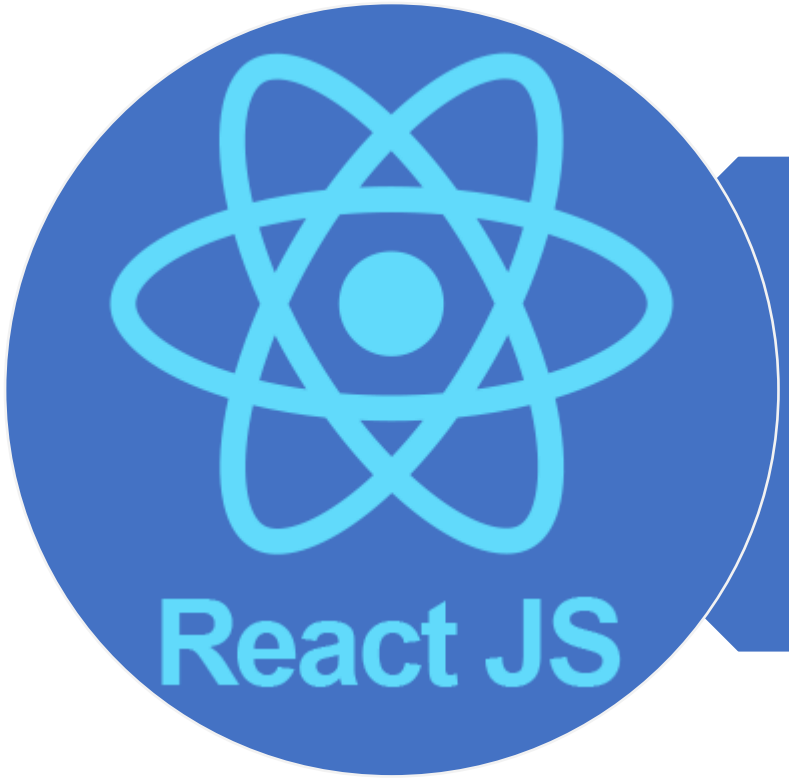
Jorge García Flores

Desarrollo Web en Entorno Cliente

Tema 21



React

Eventos en React

React maneja los eventos de forma parecida a como la hace JS de forma nativa, aunque hay sutiles diferencias.

Todos los eventos de JS también están disponibles en *React*, pero en *React* los eventos se escriben en *Camel Case*, además, siempre empiezan por la palabra "*on*", como: *onClick*, *onChange*, *onDrop*… y al igual que en JS pueden recibir como argumento el objeto *Event*.

A diferencia de JS y HTML, donde se coloca el nombre de la función en un *string* (entrecomillado), en *React* (JSX) se utiliza el nombre de la función dentro de una expresión (entre llaves *{}*).

Por ejemplo:

En HTML:

```
<button onclick="aumentar()">
   Pulsame!
</button>
```

En *React* (JSX):

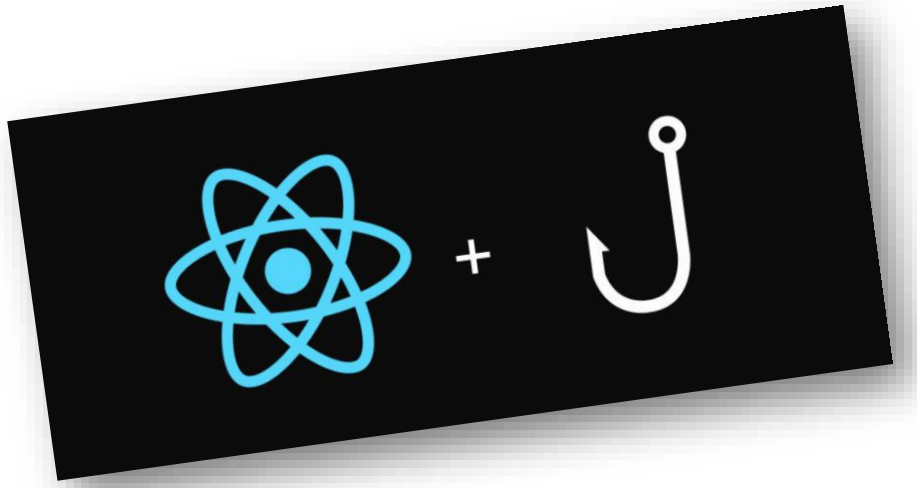
```
<button onClick={aumentar()}>
   Pulsame!
</button>
```

React Hooks

React tiene una API muy sencilla que permite crear todo tipo de aplicaciones haciendo uso de los *React Hooks*, llamados comúnmente *Hooks*.

Los *Hooks* son funciones que permiten, por ejemplo, "enganchar" el estado y el ciclo de vida desde los *functional components* de *React.*

La API de *React* dispone de *Hooks*, que pueden agruparse en *Hooks* básicos, Hooks adicionales y *Hooks* personalizados (*custom Hooks*) creados por el usuario con el fin de crear código reutilizable utilizando las ventajas que aporta *React*.



