



# Introducción a React

## Tabla de Contenidos

[Tabla de Contenidos](#)

[Configurar el entorno y crear el proyecto](#)

[Instalar Node.js y NPM](#)

[Crear el proyecto usando Create React App](#)

[Abrir el proyecto en Visual Studio Code](#)

[Ejecutar nuestra aplicación React](#)

[Estructura del proyecto](#)

[El archivo `package.json`](#)

[El fichero `index.html`](#)

[El archivo `index.js`](#)

[El archivo `App.js`](#)

[Conceptos 1](#)

[Componentes funcionales](#)

[Extraer un componente a un módulo](#)

[Componentes clase](#)

[Sobre el objeto `props`](#)

[Introducción a Hooks y estado](#)

[Aplicación de ejemplo 1: Lista de Tareas](#)

[Creamos una aplicación nueva con Create React App](#)

[Editamos App.js](#)

[Desglosando](#)

[Módulos Importados](#)

[Hooks](#)

[Manejador de Eventos](#)

[Bloque de renderizado](#)

[Conceptos 2](#)

[Componentes Clase](#)

[Métodos de ciclo de vida](#)

[Composición](#)

[Componente FilterableProductTable](#)

[Componente SearchBar](#)

[Componente ProductTable](#)  
[App.css](#)  
[Usando Axios](#)  
[Usando React-Bootstrap](#)

# Configurar el entorno y crear el proyecto

## Instalar Node.js y NPM

### Node.js

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.

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



## Crear el proyecto usando Create React App

```
npx create-react-app nombre-de-la-app
```

npx es un programa que ejecuta scripts que viene incluido con Node.js desde hace bastante tiempo.

Lo utilizamos para ejecutar un script creado por el equipo de React para construir una aplicación vacía en React con muchas de las dependencias más habituales.

Le llevará un tiempo, tiene que descargar mucho código que irá incluido en nuestra aplicación y aun mas código que se usa como herramientas de desarrollo.

## Abrir el proyecto en Visual Studio Code

```
cd nombre-de-la-app  
code .
```

Accedemos a la carpeta que ha creado Create React App y una vez dentro abrimos la carpeta con Visual Studio Code

## Ejecutar nuestra aplicación React

```
npm run start
```

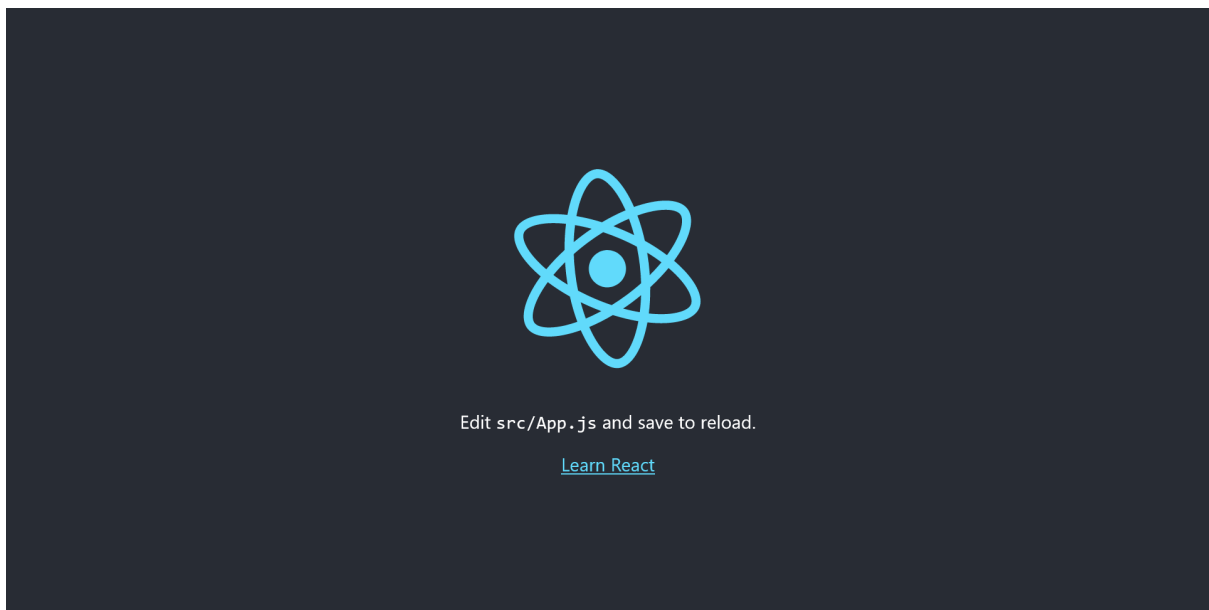
npm puede ejecutar scripts que hayamos definido en nuestro package.json.

