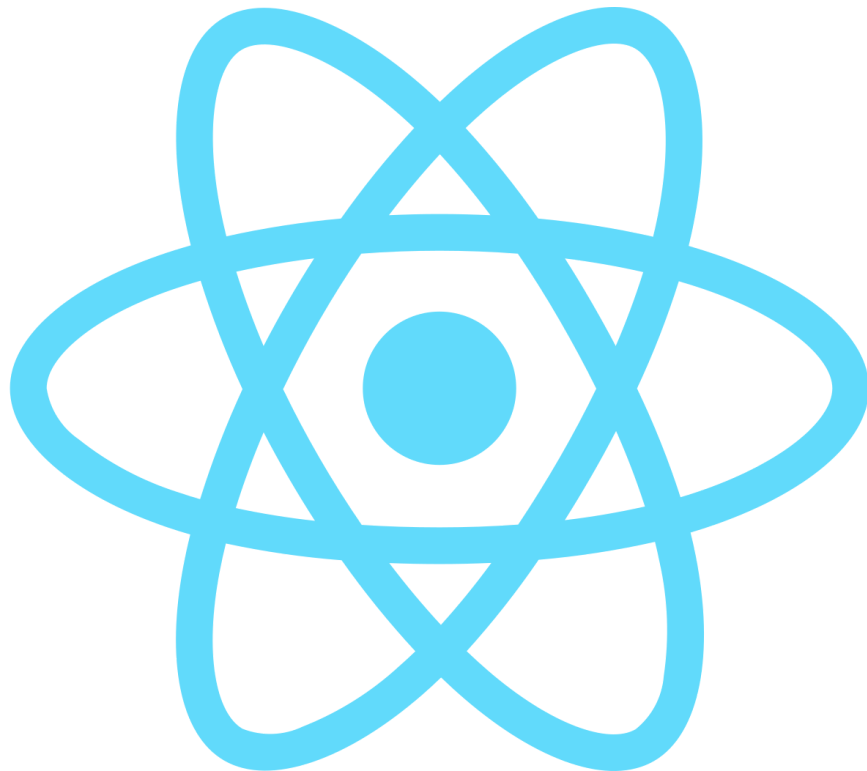


INTRODUCCIÓN

A

REACT



IGNACIO ORTEGA CARRILLO

PASCUAL JUAN SÁNCHEZ

Índice

Índice	1
Introducción	2
Características principales	2
Crear React App	3
Estructura de archivos	6
JSX	8
Componente principal	9
Componentes	11
Hook useState	13
Fragments	14
Componente TodoItem	16
Manejo de cambios con Eventos	17
Keys	18
Flujo de Props y Eventos	18
Hook useEffect	20
Bibliografía	21

Introducción

React, también llamado ReactJS, es una biblioteca de JavaScript diseñada para construir interfaces de usuario con un único objetivo: facilitar *Single Page Apps*. Mantenido actualmente por **Facebook** y la comunidad del software libre. Se utiliza el patrón MVC.

Según el ingeniero Diego Pacheco, *"React es el framework más popular y disruptivo entre los frameworks basados en JavaScript del lado del cliente"*. Ha popularizado el enlace de datos unidireccional, la programación declarativa y el estado inmutable.

Tiene el lema *"Aprenda una vez, escriba en cualquier lugar"*, ya que se puede utilizar para desarrollar aplicaciones de cualquier tipo de interfaz de usuario: web, móvil, escritorio...

Características principales

- **Virtual DOM:** Mantiene un virtual DOM propio en lugar de confiar solamente en el del navegador. La biblioteca es la que se encarga de analizar qué partes del DOM cambian comparando contenidos del estado actual con el anterior y con el resultado determinar la manera más eficiente de actualizarlo.
- **Propiedades:** también conocidas como *props*. Son los atributos de configuración de un componente. Recibidas desde un nivel superior, éstas son inmutables por definición. Esto también lo vemos en otros frameworks como **VUEJS**.
- **Estado:** representación del componente en el momento actual. Dos tipos de componentes: con o sin estado (**statefull o stateless**).
- **Ciclos de vida:** serie de estados por la que pasan los componentes statefull durante su *vida*. Son tres etapas: **de montaje o inicialización, actualización y destrucción**.
- **JSX:** parecido a HTML. No es requerido por React, pero si que hace el código más legible y permite al programador tocar algo ya conocido por él. Como alternativa, la página de Reactjs nos proporciona *Babel REPL*, para ver código JavaScript sin procesar la compilación de JSX.

Como dicen que la práctica hace al maestro, y la mejor forma de aprender un nuevo framework es directamente embarrarte con él, vamos a aprender los principios básicos que todo el mundo que pretende aprender React debería saber.

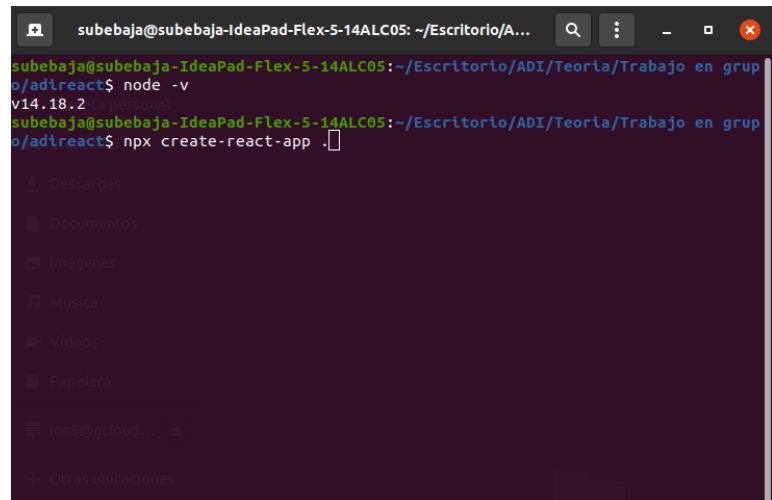
Se va a realizar una sencilla app, como ya estamos acostumbrados, ToDoList. El código está alojado en el siguiente [enlace](#).