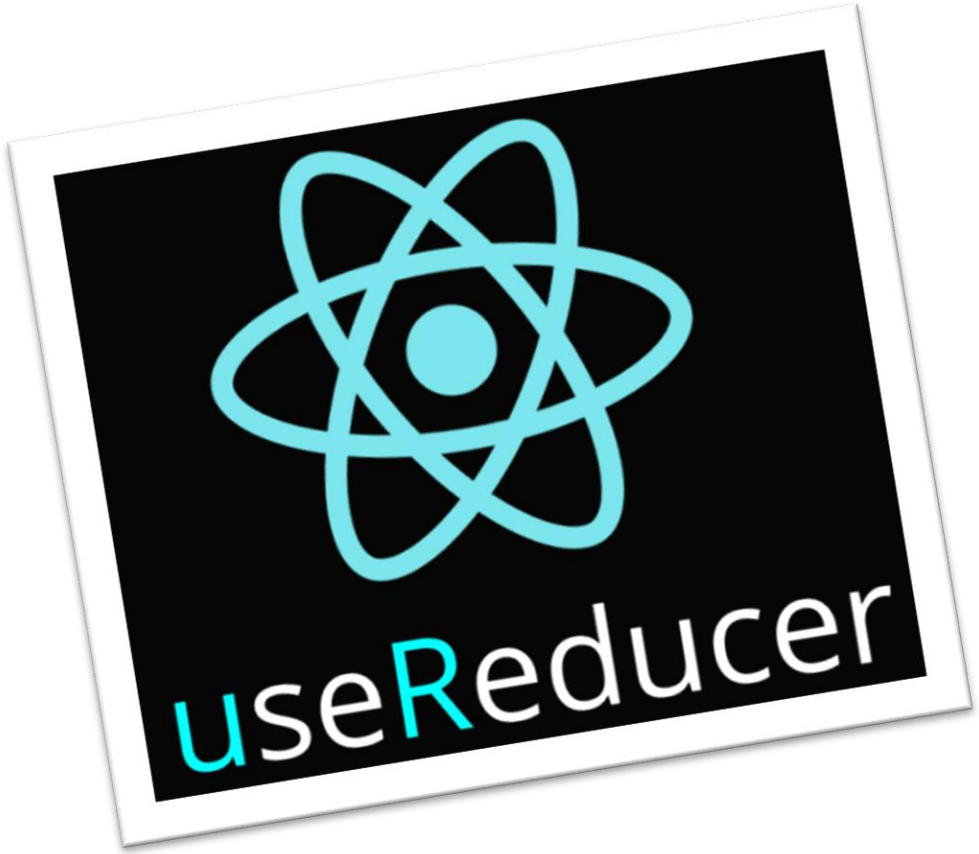
Los *Hooks* básicos, se usan con mucha frecuencia en las aplicaciones, son:

- ❑ `useState`
- ❑ `useEffect`
- ❑ `useContext`

React Hooks

Los *Hooks* adicionales, se usan con mucha menos frecuencia o casi ninguna en las aplicaciones, son:

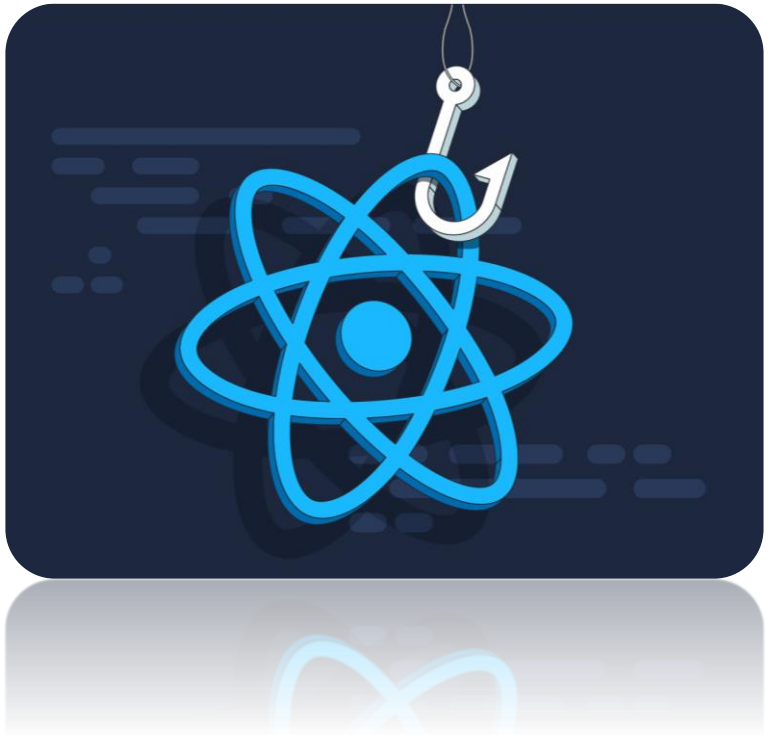
- ❑ *useReducer*
- ❑ *useCallback*
- ❑ *useMemo*
- ❑ *useRef*
- ❑ *useImperativeHandle*
- ❑ *useLayoutEffect*
- ❑ *useDebugValue*



Reglas de los Hooks

Para poder usar un *Hook* hay que importar la función del *Hook* definida en el módulo de *React*:

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



Los *Hooks* deben cumplir lo siguiente:

- No se deben de colocar dentro de condicionales, pero sí puede utilizarse el operador ternario.
- Deben aparecer antes de cualquier `return` del componente y antes de otras funciones que use el componente.
- Un componente siempre tiene que manejar o utilizar la misma cantidad de *Hooks*.

Por ello, es muy recomendable ubicarlos en la parte superior, de los componentes.

useState

El estado (*state*) contiene datos que pueden cambiar en el tiempo.

En *React*, el estado se usa para controlar los cambios en la interfaz. Cada vez que cambia el estado el componente se vuelve a dibujar (renderizar).

Para crear un estado en *React* se usa el Hook `useState`. Esta función devuelve un array de dos elementos, el primero es una referencia al valor del estado y el segundo es un *callback* (una función) para poder cambiar el valor del estado.

El estado solo debe poder cambiarse mediante la función *callback* por lo que no debe utilizarse el operador asignación (=).

La función `useState` recibe como argumento el valor inicial del estado que se está definiendo, este valor puede ser cualquier tipo: *number*, *string*, *array*…

useState

En el ejemplo de la derecha se crea un estado, con un valor inicial *0*, y se asigna a una referencia a la variable *count*.

Este estado se lee y se muestra mediante un párrafo de HTML, esto sucede cada vez que se renderiza el componente, hecho que ocurre cada vez que cambia el estado.

```
import { useState } from 'react'

function Counter() {
 const [count, setCount] = useState(0)
 return (
   <div>
     <p>Contador: {count}</p>
     <button onClick={()=>setCount(count++)}>
       Aumentar
     </button>
   </div>
)}
```

También se renderiza un botón, que al hacer clic sobre él dispara una función para cambiar el valor del estado asignado a la variable *count*, incrementándolo en *1*, mediante la función *setCount* devuelta por *useState* al crear este Hook.

Props

El estado o las funciones que se crean dentro de los componentes solo son accesibles para ese componente.

Una forma de evitar duplicar el código y reutilizar las variables, el estado o las funciones de unos componentes a otros es mediante el uso de las "*props*" o propiedades.

Las *props* son la colección de datos que un componente recibe de un componente padre, y que pueden usarse para definir los elementos de *React* que retornará el componente.

Las *props* se pasan del componente padre al componente hijo, nunca se pueden pasar del hijo al padre.