Si miramos el nuestro en la sección de scripts encontramos:

```
"scripts": {  
  "start": "react-scripts start",  
  "build": "react-scripts build",  
  "test": "react-scripts test",
```

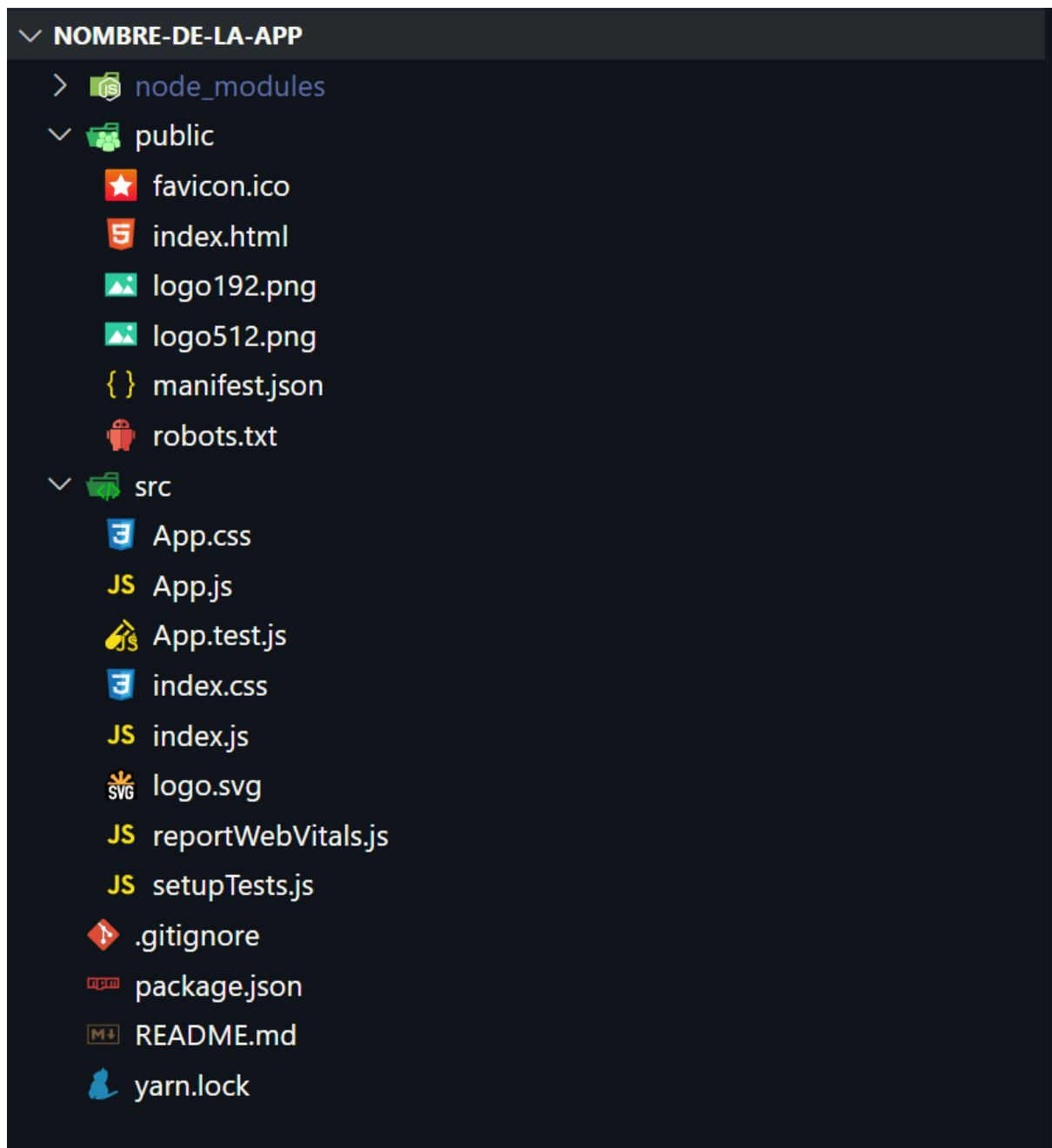
```
"eject": "react-scripts eject"
}
```

