

APUNTES REACT 1

Abel Rios - 16/05/2022

Cositas: Añadir a Chrome la extensión “React Developer Tools”

En el Visual Studio añadir la extensión: ES7 React/Redux/React-Native Snippets

Documentación:

<https://es.reactjs.org/docs/create-a-new-react-app.html>

Para empezar:

```
npx create-react-app my-app  
cd my-app  
npm start
```

Si nos fijamos en el archivo app.js, es básicamente una función (o un archivo) de javascript que retorna html (literal). Eso se denominan archivos .jsx

Estos archivos sólo pueden devolver (return) un elemento padre (un div normalmente, pero puede ser otro elemento, pero sólo uno). Tan grande como tú quieras, pero sólo un elemento.

De tal forma en react se usa la estructura de componentes, que significa que tendremos un archivo que creará el header de la página, otro archivo para el footer y luego archivo/s que sean los componentes del body de las distintas páginas de nuestra web.

Muy importante que nuestros **componentes tengan la primera letra en MAYÚSCULA**, si no la tuvieran en mayúscula, react puede (y suele) confundirlo con una etiqueta html.

Estos componentes los guardamos en una carpeta que llamaremos “components” dentro de nuestra carpeta “src” del proyecto.

Para cada componente crearemos una carpeta (con la primera letra en Mayúscula) y en esa carpeta crearemos un archivo index.js y un archivo con nuestro componente (primera letra en mayúscula). En ese index diremos:

```
export { default } from "./Title";  
// Cuando nuestro app viene a importar a la carpeta Title, lo primero  
que busca es el archivo index  
// de ésta forma, si tenemos varios componentes en la misma carpeta, al  
ir haciendo un "export" en este archivo
```

```
// de todos los componentes de la carpeta, en nuestro app.js sólo  
tenemos que hacer un import y cortito  
// no doscientos imports uno por cada componente  
// O si nuestro componente tiene a su vez más componentes, nuestro  
import no va a ser super largo
```

Para las etiquetas CSS, en React no podemos llamarlas “class” porque es un nombre reservado, para ello usamos la etiqueta “className”. Estos archivos CSS no son necesarios de importarlos en el index del componente, tan solo en el JS del componente.

Volviendo a que react sólo devuelve un elemento padre, en lugar de hacer un div, podemos importar desde react el tipo Fragment:

```
import { Fragment } from "react";
```

Y usar la etiqueta <Fragment> </Fragment>, que funciona como “contenedor” del html que queremos devolver, pero no tiene valor semántico ninguno. No es div ni nada, de hecho si inspeccionamos luego nuestra web, en el html no aparece nada, sí aparece por supuesto el código que hemos generado en react.

Pero ahora empezamos con la **MAGIA**:

Más fácil todavía, el Fragment se puede decir como <> y </>. Y no es necesario ni importar la clase Fragment desde react.

*** Propiedades / Props ***

Props es el objeto que por defecto SIEMPRE le llega al componente en React. Por ello, para no tener que ir escribiendo siempre props.name, o props.precio o lo que sea, **desestructurar el objeto props**.

Forma #1:

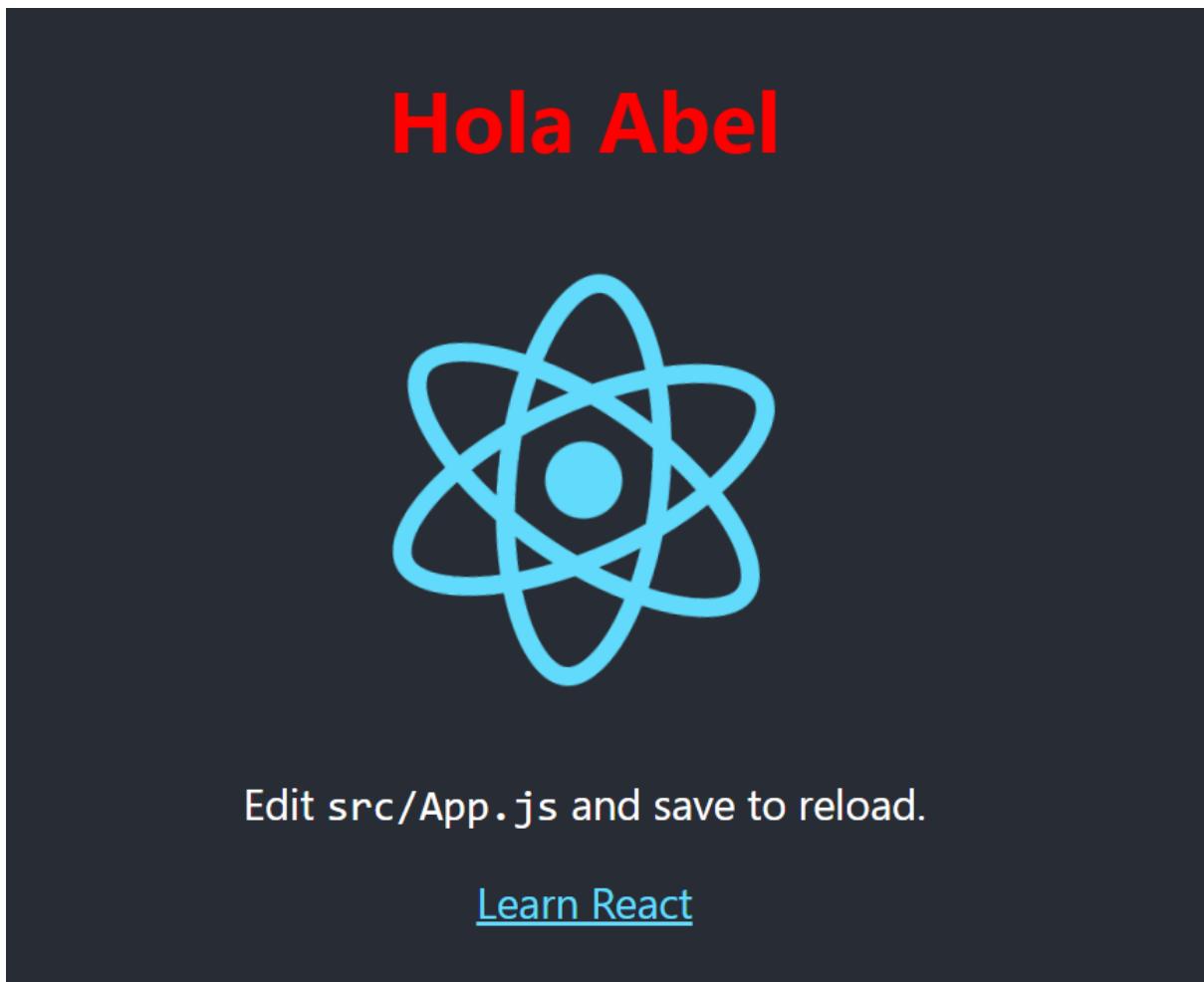
Title.js

```
proyecto1 > src > components > Title > JS Title.js > ...  
1  import "./Title.css";  
2  
3  export default function Title(props){  
4  
5      console.log(props);  
6      return(<h1 className="colorFont"> Hola {props.name} </h1>)  
7  
8  }
```

App.js

```
/          <>
8           <p> Header </p>
9           <header className="App-header">
10          <Title name="Abel" />
11          <img src={logo} className="App-logo" alt="logo" />
12          <p>
13            | Edit <code>src/App.js</code> and save to reload.
```

Web:



Forma #2 (desestructuración de objeto):

Title.js

```
 proyecto1 > src > components > Title > JS Title.js > ...
  1 import "./Title.css";
  2
  3 export default function Title(props){
  4
  5   console.log(props);
  6   const {name} = props;
  7   return(<h1 className="colorFont"> Hola {name} </h1>)
  8
  9 }
10
11
12
```

Forma #3 (desestructurando en la misma línea de argumentos de la función)

Title.js

```
 proyecto1 > src > components > title > JS Title.js > ...
  1 import "./Title.css";
  2
  3 export default function Title({ name }){
  4
  5   console.log(name);
  6   return(<h1 className="colorFont"> Hola {name} </h1>)
  7
  8 }
  9
```

*** Reglas No Escritas a la hora de Crear Componentes ***

- Siempre dentro de la carpeta “components”
- Siempre con la primera letra en Mayúscula
- Opcional, crear un archivo “index.js” que importe los componentes
- Hacer el destructuring del objeto props en los paréntesis de la función o en una línea nueva
- Exportar el componente
- Importarlo donde lo queramos usar y añadirlo al archivo jsx

*** Otro ejemplo de props ***

Alumno.js

```
export default function Alumno({alumno}) {  
  
  return (  
    <ul>  
      <li>{alumno.nombre}</li>  
      <li>{alumno.edad}</li>  
      <li>{alumno.email}</li>  
      <li>{alumno.telefono}</li>  
    </ul>  
  )  
}
```

index.js (Alumno)

```
export { default } from "./Alumno"
```

app.jsx

```
import Alumno from './components/Alumno';  
  
const alumno = { //Estamos suponiendo que ésto nos viene desde la base  
de datos  
  nombre: "Abel Rios",  
  edad: 31,  
  telefono: "666666666",  
  email: "abel@mail.com"  
}  
  
function App() {  
  return (  
    <>  
      <p> Header </p>  
      <header className="App-header">  
        <Title name="Abel" />  
        <Alumno alumno={alumno}/>  
        <img src={logo} className="App-logo" alt="logo" />  
    </>  
  )  
}  
  
export default App;
```

Y ahora vamos a hacer un componente que reciba un array de objetos (alumno)

Alumnos.js

```
import Alumno from "../Alumno/Alumno"

export default function Alumnos({alumnos}) { // donde alumnos va a ser
un array de {alumno}

    return( // react no se lleva guay con los métodos de los array,
para ello usamos un fragment <></>
    <>
    {
        alumnos.map((student) => (<Alumno alumno =
{student}>))
            // Ponemos el componente Alumno para que nos lo
transforme en el ul + los li
            // como hemos definido en el componente Alumno
        }
    </>
)

}
```

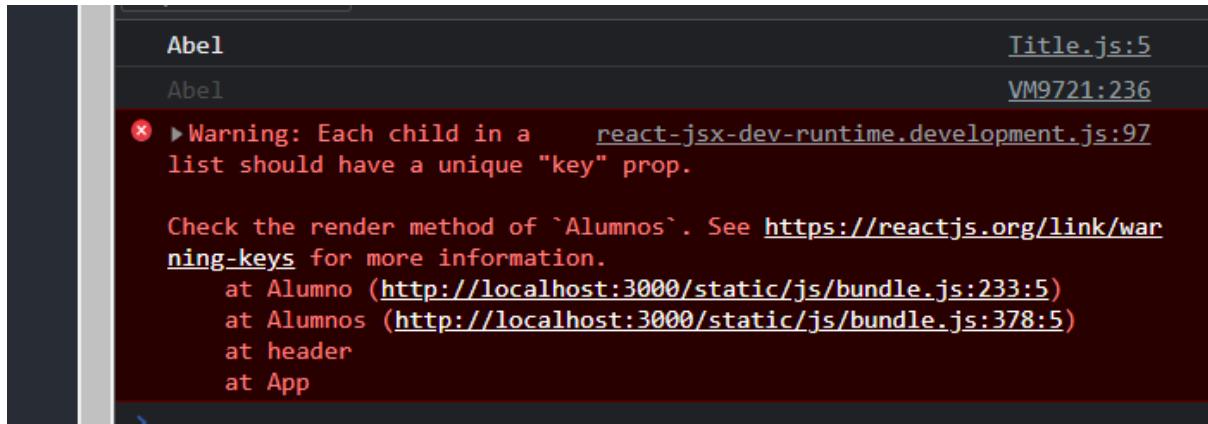
index.js (Alumnos)

```
export { default } from "./Alumnos"
```

app.jsx

```
4  import Alumno from './components/Alumno';
5  import Alumnos from './components/Alumnos';
6
7  const alumnos = [
8  {
9    nombre: "Abel Rios",
10   edad:31,
11   telefono: "666666666",
12   email: "abel@mail.com"
13 },
14 {
15   nombre: "Abel Rios",
16   edad:31,
17   telefono: "666666666",
18   email: "abel@mail.com"
19 },
20 {
21   nombre: "Abel Rios",
22   edad:31,
23   telefono: "666666666",
24   email: "abel@mail.com"
25 },
26 {
27 }
28 ]
29
30
31 const name = "Abel";
32
33
34 function App() {
35   return (
36     <>
37       <p> Header </p>
38       <header className="App-header">
39         <Title name = {name}/>
40         <Alumnos alumnos={alumnos}/>
41         <img src={logo} className="App-logo" alt="logo" />
```

esto nos va a dar un “error” en la consola:



The screenshot shows a browser's developer tools console. It displays two entries: "Abel" and "Abel". Below these, a red warning message is shown: "Warning: Each child in a list should have a unique "key" prop." The message includes a link to <https://reactjs.org/link/warning-keys>. The stack trace shows the warning originated from the "Alumno" component at "localhost:3000/static/js/bundle.js:233:5" and the "Alumnos" component at "localhost:3000/static/js/bundle.js:378:5".

porque estamos creando hijos que no tienen una “key”, para ello ahora vamos a hacer:

Alumnos.js

```
{  
  alumnos.map((student) => (<Alumno key= {student.nombre} alumno = {student}/>))  
  // Ponemos el componente Alumno para que nos lo transforme en el ul + los li  
  // como hemos definido en el componente Alumno  
  // y usamos el nombre (que se supone que es el id) como key  
}
```

Aunque lo óptimo es que cada alumno tuviera un id (único del alumno)

Dentro del map también podemos implementar funciones lógicas, ejemplo:

```
import Alumno from "../Alumno";  
  
export default function Alumnos({ alumnos }) {  
  return (  
    <>  
      {alumnos.map((student) => {  
        if (student.nombre === "Nacho Viano") {  
          return <></>;  
        }  
        return <Alumno key={student.id} alumno={student} />;  
      })}  
    </>  
  );  
}
```

Que nos devolvería “nada” si el nombre del estudiante es Nacho Viano.

O añadir un operador ternario:

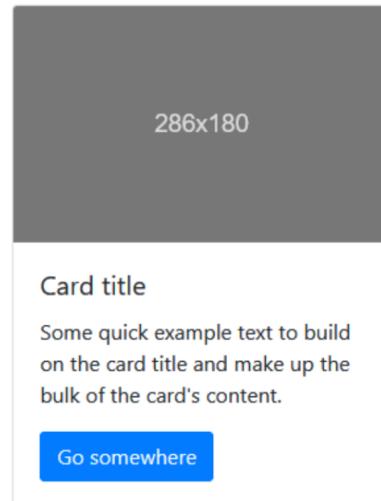
```
1 import Alumno from "../Alumno";
2
3 export default function Alumnos({ alumnos }) {
4   return (
5     <>
6       {alumnos ? (
7         alumnos.map((student) => <Alumno key={student.id} alumno={student} />)
8       ) : (
9         <h1>No hay alumnos</h1>
10      )}
11    </>
12  );
13}
14
```

Ejercicio1:

Ejercicio 1

Vamos a crear nuestro primer componente

1. Inicializaremos el proyecto con *create-react-app*.
2. Crearemos el componente Card, con un aspecto similar al de la imagen. Lo haremos directamente en *App.js*.
3. Lo utilizaremos dentro del componente principal App y comprobaremos que se muestra correctamente.
4. Refactorizaremos nuestro componente para que esté en un archivo independiente dentro de una carpeta llamada “components”.
5. Haremos que reciba por props la siguiente información: URL de la imagen, título, párrafo, enlace del botón y texto del botón.