En términos prácticos, si un componente necesita recibir información o datos para funcionar, las recibe mediante las *props*.

Props

Además, las *props* tienen ciertas características:

- ❑ Una *prop* no se modifica: son inmutables, por lo que no se puede modificar o cambiar.
- ❑ Pueden tener un valor por defecto.
- ❑ Pueden marcarse como obligatorias, cuando un componente no puede funcionar sin recibir una *prop*.

En JSX, las *props* se ven como los atributos de los elementos HTML:

```
<Saludo value = {nombre} />
```

En los componentes, las *props* se reciben como argumentos de la función:

```
const Saludo = ({nombre}) => {
  return (
    <h2>Hola {nombre}</h2>
)}
```

Props

Se pueden pasar muchos tipos de datos a un componente mediante las *props*:

```
<Header
 clientes = {nombre}
 admin = {false}
 setCarrito = {setCarrito}
 titulo = "Tienda Virtual"
/>
```

Como no se puede pasar *props* de “hijos a padres” directamente, si, por ejemplo, se tiene un estado que se debe pasar por diferentes componentes es mejor crearlo en el archivo principal, aquel que tiene el componente padre. Así, cada nivel superior jerárquico de componentes podrá pasar la *prop* hacia sus componentes hijos, y estos a su vez, a los suyos.

Existen tecnologías como *Redux* o el *Hook* `useContext` que evitan tener que hacerlo de esta forma.

Redux es una herramienta creada en 2015 para la gestión de estado en aplicaciones de JS y, aunque suele asociarse a *React*, es una librería *framework* agnósticas (f*ramework agnostic*).

Las librerías agnósticas al *framework* son conjuntos de componentes y utilidades que se desarrollan de manera independiente de cualquier *framework*, por lo que están diseñadas para poder ser compatibles con muchos de ellos.

Props

También se pueden pasar funciones de un componente a otro mediante las *props*.

En su componente padre se crea la función y se envía a su componente hijo *Header* mediante una *prop*:

```
function App() {
  const saludarFn: Function = ()=>{return <h1>Saludo</h1>}
  return (
    <Header saludar={saludarFn}/>
)}
```

En su componente hijo *Header* se lee la *prop*, en este caso una función que se manda llamar:

```
const Header= ({saludarFn}:any) => {
  return (
    <>
        {saludarFn()}
    </>
)}
```

Props

Por ejemplo, para poder actualizar el estado de un componente padre desde un componente hijo, se pasa mediante *prop* la función que cambia el estado, al invocarla desde el componente hijo, el estado del padre cambiará.

En el siguiente ejemplo el componente *Formulario* envía a su componente hijo *Error* la función *setError* que permite cambiar su estado mediante una *prop*.

```
const Formulario = ()=> {
  const [error, setError] = useState(false);
  return ({error && <Error
   setError={setError}
   mensaje="Todos los campos son obligatorios"/>})
}
```

En el componente *Error* se cambia el estado de su componente *Formulario* padre invocando dicha función.

```
const Error = ({mensaje, setError}:Props) => {
    setTimeout(() => {setError(false);}, 5000);
    return (
    <div><p>{mensaje}</p></div>
)}
```

Children

Hay otra forma de pasar *props* a los diferentes componentes mediante *children*.

children es una *prop* especial que se pasa a los componentes. Es un objeto que contiene los elementos que envuelve un componente. *children* es una palabra reservada en *React*.

Por ejemplo, si se tiene un componente llamado *Card* que muestra una tarjeta con un título y un contenido, se puede usar la *prop children* para mostrar el contenido.

En el componente padre:

```
<Card title="Título de la tarjeta">
   <p>Contenido de la tarjeta</p>
</Card>
```

En el componente *Card*:

```
function Card(props) {
  return (
    <div className="card">
      <h2>{props.title}</h2>
      <div>{props.children}</div>
    </div>
)}
```

Children

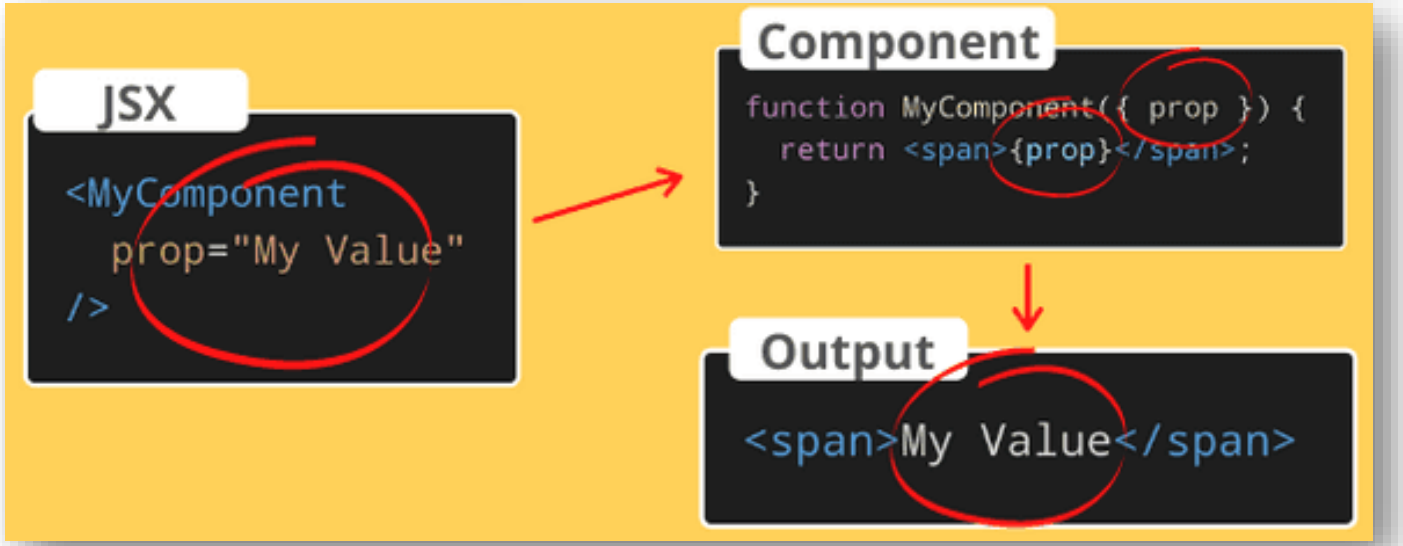
Otro ejemplo basado en un ejemplo previo de uso de *props* pero en esta ocasión usando *children*:

```
const Formulario = ()=> {
  const [error, setError] = useState(false);
  return ({error && <Error
   setError={setError}>Todos los campos son obligatorios</Error>})
}
```

```
const Error = ({children, setError}:Props) => {
    setTimeout(() => {setError(false)}, 5000);
    return (
    <div><p>{children}</p></div>
)}
```

Props vs Estado

- Las *props* son un objeto que se pasa como argumento de un componente padre a un componente hijo.
- Las *props* son inmutables y no se pueden modificar desde el componente hijo.