- `start` ejecuta un entorno de desarrollo. Comprueba si hacemos cambios para re-lanzar el servidor actualizado.
- `build` empaqueta el código para su uso en producción.
- `test` ejecuta los test que hayamos escrito en Jest.
- `eject` des-empaqueta módulos de dependencia de React para que podamos interactuar con ellos (avanzado).



La aplicación que crea React Create App

## Estructura del proyecto



## El archivo `package.json`

```
{
  "name": "nombre-de-la-app",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.11.4",
    "@testing-library/react": "^11.1.0",
    "@testing-library/user-event": "^12.1.10",
    "react": "^17.0.2",
    "react-dom": "^17.0.2",
    "react-scripts": "4.0.3",
    "web-vitals": "^1.0.1"
  },
  "scripts": {
    "start": "react-scripts start",
```

```

    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}

```

El archivo `package.json` contiene información sobre la aplicación (nombre, versión, autor, etc) y además sus dependencias.

Cuando utilizamos el comando `npm install` lo que hará será instalar dichas dependencias (se encuentran en la carpeta `node_modules`)

## El fichero `index.html`

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <!--
      manifest.json provides metadata used when your web app is installed on a
      user's mobile device or desktop. See https://developers.google.com/web/fundamentals/web-app-manifest/
    -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <!--
      Notice the use of %PUBLIC_URL% in the tags above.
      It will be replaced with the URL of the `public` folder during the build.
      Only files inside the `public` folder can be referenced from the HTML.

      Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
      work correctly both with client-side routing and a non-root public URL.
      Learn how to configure a non-root public URL by running `npm run build`.
    -->
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <!--

```

```
This HTML file is a template.  
If you open it directly in the browser, you will see an empty page.  
  
You can add webfonts, meta tags, or analytics to this file.  
The build step will place the bundled scripts into the <body> tag.  
  
To begin the development, run `npm start` or `yarn start`.  
To create a production bundle, use `npm run build` or `yarn build`.  
-->  
</body>  
</html>
```

## El archivo `index.js`

```
import React from 'react';  
import ReactDOM from 'react-dom';  
import './index.css';  
import App from './App';  
import reportWebVitals from './reportWebVitals';  
  
ReactDOM.render(  
  <React.StrictMode>  
    <App />  
  </React.StrictMode>,  
  document.getElementById('root')  
>);  
  
// If you want to start measuring performance in your app, pass a function  
// to log results (for example: reportWebVitals(console.log))  
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals  
reportWebVitals();
```

## El archivo `App.js`

```
import logo from './logo.svg';  
import './App.css';  
  
function App() {  
  return (  
    <div className="App">  
      <header className="App-header">  
        <img src={logo} className="App-logo" alt="logo" />  
        <p>  
          Edit <code>src/App.js</code> and save to reload.  
        </p>  
        <a  
          className="App-link"  
          href="https://reactjs.org"  
          target="_blank"  
          rel="noopener noreferrer"  
        >  
          Learn React  
        </a>  
      </header>  
    </div>  
  );  
}  
  
export default App;
```

# Conceptos 1

## Componentes funcionales

Modificamos el archivo `App.js`

```
import './App.css';

const Saludo = (props) => <p>Hola {props.nombre}</p>

function App() {
  return (
    <div className="App">
      <Saludo nombre="Alicia" />
      <Saludo nombre="Bernardo" />
      <Saludo nombre="Carmen" />
      <Saludo nombre="Darío" />
    </div>
  );
}

export default App;
```

Saludo es un componente de React. Se construye como una función pero en el retorno podemos escribir código **JSX**

Una forma más explícita de escribir el componente funcional es:

```
function Saludo(props) {
  return (
    <p>Hola {props.nombre}</p>
  )
}
```

Los componentes reciben un objeto como parámetro, sus propiedades o `props`

Cuando añadimos un componente a la aplicación lo hacemos usando su nombre como una etiqueta XML

