Docs_

Guia React para principiantes

Bienvenido a la guía de React para principiantes. Está diseñada para enseñarte todos los conceptos básicos de React, que necesitas conocer para empezar a construir aplicaciones React en 2021.

He creado este recurso para ofrecerte el camino más completo y fácil para que logres aprender React desde cero.

Al final tendrás conocimiento profundo de toneladas de conceptos esenciales de React, incluyendo:

- El porqué, el qué y el cómo de React.
- Cómo crear fácilmente aplicaciones en React.
- JSX y sintaxis básica.
- Elementos JSX.
- Componentes y Props.
- Eventos en React.
- Estado y gestión de estados.
- Los fundamentos de React Hooks.

Fundamentos de React

¿Qué es realmente React?

React se define oficialmente como una "biblioteca de JavaScript para crear interfaces de usuario", pero ¿qué significa eso realmente?

React es una biblioteca, hecha en JavaScript y que codificamos en JavaScript, para construir grandes aplicaciones que se ejecutan en la web.

¿Qué necesito saber para aprender React?

En otras palabras, es necesario tener una comprensión básica de JavaScript, para convertirse en un programador sólido de React.

Los conceptos más básicos de JavaScript, con los que deberías estar familiarizado son las variables, los tipos de datos básicos, los condicionales, los métodos de arreglo, las funciones y los módulos ES.

Si, React se hizo en JavaScript, ¿por qué no usamos simplemente JavaScript?

React fue escrito en JavaScript de abajo hacia arriba, con el propósito expreso de rapidez en la construcción de aplicaciones web y nos da herramientas para hacerlo.

JavaScript es un lenguaje con más de 20 años de antigüedad, el cual fue creado para añadir interacciones al navegador a través de scripts y no fue diseñado para crear aplicaciones completas.

En otras palabras, aunque se utilizó JavaScript para crear React, se crearon con fines muy diferentes.

¿Puedo utilizar JavaScript en aplicaciones React?

Sí, puedes incluir cualquier código JavaScript válido en tus aplicaciones React.

Puede utilizar cualquier API del navegador o de la ventana, como la geolocalización o la API fetch.

Además, como React (cuando esto se compila) se ejecuta en el navegador, puedes realizar acciones comunes de JavaScript como la consulta y manipulación del DOM.

Cómo crear aplicaciones React

Tres formas diferentes de crear una aplicación React

1. Poner React en un archivo HTML, con scripts externos.
2. Utilizar un entorno React en el navegador como CodeSandbox.
3. Crear una aplicación React en el ordenador usando una herramienta como Create React App.

¿Cuál es la mejor manera de crear una aplicación React?

¿Cuál es el mejor enfoque para usted? La mejor manera de crear tu aplicación depende de lo que quieras hacer con ella.

Si, quieres crear una aplicación web completa que quieres empujar en última instancia a la web, lo mejor es crear esa aplicación React en tu ordenador utilizando una herramienta como Create React App.

La forma más sencilla de crear y construir aplicaciones React para aprender y crear prototipos es usar una herramienta como CodeSandbox. ¡Puedes crear una nueva aplicación de React en segundos yendo a react.new!.

Elementos JSX

JSX es una potente herramienta para estructurar aplicaciones

JSX está pensado para facilitar la creación de interfaces de usuario con aplicaciones JavaScript. Toma prestada su sintaxis del lenguaje de programación más utilizado: JavaScript mezclado con HTML. Como resultado, JSX es una potente herramienta para estructurar nuestras aplicaciones.

El ejemplo siguiente, es el ejemplo más básico de un elemento React que muestra el texto "Hola Mundo":

```
<div>Hola React!</div>
```

Hola Mundo en React

Ten en cuenta que para ser mostrados en el navegador, los elementos de React necesitan ser **renderizados** (usando ReactDOM.render()).

En qué se diferencia JSX de HTML

Podemos escribir elementos HTML válidos en JSX, pero lo que difiere ligeramente es la forma en que se escriben algunos atributos.

Los atributos que constan de varias palabras, son escritos en la sintaxis **camelCase** (como className) y tienen nombres diferentes a los del HTML estándar (class).

```
<div id="cabecera">
  <h1 className="titulo">Hola React!</h1>
</div>
```

JSX

JSX tiene esta forma diferente de escribir atributos, porque en realidad se hace usando funciones de JavaScript (hablaremos de esto más adelante).

JSX debe tener una barra(/) al final si se compone de una etiqueta

A diferencia del HTML estándar, los elementos como input, img o br deben cerrarse con una barra oblicua final para que sean JSX válidos.

```
<input type="email" /> // <input type="email"> Esta sintaxis es un error
```

Error por no tener / al final

Los elementos JSX con dos etiquetas deben tener una etiqueta de cierre

Los elementos que deben tener dos etiquetas, como div, main o button, deben tener su cierre, la segunda etiqueta en JSX, de lo contrario se producirá un error de sintaxis.

```
<button>Clica aqui</button> // <button> or </button> es un error de sintaxis
```

Cómo se estilizan los elementos JSX

Los estilos en línea se escriben de forma diferente a los del HTML plano.