Crear React App

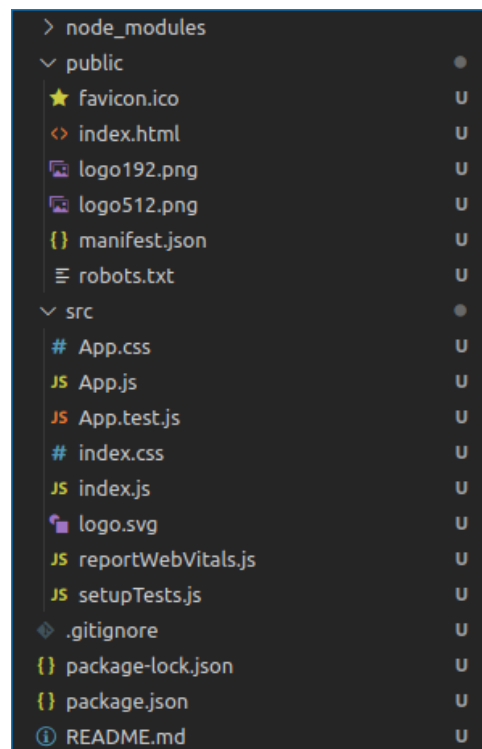
Vamos a usar Node no para trabajar sobre él si no para iniciar el proyecto React y de esta forma se generan los ficheros necesarios para empezar con nuestra aplicación. Necesitamos una versión de Node superior a la **14.0** para poder crear dicho proyecto.

A parte vamos a utilizar como administrador de paquetería **npm**, siendo este una mejora de **npm**, siendo este capaz de ejecutar los paquetes instalados localmente sin archivos de configuración (package.json) de por medio mediante una sola instrucción. Está de forma global pero temporal, ya que no se almacena en disco.

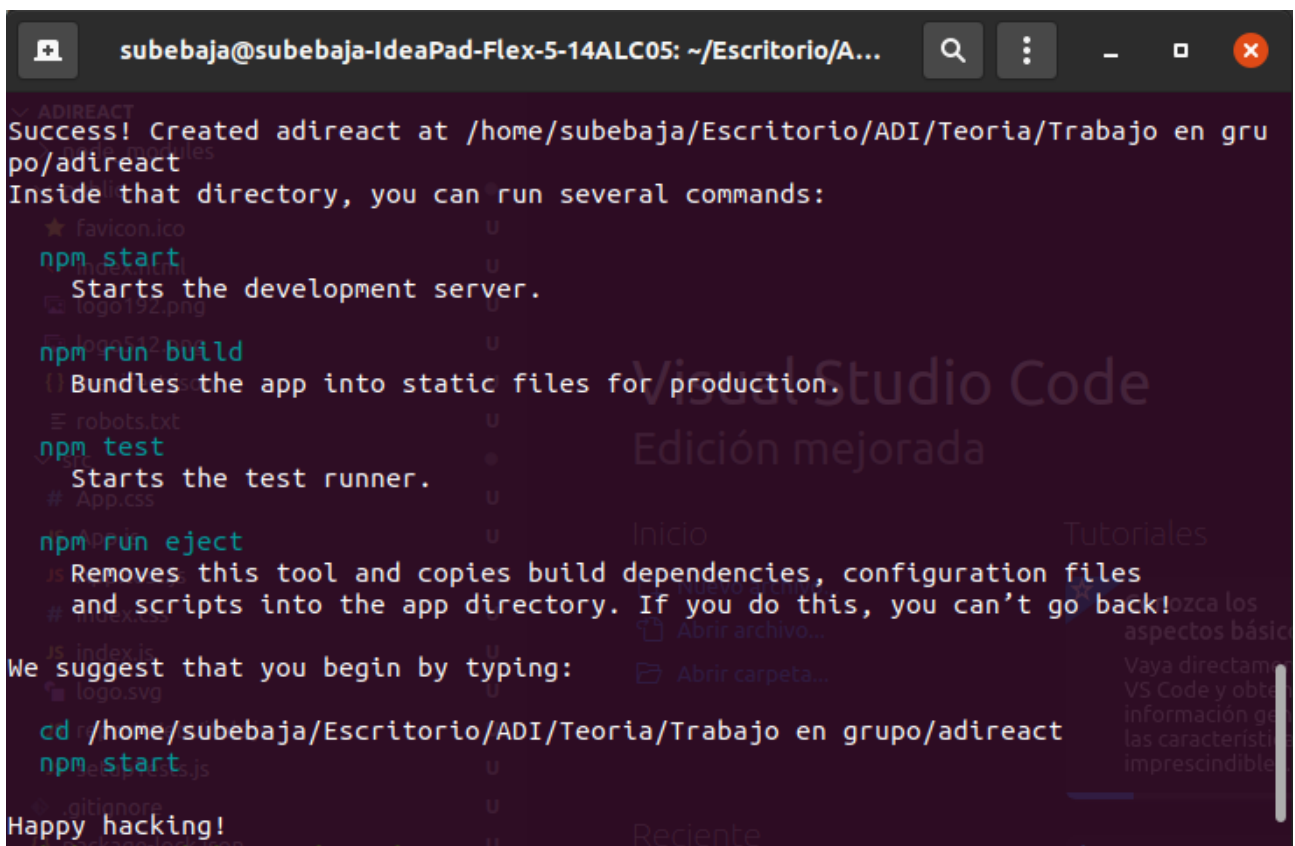
Con el anterior comando vamos a generar dicho proyecto y nos genera los siguientes archivos..



```
subebaja@subebaja-IdeaPad-Flex-5-14ALC05: ~/Escritorio/A...
subebaja@subebaja-IdeaPad-Flex-5-14ALC05:~/Escritorio/ADI/Teoria/Trabajo en grup
o/adirecta$ node -v
v14.18.2
subebaja@subebaja-IdeaPad-Flex-5-14ALC05:~/Escritorio/ADI/Teoria/Trabajo en grup
o/adirecta$ npx create-react-app .
```