```
<Saludo nombre="Alicia" />
```

El valor `Alicia` se guardará en `nombre` en el objeto `props` del elemento

También podemos deconstruir el objeto props

```
function Saludo({nombre}) {
  return (
    <p>Hola {nombre}</p>
  );
}
```

## Extraer un componente a un módulo

Creamos el fichero `Saludo.js` dentro de `src`

```
function Saludo({ nombre }) {
  return (
    <p>Hola {nombre}</p>
  );
}

export default Saludo;
```

Importamos el módulo en App.js

```
import './App.css';
import Saludo from './Saludo';

function App() {
  return (
    <div className="App">
      <Saludo nombre="Alicia" />
      <Saludo nombre="Bernardo" />
      <Saludo nombre="Carmen" />
      <Saludo nombre="Diego" />
    </div>
  );
}

export default App;
```

Para mantener mas ordenado nuestro proyecto podemos crear una carpeta components y guardar dentro los archivos de componente. Si lo hacemos el `import` cambia de esta manera:

```
import Saludo from './components/Saludo';
```

## Componentes clase

Otra forma de definir un componente es con una clase de Javascript ES6

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

class Saludo extends Component {
  render() {
    return (
      <p>Hola {this.props.nombre}</p>
    );
  }
}

export default Saludo;
```

Podemos deconstruir el `import` para solo extraer lo que necesitamos (en este caso solo la clase `Component` del módulo `react`)

El componente es una clase que extiende a `Component`

La clase puede contener los métodos que queramos pero como mínimo tiene que contener un método `render` que retorne el código **JSX**

El objeto `props` es igual al del componente funcional.



## Sobre el objeto `props`

El objeto props debe ser inmutable. Los componentes no deben modificar props, solo utilizarlo.

## Introducción a Hooks y estado

Para almacenar datos dentro de la aplicación usamos el estado.

React ha incorporado una forma muy sencilla de crear estado para componentes funcionales a través de un Hook (gancho)

Importamos `useState` de `react`

```
import { useState } from 'react';
```

Creemos un componente funcional nuevo, Contador

```
function Contador() {
  const [numero, setNumero] = useState(0);

  return (
    <div>
      <p>Has hecho click {numero} veces</p>
      <button onClick={() => setNumero(numero + 1)}>
        Haz click
      </button>
    </div>
  )
}
```

la función `useState` retorna array. El array tiene dos elementos, un estado, un "almacén" donde podemos guardar un dato, y una función para modificar su valor.

Lo habitual es usar `destructure` para separarlos en dos variables directamente.

No debemos manipular el estado directamente, en su lugar usamos la función que retorna `useState`.

Podemos modificar el componente para que tenga propiedades:

```
function Contador({ nombre, inicial }) {
  const [numero, setNumero] = useState(Number(inicial)); //Conversion de tipo

  return (
    <div>
      <h2>{nombre}</h2>
      <p>Has hecho click {numero} veces</p>
      <button onClick={() => setNumero(numero + 1)}>
        Haz click
      </button>
    </div>
  )
}
```

Y modificar la App para que use dos contadores:

```
function App() {
  return (
```

```

    <div className="App">
      <Contador
        nombre="Contador 1"
        inicial="0" />
      <Contador
        nombre="Contador 2"
        inicial="100" />
    </div>
  );
}

```

## Aplicación de ejemplo 1: Lista de Tareas

Esta aplicación (muy) sencilla utiliza un único componente y tiene como objetivo enseñar las particularidades de los Hooks, el estado y los manejadores de eventos en React.

El código completo se puede encontrar en el siguiente repo de GitHub.

GitHub - cmo7/react-notas

This project was bootstrapped with Create React App. In the project directory, you can run: Runs the app in the development mode. Open <http://localhost:3000> to view it in the browser. The page will reload if you make edits. You will also see any