- El estado (*state*) es un valor que se define dentro de un componente.
- El valor del estado es inmutable (no se puede modificar directamente) pero se puede establecer un valor nuevo del estado para que *React* vuelva a renderizar el componente.
- Tanto *props* como estado afectan al renderizado del componente, pero su gestión es diferente.

useEffect

El *Hook* `useEffect` permite realizar "efectos secundarios" en los componentes, como, por ejemplo, procesar los datos cuando llegan del servidor, actualizar directamente el DOM, manejar temporizadores…

Para poder usar `useEffect` hay que importarlo del paquete de *React* como cualquier otro Hook:

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

`useEffect` acepta dos argumentos, el primero es una función; y el segundo, llamado dependencias, es opcional, y es un array de componentes.

La función se ejecutará cuando alguna de las dependencias (los componentes) cambie de estado o se renderice (el componente esté listo).

Si el array de dependencias es un array vacío la función se ejecutará una sola vez, cuando el componente donde está definido el `useEffect` está listo o disponible.

useEffect

useEffect o devuelve *undefined* o devuelve una función que se utiliza para realizar tareas de limpieza cuando el componente se desmonta o cuando cambian las dependencias.

Una vez definida esta función de limpieza, en el próximo "efecto" esta función se ejecutará antes que la propia función definida como primer argumento.

Normalmente el primer argumento de *useEffect* es un *arrow function*, un *callback.*

Al ejecutarse automáticamente cuando un componente está listo, es en su función *callback* el lugar ideal para colocar código de consulta a una API o al *Storage*.

Es un *Hook* muy utilizado para ejecutar funciones cuando un componente está listo, ya que si se ejecutan funciones sin que se haya cargado un componente se pueden tener comportamientos no deseados e imprevistos.

useEffect

En este ejemplo, se utiliza *useEffect* para realizar la solicitud a la API cuando el componente está listo.

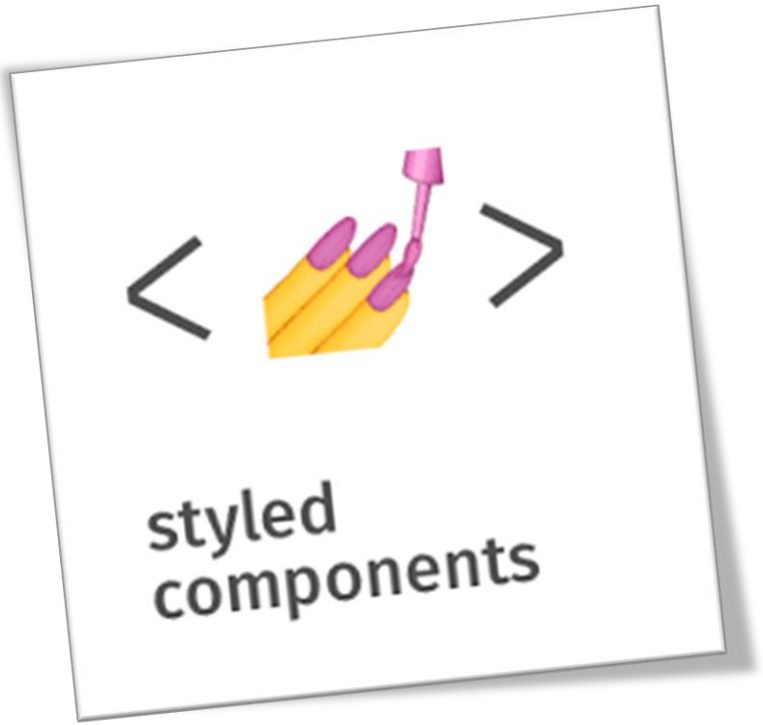
La función de limpieza no es necesaria porque no hay suscripciones o temporizadores que necesiten ser limpiados.

```
function MiComponente()
{
  const [data, setData] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch('https://api.example.com/data');
        const result = await response.json();
        setData(result);
      } catch (error) { console.error('Error fetching data:', error);}
    };
    fetchData();}, []);

   return (<div>
            {data ? (<p>{data}</p>) : (<p>Cargando...</p>)}
           </div>);
}
```

Styled Components



Existen muchas formas de utilizar estilos en React: importando una hoja estilos, usando una librería o *framework* de estilos como *Tailwind CSS*,…

JSX permite utilizar código HTML con expresiones de JavaScript, pero también es posible, utilizando una librería adicional, crear un elemento HTML con la sintaxis de un componente y añadirle propiedades CSS.

De esta manera, en lugar de tener una hoja de estilos o una librería externa, como *Bootstrap*, se puede escribir el código CSS en cada componente, llamado CSS en JS (CSS-in-JS).

Al dejar de utilizar un componente se elimina su código CSS también por lo que se mantiene una hoja estilo final con las clases que realmente usa la aplicación, haciendo el mantenimiento de la aplicación más sencilla.

Styled Components

Para usar *Styled Components* hay que instalar mediante Node.JS el paquete `styled-components`:

```
npm install styled-components
```

Emotion

No obstante, existe una biblioteca llamada *Emotion* que permite escribir estilos CSS con JS y se integra muy bien con *React*.

Emotion se ha vuelto muy popular en el ecosistema de *React*.

Al igual que *Styled Components*, *Emotion* permite escribir estilos directamente en los componentes de *React*, pero destaca por su enfoque en el rendimiento y su capacidad de ser *framework* agnóstico.

Emotion dispone de dos métodos principales, el primero es independiente del *framework* (*Framework agnostic*) y el segundo es para usarla con *React*.