Además nos muestra los diferentes comandos que podemos ejecutar y una breve descripción de cada uno de ellos.



```
subebaja@subebaja-IdeaPad-Flex-5-14ALC05: ~/Escritorio/A...
Success! Created adirect at /home/subebaja/Escritorio/ADI/Teoria/Trabajo en grupo/adirect
Inside that directory, you can run several commands:

  $ npm start
    Starts the development server.

  $ npm run build
    Bundles the app into static files for production.

  $ npm test
    Starts the test runner.

  $ npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

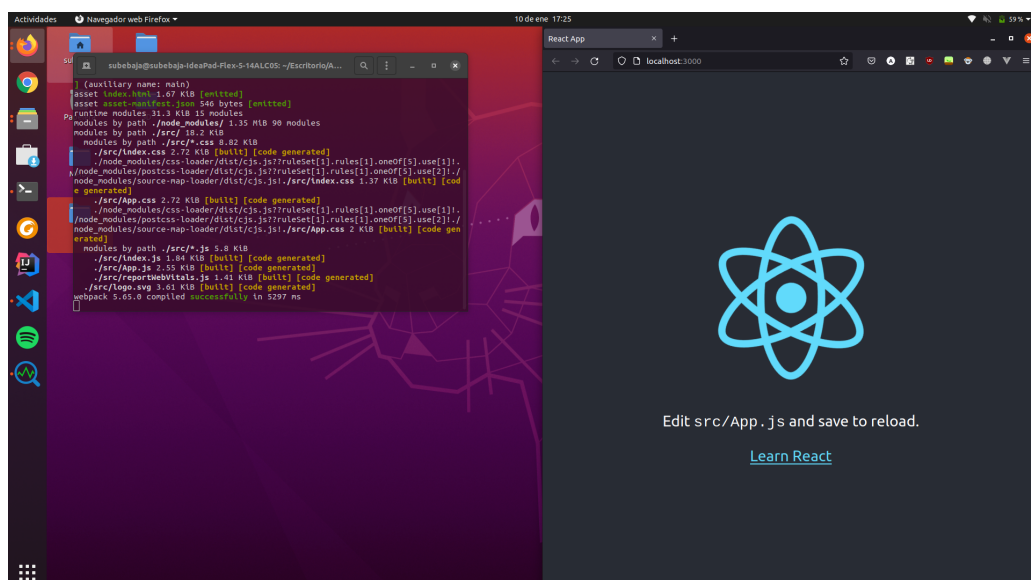
We suggest that you begin by typing:

  cd /home/subebaja/Escritorio/ADI/Teoria/Trabajo en grupo/adirect
  npm start

Happy hacking!
```

Para los que no entiendan muy bien el inglés vamos a proceder a explicar los comandos más relevantes y que usaremos:

- *npm start*. Nos permite ejecutar el servidor de desarrollo donde vamos a trabajar.
- *npm run build*. Empaqueta la aplicación para ponerla luego en producción.
- *npm test*. Ejecuta los test que creamos.



Sin tocar absolutamente nada vamos a hacer una primera ejecución, poniendo en terminal el comando `"npm start"` para ver que nos muestra React como carta de presentación.

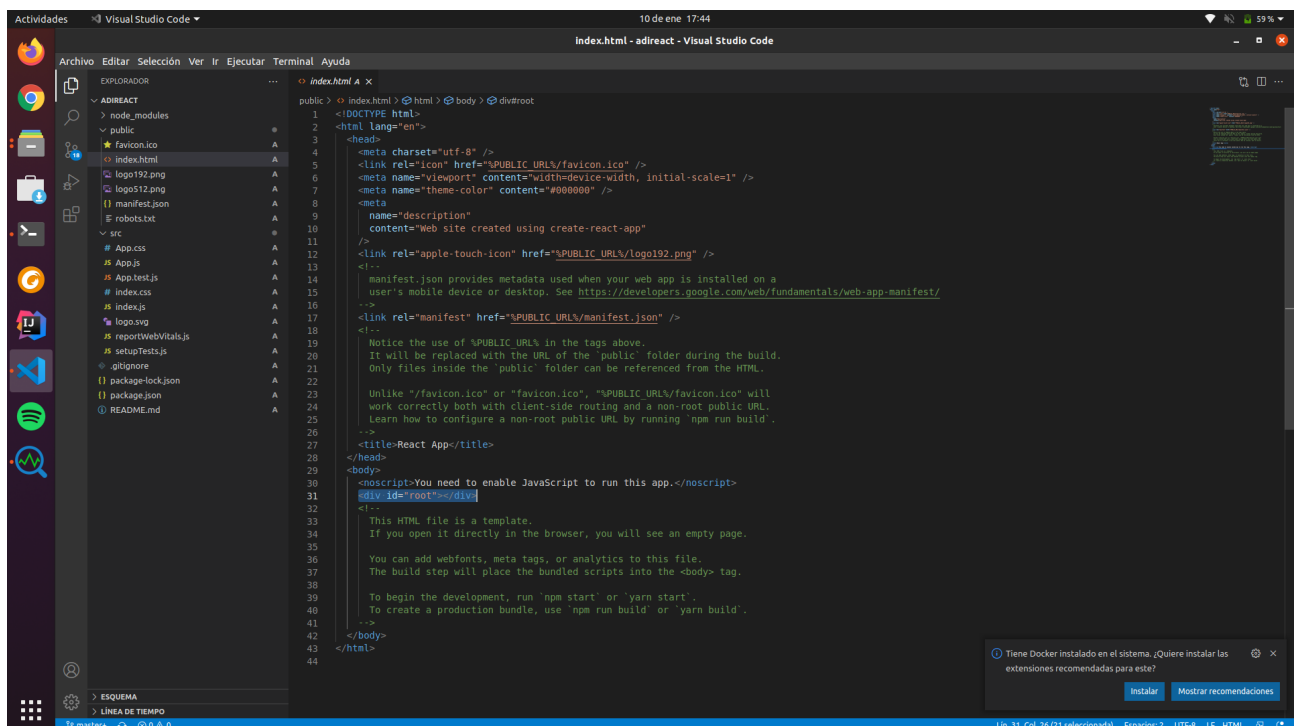
Vemos que directamente nos abre el navegador con la dirección del localhost y el puerto 3000. Ahora entraremos a modificar y agregar código en React, pero antes me gustaría remarcar que dispone de un livereload por tanto todos los cambios que realicemos en el código se verán reflejados directamente.

Hecho esto ya tenemos creado el proyecto y todos los sencillos pasos desde no tener nada hasta tener una pequeña cosa funcional.

Estructura de archivos

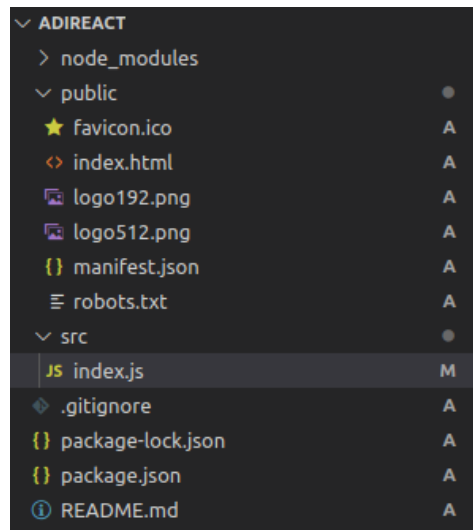
Una vez creado vamos a explicar lo más importante dentro de los archivos que se acaban de generar.