<https://github.com/cmo7/react-notas>

cmo7/react-notas



1 Contributor 0 Issues 0 Stars 0 Forks

## Creamos una aplicación nueva con Create React App

```
npx create-react-app nombre-de-la-app
```

## Editamos App.js

```

//Módulos Importados
import './App.css';
import { useState, useRef, useEffect } from 'react';

//Componente App
function App() {

  // Hooks
  const [tareas, setTareas] = useState([]);
  const tareaTexto = useRef();
  useEffect(() => {
    const tareasExistentes = localStorage.getItem('tareas');
    setTareas(tareasExistentes ? JSON.parse(tareasExistentes) : []);
  }, [])

  // Manejador de eventos
  function addTarea(event) {
    event.preventDefault();
    const next = [...tareas, tareaTexto.current.value];
    tareaTexto.current.value = "";
    setTareas(next);
    localStorage.setItem('tareas', JSON.stringify(next));
  }

  // Bloque de renderizado
  return (

```

```

<div className="App">
  <ul>
    {tareas.map(tarea => (<li key={tarea}>{tarea}</li>))}
  </ul>

  <form onSubmit={addTarea}>
    <input ref={tareaTexto} />
    <input type="submit" value="Añadir Tarea" />
  </form>
</div>
)
}

// Exportar Módulo
export default App;

```

## Desglosando

### Módulos Importados

```

import './App.css';
import { useState, useRef, useEffect } from 'react';


```

- Importamos el `App.css` predeterminado que viene con Create React App
- Importamos varios Hooks de `react`
  - `useState`, para manejar estados.
  - `useRef`, para crear un "almacén" dinámico.
  - `useEffect`, para inicializar la aplicación.

### Hooks

#### Referencia de la API de los Hooks - React

Los Hooks son una nueva incorporación en React 16.8. Te permiten usar estado y otras características de React sin escribir una clase. Esta página describe las API para los Hooks incorporados en React. Si los Hooks son nuevos para ti, es posible

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



Primero usamos `useState` para crear un estado y una función que lo modifique.

El estado se llama `tareas`, la función es `setTareas`. Lo inicializamos con un array vacío `[]`

```
const [tareass, setTareas] = useState([]);
```

Luego creamos un "almacén" llamado `tareaTexto` con `useRef`

```
const tareaTexto = useRef();
```

Ahora utilizamos el hook `useEffect` para ejecutar código. Hay algunas particularidades:

1. Pasamos una función para que se ejecute al renderizar el componente.
2. Pasamos una lista de las condiciones para que se ejecute la función. Como está vacía simplemente se ejecutará al crear por primera vez el elemento.

```
useEffect(() => {  
  const tareasExistentes = localStorage.getItem('tareas');  
  setTareas(tareasExistentes ? JSON.parse(tareasExistentes) : []);  
}, [])
```



LocalStorage es una forma "moderna" (tiene ya años) de guardar información en la memoria del navegador del usuario.

¿Como funciona?

`localStorage.setItem( <clave> , <valor> )` guardará en un diccionario lo que le pedimos, y `localStorage.getItem( <clave> )` recupera el valor

Como los valores que guardamos tienen que ser tipos "básicos" los estamos convirtiendo en un string (y viceversa) usando `JSON.stringify(<objeto>)` y `JSON.parse(<string>)`

Extraemos del LocalStorage un string guardado en la clave `'tareas'`

Fijamos el valor del estado tareas usando el operador ternario. Si el string extraído no está vacío (existía tareas en LocalStorage) pondremos su valor tras convertir el string a objetos con `JSON.parse`. En caso de que estuviese vacío usaremos un array vacío.

## Manejador de Eventos

Cuando se hace click en un botón queremos que se gestione el evento. Para ello escribimos una función que recibe un evento como parámetro.

```
function addTarea(event) {  
  event.preventDefault();  
  const next = [...tareas, tareaTexto.current.value];  
  tareaTexto.current.value = "";  
  setTareas(next);  
  localStorage.setItem('tareas', JSON.stringify(next));  
}
```

Por orden, estas líneas realizan las siguientes tareas:

```
event.preventDefault();
```

Evita que se realice la acción predeterminada de pulsar el botón submit del formulario (enviar el formulario)

```
const next = [...tareas, tareaTexto.current.value];
```

Añade, de forma moderna, un elemento al array de tareas y lo guarda en next.