Para este ejercicio vamos a importar Bootstrap en un proyecto de React.

<https://www.npmjs.com/package/bootstrap>

“npm i bootstrap”

y en el index importamos:

```
import "bootstrap/dist/css/bootstrap.min.css"
```

Solución:

```
ejercicio1 > src > components > Card > JS Card.js > Card
1  export default function Card({tarjeta}) {
2
3    return(
4      <div className="card" style={{ width: '18rem' }}>
5        <img src={tarjeta.img} className="card-img-top" alt="Imagen" />
6        <div className="card-body">
7          <h5 className="card-title">{tarjeta.titulo}</h5>
8          <p className="card-text">{tarjeta.texto}</p>
9          <a href={tarjeta.enlace} className="btn btn-primary">{tarjeta.textoBoton}</a>
10         </div>
11       </div>
12     )
13   }
14 }
```

APUNTES REACT 2

Abel Ríos - 17/05/2022

Ejercicio 2:

Ejercicio 2

1. Creamos un nuevo proyecto React.
2. Creamos un componente llamado `Tienda.js` que tendrá un objeto como el que se muestra en la imagen:

```
const tienda = {
  electronica: [
    {id: 27, producto: "Televisor", marca: "LG", modelo: "XP7302", precio: 399},
    {id: 28, producto: "Equipo Hi-FI", marca: "Samsung", modelo: "VF235", precio: 179},
    {id: 29, producto: "Televisor", marca: "Sony", modelo: "Bravia-173", precio: 498},
  ],
  alimentacion: [
    {id: 30, producto: "Galletas", marca: "María", precio: 0.90},
    {id: 31, producto: "Ensalada", marca: "Imizurra", precio: 1.30},
    {id: 32, producto: "Patatas", marca: "McKain", precio: 0.90},
    {id: 33, producto: "Pasta", marca: "Gallo", precio: 0.90},
  ],
  mascotas: [
    {id: 34, producto: "Croquetas para gato", marca: "Purina", precio: 4.90},
    {id: 35, producto: "Arena de gato", marca: "Limpior", precio: 1.10},
  ]
};
```

Ejercicio 2 (cont.)

3. Creamos un nuevo componente llamado `Lista.js` y lo renderizamos tres veces en `Tienda.js`.
4. A las listas les pasamos un atributo con el nombre de cada categoría y otro con los productos correspondientes.
5. En la extensión del navegador, comprobamos que los componentes reciben los props.
6. Lista tendrá un título con el nombre de la categoría y tantos list items como productos reciba.

Creamos nuestro componente Tienda y guardamos el objeto tienda tocho en él (aunque debería venir de la base de datos). Para empezar vamos a desestructurar el objeto tienda en tres objetos que sean las distintas categorías:

```
20  const {electronica,mascotas,alimentacion} = tienda
21  // Desestructuramos el objeto tienda en tres objetos: electronica, mascotas y alimentación
22  // estos objetos los podemos usar ahora sin necesidad de decir tienda.electronica (por ejemplo)
23
```

Ahora creamos el componente Lista, que será un map de los arrays (categorías) pintados como ul y li.

```
ejercicio2 > src > components > Lista > JS Lista.js > >List
1  export default function Lista({ productos , categoria }) {
2  // Recibe dos objetos distintos, pero van dentro de la misma llave
3  |
```

Lista.js (le añadimos la key para que no nos de el warning en la consola)

```
return (
  <>
    <h1>{categoria}</h1>
  {
    productos.map((item) => {
      return (
        <ul key={item.id}>
          <li>{item.id}</li>
          <li>{item.producto}</li>
          <li>{item.marca}</li>
          <li>{item.modelo}</li>
          <li>{item.precio}</li>
        </ul>
      )
    })
  }
)
```

Ahora en nuestro componente Tienda.js importamos Lista.js

```
ejercicio2 > src > components > Tienda > JS Tienda.js > ...
1   import Lista from "../Lista/Lista"
2
```

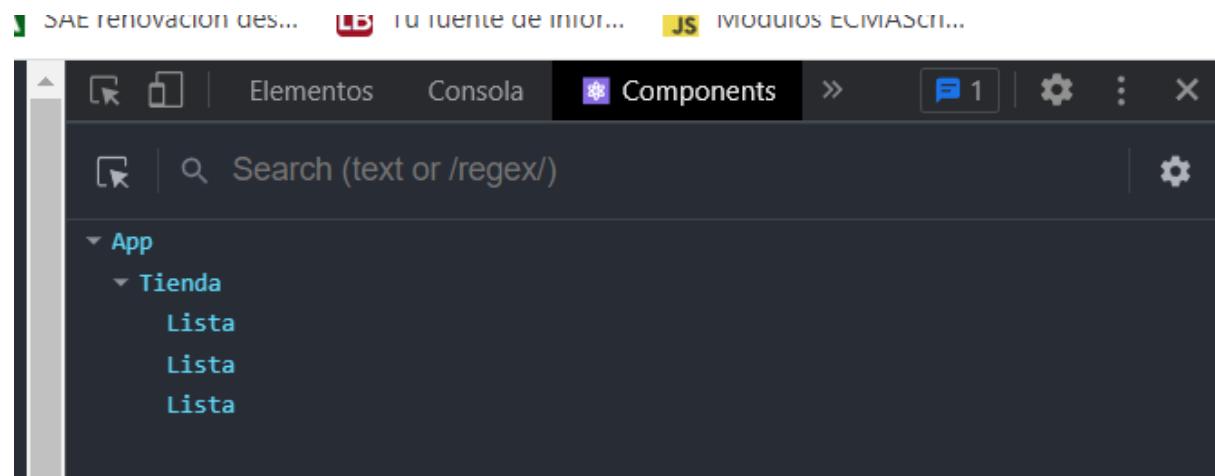
Y creamos la función del componente:

```
24  export default function Tienda(){
25    return(
26      <>
27        <Lista productos = {electronica} categoria="Electronica"/>
28        <Lista productos = {alimentacion} categoria="Alimentacion"/>
29        <Lista productos = {mascotas} categoria="Mascotas"/>
30      </>
31    )
32 }
```

Y en nuestro archivo App.jsx sólo tenemos que importar y llamar al componente Tienda:

```
3   import Tienda from './components/Tienda/Tienda';
4
5   function App() {
6     return (
7       <div className="App">
8         <header className="App-header">
9           <p>Ejercicio 2 - React</p>
10          <p>Abel Rios</p>
11          <Tienda/>
12        </header>
13      </div>
14    );
15  }
```

Si entramos al navegador, con la extensión que instalamos ayer, podemos ver los componentes:



Desestructuración

Podemos “dividir” el objeto, o mejor dicho, sacar las propiedades a una copia fuera para mejor manejo. Para ello, por ejemplo:

```
const usuario = {  
    name: "Nacho",  
    surname: "Viano",  
    age: 23,  
    phone: 66666666  
};
```

```
const { name, surname, ...rest } = usuario;
```

Estamos creando una constante “name”, una constante “surname”, una constante “rest” con el resto de propiedades. Si las propiedades son simples (string, número, boolean) se pasan por valor, pero si la propiedad es un objeto, se pasa por referencia y si se modifica, se modifica en el objeto primario:

```
2  const usuario = {  
3      name: "Nacho",  
4      surname: "Viano",  
5      age: 23,  
6      phone: 66666666,  
7      address: {  
8          street: "Calle larios",  
9          number: 3  
10     }  
11 };  
  
12  
13 const array = ["Nacho", "Raul", "Rocio", "Paula"];  
14  
15 let { name, surname, address } = usuario;  
16 address.street = "Pepito";  
17 console.log(usuario);  
18
```

Consola:

```
▼ {name: "Nacho", surname: "Viano", age: 23, phone: 66666666, address: Object}
  name: "Nacho"
  surname: "Viano"
  age: 23
  phone: 66666666
  ▼ address: Object
    street: "Pepito"
    number: 3
```

Ésto mismo se puede hacer con arrays también, y metemos el valor de la posición del array en una variable:

```
const array = ["Nacho", "Raul", "Rocio", "Paula"];
const [profe, ...rest]=array

console.log(profe)
```

consola:

```
Nacho
```

APUNTES REACT 3

Abel Rios - 18/05/2022

Hooks

Documentación: <https://es.reactjs.org/docs/hooks-state.html>

Un Hook es una función especial que permite “conectarse” a características de React. Hay que importarlos desde “react”.

En el ejemplo vamos a usar el hook **useState** que permite añadir el estado de React a un componente de función.

Cada hook se invoca con un “estado” y con una función:

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

En el ejemplo estamos haciendo un contador. La variable count es el estado (que será un número) y setCount es una función que deberemos implementar:

```
1 import "./styles.css";
2 import {useState} from 'react'
3
4 export default function App() {
5   const [count, setCount] = useState(0);
6
7   function increment(){
8     setCount(count + 1)
9   }
10
11  return (
12    <div className="App">
13      <h1>Hello useState</h1>
14      <p>Contador: {count}</p>
15      <button onClick={increment}>Incrementar</button>
16    </div>
17  );
18}
19
```

Estos hooks pueden generar fallos al funcionar de manera asíncrona. Por ejemplo vamos a usar una función setTimeout:

(Forma Imperativa)

```
4  export default function App() {
5      const [count, setCount] = useState(0);
6
7      function increment() {
8          setTimeout(function () {
9              return setCount(count + 1);
10         }, 5000);
11     }
12
13     return (
14         <div className="App">
15             <h1>Hello useState</h1>
16             <p>Contador: {count}</p>
17             <button onClick={increment}>Incrementar</button>
18         </div>
19     );
20 }
21
22 /**
23 * Estado inicial (count = 0)
24 * Estado 2 => count + 1 cuando count vale 0
25 * Estado 3 => count + 1 cuando count vale 0
26 */
```

De tal forma, aunque clickemos muchas veces en el botón de incrementar, hasta que no pasen los 5 segundos y la variable count tenga como valor 1 no funcionará el botón. Para ello, en lugar de pasarle a la función setCount la variable, le pasamos otra función (flecha):

(Versión Funcional)

```
4  export default function App() {
5      const [count, setCount] = useState(0);
6
7      function increment() {
8          setTimeout(function () {
9              return setCount((currentCount) => currentCount + 1);
10         }, 5000);
11     }
12
13     return (
14         <div className="App">
15             <h1>Hello useState</h1>
16             <p>Contador: {count}</p>
17             <button onClick={increment}>Incrementar</button>
18         </div>
19     );
20 }
```

Esto es muy útil a la hora de hacer llamadas a APIs puesto que éstas no nos devuelven los datos instantáneamente. Aquí creamos una variable currentCount. El setCount (que es una función que hemos llamado como queremos, pero que se crea desde el hook useState) sabe que si recibe una función, el parámetro que le llegue va a ser lo mismo que la variable de cuando lo hemos definido, por lo que automáticamente asigna a currentCount el valor de count, ahorrándonos nosotros el inicializar la variable de la función flecha.

** En la carpeta de React tenemos otro ejemplo llamado “ejemplohookstate” donde hacemos una lista de personas usando dos useState distintos con un formulario.

Ejercicio 3 – Agenda

1. Crear un nuevo proyecto React y limpiamos lo que no nos haga falta.
 2. Nuestra App contendrá dos componentes: una agenda y un formulario, cada uno en su archivo.
 3. Al nivel de la App, definiremos un array de contactos que serán objetos con la siguiente información:
 - Nombre, Apellidos, Dirección, Ciudad, Código Postal y teléfono.
- Crear 3 contactos iniciales.
4. Crear el componente agenda que recibirá por props todos los contactos y devolverá una `ul` por cada uno. La `ul` tendrá tantos `li` como datos tenga el contacto. Los `li` se pueden *hardcodear* o hacer dinámicos.
 5. Crear la estructura del formulario en el componente, aún sin lógica.
 6. Cambiar el array de contactos para que pase a ser un estado con su valor inicial. (hook useState). Pasar el array por props al componente agenda y deberá funcionar igual que el Apartado 4.

Ejercicio 3 – Agenda (cont.)

7. Dentro del componente del formulario, crearemos tantos hooks de useState como inputs tenga. Iremos actualizando los estados con los eventos change de los inputs de manera individual.
8. Cuando se produzca el submit en el form:
 1. Evitaremos el comportamiento por defecto.
 2. Crearemos un nuevo objeto con la información de todos los estados (*inputs*) en ese momento.
 3. Llamaremos a la función que modifica el estado de los contactos (apartado 3) y le pasaremos el array tras añadirle el nuevo objeto.
 4. Vacaremos los campos del formulario.
9. **Avanzado:** Cada lista (contacto) tendrá un botón que eliminará dicho contacto del array global.

Nota: Para añadir un nuevo elemento a la lista, se puede utilizar el operador spread (...) : `setContactos(contactosActual => [...contactosActual, nuevoContacto]);`

Ejercicio 3 – Agenda (cont.)

Aspecto final aproximado usando Bootstrap 4

Nota: El formulario irá debajo de la lista.