Para usar la versión de *Emotion* de *React* hay que instalar el paquete `@emotion/react`:

```
npm install @emotion/react
```

Styled Components

Además, *Emotion* cuenta con un paquete llamado *@emotion/styled* que facilita aún más la creación de estilos al integrarlo directamente en el componente.

```
import { css } from '@emotion/react'

const color = 'white'

render(
  <div
    css={css`
      padding: 32px;
      background-color: hotpink;
      font-size: 24px;
      border-radius: 4px;
      &:hover {
        color: ${color};
      }
    `}
  >
    Colócate encima.
  </div>
)
```

```
import styled from '@emotion/styled'

const Button = styled.button`
  padding: 32px;
  background-color: hotpink;
  font-size: 24px;
  border-radius: 4px;
  color: black;
  font-weight: bold;
  &:hover {
    color: white;
  }
`

render(<Button>
Mi componente botón.
</Button>)
```

Styled Components

Para usar esta API de *Emotion*, además de instalar *@emotion/react*, hay que instalar este paquete *@emotion/styled*:

```
npm install @emotion/styled
```

Para usar los *Styled Components* se hay que importar *styled* de *@emotion/styled* y crear el componente que debe ser nombrado en *Pascal Case*.

Un *Styled Component* se crea poniendo la palabra *styled* seguida de un punto **.** y una etiqueta HTML válida que se utilizará para el componente y, justo seguido, se coloca un *Template String* con las propiedades CSS que se quieren asignar a dicho componente.

Y ya se puede utilizar el componente creado como cualquier otra etiqueta HTML:

```
import styled from '@emotion/styled'

const Heading = styled.h1`
  font-family: 'Poppins', sans-
serif;
  color: #FFF;`

function App() {
  return (
    <Heading>
    Hola Mundo
    </Heading>)}

export default App
```

Custom Hooks

React permite crear *Hooks* personalizados, conocidos como *Custom Hooks*, con el fin de poder aprovechar el potencial de los *Hooks* y, de igual manera, poder reutilizarlos en nuevos proyectos o en otros partes de la aplicación.

Hay que recordad que un Hook no es más que una función proporcionada por la API de *React*. Así, estos *Hooks* personalizados, como cualquier otro Hook de React, pueden tener estado, manteniendo la función persistente, y hacer uso de las *props* para comunicarse con otros componentes.

Las ventajas de los Hook personalizados es que tienen todas las mejorías de *React* como son el estado, los *effects*, integrar otros *Hooks* y además del excelente rendimiento de los *Hooks* de *React*.

Los *Hooks* personalizados, siguiendo la nomenclatura de *React*, se nombran con *Camel Case*, empezando con la palabra `use`.

Custom Hooks

Los *Hooks*, a diferencia de los componentes, no devuelven código JSX, normalmente devuelven un array o un objeto literal.

El siguiente ejemplo, es un *Hook* personalizado, muy sencillo, que devuelve un array que contiene una función llamada *saludar*, cuando es invocada la función *saludar* muestra por consola el nombre que se le pasa como argumento.

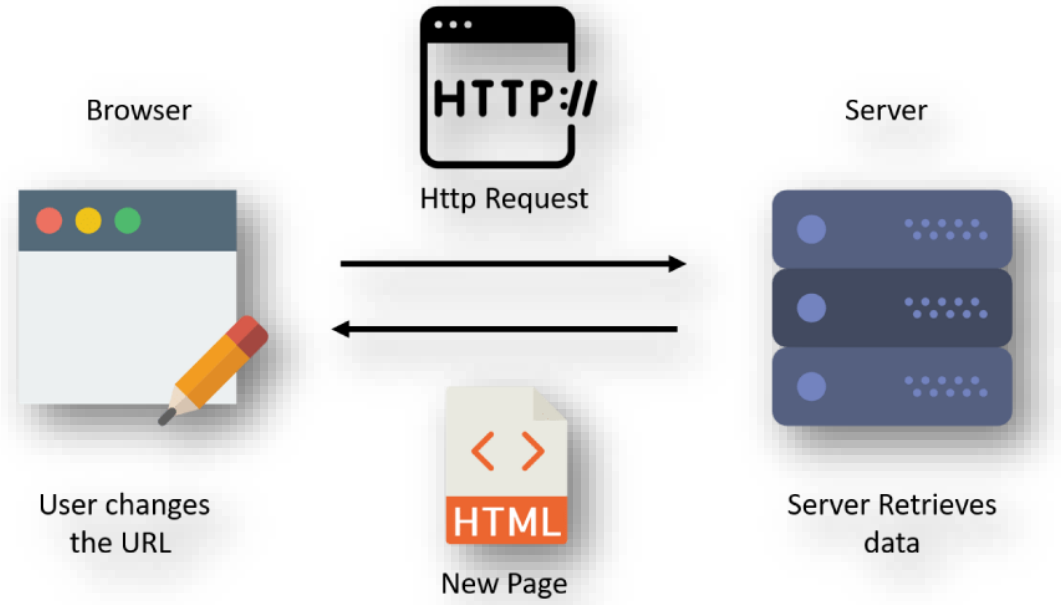
```
export const useSimpleHook = (inicial="") => {
  const [nombre, setNombre] = useState(inicial);
  useEffect(() => {
    console.log(`Hola ${nombre}`)}, [nombre]);

    const saludar = (nombre) => {
      setNombre(nombre);
    }

  return [saludar];
}
```

Este Hook, llamado *useSimpleHook*, hace uso de los Hooks *useState* y *useEffect* de *React*, además, puede ser inicializado con valor inicial.

Routing

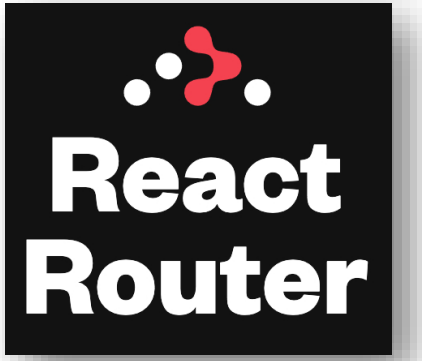


Usando únicamente *React*, se pueden construir aplicaciones SPA (*Single-Page Application*) es una aplicación web o un sitio web que cabe en una sola página con el propósito de dar una experiencia más fluida a los usuarios, como si fuera una aplicación de escritorio.