- `...tareas` desestructura el array `tareas` en sus valores separados por comas.
- `tareaTexto.current.value` es el valor actualmente almacenado en `tareaTexto`, lo usamos en el formulario más adelante.
- Ambos elementos están rodeados de corchetes `[]`, de forma que se crea un nuevo array.

```
tareaTexto.current.value = "";
```

Reiniciamos el valor de `tareaTexto` a un string vacío para borrar la entrada en el formulario.

```
setTareas(next);
```

Guardamos en el estado el nuevo array que incorpora las tareas que había antes y la nueva.

```
localStorage.setItem('tareas', JSON.stringify(next));
```

Guardamos en la clave `'tareas'` en el LocalStorage el resultado de convertir el array contenido en el estado tareas en un string con `Json.stringify`.

## Bloque de renderizado

```
return (  
  <div className="App">  
    <ul>  
      {tareas.map(tarea => (<li key={tarea}>{tarea}</li>))}  
    </ul>  
  
    <form onSubmit={addTarea}>  
      <input ref={tareaTexto} />  
      <input type="submit" value="Añadir Tarea" />  
    </form>  
  </div>  
)
```

- Creamos un `<div>` con el nombre de clase "App"
- Creamos un elemento `<ul>` y dentro usamos `map` para convertir cada elemento del array tareas en un elemento `<li>`
- Creamos un `<form>`. Establecemos como acción de submit ejecutar `addTarea`
- Creamos un `<input>` y usamos ref para enlazarlo con el almacén `tareaTexto`
- Creamos un `<input>` de tipo submit con el value "Añadir Tarea"

## Conceptos 2

### Componentes Clase

Otra forma de definir un componente es como una clase de Javascript ES6.

Por ejemplo, un interruptor podría ser:

```
import {Component} from 'react'

class Toggle extends Component {
  constructor(props) {
    //Ejecuta el constructor de Component con el argumento props
    super(props);
    //El estado es un objeto. Solo hay un estado.
    this.state = {isToggleOn: true}
  }
  //Usar esta sintaxis nos evita ciertos problemas con this
  handleClick = () => {
    //Usamos una función flecha para no mutar el estado.
    this.setState(prevState => ({
      isToggleOn: !prevState.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

export default Toggle;
```

```
import Toggle from './Components/Toggle';

function App() {
  return (
    <div className="App">
      <Toggle />
    </div>
  );
}

export default App;
```

Los componentes clase tienen tanto un objeto props (las propiedades que definamos) como la posibilidad de tener un state local, que contiene el estado del componente.

Cada componente solo puede ver su propio estado, no puede ver el estado de sus componentes padres. Podemos pasar el estado o parte del estado como propiedad a otros componentes.

## Métodos de ciclo de vida

Existen métodos especiales que se llaman antes de pintar el componente por primera vez, o cuando va a ser eliminado.

Por ejemplo, creamos un contador:

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

class Clock extends Component {
```

```

constructor(props) {
  super(props);
  this.state = { date: new Date() }
}

componentDidMount() {
  this.timerID = setInterval(
    () => this.tick(),
    1000
  );
}

componentWillUnmount() {
  clearInterval(this.timerID);
}

tick() {
  this.setState({
    date: new Date()
  });
}

render() {
  return (
    <div>
      Son las {this.state.date.toLocaleTimeString()}
    </div>
  )
}
}