Lista de contactos

Contacto 0	Contacto 1
Federica	Rafael
Rica America	Remar Martinez
748452178	667542184
Calle Ángustias, Nº 27, 2ºB, 29006, Málaga	Calle Hipófeses, Nº 11, 3ºA, 29006, Málaga
<button>Eliminar</button>	<button>Eliminar</button>

Nuevo contacto

Introduce un nombre
Introduce los apellidos
Introduce la dirección
Introduce la provincia
Introduce el código postal
Introduce el número de teléfono
<button>Registrar</button>

APUNTES REACT 4

23/05/2022

Haciendo el Ejercicio 3:

Comenzamos declarando el array de contactos (arrayAgenda) en el archivo app.jsx

A nuestra agenda (o array de contactos mejor dicho) le haremos un useState, en el que el estado inicial será el arrayAgenda que hemos declarado al principio y una función setAgenda que nos permitirá modificar el estado del array.

```
37  function App() {  
38  
39    // const [agenda, setAgenda] = useState(JSON.parse(localStorage.getItem('agenda')));  
40    const [agenda, setAgenda] = useState(arrayAgenda);  
41
```

Creamos el componente Agenda, que básicamente es un map que nos crea una tarjeta por cada elemento del array.

```
contacts.map(({ name, surname, address, city, postCode, phoneNumber },index) => [  
  // el map acepta un parámetro index (o i o como lo quieras llamar) que guarda el valor de  
  // la iteración en la que se encuentra  
  return (  
    <div className="card" style={{ width: '18rem' }}>  
      <div className="card-header" style={{background: '#00AA9E'}}>Contacto {index+1}</div>  
      <ul className="list-group list-group-flush" key={index+1}>  
        <li className="list-group-item">{name}</li>  
        <li className="list-group-item">{surname}</li>  
        <li className="list-group-item">{address}</li>  
        <li className="list-group-item">{city}</li>  
        <li className="list-group-item">{postCode}</li>  
        <li className="list-group-item">{phoneNumber}</li>  
        <button type="submit" className="btn btn-danger">Eliminar</button>  
      </ul>  
    </div>  
)  
]
```

Creamos el componente Formulario, en el que hacemos un formulario de entrada de datos de nuevos contactos:

```
<form className="formularioEjemplo">
  <div className="form-group">
    <label for="inputName">Name</label>
    <input type="name" className="form-control" placeholder="Introduce Name"/>
  </div>
  <div className="form-group">
    <label for="inputSurname">Surname</label>
    <input type="surname" className="form-control" placeholder="Introduce Surname"/>
  </div>
  <div className="form-group">
    <label for="inputAddress">Address</label>
    <input type="address" className="form-control" placeholder="Introduce Address"/>
  </div>
  <div className="form-group">
    <label for="inputCity">City</label>
    <input type="city" className="form-control" placeholder="Introduce City"/>
  </div>

  <div className="form-group">
    <label for="inputPostCode">PostCode</label>
    <input type="postcode" className="form-control" placeholder="Introduce PostCode"/>
  </div>
  <div className="form-group">
    <label for="inputPhoneNumber">Phone Number</label>
    <input type="phonenumber" className="form-control" placeholder="Introduce Phone Number"/>
  </div>
  <button type="submit" className="btn btn-primary">Registrar</button>
</form>
```

Una vez creados éstos, vamos a proceder a usar los estados y el useState. En nuestro componente agenda no nos hace falta por ahora, así que seguimos en el Formulario.

Creamos un objeto base (baseContact) y lo usamos como estado base de contact.

```
const baseContact = {
  name:"",
  surname:"",
  address:"",
  city:"",
  postCode:"",
  phoneNumber:""
}

const [contact, setContact] = useState(baseContact);
```

Ahora, para cada input del formulario tenemos que crear una función que lo maneje (handle) y que nos actualice ese valor en nuestro objeto contact.

```
function handleInputName(event){
    setContact({...contact, name:event.target.value})
}

function handleInputSurname(event){
    setContact({...contact, surname:event.target.value})
}

function handleInputAddress(event){
    setContact({...contact, address:event.target.value})
}

function handleInputCity(event){
    setContact({...contact, city:event.target.value})
}

function handleInputPostCode(event){
    setContact({...contact, postCode:event.target.value})
}

function handleInputPhoneNumber(event){
    setContact({...contact, phoneNumber:event.target.value})
}
```

Por lo que en nuestro formulario, en cada input, tendremos que decirle que onChange llame a su correspondiente función:

```
<div className="form-group">
  <label for="inputName">Name</label>
  <input type="name" onChange={handleInputName} className="form-control" placeholder="Introduce Name"/>
</div>
```

Ahora, para poder añadir el objeto contact que hemos creado a nuestro array de contactos, necesitamos que el componente Formulario se comunique con su padre (App).

Primero vamos a decirle al formulario que cuando se cliqueé el submit nos llame a la función que lo maneja (handleSubmit)

```
<form onSubmit={handleSubmit}>
```

Y declaramos nuestra función handleSubmit. Esta función necesita usar el array que está en App, por lo que al componente Formulario le tendremos que pasar por props una función manejadora del array de contactos (handleContacts), que será la misma función setAgenda que declaramos al principio.

Formulario.js

```
export default function Formulario({handleAgenda}) {
```

App.js

```
<Formulario handleAgenda={setAgenda}/>
```

Formulario.js

```
function handleSubmit(event){  
    event.preventDefault();  
    handleAgenda((agenda) =>[...agenda,contact])  
    setContact(baseContact);  
}
```

Se me había olvidado en los inputs del formulario agregar el valor “value”, de ésta forma, al agregar en el submit y resetear el contact (setContact(baseContact)) nuestro formulario se reinicia y queda en blanco.

```
<label for="inputName">Name</label>  
<input type="name" value={contact.name} onChange={handleInputName}>
```

Y así en todos los demás.

Además, he agregado las funciones de almacenamiento en el local storage para guardar mi agenda en el navegador y que no se reinicie al recargar la página.

App.jsx

```
const [agenda, setAgenda] = useState(JSON.parse(localStorage.getItem('agenda')) || arrayAgenda);
```

y más abajo:

```
<Formulario handleAgenda={setAgenda}/>  
{localStorage.setItem('agenda', JSON.stringify(agenda))}  
</header>
```

Ahora nos queda el botón de eliminar (y también vamos a refactorizar los handle de los inputs, para hacer un sólo handle y ahorrarnos pechá de código).

Para el botón eliminar nos hace falta un id individual de cada objeto para poder interactuar con él. Para ello he usado la función Date.now() que genera el número de milisegundos de ahora mismo contando desde hace un año en concreto.

App.jsx

```
const arrayAgenda = [
  {
    id: Date.now(),
    name: "James",
    surname: "Hook",
    address: "Boat St.",
    city: "Neverland",
    postCode: "001",
    phoneNumber: "666111222"
  },
]
```

Formulario.js

```
const [contact, setContact] = useState(baseContact);

contact.id = Date.now();

function handleInputName(event){
  setContact({...contact, name: event.target.value})
}
```

Y en nuestro archivo Agenda, modificamos el botón y le añadimos la función handleBoton:

```
<li className="list-group-item">{phoneNumber}</li>
<button type="button" onClick={()=>handleBoton(id)} className="list-group-item">
```

OJO si no hacemos la función flecha, aunque esté en el onClick, lo ejecutaría automáticamente y tal como te pinta todas las tarjetas en pantalla te las elimina del tirón, dejándote la pantalla en blanco.

También hemos modificado el map añadiendo el id (aunque no lo usemos en los li):

```
contacts.map(({id, name, surname, address}) =>
```

Ahora, como vamos a modificar el estado del array, nuestro componente Agenda necesita el setter del estado del array (setAgenda)

App.jsx

```
<Agenda contacts={agenda} handleDelete={setAgenda}/>
```

Agenda.js

```
export default function Agenda({ contacts, handleDelete }) {
```

y definimos nuestra función handleBoton (que es la que borrará el elemento del array)

```
function handleBoton(id){
  handleDelete(contacts.filter((item) => item.id !== id));
}
```

OJO Al generarse a la vez los tres primeros contactos, su id va a ser el mismo, por lo que si borramos cualquiera de ellos se borrarán todos. Para evitar ésto, deberíamos asignar el id de otra forma (quizá una función, aunque Nacho lo ha hecho hardcodeao)

Vamos ahora a refactorizar lo gordo:

Nacho:

```
function handleInputs(event) {
  console.log(event.target.name);
  setContact((contact) => ({
    ...contact,
    [event.target.name]: event.target.value,
  }));
}
```

Lo que hace es a cada input del formulario le da la propiedad “name”, y al input del name lo llama name, al input del surname lo llama surname, etc. y en cada uno de los inputs, cambiamos el handle de handleInputName (por ejemplo) a handleInputs.

Formulario.js:

```
<div className="form-group">
  <label htmlFor="inputName">Name</label>
  <input name = "name" type="text" value={contact.name} onChange={handleInputs} clas
</div>
<div className="form-group">
  <label htmlFor="inputSurname">Surname</label>
  <input name="surname" type="text" value={contact.surname} onChange={handleInputs} c
</div>
```

APUNTES REACT 5

24/05/2022

Nuevo Hook: useEffect

Genera un efecto secundario después de la carga del componente.

Para explicarlo vamos a empezar con un ejemplo simple. Hacemos un contador, y queremos que si el contador es divisible entre 3 nos salte una alerta.

```
1 import { useState } from "react";
2
3 export default function Counter() {
4     const [count, setCount] = useState(0);
5
6     function incrementAccount() {
7         setCount(count + 1);
8     }
9
10    if (count % 3 === 0 && count !== 0) {
11        alert("Número multiplo de 3!!");
12    }
13    return (
14        <div>
15            <p>Valor del contador: {count}</p>
16            <button onClick={incrementAccount}>Incrementar</button>
17        </div>
18    );
19 }
```

El problema de ésto, es que nos salta la alerta antes de hacer el return, por lo que la alerta saltará pero en pantalla sólo dirá “Valor del contador: 2”, y una vez le demos a Aceptar entonces nos incrementará en pantalla el valor del contador a 3.

Este es un ejemplo muy sencillo, pero imaginemos que en lugar de un contador es una llamada a una API. Si fuese así la pantalla se quedaría estática o en blanco hasta recibir los datos de la API. Para ello usamos useEffect.

useEffect recibe dos argumentos, aunque por ahora nos centraremos en el primero. El primer argumento es una función.

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

export default function Counter() {
  const [count, setCount] = useState(0);

  function incrementAccount() {
    setCount(count + 1);
  }

  useEffect(function () {
    if (count % 3 === 0 && count !== 0) {
      alert("Número multiplo de 3!!");
    }
  });

  return (
    <div>
      <p>Valor del contador: {count}</p>
      <button onClick={incrementAccount}>Incrementar</button>
    </div>
  );
}
```

El useEffect se ejecuta **después de haber actualizado el DOM**. (en teoría)

Ahora vamos a usarlo con una llamada a una API de ejemplo (reqres.in):

****OJO**** a useEffect **no** se le puede pasar una función asíncrona. Para ello se crea la función asíncrona dentro y se la llama después.

Estos son los datos que nos devuelve nuestra API:

```
{
  "page": 2,
  "per_page": 6,
  "total": 12,
  "total_pages": 2,
  "data": [
    {
      "id": 7,
      "email": "michael.lawson@reqres.in",
      "first_name": "Michael",
      "last_name": "Lawson",
      "avatar": "https://reqres.in/img/faces/7.jpg"
    },
    {
      "id": 8,
      "email": "lindsay.ferguson@reqres.in",
      "first_name": "Lindsay",
      "last_name": "Ferguson",
      "avatar": "https://reqres.in/img/faces/8.jpg"
    },
    {
      "id": 9,
      "email": "tobias.funke@reqres.in",
      "first_name": "Tobias",
      "last_name": "Funke",
      "avatar": "https://reqres.in/img/faces/9.jpg"
    }
  ]
}
```

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

export default function Users() {
  const [users, setUsers] = useState(null);

  useEffect(function () {
    async function fetchApi() {
      const response = await fetch("https://reqres.in/api/users");
      const json = await response.json();
      setUsers(json.data);
      console.log(json.data);
    }
    fetchApi();
  });
}

if (!users) {
  return <div>Todavia no hay usuarios</div>;
}

return (
  <ul>
    {users.map((user) => (
      <li key={user.id}>{user.first_name}</li>
    )));
  </ul>
);
```

Ésto PETA. ¿Por qué? Porque nuestro useEffect llama a la API, ésta nos devuelve un dato, la web se renderiza y vuelve a llamarse al useEffect, y vuelve a llamar a la API, y vuelve a recibir un dato, y vuelve a renderizar, y así hasta el infinito (app se desborda).

Para ello, al useEffect le pasamos un **array de dependencia**. Este array le dice al useEffect la cantidad de veces que tiene que ejecutarse. En el primer render se ejecuta seguro, pero si le pasamos un array vacío le estamos diciendo que se ejecute una sola vez.

Así sí: (además hemos modificado el final con el operador ternario)

```
1 import { useState, useEffect } from "react";
2
3 export default function Users() {
4   const [users, setUsers] = useState(null);
5
6   useEffect(function () {
7     async function fetchApi() {
8       const response = await fetch("https://reqres.in/api/users");
9       const json = await response.json();
10      setUsers(json.data);
11      console.log(json.data);
12    }
13    fetchApi();
14  }, []);
15
16
17
18   return (
19     <ul>
20       {users ? users.map((user) => (
21         <li key={user.id}>{user.first_name}</li>
22       )) : <div>Todavia no hay usuarios</div>}
23     </ul>
24   );
25 }
26
27 }
```

¿Cómo llamamos a una API?

Antes de **function** le tenemos que decir que es asíncrona (**async**) a la función. Y, como no sabemos cuándo nos va a responder la función, cuando le metemos a la variable **response** lo que nos venga de la api (**fetch** → para llamar a la API) tenemos que decirle que espere (**await**) puesto que es una promesa. Y como **response** también es una promesa, cuando la transformamos en **json** y la metemos en la variable que hemos llamado **json**, también es una promesa.

Volviendo al **useEffect**

Tenemos efectos que no necesitan ser limpiados, porque hacen algo que no se mantiene en el tiempo ni se mantiene en ningún espacio, pero también hay efectos que necesitan de limpieza después de usarlos.

Veamos un ejemplo con un componente llamado ‘Scroll’:

(*** tenemos el objeto ‘window’ que como propiedades tiene una llamada scrollY que por defecto será cero)

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

export default function Scroll() {
  const [currentScroll, setCurrentScroll] = useState(window.scrollY);

  useEffect(function () {
    function onScroll() {
      console.log("onScroll");
      setCurrentScroll(window.scrollY);
    }

    window.addEventListener("scroll", onScroll);
  });

  return (
    <div
      style={{
        color: "black",
        position: "fixed",
        background: "red",
        width: "100%",
        bottom: 0
      }}
    >
      Scroll: {currentScroll}
    </div>
  );
}
```

Aquí estamos creando un estado (currentScroll) que almacena simplemente en qué punto de scroll estamos. Y creamos un efecto con la función “onScroll”, que nos haga un console log cada vez que lo llamamos y llame al setter del estado (setCurrentScroll) con el nuevo valor. ¿Qué ocurre? Que como hemos añadido un eventListener que escuche el evento “scroll” y cuando ocurra el evento nos llame a nuestra función “onScroll” para modificar el estado, en cada iteración del setter hay un render. Y nuestro useEffect se lanza después de cada render. Ésto hace que si vemos la consola, aunque el scroll sea de 32, las veces que se ha hecho el console log han sido 128, porque en cada iteración se vuelve a hacer un listen del evento sobre las anteriores existentes. Esto es un fallo grave en el rendimiento de la app. Para ello creamos una función de limpieza (que llamaremos cleanUp pero puede ser anónima).

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

export default function Scroll() {
  const [currentScroll, setCurrentScroll] = useState(window.scrollY);

  useEffect(function () {
    function onScroll() {
      console.log("onScroll");
      setCurrentScroll(window.scrollY);
    }

    window.addEventListener("scroll", onScroll);

    return function cleanUp(){
      window.removeEventListener("scroll",onScroll)
    }
  });
  return (
    <div
      style={{
        color: "black",
        position: "fixed",
        background: "red",
        width: "100%",
        bottom: 0
      }}
    >
      Scroll: {currentScroll}
    </div>
  );
}
```

Esta función cleanUp se la guarda React en la manga. Por lo que al cargar la página al principio, se hace un primer render, nuestro efecto se realiza y el estado del currentScroll está a cero. En cuanto se hace scroll se hace un nuevo render, pero antes de hacer el efecto, React saca la función cleanUp de la manga y elimina el eventListener que teníamos creado. Y ahora se hace el efecto y se crea un nuevo listener, y vuelve a guardarse el nuevo cleanUp en la manga. Así evitamos problemas de rendimiento.