Para utilizar más de una página, sobre todo en proyectos grandes sea necesario utilizar diferentes páginas, por lo que se suele utilizar un sistema de enrutado, pero *React* como librería, a diferencia de un *framework*, no incluye, de manera oficial un sistema de enrutado de aplicaciones.

Con una librería de enrutado se puede tener diferentes *URLs*, mostrar diferentes componentes, restringir el acceso a ciertas páginas…

Routing



Para utilizar sistema de enrutado se debe usar una biblioteca externa, como *React Router*, o usar un *framework*, como *Next.js*, que incluya esta característica.

Así, *Next.js* se puede considerar un *framework* de *React* porque incluye React, un sistema de enrutado, un sistema de renderizado del lado del servidor, etc.

Existen muchas librerías de enrutado como:

- ❑ *React Router* o *React Router DOM*
- ❑ *React Location*
- ❑ …

Y muchos *frameworks* que incluyen *routing* y *React*, como:

- ❑ *Next.js*
- ❑ *Remix Run*
- ❑ *Gatsby*
- ❑ *Astro*
- ❑ *Hydrogen*
- ❑ ….

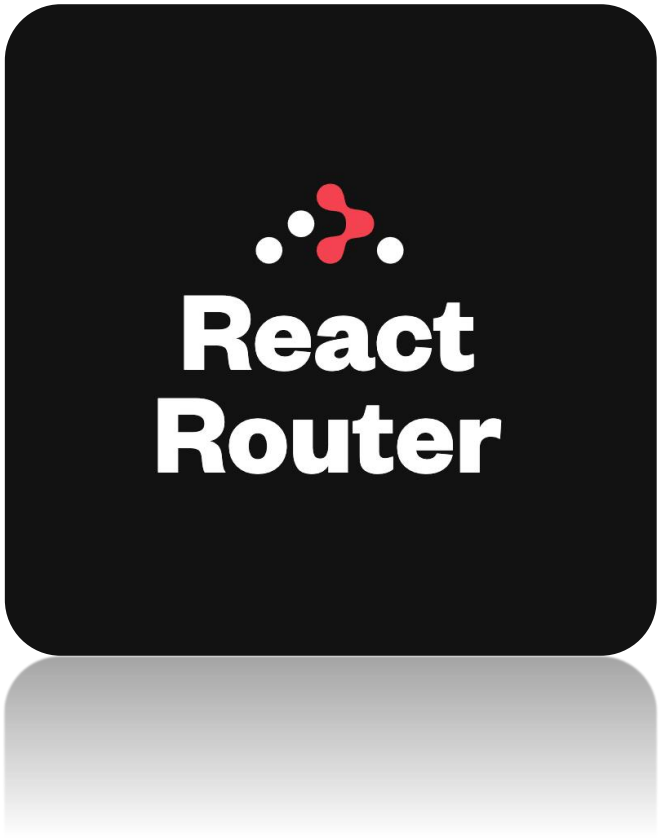
React Router

React Router es una librería para crear aplicaciones con *routing* (diferentes *URLs*) lo que favorece un mejor código, más organizado, mejor reutilización de componentes…

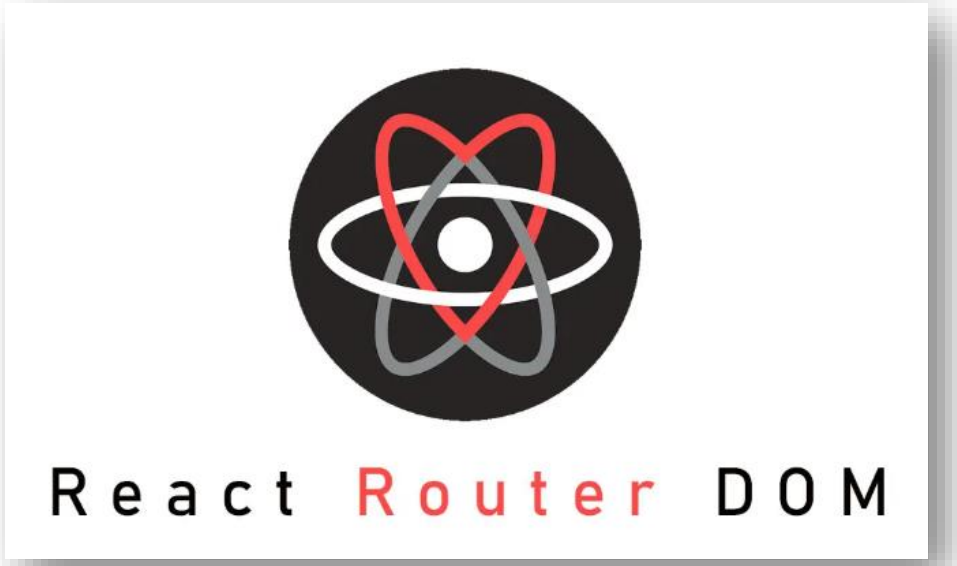
En la actualidad coexisten dos versiones mayores: la v.5, versión *legacy*, y la v.6.4, a partir de esta última versión prácticamente se convierte en un *framework* con manejo de rutas, peticiones HTTP, formularios y datos.

React Router, al contrario de otros enrutadores en los que la declaración de las rutas de la aplicación es estática, usa un enrutamiento dinámico.

Las rutas estáticas se declaran al iniciar la aplicación, cuando se ejecuta por primera vez la aplicación y no pueden cambiarse en tiempo de ejecución; en cambio, las rutas dinámicas pueden cambiarse durante la ejecución de la aplicación.