Nos interesa que dentro de la carpeta *public* el *index.HTML* porque será nuestra plantilla o layout de nuestra aplicación ya que como podéis adivinar se trata de una *single page application*, es decir una aplicación web de una sola página.



Como vemos tenemos un *div* con *id root* este *div* vamos a utilizarlo para incluir en el todo el código HTML que vayamos a generar con React.

Por defecto en la carpeta *src* se generan múltiples archivos pero para aprender React vamos a eliminarlos y generamos dentro de esta un nuevo archivo llamado *index.js*.



En el *index.js* vamos a importar React y *ReactDOM*, no hace falta instalar nada ya que en el *package.json* al crear un proyecto React se instalan directamente, esto lo que nos permite es usar JSX.

```
JS index.js M X
src > JS index.js
1 | import React from "react";
2 | import ReactDOM from "react-dom";
3 |
```

JSX

JSX es un lenguaje que al principio puede recordar mucho a HTML pero realmente es JavaScript.

En lugar de separar artificialmente tecnologías poniendo el maquetado y la lógica en archivos separados, React separa intereses con unidades ligeramente acopladas llamadas “*componentes*” que contienen ambas. React no requiere usar JSX, pero la mayoría de la gente lo encuentra útil como ayuda visual cuando trabajan con interfaz de usuario dentro del código JavaScript.

Por ejemplo si quisiéramos crear un div y dentro de ese div además añadir otra cosa en React sería de la siguiente manera:

```
JS index.js 1, M ●
src > JS index.js > ...
1 | import React from "react";
2 | import ReactDOM from "react-dom";
3 |
4 | React.createElement('div', {objeto:propiedades}, React.createElement(...))
```

Como vemos es un poco engorroso y solo hacemos algo bastante habitual que es anidar algo dentro de un div. En cambio usando JSX el mismo código queda de la siguiente manera:

Como hemos dicho antes es muy parecido a HTML pero no lo es.

Vamos a hacer por tanto un hola mundo algo tonto para que explicar un poco como funciona React con JSX.

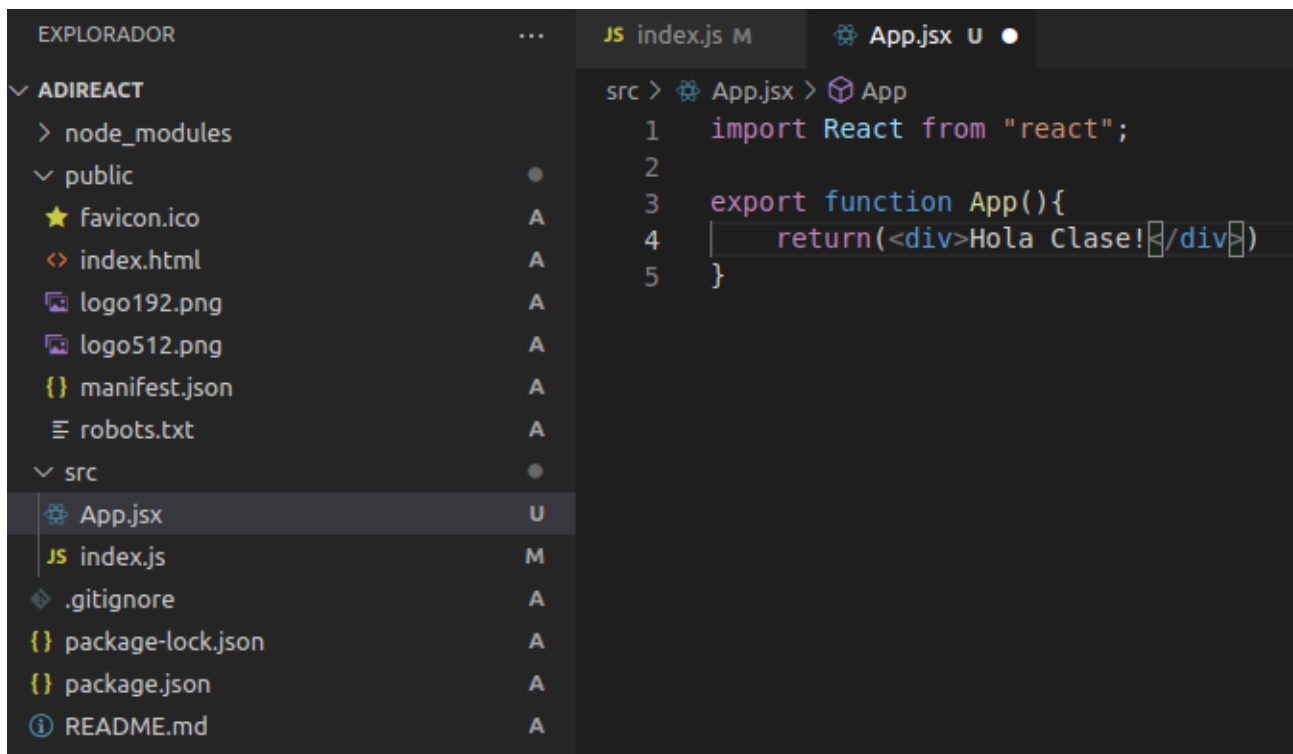
```
JS index.js M ●
src > JS index.js
1 | import React from "react";
2 | import ReactDOM from "react-dom";
3 |
4 | <div>...</div>
```

El *ReactDOM* lo que me va a permitir es renderizar todos los elementos que le pasamos como parámetros. De la anterior forma vemos que añade el div con el mensaje "Hola Clase" y el segundo elemento es el div de la plantilla *index.js* donde queremos embeber el mensaje.



