```
    const headers = {
      headers: { Authorization: token }
    };
   let respuesta = await axios.get( ruta,headers);
```

Recordar que el token debe ir en una cabecera Authorization Estamos suponiendo también que el propio texto de la variable token ya incluye la palabra: Bearer pero en general la cabecera debe ser:

```
Authorization: Bearer tokenjwt
```

Observar que hemos supuesto que el token lo hemos almacenado en el localStorage y lo hemos recuperado de ahí

Control de acceso a rutas protegidas

Vamos a ponernos de nuevo en el caso de la aplicación monedas y no queremos que estén accesibles las rutas a los componentes protegidos, como el de Usuarios.tsx que nombramos antes

Para resolver eso anidamos nuestro componente que queremos proteger por otro que va a controlar el acceso en nuestro Router. Veamos ejemplo:

```
export default function App() {

  return (
    <BrowserRouter>
      <h1>Aplicación Monedas</h1>
      <Navbar />

      <Routes>
        <Route path="/" element={<Cronometro />} />
        <Route path="/cronometro" element={<Cronometro />} />
        <Route path="/reloj" element={<Clock />} />
        <Route path="/monedas" element={<Monedas />} />
        <Route path="/moneda/:idmoneda" element={<ManageMoneda  />} />
        <Route path="/crearmoneda" element={<CreateMoneda  />} />
        <Route path="/login" element={<Login  />} />
        <Route path="/usuarios" element={
          <RequireAuth >
              <Usuarios  />
          </RequireAuth>

        } />


      </Routes>
    </BrowserRouter>
  )
}
```

Vemos que se ha anidado el componente Usuarios dentro de uno llamado RequireAuth.

En React tenemos unas props especiales que son: **children** podemos acceder a los objetos JSX/TSX anidados gracias a esa opción

Vamos a ver el componente RequireAuth para ver como permite o deniega el acceso al componente Usuarios.tsx

```
import React from 'react'
```

```
import { Navigate,  useNavigate } from 'react-router-dom';

interface IProps{
    children: JSX.Element;
}

export const RequireAuth = ({children}: IProps) => {
    let autorizado = localStorage.getItem("token");

    if(autorizado){
        return children
    }
    return <Navigate to="/login" />
}
```

Lo primero que vamos a destacar es que hemos desestructurado de las props que hemos recibido, children para así obtener el objeto JSX.Element que está anidado: Usuarios.tsx

```
export const RequireAuth = ({children}: IProps) => {
```

Como dijimos antes en Typescript hay que decirle que children es del tipo JSX.Element

```
interface IProps{
    children: JSX.Element;
}
```

Y finalmente vamos a ver un nuevo objeto del Router: <Navigate>

Con <Navigate> podemos devolver un componente que nos lleva a otra ruta (en este caso al login), es como usar el hook useNavigate() pero con componentes. Observar que hay que hacer un import explícito

```
import { Navigate,  useNavigate } from 'react-router-dom';
```

ahora el objeto Navigate devuelto:

```
return <Navigate to="/login" />
```