export default Clock;

```

`componentDidMount()` se ejecutará cada vez que un nuevo `Clock` aparezca en la página web. Aprovechamos la oportunidad para registrar un nuevo intervalo, con duración 1 segundo, que ejecutará el método `tick`.

`componentWillUnmount()` se ejecutará justo antes de que el `Clock` se borre. Aprovechamos para hacer limpieza y eliminar el intervalo para evitar problemas y errores en la consola.

## Composición

Una buena estrategia al crear componentes es que sean pequeños y re-utilizables. Nuestros componentes pueden contener otros componentes más pequeños por ejemplo:

## Productos

Filtrar:		
NOMBRE	PRECIO	STOCK
Pepper - Paprika, Hungarian	€2,25	10
Bread - Sour Batard	€8,65	30
Gatorade - Cool Blue Raspberry	€2,47	62
Chips Potato Swt Chili Sour	€1,22	2
Beef - Tenderloin Tails	€5,54	39
Straws - Cocktale	€0,94	19
Cheese Cloth No 100	€0,24	95
Curry Paste - Madras	€9,02	82
Wine - Magnotta - Pinot Gris Sr	€1,03	14
Mushroom - Chanterelle, Dry	€2,85	90
Sauce - Caesar Dressing	€6,14	6
Gin - Gilbey's London, Dry	€6,26	32
Toamtoes 6x7 Select	€2,30	21
Wine - Jaboulet Cotes Du Rhone	€9,17	56
Onions - Green	€0,85	11
Muffin Batt - Ban Dream Zero	€6,96	29
Nantucket - Pomegranate Pear	€4,81	86
Silicone Parch. 16.3x24.3	€1,21	26
Bread - Wheat Baguette	€1,73	33
Clams - Bay	€9,29	68
Snapple Lemon Tea	€8,76	60
Corn Shoots	€7,76	65
Soap - Pine Sol Floor Cleaner	€4,27	81
Apricots - Dried	€3,91	8
Celery	€9,33	62
Milk - 2% 250 Ml	€1,96	51
Veal - Heart	€9,15	11
Lettuce - Belgian Endive	€7,76	80
Tart Shells - Sweet, 3	€2,64	94
Bar Energy Chocchip	€1,29	9
Bread - Pullman, Sliced	€0,97	49
Lamb - Bones	€0,86	85
Plastic Arrow Stir Stick	€9,35	51
Mustard - Individual Pkg	€8,14	100
Beans - Yellow	€2,11	73
Energy Drink	€6,52	4
Basil - Primerba, Paste	€4,99	33
Pastry - Banana Muffin - Mini	€5,36	92
Tortillas - Flour, 12	€4,87	70
Oven Mitt - 13 Inch	€0,40	72
Flour - Rye	€9,82	67
Salami - Genova	€8,24	11
Chocolate - Pistoles, Lactee, Milk	€8,10	92
Wine - Port Late Bottled Vintage	€3,21	35
Vegetable - Base	€7,92	5
Spice - Onion Powder Granulated	€3,80	76
Duck - Fat	€4,27	17
Cheese - St. Paulin	€7,77	33
Banana	€4,37	1
Pear - Halves	€3,68	98

Tabla de productos con posibilidad de filtrar por nombre

Para diseñar este componente en React lo dividimos en sub-componentes:

- App
  - H1
  - FiltrableProductTable
    - SearchBar
    - ProductTable
      - Product


El código completo de esta aplicación está en el siguiente repositorio de GitHub.

GitHub - cmo7/ejemplo-tabla

This project was bootstrapped with Create React App. In the project directory, you can run: Runs the app in the development mode. Open <http://localhost:3000> to view it in the browser. The page will reload if you make edits. You will also see any

<https://github.com/cmo7/ejemplo-tabla>

**cmo7/ejemplo-tabla**



Contributor: 1, Issues: 0, Stars: 0, Forks: 0

## Componente FilterableProductTable

```
const FilterableProductTable = ({ products }) => {
  const [query, setQuery] = useState("");
  const searchKey = Object.keys(products[0])[0];
  return (
    <div className="filtrable-product-table">
      <SearchBar
        callback={setQuery} />
    </div>
  )
}
```



```

    <div className="header row">
      {Object.keys(products[0]).map(x => <div> {x.toUpperCase()} </div>)}
    </div>
    <ProductTable
      products={query
        ? products.filter(p => p[searchKey].includes(query))
        : products} />
  </div>
)
}

```

- Este componente recibe en props solo un array de productos.
- Las props están deconstruidas `{ products }`
- El componente es funcional pero usa el Hook `useState` para crear un estado `"query"` con su función set correspondiente.
- En `searchKey` guardamos el nombre del campo que usaremos para filtrar los resultados.
- El componente `SearchBar` recibe una función callback para poder usarla para modificar el valor de `query` con su input.
- Dentro de un div con la clase `header` pintamos los nombres de todas las claves de uno de los objetos del array. Se podrían pintar los nombres "a mano" también.
- El componente `product table` recibe un array de productos.
  - Si `query` es undefined, null o cadena vacía, le pasamos el array `products` intacto.
  - En cualquier otro caso pasamos un array `products` filtrado, donde el campo `searchKey` tiene que contener el `query`.

## Componente SearchBar