Componente principal

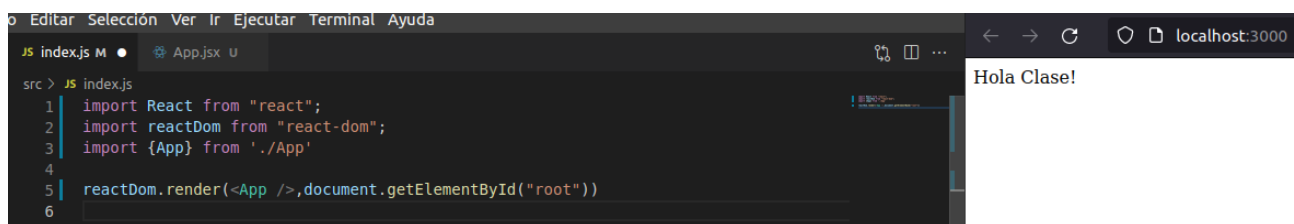
Una vez hecho todo lo anterior vamos a hacer el componente principal es decir el componente que va a contener toda la aplicación. Por lo tanto vamos a crear en la carpeta *src* un archivo *App.jsx* (esta extensión la usaremos sobre todo para aclarar que vamos a usar React, pero se puede dejar la extensión *.js* y funcionaria igual).

The image shows a VS Code window with the Explorer sidebar on the left and a code editor on the right. The Explorer sidebar shows a file tree for a project named 'ADIRECT'. The 'src' folder is expanded, showing 'App.jsx' (highlighted with a 'U' icon) and 'index.js' (with an 'M' icon). The code editor shows the content of 'App.jsx', which contains a single function component named 'App' that returns a JSX element: a div with the text 'Hola Clase!'.

```
1 import React from "react";
2
3 export function App(){
4   return(<div>Hola Clase!</div>)
5 }
```

En la App vamos a importar *React* y creamos una función de nombre *App* la cual va a hacer prácticamente lo mismo que el código anterior en el *index.js* pero le hemos añadido una exclamación para ver que llamamos bien a dicha app.

Pero claro esto de forma independiente no tiene mucho sentido por tanto tenemos que llamar desde el *index.js* a la app esto se hace de la siguiente forma:

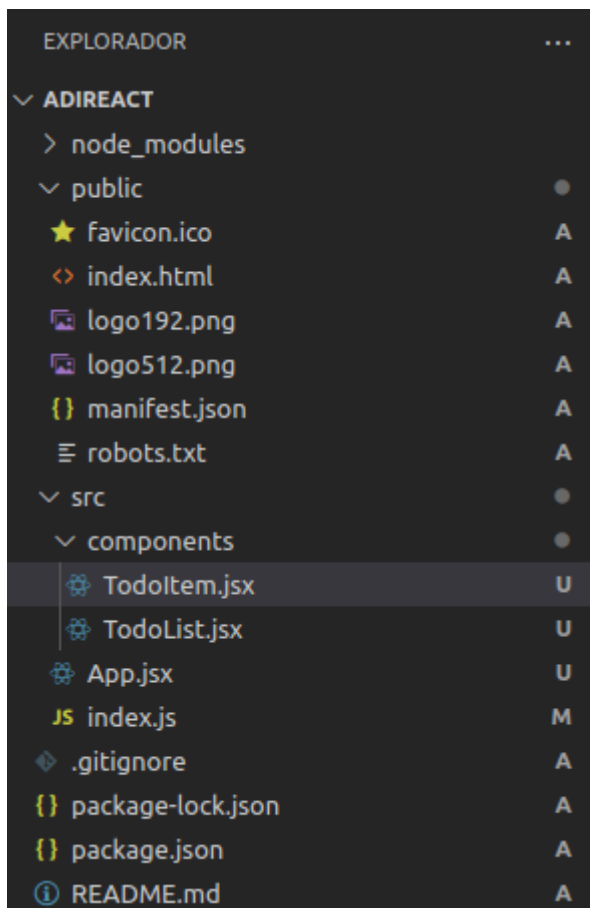
The image shows a VS Code window with the code editor and a browser preview. The code editor shows the content of 'index.js', which imports 'React', 'ReactDOM', and the 'App' component from './App'. It then uses 'ReactDOM.render' to render the 'App' component into the root element of the document. The browser preview on the right shows the rendered output: 'Hola Clase!'.

```
1 import React from "react";
2 import ReactDOM from "react-dom";
3 import {App} from './App'
4
5 ReactDOM.render(<App />,document.getElementById("root"))
6
```

Como se puede observar la App parece un elemento HTML pero realmente es el componente que hemos creado nosotros. También hay diferencias con los elementos HTML y es que empiezan por mayúscula esto es un convenio entre todos los programadores de React y sobre todo para que React no haga cosas extrañas e interprete componentes como elementos HTML.

Componentes

Una vez creado este componente principal podemos crear diferentes componentes así podemos ver cómo se van pasando propiedades entre ellos, cómo van escuchando los eventos y cómo se

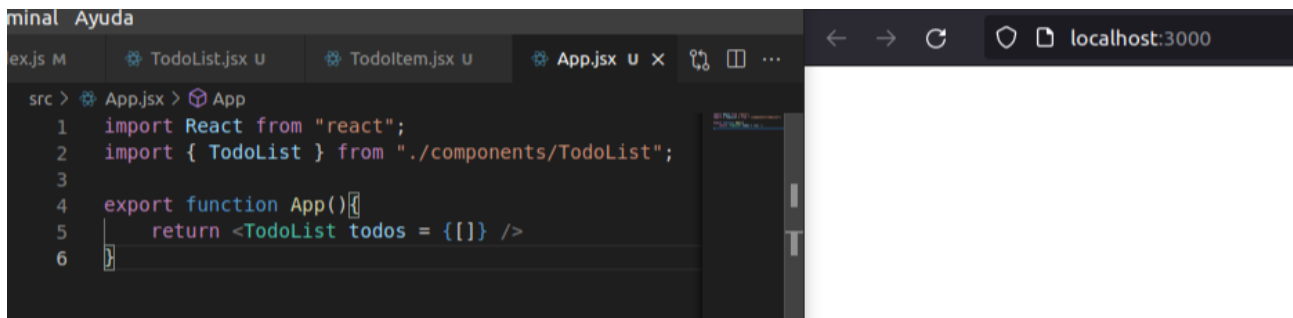


pueden crear componentes compuesto de otros componentes a su vez.-

Siguiendo los pasos para hacer el todolist creamos una carpeta dentro de `src` de nombre `components` y dentro creamos dos archivos: `TodoList.jsx` y `TodolItem.jsx`

Volvemos a importar React y generamos la función de nuestro componente en este caso vamos a empezar por el `TodoList` y este componente nos va a devolver una lista. Como parámetros va a recibir una array de cosas por hacer (*todos*). Tenemos la opción de meter código jsx directamente entre las tag `ul` de `HTML` con las llaves.