React Router



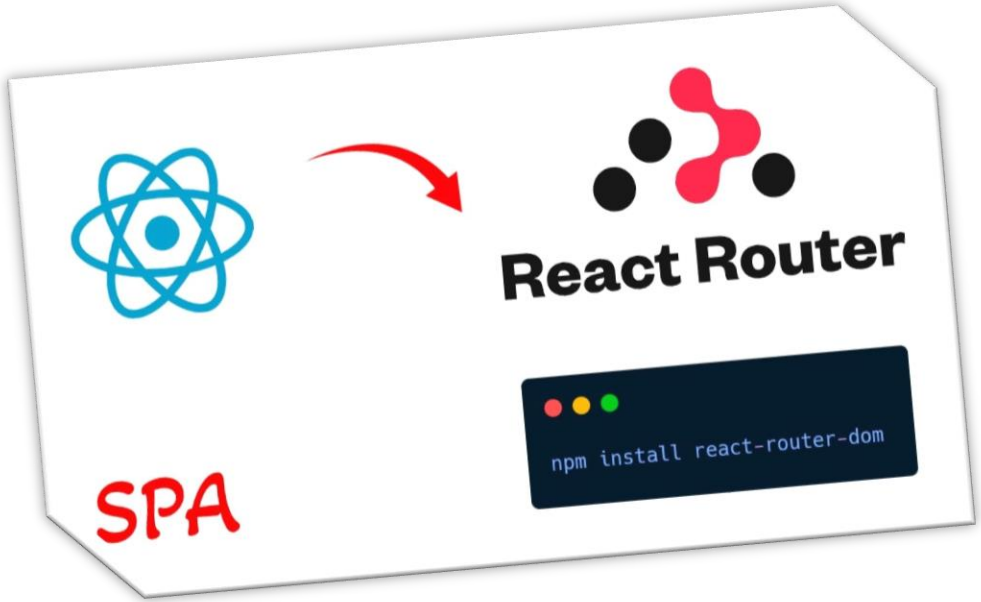
React Router consta de dos paquetes:

- `react-router-dom`, anteriormente coexistía otro paquete llamado `react-router` pero en la actualidad este integrado en este, permite realizar el a navegación en aplicaciones de una sola página (SPA) de manera declarativa, facilitando la transición entre diferentes vistas o componentes en función de la *URL*. Está optimizado para su uso en entornos web, ya que utiliza la API del DOM para gestionar la navegación basada en el navegador.

- `react-router-native` diseñado específicamente para aplicaciones *React Native*, un *framework* que permite desarrollar aplicaciones móviles utilizando JS y *React*.

React Router DOM

Los elementos fundamentales de *react-router-dom* son los siguientes:



- ***Componentes de enrutamiento***: son componentes de *React* que permiten definir las rutas de la aplicación.

- ***Parámetros de ruta***: permiten incluir parámetros dinámicos en las rutas para facilitar la creación de rutas dinámicas. Por ejemplo, */users/:id* podría representar una ruta para mostrar detalles de un usuario específico.

- ***Navegación***: son los componentes *Link* y *NavLink* que actúan como enlaces de navegación a diferentes rutas, similares a la etiqueta *<a>* de HTML, pero con la ventaja de que cambia la URL del navegador, pero renderizando únicamente los componentes afectados.

- ***Historial***: permite gestionar de manera automática el historial del navegador, con el fin de que se puedan realizar acciones como retroceder o avanzar en la navegación de la aplicación.

React Router DOM

Las rutas se pueden anidar para estructurar la aplicación de manera jerárquica y modular.

Para crear una ruta se utilizan, principalmente, los siguientes componentes:

- *`Router`*: Conecta la aplicación a la URL del navegador, manteniendo la interfaz de usuario en sincronía con la URL del navegador mediante la API *History* de HTML5.

- *`Routes`*: Genera un árbol de rutas, que permite reemplazar cada componente con el que coincide con la URL de la barra de navegación.

- *`Route`*: Es el componente fundamental que se utiliza para definir cómo se renderizará un componente cuando la URL coincida con cierta ruta. Se corresponde con cada una de las rutas en el árbol de rutas, y necesita, para representar correctamente una ruta, al menos, que se hayan definido las propiedades *`path`* y *`element`*.

- *`Switch`*: Similar a *`Route`*, se utiliza para envolver varias rutas y garantizar que solo se renderice la primera coincidencia.

React Router DOM

Para utilizar *React Router DOM* hay que instalar el paquete *react-router-dom*:

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

E importar los componentes que se necesiten, como, por ejemplo, *createBrowserRouter* y *RouterProvider*:

```
import
{createBrowserRouter,RouterProvider}
from 'react-router-dom'
```

React Router DOM

createBrowserRouter es un enrutador, que permite definir las rutas por medio de objetos.

Es realmente una función que recibe como argumento un array donde hay definido un objeto para cada una de las rutas de la aplicación con las propiedades:

- *path* que contiene las diferentes rutas de la aplicación.
- *element* es código JSX o el componente que será mostrado.

```
const router = createBrowserRouter
([
  {
    path: '/',
    element: <h1>Inicio</h1>
  },
  {
    path: '/login',
    element: <Formulario />
  }
])
```

React Router DOM

RouterProvider es una función, que actúa como el componente de entrada de la aplicación y desde él fluyen los datos y las acciones al resto de componentes.

Esta función necesita conocer las diferentes rutas de la aplicación, por lo que recibe un enrutador, por ejemplo, el creado por la función *createBrowserRouter*, mediante la *prop* llamada *route*:

```
ReactDOM.createRoot(
document.getElementById('root')).render(
  <React.StrictMode>
    <RouterProvider router={router}/>
  </React.StrictMode>
)
```

Un *helper* (ayudante) es una función que realiza tareas específicas y comunes en una aplicación, diseñadas para ser reutilizables y facilitar el código, ayudando a realizar operaciones comunes de una manera más sencilla. A menudo se utilizan para encapsular lógica común que no está directamente relacionada con la interfaz de usuario, pero que es necesaria para realizar ciertas operaciones.

Por ejemplo, el siguiente código muestra un *helper* personalizado que calcula el precio total, incluyendo el IVA, de productos:

```
function calculaTotal(precio=1, iva=21)
{
  return (precio * iva/100) + precio;
}
```

Outlet

Outlet es un componente de *React Router DOM* que se utiliza para representar los elementos HTML o componentes que deben aparecer en todas las rutas, evitando tener que escribir o reimportar el mismo componente en cada una de las rutas secundarias.

Actúa como un *layout*, como un contenedor dinámico, haciendo que el resto de los componentes se aniden dentro de él, y consiguiendo que la aplicación tenga la misma apariencia en todas las rutas.

El componente *Outlet* se define en la ruta principal, y justo donde aparezca *<Outlet />* se inyectará el código JSX de los demás componentes.