Imagen de pantalla:



Código de Nacho comentado:

<https://codesandbox.io/s/useeffect-with-comments-lch8r?file=/src/Counter.js>

Ejercicio 4:

Ejercicio 4 – Lista To-do

1. Crear un nuevo proyecto React y limpiar lo que no haga falta.
2. Nuestra App tendrá dos componentes principales: una lista de to-dos y un input para añadir uno nuevo.
3. Al nivel de la App, definiremos como estado un array de to-dos.

Los to-do serán objetos con la siguiente información:

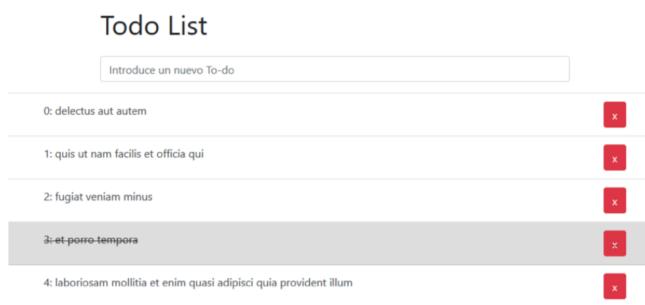
- userId, title, completed

Hacer *fetch* (utilizando *useEffect*) a <https://jsonplaceholder.typicode.com/todos> y actualizar el estado con los primeros 20 elementos.

4. En el componente de la lista, renderizar una *ul* que tenga un *li* por cada to-do. En cada elemento estará en primer lugar el índice del to-do y luego el título. Si se hace click en un to-do, se hará *toggle* sobre su propiedad *completed*. Este cambio automáticamente deberá tachar el título y cambiará el estilo del fondo y del texto.

Ejercicio 4 – Lista To-do (cont.)

6. Dentro del componente del input, utilizaremos un estado para guardar lo que *se* introduzca por teclado. Cuando se produzca el evento *submit* añadiremos el to-do al principio del array y vaciaremos el input.
7. **Avanzado:** Cada to-do tendrá un botón al final del *list item* que lo eliminará.



Antes de meterle mano al ejercicio, vamos a hacernos un esquema de los componentes que vamos a necesitar y de los *useState* o *useEffects* que vayamos a necesitar y que necesiten enviarse entre los componentes.

<https://jsonplaceholder.typicode.com/todos>

(*toggle*: cambiar el estado de algo que tiene sólo dos estados posibles (on/off por ejemplo).

librería para generar id's random: <https://www.npmjs.com/package/uuid>

Corrección Ejercicio 4: (por Nacho)

Teniendo la opción de que el componente Todos y el componente Input sean hermanos o que 'Input' sea hijo de 'Todos', lo que más sentido tiene es que sean hermanos. Puesto que vamos a trabajar con "vistas/pages" y los que modifican esa vista son los componentes. Al

hacer que ‘Todos’ sea padre de Input, estamos creando un componente que no es genérico y que no podría ser usado en otro lado. No es un componente “listado” sino que al final mi Todos (padre de input) se acaba transformando en una vista. Por ello, Nacho ha elegido que ‘Todos’ e ‘Input’ sean hermanos.

OJO: los componentes deben ser siempre lo más genéricos posible, para que puedan ser utilizados por cualquier vista/page que lo necesite.

Comenzamos:

Primero se crea el componente TodoList (con su carpeta y sus archivos index y TodoList.js) y el componente NewTodo (de igual forma).

En nuestro componente TodoList.js necesitamos recibir ‘todos’ (el array de tareas). En el archivo App usa el useEffect para hacer el fetch de la API (y quedarnos sólo con los 20 primeros):

```
JS TodoList.js U ● JS App.js 1, M X
todos > src > JS App.js > ⚡ App
1 import { useState } from "react";
2 import { useEffect } from "react";
3 import TodoList from "./components/TodoList/TodoList";
4
5 function App() {
6   const [todos, setTodos] = useState(null);      'todos' is assigned a value but never used.
7
8   useEffect(function () {
9     async function fetchTodos() {
10       const response = await fetch(
11         "https://jsonplaceholder.typicode.com/todos"
12       );
13       const todoList = await response.json();
14       setTodos(todoList.splice(0, 20));
15     }
16     fetchTodos();
17   }, []);
18
19   return (
20     <div>
21       <TodoList />
22     </div>
23   );
24 }
```

y en la línea 21 al componente le pasaremos la variable todoList (que es nuestro array de tareas).

OJO: que no se nos olvide en el archivo index primario importar Bootstrap.

Ahora hace el html del archivo **TodoList**. Crea el ul con la clase de bootstrap y dentro hace el map del array, poniendo un li (en el que si la propiedad completed está a true lo pone en oscuro), un span (en el que si completed, tacha el texto) y un botón.

TodoList.js

```
JS TodoList.js U X JS App.js M
todos > src > components > TodoList > JS TodoList.js > TodoList > todos.map() callback
1  export default function TodoList({ todos }) {
2    return (
3      <ul className="list-group container">
4        {todos.map(function (todo, index) {
5          return (
6            <li
7              key={todo.id}
8              className={`list-group-item d-flex justify-content-between ${{
9                todo.completed && "list-group-item-dark "
10               }}`}>
11              >
12                <span
13                  className={`${todo.completed && " text-decoration-line-through"}`}
14                  >
15                  {index + 1}: <strong>{todo.title}</strong>
16                </span>
17                <button className="btn btn-danger">X</button>
18              </li>
19            );
20          });
21        </ul>
22      );
23    }
24  }
```

también modifica el principio para pasarle la función setTodos también:

```
export default function TodoList({ todos, setTodos }) {
```

Ahora crea la función de borrar tareas:

```
3  function removeTodo(id){  'removeTodo' is defined but never used.
4    setTodos(todos.filter(todo => todo.id !== id))
5  }
```

y se la asigna al botón del li con el onClick:

```
21          </span>
22          <button className="btn btn-danger" onClick={()=> removeTodo(todo.id)}>X</button>
23        </li>
```

Ahora crea la función de **toggle**:

Su función toggle recibe el index (el mismo que ha generado el map) del array en el que queremos hacer toggle. Primero crea una copia del array de todos, luego en la del índice le modifica la propiedad completed y ahora devuelve el nuevo array creado:

TodoList.js

```
5   function toogleTodo(index) {    'toogleTodo' is defined but never used.
6     const newTodos = [...todos];
7     newTodos[index].completed = !newTodos[index].completed;
8     setTodos(newTodos);
9   }
10 }
```

y se lo pasa al li con la propiedad onClick:

```
16   return (
17     <ul class="list-group container">
18       {todos.map(function (todo, index) {
19         return (
20           <li
21             key={todo.id}
22             className={`list-group-item d-flex justify-content-between ${
23               todo.completed && "list-group-item-dark"
24             }`}
25             onClick={() => toogleTodo(index)}
26           >
27             -----
```

Pero de esta forma se ha cargado el botón de eliminar, puesto que el botón está dentro del 'li' y le ha dado la propiedad de toggle al onClick del 'li' entero. Esto se llama **propagación de eventos**, porque cuando le da al botón, también está creando el evento del 'li' que lo engloba.

Para evitar ésto, a la función removeTodo le pasa también un evento (e) y usa la función **stopPropagation()**.

<https://developer.mozilla.org/es/docs/Web/API/Event/stopPropagation>

```
2   function removeTodo(id, e) {
3     e.stopPropagation();
4     setTodos(todos.filter(todo => todo.id !== id));
5   }
6 }
```

```
<button
  className="btn btn-danger"
  onClick={(e) => removeTodo(todo.id, e)}
>
```

También está la opción de condicionar la función toggle para que no afecte a la etiqueta "button":

```
7   function toggleTodo(index, e) {
8     console.log(e.target.tagName);
9     if (e.target.tagName !== "BUTTON") {
10       const newTodos = [...todos];
11       newTodos[index].completed = !newTodos[index].completed;
12       setTodos(newTodos);
13     }
14 }
```

Cualquiera de las dos opciones es válida.

Ahora vamos al componente de añadir, que llama **NewTodo** (en el archivo App, llama a NewTodo justo en la línea anterior a TodoList):

```
JS NewTodo.js 1, U X JS App.js M
todos > src > components > NewTodo > JS NewTodo.js > ⚡ NewTodo
1 import { useState } from "react";
2
3 export default function NewTodo() {
4   const [newTodo, setNewTodo] = useState("");    'setNewTodo' is assigned a value but never used.
5
6   return (
7     <form className="form-group container">
8       <h1 className="my-4">Todo List</h1>
9       <input
10         className="form-control mb-3"
11         type="text"
12         value={newTodo}
13         placeholder="Introducir un nuevo To-do"
14         onChange={handleNewTodo}
15       />
16     </form>
17   );
18 }
19
```

Ahora crea un handleNewTodo:

```
JS NewTodo.js U X JS App.js M

todos > src > components > NewTodo > JS NewTodo.js > ⚡ NewTodo
1 import { useState } from "react";
2
3 export default function NewTodo() {
4     const [newTodo, setNewTodo] = useState("");
5
6     function handleNewTodo(e) {
7         setNewTodo(e.target.value);
8     }
9
10    return [
11        <form className="form-group container">
12            <h1 className="my-4">Todo List</h1>
13            <input className="form-control mb-3"
14                type="text"
15                value={newTodo}
16                placeholder="Introducir un nuevo To-do"
17                onChange={handleNewTodo}
18            />
19        </form>
20    ];
21}
22
23
```

con esto creado, ahora falta la comunicación con el padre para enviarle el newTodo que se crea en el formulario. Para ello necesitan comunicación por el setTodo:

```
3 export default function NewTodo({setTodos}) {
4     const [newTodo, setNewTodo] = useState("");
5 }
```

y crea un handleSubmit del formulario, para que al hacer submit se envíe el newTodo al padre (App):

```
10 | function handleSubmit(e) {    'handleSubmit' is defined but never used.
11 |   e.preventDefault();
12 |   setTodos((todos) => [...todos, {
13 |     id: 234,
14 |     title: newTodo,
15 |     completed: false,
16 |     userId: 1,
17 |   }]);
18 | }
```

aquí está creando el objeto en la misma función handleSubmit y metiéndolo al array de todos. Y se la pasamos al formulario:

```
24 | return (
25 |   <form onSubmit={handleSubmit} className="form-group container">
26 |     <h1 className="my-4">Todo List</h1>
27 |     <input
28 |       className="form-control mb-3"
29 |       type="text"
30 |       value={newTodo}
31 |       placeholder="Introducir un nuevo To-do"
32 |       onChange={handleNewTodo}
33 |     />
34 |   </form>
35 | );
```

EJERCICIO RICK & MORTY

Traer los personajes de la API de Rick and Morty y pintarlos en tarjetas

(buscar cómo hacerle **paginación**, para que te salga un número de tarjetas por página y poder pasar a la siguiente página y botón de **eliminar**)

(hacerle también un “mensaje de loading” de mientras te trae los datos de la API) → Ésto es lo que hace él cuando crea el array a “null” y pone un mensaje de cuando: !todos

<https://rickandmortyapi.com/api/character>

Apuntes React Día 30/05/2022

Nacho va a corregir el ejercicio con Material, en lugar de Bootstrap:

<https://mui.com/material-ui/getting-started/installation/>

De la misma forma que hacemos con Bootstrap, ha buscado un componente CharacterCard en la documentación de material UI, copiado y modificado el html y creado una tarjeta.

Ha creado un componente CharacterList que sea un map de un array de items que luego llama al componente CharacterCard y pinta una tarjeta por cada ítem.

Ahora, en App.js, llamamos a la API y nos traemos los datos de los personajes.

Apuntes React Día 31/05/2022

**** Creando un Custom Hook ****

App.js:

```
1 import "./styles.css";
2 import { useEffect, useState } from "react";
3 import { useFetch } from "./useFetch";
4
5 export default function App() {
6   const [characters, setCharacters] = useState(null);
7
8   useEffect(() => {
9     useFetch("https://rickandmortyapi.com/api/character", setCharacters);
10
11   return (
12     <div className="App">
13       <h1>Hello CustomHook</h1>
14       {characters &&
15         characters.results.map((character) => (
16           <p key={character.id}>{character.name}</p>
17         )));
18     </div>
19   );
20 }
```

Nuestro hook personalizado se llama 'useFetch' y lo hemos creado a nuestra idea de traer los datos de la API de Rick and Morty.

useFetch.js

```
JS App.js      JS useFetch.js x
1 import { useEffect } from "react";
2
3 export function useFetch(URL, setState) {
4   useEffect(
5     function () {
6       async function fecthData() {
7         const response = await fetch(URL);
8         const data = await response.json();
9         setState(data);
10    }
11    fecthData();
12  },
13  [URL, setState]
14 );
15 }
16
```

Ésto debería ir dentro de una carpeta que llamemos ‘hooks’ y ahí guardaremos todos los hooks personalizados que queramos. Como si fuese nuestra propia librería.

** Nuevo Hook: useContext **

<https://es.reactjs.org/docs/hooks-reference.html#usecontext>

<https://codesandbox.io/s/silly-zeh-l5g9rg?file=/src/Bar.js>

Este hook es como un useState, lo que pasa es que el estado es el GLOBAL de la aplicación. A ese estado pueden acceder todos los componentes (ya sean hijos, nietos o bisnietos o lo que sea de App).

Para crear nuestros contextos, al igual que los componentes, haremos una carpeta que se llama ‘contexts’.