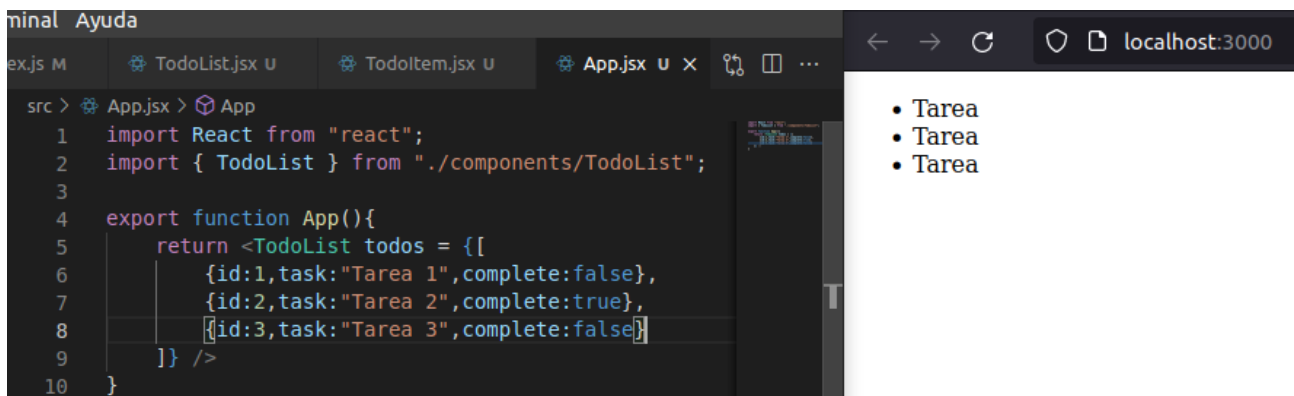
Una vez hecho esto vamos a llamar desde `App.jsx` a este componente, lo importamos y hacemos la llamada pasándole como parámetro, de momento, un array vacío.



The screenshot shows a web browser at localhost:3000 displaying a blank page. The code editor on the left shows the following code in App.js:

```
src > App.jsx > App
1 import React from "react";
2 import { TodoList } from "../components/TodoList";
3
4 export function App() {
5   return <TodoList todos = {} />
6 }
```

Como podemos observar al pasarle un array vacía no se dibuja nada en la página web. En cambio si le a pelo le pasamos un array con atributos vemos que ya imprime algo.



The screenshot shows the web browser at localhost:3000 displaying a list of three items: "Tarea", "Tarea", and "Tarea". The code editor on the left shows the following code in App.js:

```
src > App.jsx > App
1 import React from "react";
2 import { TodoList } from "../components/TodoList";
3
4 export function App() {
5   return <TodoList todos = [
6     {id:1,task:"Tarea 1",complete:false},
7     {id:2,task:"Tarea 2",complete:true},
8     {id:3,task:"Tarea 3",complete:false}
9   ] />
10 }
```

Como vemos al pasarle un array con algún dato muestra ya cosas aunque de momento no es que sea muy útil pero vemos lo sencillo que es y amigable.

Hook useState

Pero nadie va pasar de la forma anterior los datos a la app y por la forma que tienen los datos de tareas tiene pinta de que vamos a tener que usar una especie de estados, el estado hace que cada vez que guardemos o cambiemos algún dato de dicho estado se vuelve a renderizar el componente para mostrar los cambios, entonces vamos a importar junto a *React* el *hook useState* para poder usarlos.

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

El *useState* es un array que devuelve el estado en sí y la función que hace modificar ese estado por tanto creamos dicho *useState* le decimos que la función que modifica se va a llamar *setTodos* y llamamos a los *todos* para la componente.

```
src > App.jsx > App
1  import React, {useState} from "react";
2  import { TodoList } from "../components/TodoList";
3
4  export function App(){
5      const [todos, setTodos] = useState([
6          {id:1, task: "Tarea 1", complete: false},
7          {id:2, task: "Tarea 2", complete: true},
8          {id:3, task: "Tarea 3", complete: false}
9      ])
10     return <TodoList todos = {todos} />
11 }
```

Fragments

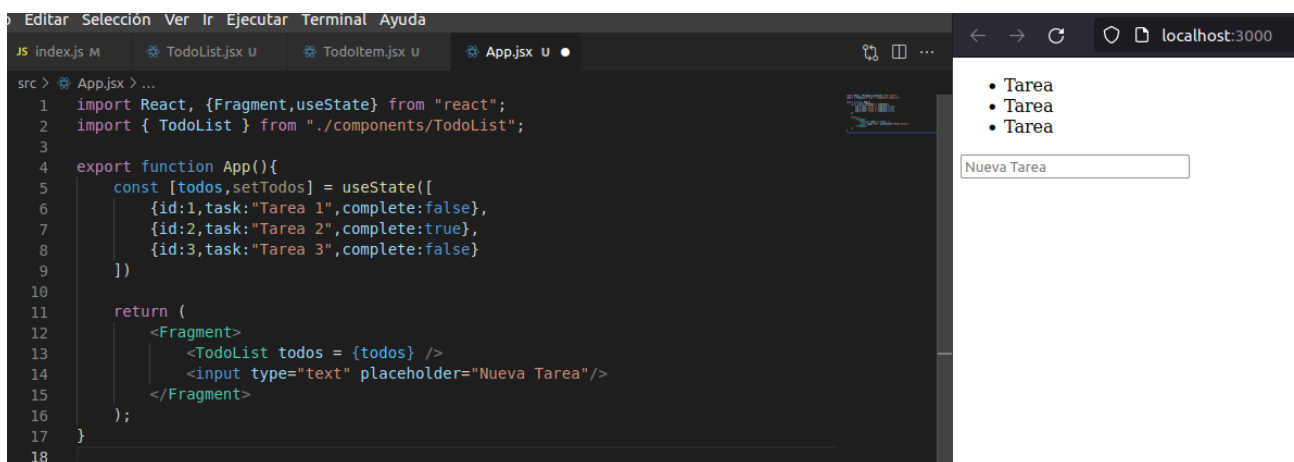
Pero bueno hasta el momento tenemos un app que reactiva, lo que se dice reactiva, no es.

Tampoco es que sea útil ya que nos gustaría poder añadir tareas, eliminarlas, marcarlas como completadas y demás.