```

const SearchBar = ({ callback }) => {
  const queryValue = useRef();

  const handleOnChange = () => {
    callback(queryValue.current.value);
  }

  return (
    <div className="search-bar">
      <label htmlFor="query">
        Filtrar:
      </label>
      <input
        id="query"
        ref={queryValue}
        onChange={handleOnChange}
      />
    </div>
  )
}

```

- El componente `SearchBar` utiliza un ref (creado con `useRef`) para almacenar el input que contiene en su render.

- El componente contiene una función `handleOnChange` para gestionar el cambio del input. Esta función simplemente utiliza la función callback recibida en props (los props están deconstruidos) pasando el valor actual del ref como parámetro.
- En cuanto a renderizado, simplemente pinta un div con un label y un input. El input tiene asignados el ref y el `onChange` correspondientes.

## Componente ProductTable

```
const ProductTable = ({ products }) => {
  return (
    <div>
      {products.map(product => <Product key={product.name} data={product} />)}
    </div>
  )
}
```

- Recibe un array de productos crea un componente `Product` para cada uno.

## Componente Product

```
const Product = ({ data }) => {
  return (
    <div className="product row">
      {Object.values(data).map(x => <div> {x} </div>)}
    </div>
  )
}
```

- Recibe unos datos en forma de objeto. Convierte los valores del objeto en un array y pinta cada uno en un div.

## App.css

```
.row {
  display: grid;
  grid-template-columns: 3fr 1fr 1fr;
}

.search-bar {
  display: flex;
  justify-content: space-between;
  margin: auto;
}

.search-bar input {
  width: 80%;
  margin-left: 25px;
}

.header {
  border-bottom: 2px solid black;
  padding-bottom: 5px;
  margin-bottom: 5px;
}

.filterable-product-table {
  margin: auto;
  padding: 15px;
}
```

```
width: 800px;
border: 1px solid black;
}

h1 {
  text-align: center;
}
```

Simplemente las clases necesarias para mostrar la aplicación como aparece en los ejemplos.

## Usando Axios

GitHub - axios/axios: Promise based HTTP client for the browser and node.js

Promise based HTTP client for the browser and node.js New axios docs website: click here Make XMLHttpRequests from the browser Make http requests from node.js Supports the Promise API Intercept request and response Transform request and

 <https://github.com/axios/axios>

axios/axios

Promise based HTTP client for the browser and node.js



357 Contributors 7m Used by 24 Discussions 95k Stars 10k Forks

Para poder hacer llamadas a APIs podemos utilizar la librería Axios. Para instalar Axios navegamos con la consola a la carpeta de nuestro proyecto y utilizamos el comando:

```
npm install axios
```

De forma que en nuestras dependencias en `package.json` aparecerá:

```
"dependencies": {
  "@testing-library/jest-dom": "^5.11.4",
  "@testing-library/react": "^11.1.0",
  "@testing-library/user-event": "^12.1.10",
  "axios": "^0.21.4",
  "react": "^17.0.2",
  "react-dom": "^17.0.2",
  "react-scripts": "4.0.3",
  "web-vitals": "^1.0.1"
},
```

Para utilizar Axios en nuestra aplicación tenemos que importar el módulo:

```
import './App.css';
import axios from 'axios';
```

El lugar ideal para obtener información de una API es en el hook `useEffect`:

```
import './App.css';
import axios from 'axios';
import { useEffect, useState } from 'react';

function App() {

  const [blogInfo, setBlogInfo] = useState([]);
  const [blogPosts, setBlogPosts] = useState([]);
```

```

useEffect(() => {
  const consultaAPI = async () => {
    const posts = await axios.get('https://apitest.nohaywebs.com/wp-json/wp/v2/posts');
    const info = await axios.get('https://apitest.nohaywebs.com/wp-json/');
    setBlogPosts(posts.data);
    setBlogInfo(info.data);
  }
  consultaAPI();
}, [])

return (
  <div className="App">

    </div>
);
}

export default App;

```

Si lanzamos esta aplicación con `npm start` podremos ver una página en blanco, pero en el inspector podremos ver los valores en el estado:



A partir de aquí podemos escribir componentes y usarlos para mostrar nuestro blog a partir de los datos obtenidos.

## Usando React-Bootstrap

### React-Bootstrap

The most popular front-end framework, rebuilt for React.

<https://react-bootstrap.github.io/>

En resumen una vez realicemos el proceso de instalación podemos importar componentes a nuestra aplicación con:

```

import Button from 'react-bootstrap/Button';
import Card from 'react-bootstrap/Card';

```