Importamos desde react el createContext, el useState y el useContext. Y creamos el contexto:



```
JS App.js      JS UiModeContext.js •  JS Zeta.js      JS Bar.js      # Bar.css      JS □ ▢ ▤ ...  
1 import { createContext, useState, useContext } from "react";  
2  
3 const UiModeContext = createContext({  
4     uiMode: "",  
5     toggleUiMode: () => {}  
6 });  
7
```

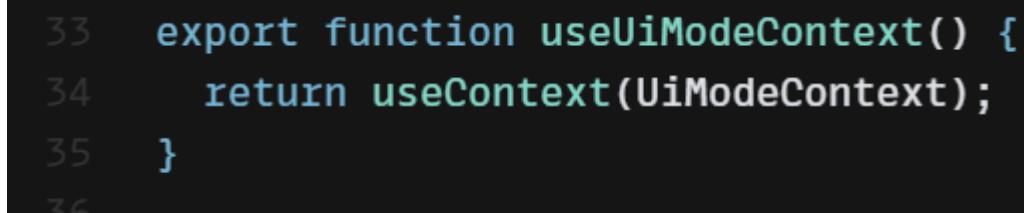
Ahora creamos un componente que nos devuelve el contexto en su estado inicial:



```
10 export function UiModeContextProvider({ children }) {  
11     const [uiMode, setUiMode] = useState("light");  
12  
13     function toggleUiMode() {  
14         setUiMode((oldUiMode) => {  
15             if (oldUiMode === "light") {  
16                 return "dark";  
17             } else {  
18                 return "light";  
19             }  
20         });  
21     }  
22  
23     const value = {  
24         uiMode,  
25         toggleUiMode  
26     };  
27  
28     return (  
29         <UiModeContext.Provider value={value}>{children}</UiModeContext.Provider>  
30     );  
31 }
```

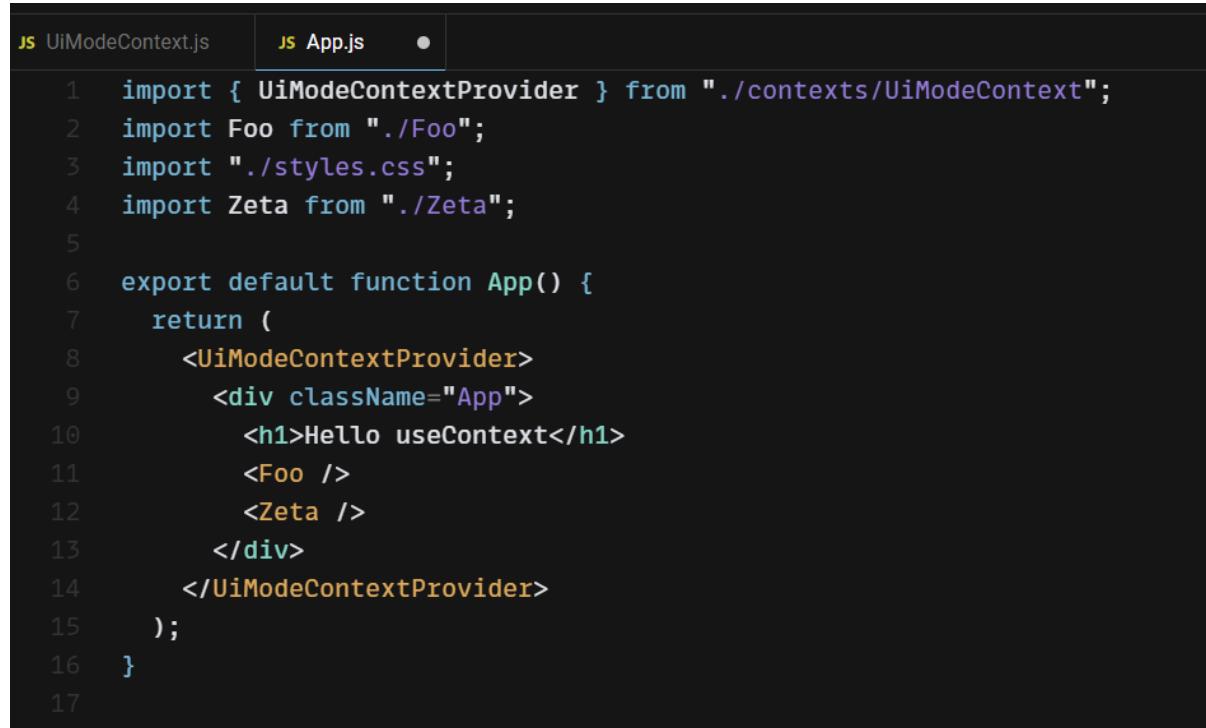
Y retorna el proveedor del contexto. En el return si nos fijamos bien, estamos hablando de “children” que ahora veremos que ésto es el App al completo.

También crea un hook personalizado para exportarlo y poder usarlo del tirón:



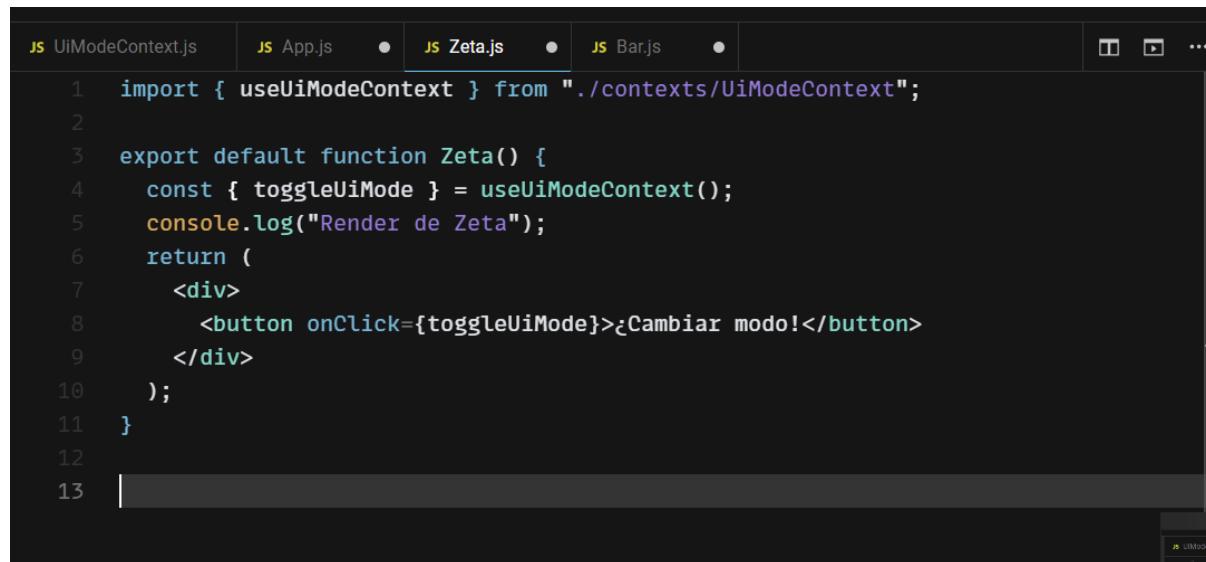
```
33 export function useUiModeContext() {  
34     return useContext(UiModeContext);  
35 }  
36
```

Aquí vemos la estructura de App:



```
js UiModeContext.js    js App.js    ●
1 import { UiModeContextProvider } from "./contexts/UiModeContext";
2 import Foo from "./Foo";
3 import "./styles.css";
4 import Zeta from "./Zeta";
5
6 export default function App() {
7   return (
8     <UiModeContextProvider>
9       <div className="App">
10         <h1>Hello useContext</h1>
11         <Foo />
12         <Zeta />
13       </div>
14     </UiModeContextProvider>
15   );
16 }
17
```

Aquí, en Zeta, estamos pintando el botón que modificará el contexto (modo light o dark):



```
js UiModeContext.js    js App.js    ●    js Zeta.js    ●    js Bar.js    ●    ...    ⏹    ⏹    ...
1 import { useUiModeContext } from "./contexts/UiModeContext";
2
3 export default function Zeta() {
4   const { toggleUiMode } = useUiModeContext();
5   console.log("Render de Zeta");
6   return (
7     <div>
8       <button onClick={toggleUiMode}>Cambiar modo!</button>
9     </div>
10   );
11 }
12
13 |
```

Y en Bar, que es quien pinta las letras, le damos un className que sea modificado dependiendo del contexto:

silly-zeh-l5g9rg

JS	UiModeContext.js	JS	App.js	●	JS	Zeta.js	●	JS	Bar.js	●
----	------------------	----	--------	---	----	---------	---	----	--------	---

```
1 import { useUiModeContext } from "./contexts/UiModeContext";
2 import "./Bar.css";
3
4 export default function Bar() {
5   console.log("Render de Bar");
6
7   const { uiMode } = useUiModeContext();
8   return <div className={uiMode}>El valor de uiMode es: {uiMode}</div>;
9 }
10
```

Ésto lo vemos mejor en el Bar.css:

```
JS UiModeContext.js   JS App.js   ●   JS Zeta.js   ●   JS Bar.js   ●   # Bar.css   X
1  .dark {
2    color: red;
3  }
4
5  .light {
6    color: green;
7  }
8
```

Apuntes React Día 01/06/2022

Ejemplo de hacer un login con el useContext:

<https://codesandbox.io/s/frosty-pond-bdvuk2?file=/src/components/Login.js>

**** Página de Custom Hooks (comunidad) ****

<https://usehooks-ts.com/react-hook/use-debounce>

**** Creando un nuevo Hook Personalizado (useDebounce) ****

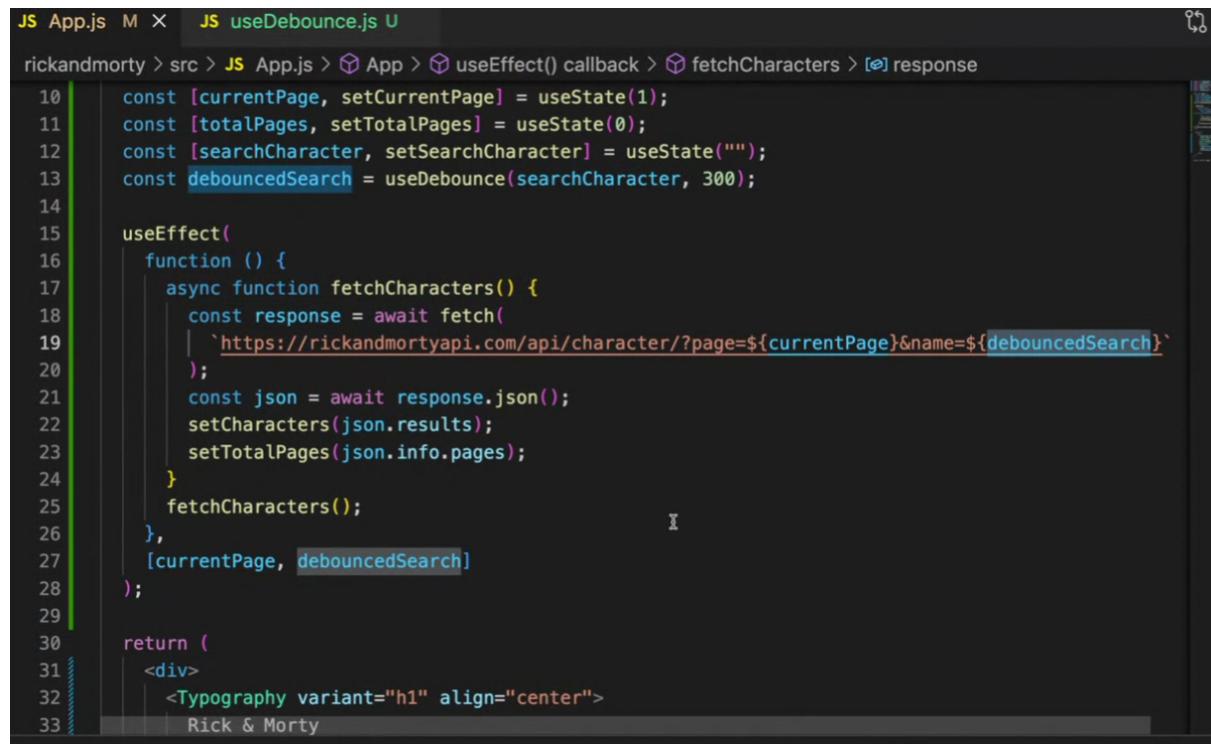
Nacho crea aquí un Hook que actualice el valor de una variable al pasar un tiempo. Esto lo utilizamos para evitar en el buscador que cada vez que escribe una letra haga una llamada a la API, de esta forma ejercerá la llamada a la API únicamente cuando dejemos de teclear.

A este Hook lo llama useDebounce, y lo único que hace es eso, almacenar los cambios de la variable y generar el cambio acumulado al final del delay.

```
import { useEffect, useState } from "react";
export default function useDebounce(value = "", delay) {
  const [debouncedValue, setDebouncedValue] = useState(value);
  useEffect(
    function () {
      const handler = setTimeout(() => {
        setDebouncedValue(value);
      }, delay);
      return () => {
        clearTimeout(handler);
      };
    },
    [value, delay]
  );
  return debouncedValue;
}
```

Este useEffect necesita de una función de limpieza del timeOut que hemos declarado. Para ello hacemos el return con la función clearTimeout y le pasamos el valor del handler.

Y lo usamos así:



```
JS App.js M X JS useDebounce.js U
rickandmorty > src > JS App.js > ⚡ App > ⚡ useEffect() callback > ⚡ fetchCharacters > [⌚] response
10 |   const [currentPage, setCurrentPage] = useState(1);
11 |   const [totalPages, setTotalPages] = useState(0);
12 |   const [searchCharacter, setSearchCharacter] = useState("");
13 |   const debouncedSearch = useDebounce(searchCharacter, 300);
14 |
15 |   useEffect(
16 |     function () {
17 |       async function fetchCharacters() {
18 |         const response = await fetch(
19 |           `https://rickandmortyapi.com/api/character/?page=${currentPage}&name=${debouncedSearch}`
20 |         );
21 |         const json = await response.json();
22 |         setCharacters(json.results);
23 |         setTotalPages(json.info.pages);
24 |       }
25 |       fetchCharacters();
26 |     },
27 |     [currentPage, debouncedSearch]
28 |   );
29 |
30 |   return (
31 |     <div>
32 |       <Typography variant="h1" align="center">
33 |         Rick & Morty
```