Como es habitual, para poder utilizar el componente *Outlet* hay que importarlo de *React Router DOM*:

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

Outlet

Para que afecte a las rutas secundarias hay que añadir una propiedad, llamada *children* en el enrutador definido por *createBrowserRouter*.

children es un array de objetos, que se corresponde con las rutas secundarias donde se quiere se inyecte su código JSX justo donde aparece el componente *<Outlet/>*.

En el siguiente ejemplo, cuando se acceda a la ruta */login* se mostrará el elemento *h1* y justo debajo el *h2*:

```
const router = createBrowserRouter ([{
    path: '/',
    element: <Layout />,
    children: [{
        path: '/login',
        element: <h2>Login</h2>,
      }]
  }])
```

```
import { Outlet } from 'react-router-dom'

export const Layout = () => {
  return (
    <>
      <h1>Cartelera de Películas</h1>
      <Outlet/>
    </>
)}
```

Outlet

Dentro del array de objetos rutas definido por *children* se puede crear un objeto con la propiedad *index: true* para indicar que esa ruta hace referencia a la página de inicio.

De esta manera se puede crear la vista o *layout* de la página raíz o de inicio, accedida por */*:

```
const router = createBrowserRouter ([{
    path: '/',
    element: <Layout />,
    children: [
      {
        index: true,
        element: <Inicio />
      },
      {
        path: '/login',
        element: <h2>Login</h2>,
      }]
  }])
```

Router

Otra manera mucho más frecuente de definir las rutas es utilizando una estructura anidada mediante los componentes *BrowserRouter*, *Routes*, *Route* de *React Router DOM*.

El componente *BrowserRouter* define el *router* y anida dentro de él las diferentes rutas mediante una entrada *Routes* y una entrada *Route* por cada ruta.

```
const Router = () => {
  return (
    <BrowserRouter basename="/jorge/">
      <Routes>
        <Route path='/' element={<Layout/>}>
          <Route index  element={<Inicio/>}  />
          <Route path='buscador' element={<Buscador/>}/>
          <Route path='insertar' element={<Insertar/>}/>
        </Route>
      </Routes>
    </BrowserRouter>
)};

export default Router;
```

```
ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
      <Router />
  </React.StrictMode>)
```

Router

BrowserRouter puede tener la propiedad *basename* que indica la ruta o carpeta donde se ha desplegado la aplicación en el servidor cuando no se despliega en un directorio raíz.

Las rutas y enlaces de la aplicación serán relativos a la ruta indicada por *basename*. Por ejemplo:

```
<BrowserRouter basename="/jorge/">
   <Routes>
     <Route path='/' element={<Layout/>}>
        <Route index  element={<Inicio/>}  />
        <Route path='buscador' element={<Buscador/>}/>
        <Route path='insertar' element={<Insertar/>}/>
     </Route>
   </Routes>
</BrowserRouter>
```

Se corresponde con:

/jorge o /jorge

/jorge o /jorge

/jorge/buscador

/jorge/insertar

Además, cuando se aniden rutas las rutas hijas inyectarán su código en el elemento *Outlet* de *React Router DOM* definido en el elemento de *React* de la ruta padre.

Así, en el ejemplo, se inyectará el elemento de *React* *<Buscador/>* en la etiqueta *<Outlet/>* de *<Layout/>* cuando se acceda a la ruta */jorge/buscador*

Route

Cada entrada *Route* debe tener al menos las dos propiedades siguientes:

- ❑ *path*: especifica en qué ruta de la aplicación se encuentra localizada una ruta determinada.

- ❑ *element*: que especifica el elemento de *React* que será renderizado cuando se acceda a dicha ruta:

```
<Route path='insertar' element={<Insertar/>}
```

En lugar de la propiedad *element* puede usarse la propiedad *Component* cuando no se dispone de un elemento, en este caso *React Router Dom* creará el elemento *React* de manera automática:

```
<Route path='/*' Component={()=>{ return (
  <p style={{color: "white",
             fontFamily: "'Poppins', sans-serif",
             textAlign: "center",
             fontWeight: 700,
             fontSize: "34px"}}>
    Ha ocurrido un error inesperado.</p>)}}
/>
```

Route

En la mayoría de las ocasiones es necesario definir un *layout* común para todas los componentes de *React* en la aplicación, este se define en la ruta principal, en la ruta padre, y es utilizado por todas las rutas hijas, inyectando su código en el elemento *<Outlet/>* de *React Router DOM* como ya se ha explicado.

No obstante, también es necesario crear una página de inicio que utilice el *layout* definido pero que también haga uso de su propio elemento de *React*.

Para ello se crea una ruta con *Route* sin la propiedad *path* pero con la propiedad *index*, el elemento asociado a su propiedad *element* será renderizado cuando se acceda a la ruta base:

```
<Routes>
  <Route path='/' element={<Layout/>}>
     <Route index  element={<Inicio/>}  />
     <Route path='buscador' element={<Buscador/>}/>
     <Route path='insertar' element={<Insertar/>}/>
  </Route>
</Routes>
```

Link

Link es un elemento de *React Router DOM* que representa una etiqueta *<a>* HTML y permite al usuario navegar a otra URL haciendo clic sobre él. Es un elemento importante ya que realmente no se recarga la página entera solo los elementos y componentes afectados por la nueva ruta.

Link tiene asociado una propiedad *to* para indicar el enlace a la nueva ruta, este valor es una ruta relativa por lo que se resuelve en relación con la ruta principal. Este valor puede contener varios **..** indicando enlaces a rutas superiores en la jerarquía, actuando exactamente igual que la orden cd de línea de comandos.

También puede utilizarse la propiedad *reloadDocument* para omitir el enrutamiento del lado del cliente y dejar que el navegador maneje la transición normalmente, como si fuera un elemento *<a href>*.

```
import { Link } from "react-router-dom";

const Layout = () => {
  return (
    <nav>
      <Link to="inicio">Home</Link>
      <Link to="nosotros">About us</Link>
    </nav>
)};

export default Layout;
```

Bibliografía y recursos online

- https://es.react.dev/reference/react/hooks
- https://reactrouter.com/en/main
- https://styled-components.com/
- https://emotion.sh/
- https://www.npmjs.com/package/@emotion/react

Bibliografía y recursos online

- https://reactrouter.com/en/main/routers/create-browser-router
- https://reactrouter.com/en/main/components/outlet
- https://reactrouter.com/en/main/utils/create-routes-from-elements
- https://www.escuelafrontend.com/react-router-6