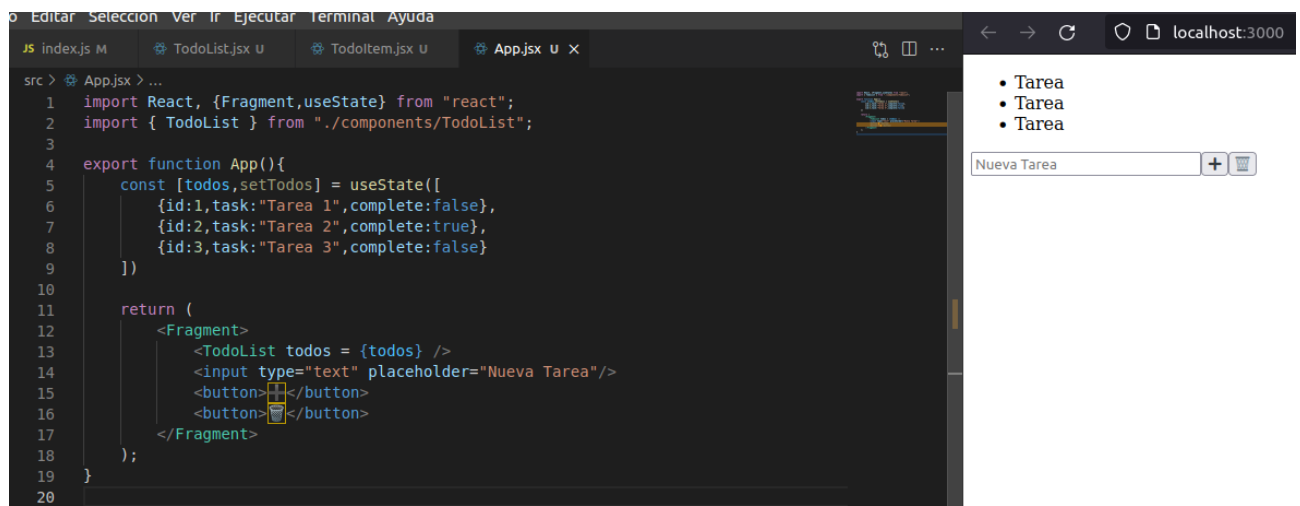
Por tanto vamos a añadir algo básico que es el poder añadir tareas. Lo lógico sería añadir un input type text para añadir el título de la tarea. Pero esto falla ya que no podemos porque estamos mandando dos componentes distintos y React por como renderiza solo puede devolver una sola componente, anteriormente se hacía la chapuzilla de englobar todo dentro de un div por tanto era una misma componente.

```
1  port React, {useState} from "react";
2  port { TodoList } from "../components/TodoList";
3
4  port function App(){
5    const [todos,setTodos] = useState([
6      {id:1,task:"Tarea 1",complete:false},
7      {id:2,task:"Tarea 2",complete:true},
8      {id:3,task:"Tarea 3",complete:false}
9    ])
10
11   return (
12     <TodoList todos = {todos} />
13     <input type="text" placeholder="Nueva Tarea"/>
14   );
15
```

Para solucionar eso usaremos los *fragments* de React. Como las anteriores herramientas de React debemos importarlas y su uso es muy sencillo:



Una vez hecho esto podemos añadir más elementos, como por ejemplo botones para añadir o eliminar una tarea.



Componente TodoItem

Es el hijo de TodoList, por lo tanto lo definiremos como un ``. Debemos crear dicho componente en el que vemos las **propiedades**, **que** corresponden a la variable `todo`. Es lo que nos manda el padre (elemento de nivel superior) cuando se realiza una llamada. En este caso sería un elemento de la lista, que tiene como propiedades el *id*, *tarea* y *completada*.

```
import React from 'react' 7.2K (gzipped: 2.9K)

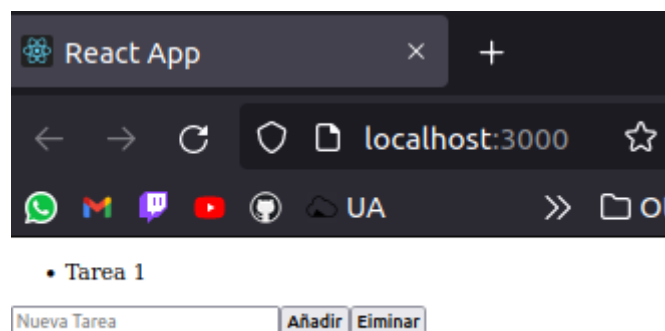
export function TodoItem({todo}) {
  return (
    <li>
      {todo.task}
    </li>
  )
}
```

A parte vemos una llamada a una propiedad del objeto, mostrándolo en el código mediante llaves.

En cuanto al elemento padre, tendremos que realizar dicha llamada. En ella indicamos las variables que pasamos con los datos del array de la lista, siempre entre llaves. En general, mapeamos dicho array en el que en cada elemento imprimimos un componente *TodoItem*.

```
<ul>
  { todos.map((todo) => {
    return(
      <TodoItem todo={todo}></TodoItem>
    )
  }) }
</ul>
```

Comprobamos el resultado actual al recargar.



Manejo de cambios con Eventos

Dentro de *App.jsx*, añadiremos al elemento, en este caso el botón, donde se quiera manejar el evento. En este caso definiremos un evento cuando se clique sobre él (**onClick**). Definiremos a su vez dicha función a la que se llamará cuando se realice el evento.

```
<button onClick={handleTodoAdd}>Añadir</button>
```

Analicemos la definición del manejador del evento. Utilizamos `useRef`, que deberemos importar de la librería de React, y

básicamente lo utilizamos para que React identifique a qué elemento del JSX hacemos referencia, y así obtener el valor del input de forma dinámica. A parte hay que crear en el

```
const todoTaskRef = useRef()
const handleTodoAdd = () => {
  const task = todoTaskRef.current.value;

  if(task === '') return

  setTodos((prevTodos) => {
    return [...prevTodos, {id: uuidv4(), task, completed: false}]
  })
}
```

input un valor **ref** en el que indicamos la variable que hace referencia, en este caso *todoTaskRef*.

Ya añadimos esa tarea al array de *todos*. Lo hacemos mediante la función que declaramos anteriormente **setTodos**, que nos permite acceder al estado previo, tratando así al array de manera inmutable. Si haces cambios, siempre se tienen que generar mediante una copia del estado anterior. Si no React no puede comparar el estado anterior con el actual y no puede llegar a comparar cambios en el DOM. Por tanto devolvemos el estado anterior más la nueva tarea.