Apuntes React Día 06/06/2022

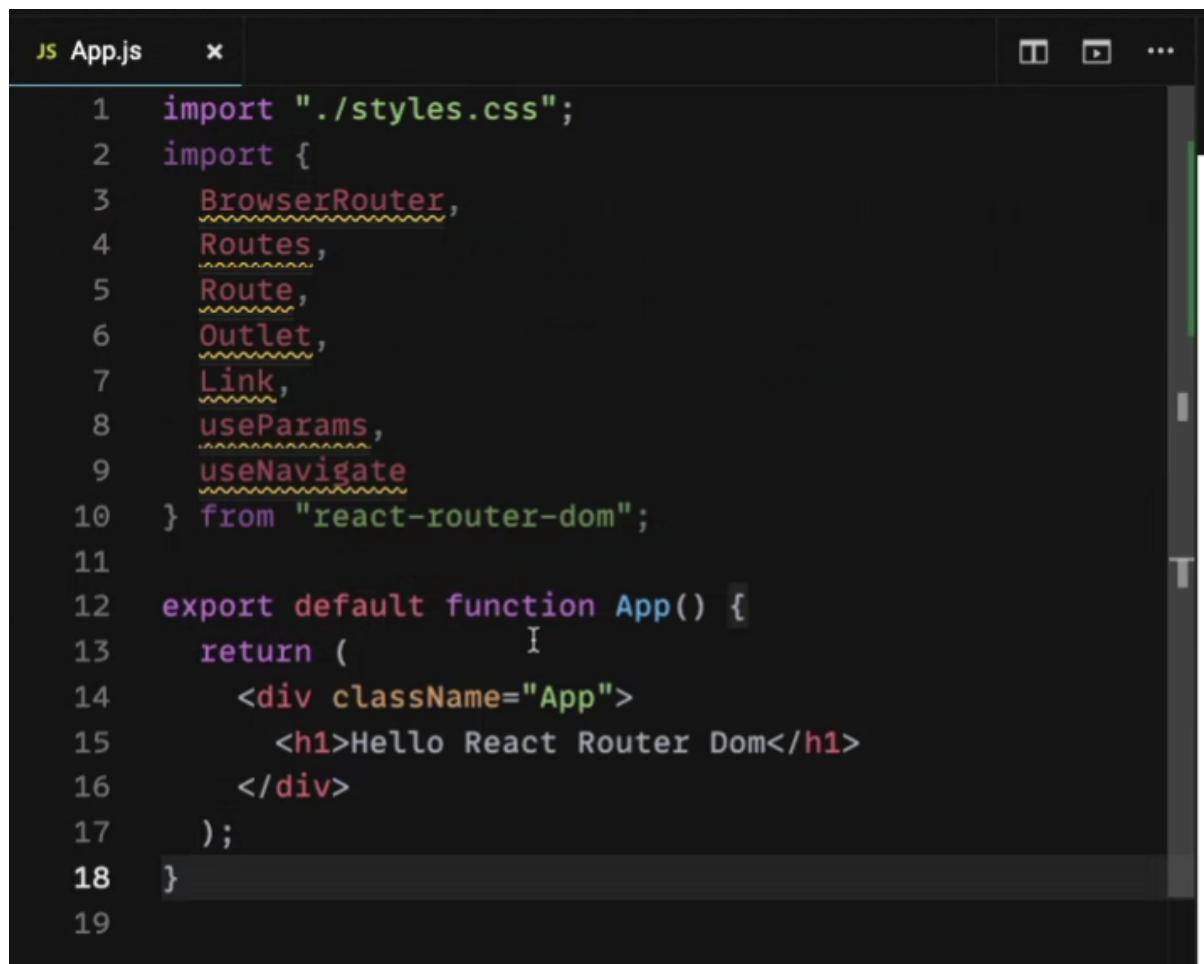
** Rutas **

Documentación:

<https://reactrouter.com/docs/en/v6/getting-started/tutorial>

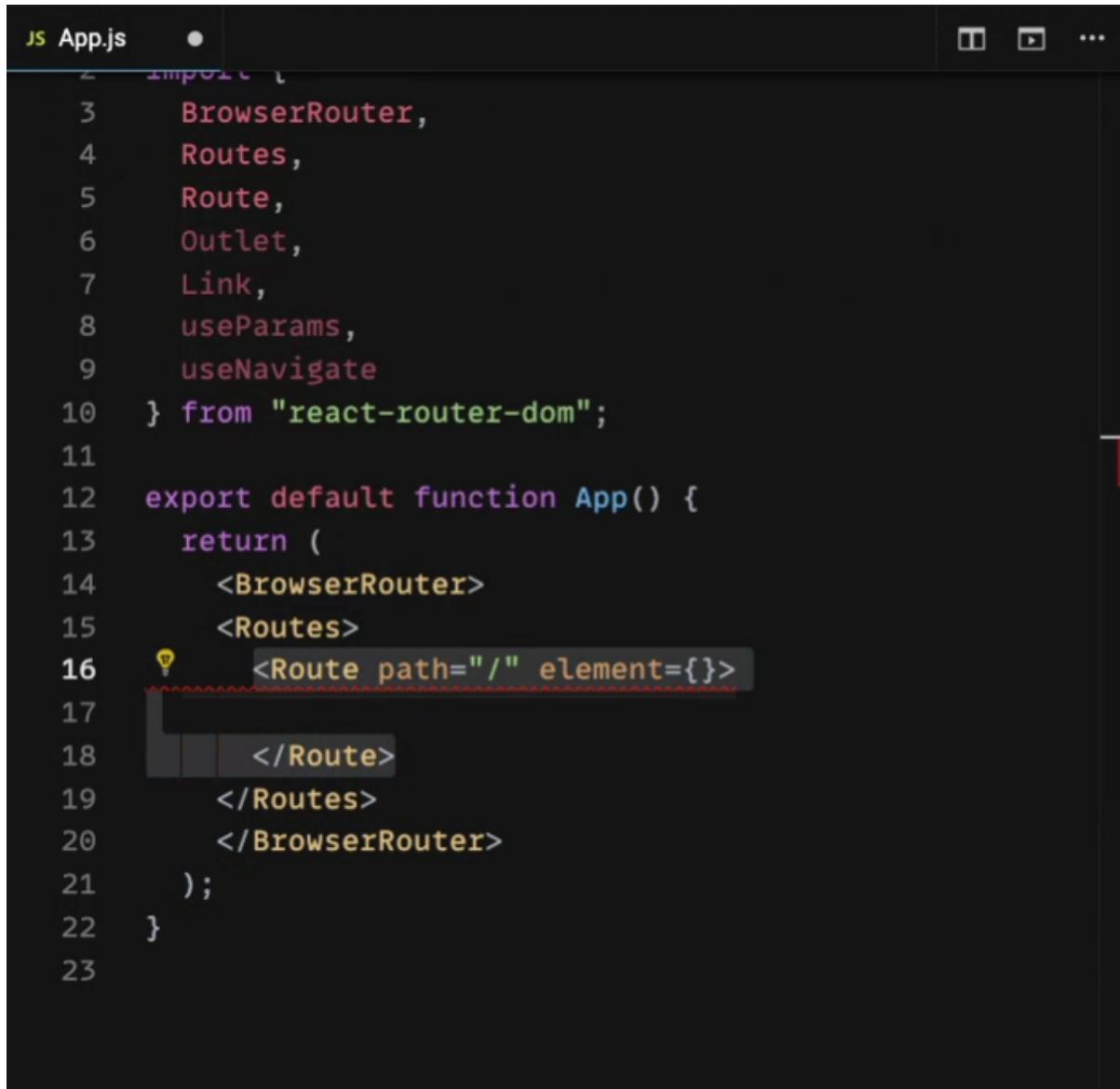
Para poder utilizar la librería de enrutación hay que instalarla en nuestro proyecto de React con “npm i react-router-dom”

Al principio de nuestro proyecto tendremos que importar varias cosas desde la librería React Router Dom:



```
JS App.js x
1 import "./styles.css";
2 import {
3   BrowserRouter,
4   Routes,
5   Route,
6   Outlet,
7   Link,
8   useParams,
9   useNavigate
10 } from "react-router-dom";
11
12 export default function App() {
13   return (
14     <div className="App">
15       <h1>Hello React Router Dom</h1>
16     </div>
17   );
18 }
19
```

Nuestro App se quedará tal que así:



```
JS App.js
  1 import {
  2   BrowserRouter,
  3   Routes,
  4   Route,
  5   Outlet,
  6   Link,
  7   useParams,
  8   useNavigate
  9 } from "react-router-dom";
10
11
12 export default function App() {
13   return (
14     <BrowserRouter>
15       <Routes>
16         <Route path="/" element={}>
17           </Route>
18       </Routes>
19     </BrowserRouter>
20   );
21 }
22
23
```

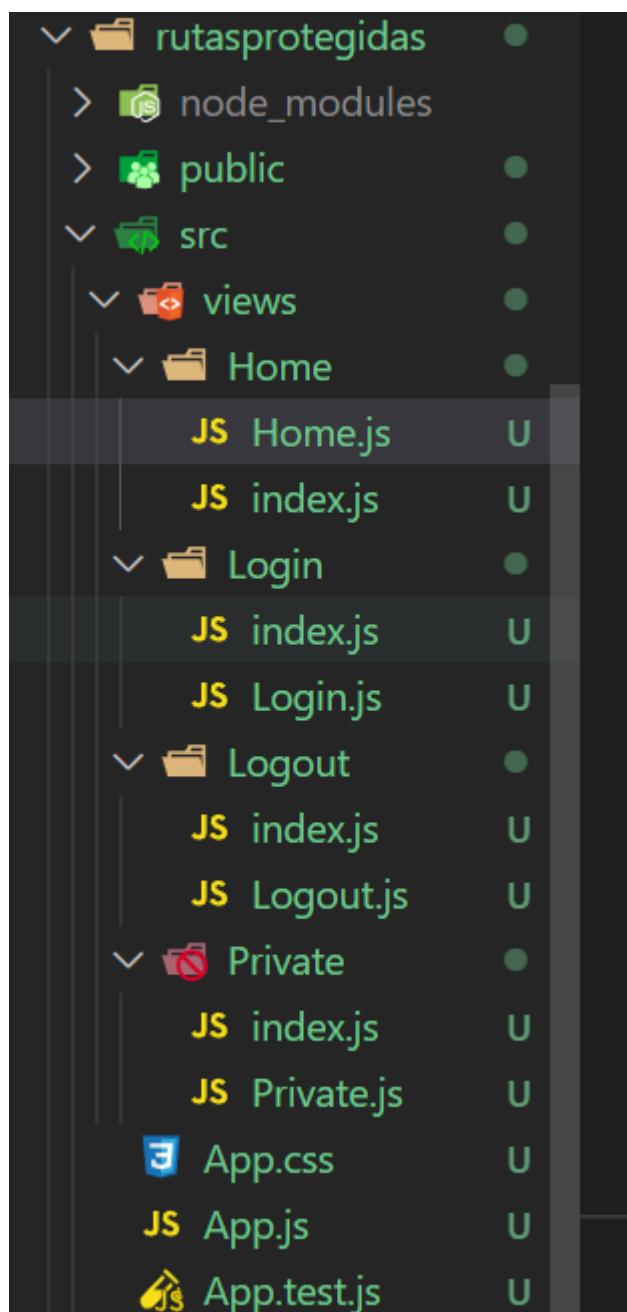
Donde BrowserRouter es el padre supremo de todo lo demás, Routes es el padre de todas las rutas y Route

Apuntes Día 08/06/2022

** Rutas con React Router Dom **

Vamos a crear un nuevo proyecto. En este nuevo proyecto tendremos dos vistas públicas (home, login) y dos vistas privadas a las que solo tenga acceso usuario registrado (logout, private).

La arquitectura de carpetas sería así:



Ahora vamos a ir a App.js y configuraremos nuestras rutas. Habrá que haber instalado la librería de react router dom. Y nos quedaría tal que así:

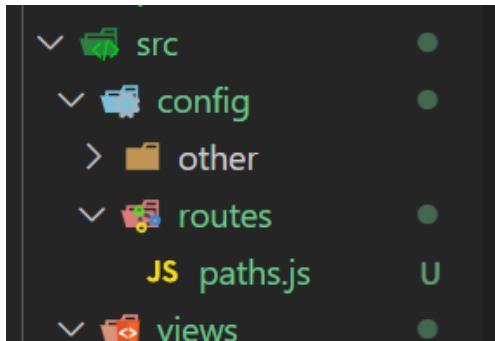
```
JS App.js U X JS Home.js U JS Login.js U JS Logout.js U JS Private.js U
rutasprotegidas > src > JS App.js > ...
1 import {BrowserRouter, Routes, Route} from "react-router-dom";
2 import Home from "./views/Home";
3 import Login from "./views/Login";
4 import Logout from "./views/Logout";
5 import Private from "./views/Private";
6
7 function App() {
8     return (
9         <BrowserRouter>
10            <Routes>
11                <Route path="/" element={<Home/>} />
12                <Route path="/login" element={<Login/>} />
13                <Route path="/private" element={<Private/>} />
14                <Route path="/private/logout" element={<Logout/>} />
15            </Routes>
16        </BrowserRouter>
17    );
18 }
19
20 export default App;
```

Para quitarnos los strings puestos a fuego en el código vamos a crear una carpeta config, en ella una carpeta 'routes' y en ella un archivo paths.js

Paths.js

```
JS App.js U JS paths.js U X JS Home.js U JS Login.j
rutasprotegidas > src > config > routes > JS paths.js > [?] LOGOUT
1 export const HOME = "/"
2 export const LOGIN = "/login"
3 export const PRIVATE = "/private"
4 export const LOGOUT = "/private/logout"
```

Carpetas:



Y en App importamos las constantes desde paths y las modificamos en las rutas:

```
JS App.js U X JS paths.js U JS Home.js U JS Login.js U JS Logout.js U JS Private.js U
rutas protegidas > src > JS App.js > ...
1  import {BrowserRouter, Routes, Route} from "react-router-dom";
2  import Home from "./views/Home";
3  import Login from "./views/Login";
4  import Logout from "./views/Logout";
5  import Private from "./views/Private";
6  import {HOME, LOGIN, PRIVATE, LOGOUT} from "./config/routes/paths";
7
8  function App() {
9    return (
10      <BrowserRouter>
11        <Routes>
12          <Route path = { HOME } element = { <Home/> } />
13          <Route path = { LOGIN } element = { <Login/> } />
14          <Route path = { PRIVATE } element = { <Private/> } />
15          <Route path = { LOGOUT } element = { <Logout/> } />
16        </Routes>
17      </BrowserRouter>
18    );
19  }
20
21  export default App;
22
```

Ahora si vamos a <http://localhost:3000/> nos mostrará la página de Home, y si vamos a <http://localhost:3000/login> nos mostrará la página de Login y así con los demás.

Ahora para la autentificación de usuario vamos a crear un contexto AuthContext:

```
JS App.js U JS AuthContext.js U X JS paths.js U JS Home.js U JS Lo
rutasprotegidas > src > contexts > JS AuthContext.js > ...
1 import { useContext, createContext, useState } from "react";
2
3 const AuthContext = createContext({
4
5   login:() => {},
6   logout:() => {},
7   isAuthenticated: false
8
9 });
10
11 export default function AuthContextProvider({ children }) {
12
13 }
```

Creamos una constante global (en MAYÚSCULAS) que usaremos como key para almacenar en el localStorage, y un estado de si el usuario está autenticado o no.

```
10
11 const MY_AUTH_APP = "MY_AUTH_APP";
12
13 export default function AuthContextProvider({ children }) {
14
15   const [isAuthenticated, setIsAuthenticated] = useState(
16     window.localStorage.getItem(MY_AUTH_APP) ?? false
17   )
}
```

Usamos el operador '??' que significa que si lo primero es nulo, entonces toma lo segundo.

Ahora vamos a usar un hook nuevo que no hemos usado aún, el **useCallback**. Este nuevo hook lo que hace es memorizar la función para que cada vez que haya un render no tenga que volver a crearla.

Para el **value** de nuestro provider vamos a usar otro hook nuevo llamado **useMemo**, este hook es como el anterior pero que en lugar de memorizar una función, este nuevo hook va a memorizar datos/resultados.

Finalmente retornamos el provider del contexto, y un hook que creamos para ejecutar sus valores. (Capturas a continuación)

```
JS App.js U JS AuthContext.js U X JS paths.js U JS Home.js U JS Login.js U JS Logout.js U JS Private.js U
rutas protegidas > src > contexts > JS AuthContext.js > useAuthContext
12 export default function AuthContextProvider({ children }){
13   const [isAuthenticated, setIsAuthenticated] = useState(window.localStorage.getItem(MY_AUTH_APP) ?? false);
14
15   const login = useCallback(function(){
16     window.localStorage.setItem(MY_AUTH_APP, true);
17     // Aquí en lugar de true, podríamos guardar el token que nos viene desde Node
18     setIsAuthenticated(true);
19   }, []);
20
21   const logout = useCallback(function(){
22     window.localStorage.removeItem(MY_AUTH_APP);
23     setIsAuthenticated(false);
24   }, []);
25
26   const value = useMemo(
27     () => ({
28       login,
29       logout,
30       isAuthenticated
31     }),
32     [login, logout, isAuthenticated]
33   );
34
35   return <AuthContext.Provider value={value}> {children} </AuthContext.Provider>
36 }
37
38 }
39
40 export function useAuthContext(){
41   return useContext(AuthContext);
42 }
43
```

Con nuestro contexto creado, lo importamos en App.js.

```
6 import { HOME, LOGIN, PRIVATE, LOGOUT } from "./config/routes/paths";
7 import AuthContextProvider from "./contexts/AuthContext";
8
9 function App() {
10   return (
11     <AuthContextProvider>
12       <BrowserRouter>
13         <Routes>
14           <Route path={HOME} element={<Home />} />
15           <Route path={LOGIN} element={<Login />} />
16           <Route path={PRIVATE} element={<Private />} />
17           <Route path={LOGOUT} element={<Logout />} />
18         </Routes>
19       </BrowserRouter>
20     </AuthContextProvider>
21   );
22 }
```

Ahora vamos a dividir las rutas en privadas y públicas:

```
9  function App() {
10    return (
11      <AuthContextProvider>
12        <BrowserRouter>
13          <Routes>
14            <Route path={HOME} element={<PublicRoute/>}>
15              <Route path={HOME} element={<Home />} />
16              <Route path={LOGIN} element={<Login />} />
17            </Route>
18            <Route path={PRIVATE} element={<PrivateRoute/>}>
19              <Route path={PRIVATE} element={<Private />} />
20              <Route path={LOGOUT} element={<Logout />} />
21            </Route>
22          </Routes>
23        </BrowserRouter>
24      </AuthContextProvider>
25    );
26 }
```

Ahora creamos carpeta components, y dentro de ella carpeta ‘router’ y en ella dos archivos, PublicRoute.js y PrivateRoute.js para ellos vamos a usar el componente Outlet, que nos muestra los hijos de la ruta, y el componente Navigate, que nos manda a la ruta que le digamos.

PublicRoute.js

The screenshot shows a code editor with four tabs at the top: App.js, PublicRoute.js (which is the active tab), PrivateRoute.js, and AuthContext.js. The PublicRoute.js tab has a red 'X' icon. The code editor displays the following code:

```
JS App.js U JS PublicRoute.js U X JS PrivateRoute.js U JS AuthContext.js U

rutasprotegidas > src > components > Router > JS PublicRoute.js > ...

1 import { Outlet, Navigate } from "react-router-dom";
2 import {PRIVATE} from "../../config/routes/paths";
3 import { useAuthContext } from "../../contexts/AuthContext";
4
5 export default function PublicRoute(){
6
7     const {isAuthenticated} = useAuthContext();
8
9     if(isAuthenticated){
10         return <Navigate to={PRIVATE} />
11     }
12
13     return (
14         <Outlet/>
15     )
16
17 }
```

Aquí estamos diciendo que si el usuario está autenticado, entonces entra a la zona de PRIVATE, y si no, pues devolvemos los hijos con el return Outlet.

PrivateRoute.js

```
JS App.js U JS PublicRoute.js U JS PrivateRoute.js U X JS AuthContext.js U
rutasprotegidas > src > components > Router > JS PrivateRoute.js > ...
1 import { Outlet, Navigate } from "react-router-dom";
2 import {LOGIN} from "../../config/routes/paths";
3 import { useAuthContext } from "../../contexts/AuthContext";
4
5 export default function PrivateRoute(){
6
7     const {isAuthenticated} = useAuthContext();
8
9     if(!isAuthenticated){
10         return <Navigate to={LOGIN} />
11     }
12
13     return (
14         <Outlet/>
15     )
16
17 }
18
```

Y aquí hacemos similar, pero le decimos que si no está autenticado le mandamos a LOGIN.

En App.js modificaremos las rutas principales tanto de autenticado como de no y las nombramos “index”

App.js

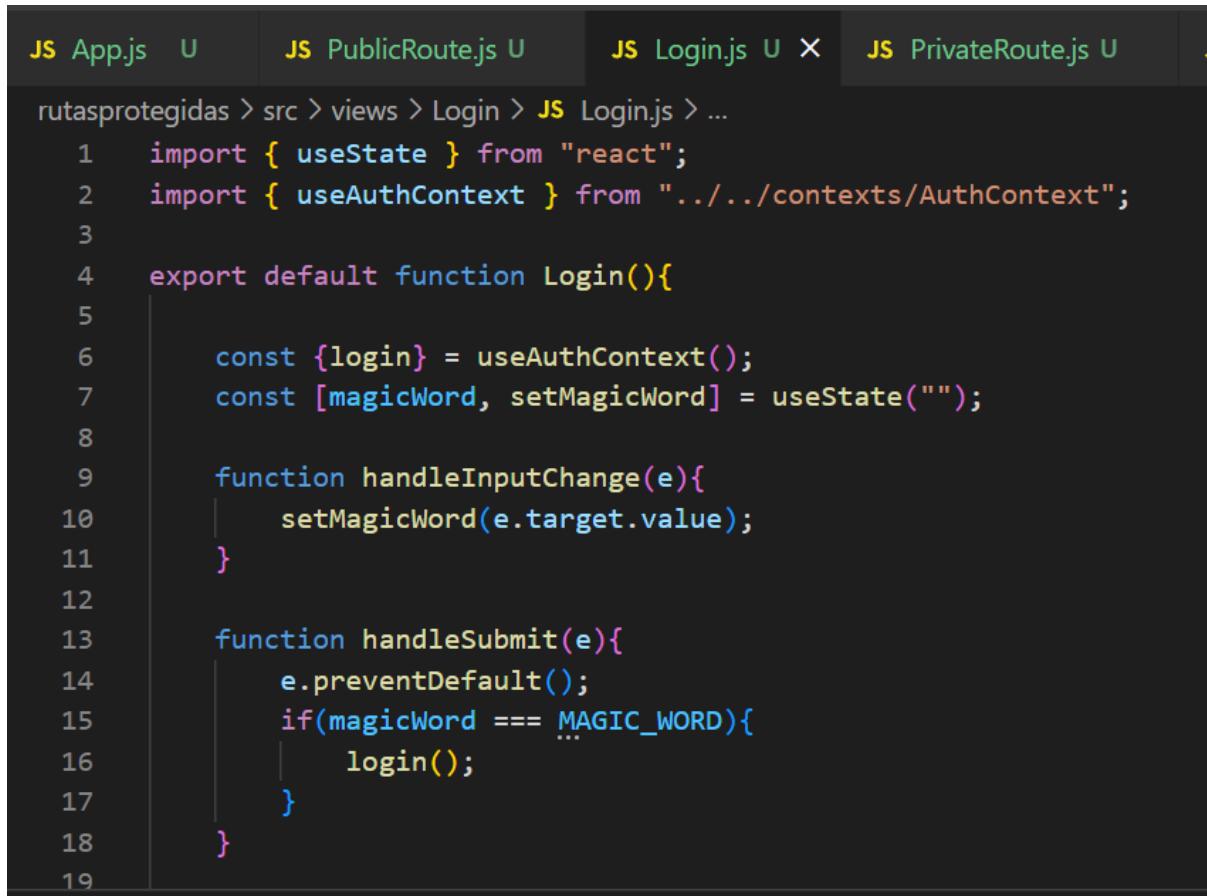
```
12  function App() {
13    return (
14      <AuthContextProvider>
15        <BrowserRouter>
16          <Routes>
17            <Route path={HOME} element={<PublicRoute/>}>
18              <Route index element={<Home />} />
19              <Route path={LOGIN} element={<Login />} />
20            </Route>
21            <Route path={PRIVATE} element={<PrivateRoute/>}>
22              <Route index element={<Private />} />
23              <Route path={LOGOUT} element={<Logout />} />
24            </Route>
25          </Routes>
26        </BrowserRouter>
27      </AuthContextProvider>
28    );
}
```

De esta forma si ahora intentamos entrar a <http://localhost:3000/private>, al no estar logeados nos envía del tirón a la página de Login.

Ahora, para ello vamos a hacer un login. Nos vamos a la vista Login. Crearemos un formulario simple, que sólo nos va a pedir una palabra mágica para logearse:

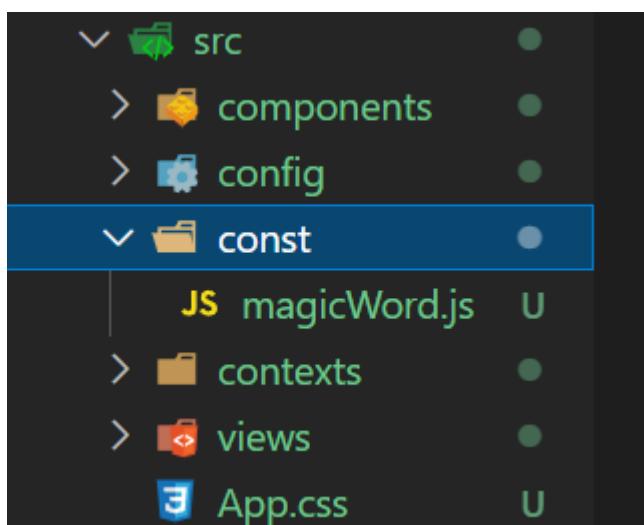
```
20  return(
21    <div>
22      <h1> Login </h1>
23      <form onSubmit={handleSubmit}>
24        <input type="text" value={magicWord} onChange={handleInputChange} />
25        <button type="submit"> Iniciar Sesión </button>
26      </form>
27    </div>
28  )
29 }
```

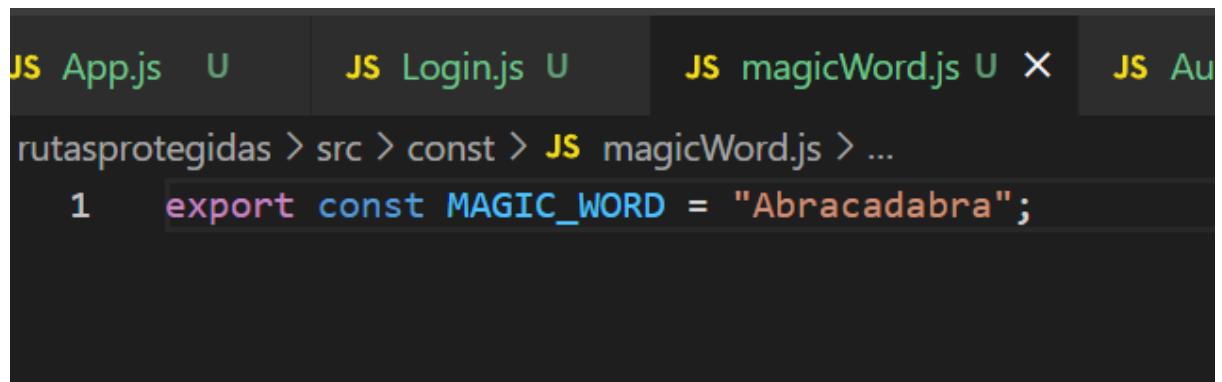
Y las funciones respectivas para el manejo del input y del submit:



```
JS App.js U JS PublicRoute.js U JS Login.js U X JS PrivateRoute.js U
rutasprotegidas > src > views > Login > JS Login.js > ...
1 import { useState } from "react";
2 import { useAuthContext } from "../../contexts/AuthContext";
3
4 export default function Login(){
5
6     const [login] = useAuthContext();
7     const [magicWord, setMagicWord] = useState("");
8
9     function handleInputChange(e){
10         setMagicWord(e.target.value);
11     }
12
13     function handleSubmit(e){
14         e.preventDefault();
15         if(magicWord === MAGIC_WORD){
16             login();
17         }
18     }
19 }
```

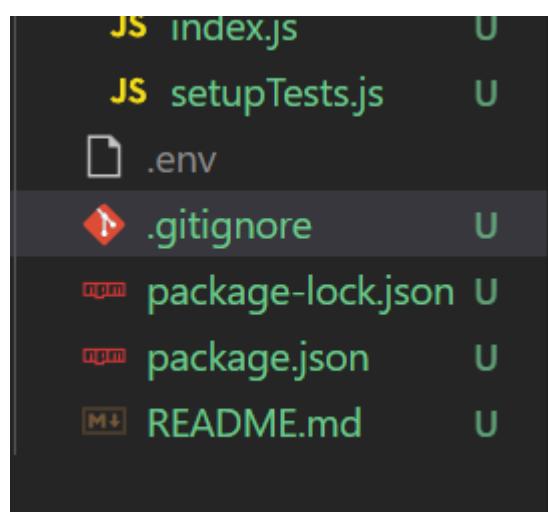
Ahora vamos a crear una carpeta dentro de 'src' que se llame 'const' donde vamos a guardar las constantes que se van a usar.