A modo de curiosidad, para generar el *id* se ha hecho uso de la librería **uuid**, la cual te permite crear un id de forma aleatoria. Se podría hacer manualmente, pero siempre es preferible hacer uso de herramientas que ya nos proporcionan los propios desarrolladores. Importamos la librería y realizamos la llamada para que se ejecute.

```
import { v4 as uuidv4 } from 'uuid'
```


Keys

Necesario para que React distinga qué elemento es e identificar qué elementos han cambiado. En este caso es necesario en cada elemento de la lista, para que React pueda llevar a cabo un renderizado óptimo. Sencillamente añadimos el atributo **key** a la etiqueta de llamada a cada elemento *ToDoItem*. Para ello hacemos uso del elemento id que sabemos que siempre va a ser único.

```
<ToDoItem key={todo.id} todo={todo}></ToDoItem>
```

A su vez estas deben ser únicas solamente entre sus hermanos, no de forma global.

Flujo de Props y Eventos

Aquí vamos a la cuestión importante de la app: marcar como completada o no una tarea. Para ello vamos a realizar una comunicación del nivel más alto al más bajo para realizar el cambio de estado.

En *App.jsx* creamos la función que realiza el cambio. En él realizamos una copia del estado actual de *todos*, buscamos el

```
const toggleTodo = (id) => {  
  const newTodos = [...todos]  
  const todo = newTodos.find((element) => element.id === id)  
  todo.completed = !todo.completed;  
  setTodos(newTodos)  
}
```

elemento que tenga el mismo *id* y cambiamos el estado de *completed* (si ya estaba completada se marcará como no completada y al revés). Después de la definición, le pasamos al elemento hijo la función.

```
<ToDoList todos={todos} toggleTodo={toggleTodo}/>
```

Hacemos lo mismo en *ToDoList*, en el que además le llegará ya del elemento padre.

```
export function ToDoList({ todos, toggleTodo }) {  
  return (  
    <ul>  
      { todos.map((todo) => {  
        return(  
          <ToDoItem key={todo.id} todo={todo} toggleTodo={toggleTodo}></ToDoItem>  
        )  
      })}  
    </ul>  
  )  
}
```

Ya en *ToDoItem*, lo primero será crear un checkbox que permita la modificación del estado. Lo crearemos con dos atributos:

- **Checked:** que recibirá el estado actual de *completed*.
- **onChange:** evento que salta cuando se realiza un cambio en el atributo. Este ejecutará la función que le ha sido enviada desde el padre.

Aquí tenemos el código correspondiente a lo anteriormente citado:

```
export function ToDoItem({ todo, toggleTodo }) {
  const { id, task, completed } = todo;

  const handleTodoClick = () => {
    toggleTodo(id);
  }

  return (
    <li>
      <input type="checkbox" checked={completed} onChange={handleTodoClick}/>
      {task}
    </li>
  )
}
```

Vamos a realizar el mismo proceso pero esta vez eliminando las que ya están completadas cada vez que pulsamos el botón de eliminar.

Basta con crear la siguiente función y la llamada correspondiente de *onClick* al botón. Simplemente filtramos el *todos* actual quedándonos solo con los que no están completados.

```
const handleClearAll = () => {
  const todosNoCompleted = todos.filter((todo) => !todo.completed);
  setTodos(todosNoCompleted)
}
```

Hook useEffect

Hasta ahora todo lo que hemos realizado, todas las modificaciones, una vez se recarga la página, se pierde. Para hacer el sistema más persistente hacemos uso de este hook, *useEffect*.

```
import React, { Fragment, useState, useRef, useEffect } from "react";
```

Lo importamos de la librería de React y ya podemos hacer uso de él. También vamos a utilizar el *localStorage*, consiguiendo así la persistencia aunque solamente sea en el navegador.

Haciendo uso entonces del *useEffect*, como vemos en la imagen, este nos permite llevar a cabo efectos secundarios en componentes funcionales. Definimos dos acciones:

```
useEffect(() => {  
  const storedTodos = JSON.parse(localStorage.getItem(KEY));  
  if(storedTodos)  
    setTodos(storedTodos)  
}, [])  
  
useEffect(() => {  
  localStorage.setItem(KEY, JSON.stringify(todos))  
}, [todos])
```

guardar el estado del array *todos* cada

vez que sufre alguna modificación, y recargar lo almacenado en *localStorage* cuando se recarga la página (vemos como el array no tiene valor ninguno en la segunda llamada).

Básicamente le decimos a React mediante este *hook* que el componente tiene que hacer algo después de renderizarse. React la llamará después de actualizar el DOM.

Hay dos tipos de efectos secundarios en los componentes de React:

- **Efectos sin saneamiento:** hay veces que queremos ejecutar código adicional después de que React haya actualizado el DOM. Peticiones de Red, registros... Todos ellos lo único que se quiere es ejecutar el código y después ya olvidarse.
- **Efectos con saneamiento:** algunos efectos sí que lo necesitan. Por ejemplo, si queremos establecer una suscripción a alguna fuente de datos externa. Aquí es importante el saneamiento para evitar una fuga de memoria. Se sanea cuando el componente se desmonta, pero se ejecuta después de cada renderizado.

Conclusión

Y así tendríamos realizada una primera versión de una aplicación TodoList, y a su vez, una introducción hacia la librería React para programación web.

Aquí se han visto los principales conceptos necesarios para llevar a cabo proyectos sencillos: creación de componentes, comunicación de nivel superior a inferior y viceversa, persistencia, uso de elementos nativos de React...

En cuanto a Vue, React nos ofrece muchas herramientas por sí sola para facilitarnos bastante la vida, como es el caso de JSX. Habiendo hecho la misma aplicación en una y otra, me parece más legible y más manejable el sistema de Reac que el de Vue.

De todas formas, si algo no ha quedado claro o no se ha entendido del todo, aquí dejamos un [vídeo](#) a modo de demo donde se ve de manera visual el avance del desarrollo de la aplicación.

Esperamos que haya servido de ayuda para iniciarse en esta tecnología puntera actualmente y altamente demandada. ¡A seguir programando y a seguir aprendiendo!

Bibliografía

Información acerca de React:

<https://es.wikipedia.org/wiki/React>

<https://es.reactjs.org/>

Ranking top 20 frameworks desarrollo web:

<https://diegooo.com/frameworks-de-desarrollo-web-mas-solicitados-en-2021/>

Tutorial basado en el vídeo de Carlos Azaustre:

<https://www.youtube.com/watch?v=EMk6nom1aS4&list=WL&index=27>