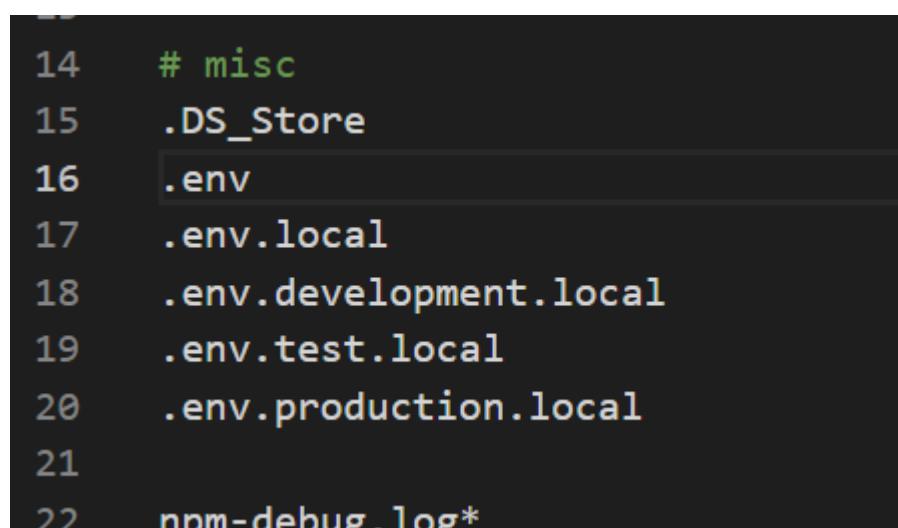


```
JS App.js U JS Login.js U JS magicWord.js U X JS Au
rutasprotegidas > src > const > JS magicWord.js > ...
1   export const MAGIC_WORD = "Abracadabra";
```

El problema es que a este archivo cualquier persona tiene acceso, entonces para ello vamos a usar las **variables de entorno**. Para ello, en la carpeta raíz (a la altura del package.json) creamos un archivo que se llame '.env' (environment) y en el archivo gitignore, en #misc lo añadimos:

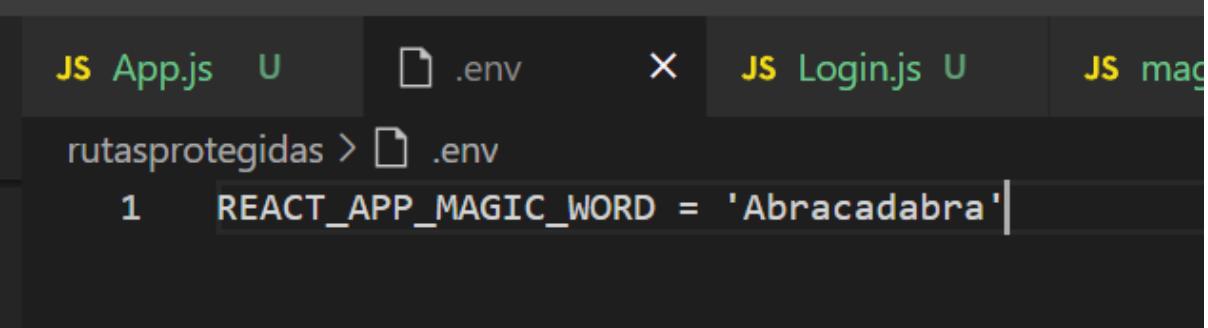


```
JS index.js U
JS setupTests.js U
└ .env
  .gitignore U
  package-lock.json U
  package.json U
  README.md U
```



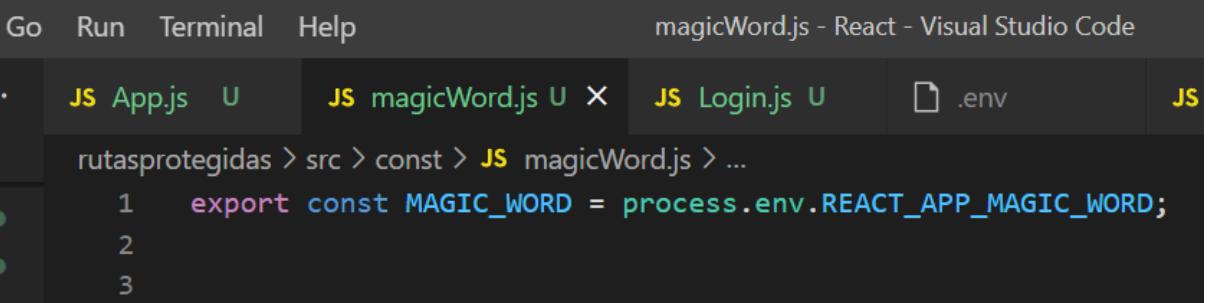
```
-- 
14  # misc
15  .DS_Store
16  .env
17  .env.local
18  .env.development.local
19  .env.test.local
20  .env.production.local
21
22  npm-debug.log*
```

Y ahora en el archivo .env vamos a declarar que es una constante:



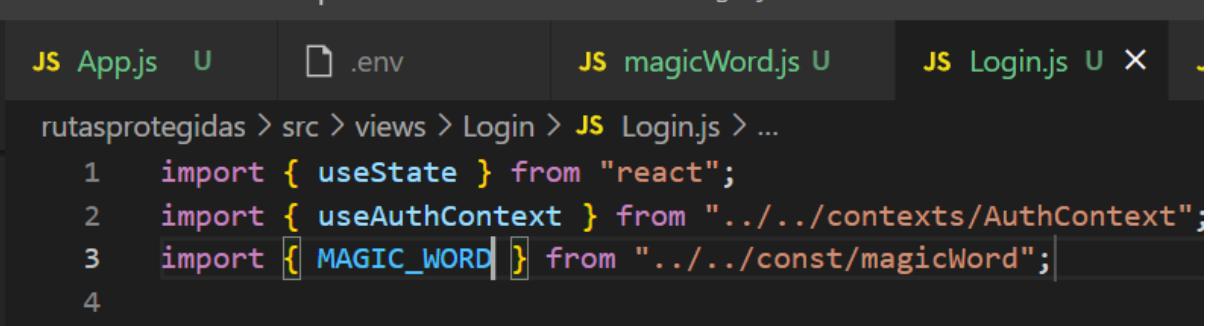
```
JS App.js U .env X JS Login.js U JS mag
rutasprotegidas > .env
1 REACT_APP_MAGIC_WORD = 'Abracadabra'
```

Y ahora que esta información está escondida, ahora lo podemos usar en nuestro archivo magicWord:



```
Go Run Terminal Help magicWord.js - React - Visual Studio Code
. JS App.js U JS magicWord.js U X JS Login.js U .env JS
rutasprotegidas > src > const > JS magicWord.js > ...
1 export const MAGIC_WORD = process.env.REACT_APP_MAGIC_WORD;
```

y la importamos en nuestro componente Login:



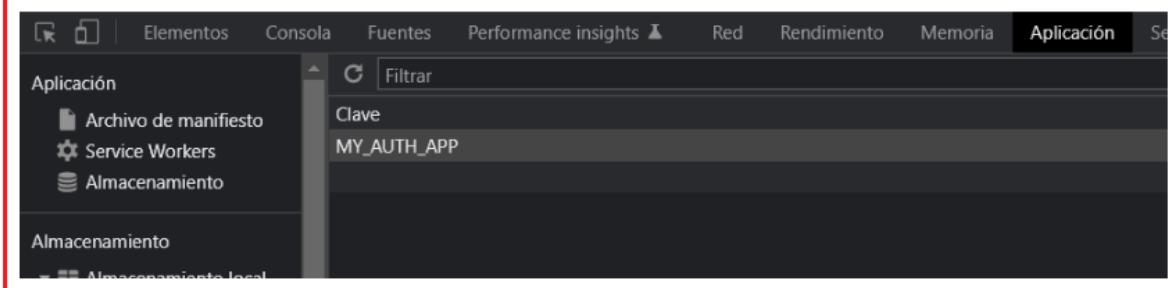
```
JS App.js U .env JS magicWord.js U JS Login.js U X .
rutasprotegidas > src > views > Login > JS Login.js > ...
1 import { useState } from "react";
2 import { useAuthContext } from "../../contexts/AuthContext";
3 import { MAGIC_WORD } from "../../const/magicWord";
4
```

OJO

Al usar archivos de entorno, hay que echar abajo el servidor y volverlo a iniciar.

Aquí podemos ver cómo se almacena en el local storage:

Private

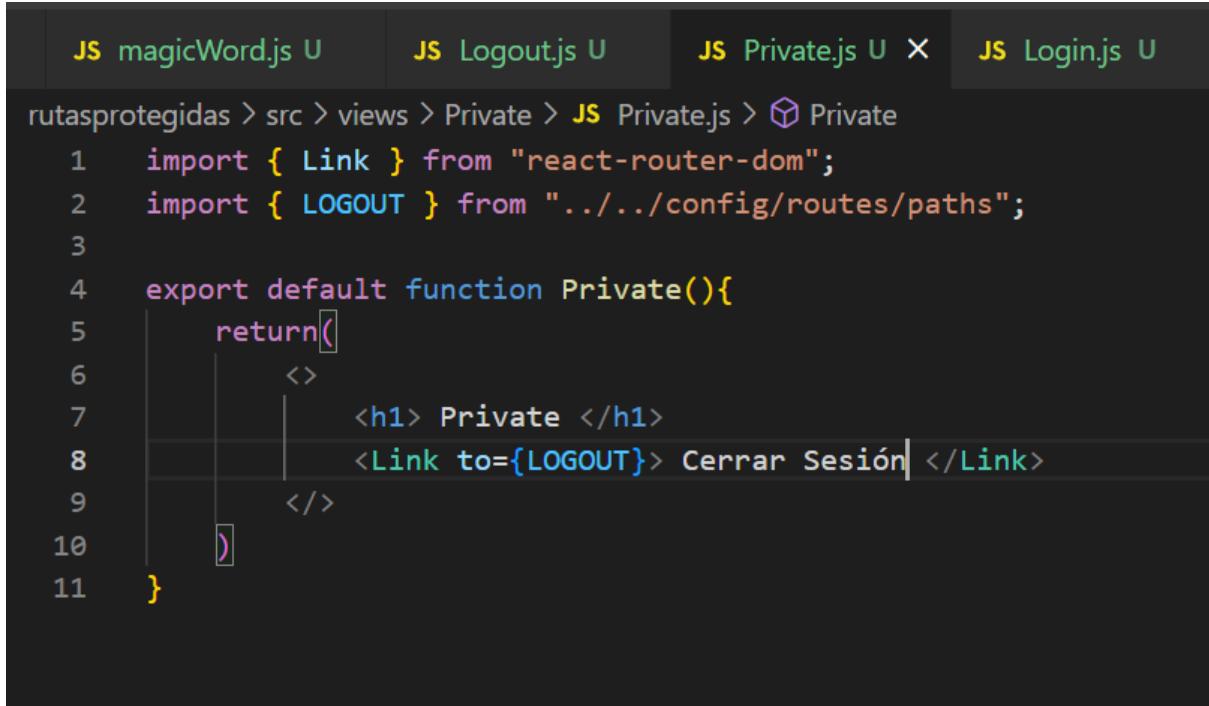


Ahora creamos el logout:

```
rutasprotegidas > src > views > Logout > JS Logout.js U X JS Login.js U .env
1 import {useEffect} from 'react';
2 import { useAuthContext } from '../../../../../contexts/AuthContext';
3
4 export default function Logout(){
5
6     const {logout} = useAuthContext();
7
8     useEffect(() => logout());
9
10    return null;
11}
```

The image shows a code editor with several tabs at the top: magicWord.js (U), Logout.js (U X), Login.js (U), and .env. The Logout.js tab is active. The code in the editor is a functional component named Logout. It imports useEffect and useAuthContext from react and AuthContext respectively. It uses the useAuthContext hook to get the logout function. Then it creates an effect that calls the logout function when the component mounts. Finally, it returns null.

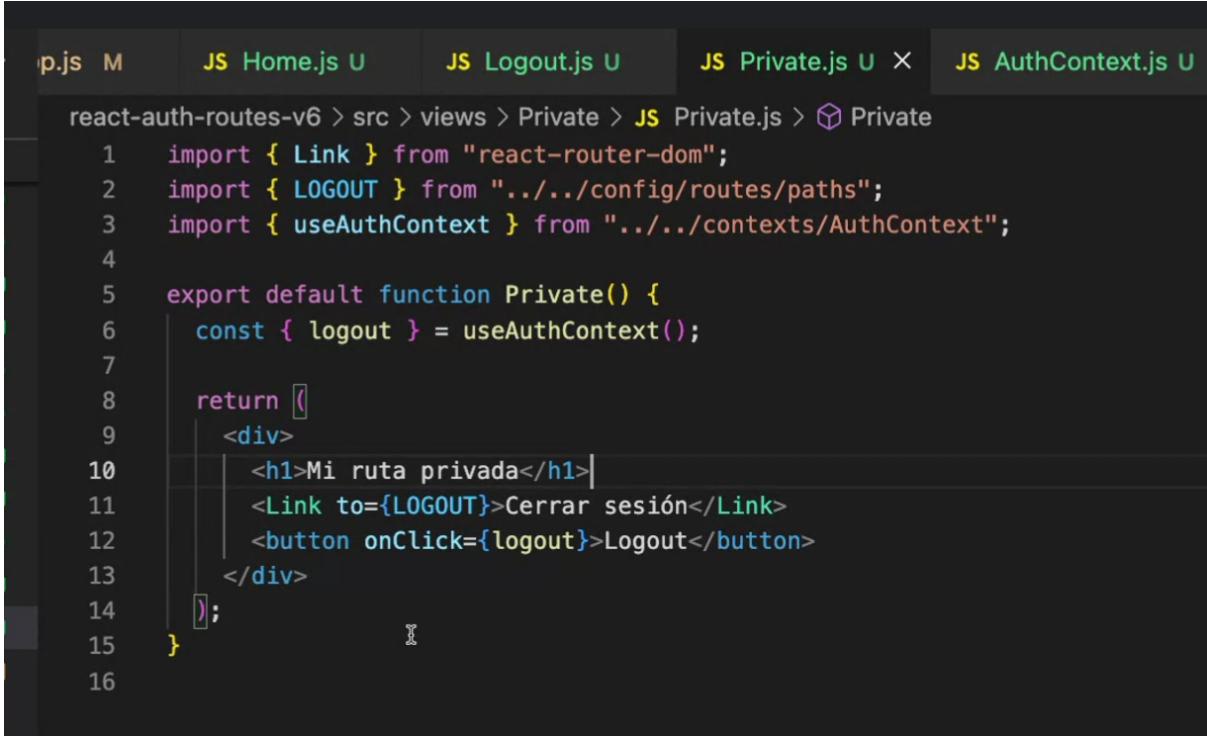
Y en Private haremos el botón de logout:



```
JS magicWord.js U JS Logout.js U JS Private.js U X JS Login.js U
rutasprotegidas > src > views > Private > JS Private.js > Private
1 import { Link } from "react-router-dom";
2 import { LOGOUT } from "../../config/routes/paths";
3
4 export default function Private(){
5   return(
6     <>
7       <h1> Private </h1>
8       <Link to={LOGOUT}> Cerrar Sesión </Link>
9     </>
10   )
11 }
```

Entonces en Private tenemos un enlace a Logout, y Logout ejecuta la función que elimina lo almacenado en LocalStorage sin devolver nada.

También podríamos haber creado un botón en Private, que le asignáramos en el onClick la función logout. Ejemplo de Nacho:



```
JS Home.js U JS Logout.js U JS Private.js U X JS AuthContext.js U
react-auth-routes-v6 > src > views > Private > JS Private.js > Private
1 import { Link } from "react-router-dom";
2 import { LOGOUT } from "../../config/routes/paths";
3 import { useAuthContext } from "../../contexts/AuthContext";
4
5 export default function Private() {
6   const { logout } = useAuthContext();
7
8   return (
9     <div>
10       <h1>Mi ruta privada</h1>
11       <Link to={LOGOUT}>Cerrar sesión</Link>
12       <button onClick={logout}>Logout</button>
13     </div>
14   );
15 }
16
```

De esta forma nuestra aplicación funciona así:

En la página Private al clickar nos lleva a Logout, que retorna null y hace la función de logout. Al hacer la función de logout, el isAuthenticated se queda en falso, y en PrivateRoute le tenemos dicho que si no está autenticado el usuario, que lo mande de vuelta a la pagina Login.