- Los estilos en línea no deben incluirse como una cadena de texto, sino dentro de un objeto.
- Una vez más, las propiedades de estilo que utilicemos deben estar escritas en estilo camelCase.

```
<h1 style={{ color: "blue", fontSize: 22, padding: "0.5em 1em" }}>
  Hola React!
</h1>;
```

Estilos en React

Las propiedades de estilo que aceptan valores en píxeles (como la anchura, la altura, el relleno, el margen, etc.), pueden utilizar enteros en lugar de cadenas de texto. Por ejemplo, fontSize: 22 en lugar de fontSize: "22px".

JSX puede mostrarse condicionalmente

Los nuevos desarrolladores de React pueden preguntarse cómo es de beneficioso que React pueda utilizar código JavaScript.

Un ejemplo sencillo que oculta o muestra condicionalmente el contenido JSX, es usar cualquier condicional JavaScript válido, como una sentencia if o switch.

```
const isAuthUser = true;

if (isAuthUser) {
  return <div>Hola usuario!</div>
} else {
  return <button>Ingresar</button>
}
```

Esta autenticado el usuario

¿Dónde devolvemos este código? Dentro de un componente React, que cubriremos en una sección posterior.

JSX no puede ser entendido por el navegador

Como se ha mencionado anteriormente, JSX no es HTML, sino que se compone de funciones de JavaScript.

De hecho, escribir Hola React en JSX no es más que una forma más cómoda y comprensible de escribir código como el siguiente:

```
React.createElement("div", null, "Hola React!")
```

JSX elements

Ambos trozos de código tendrán la misma salida de "Hola React".

Para escribir JSX y que el navegador entienda esta sintaxis diferente, debemos utilizar un **transpilador** para convertir JSX a estas llamadas de función.
El transpilador más común se llama **Babel**.

Componentes de React

¿Qué son los componentes de React?

En lugar de limitarse a renderizar uno u otro conjunto de elementos JSX, podemos incluirlos dentro de **componentes** React.
Los componentes se crean utilizando lo que parece una función normal de JavaScript, pero es diferente porque devuelve elementos JSX.

```
function Saludos() {
  return <div>Hola React!</div>;
}
```

Componente React

¿Por qué utilizar componentes React?

Los componentes React nos permiten crear lógicas y estructuras más complejas dentro de nuestra aplicación de React, más de lo que haríamos con elementos JSX solos.

Piensa en los componentes React como nuestros elementos React personalizados, que tienen su propia funcionalidad.

Como sabemos, las funciones nos permiten crear nuestra propia funcionalidad y reutilizarla donde queramos en nuestra aplicación.

Los componentes son reutilizables donde queramos en nuestra aplicación y tantas veces como queramos.

Los componentes no son funciones normales de JavaScript

¿Cómo podemos mostrar el JSX devuelto por el componente anterior?

```
import React from 'react';
import ReactDOM from 'react-dom';

function Saludos() {
  return <div>Hola React!</div>;
}

ReactDOM.render(<Saludos />, document.getElementById("root"));
```

Renderización componente Saludos

Usamos la importación de React para parsear el JSX y ReactDOM para renderizar nuestro componente en un **elemento raíz** con el id de "root".

¿Qué pueden retornar los componentes de React?

Los componentes pueden devolver elementos JSX válidos, así como cadenas de texto, números, booleanos, valores null, arreglos y fragmentos.
¿Por qué debemos retornar null? Es común retornar null, si queremos que un componente no muestre nada.

```
function Saludos() {
  if (isAuthUser) {
    return "Hola otra vez!";
  } else {
    return null;
  }
}
```

JavaScript

Otra regla es que los elementos JSX deben estar envueltos en un elemento padre. No se pueden devolver múltiples elementos hermanos.

Si, necesitas devolver varios elementos, pero no necesitas añadir otro elemento al DOM (normalmente para un condicional), puedes utilizar un componente especial de React llamado fragmento(fragment).

Los fragmentos se pueden escribir como <></> o cuando importas React en tu archivo puedes crear un fragmento con: <React.Fragment></React.Fragment>.

```
function Saludos() {
  const isAuthUser = true;

  if (isAuthUser) {
    return (
      <>
        <h1>Hola otra vez!</h1>
        <button>Salir</button>
      </>
    );
  } else {
    return null;
  }
}
```

Componente en React

Tenga en cuenta que cuando se intenta devolver un número de elementos **JSX** que se extienden a lo largo de **varias líneas**, podemos devolverlo todo utilizando un conjunto de paréntesis () como se ve en el ejemplo anterior.

Los componentes pueden devolver otros componentes

El hecho más importante, es que los componentes pueden devolver otros componentes.

A continuación se muestra un ejemplo básico de una aplicación React contenida con en un componente llamado App que devuelve múltiples componentes:

```
import React from 'react';
import ReactDOM from 'react-dom';

import Layout from './components/Layout';
import Navbar from './components/Navbar';
import Aside from './components/Aside';
import Main from './components/Main';
import Footer from './components/Footer';

function App() {
 return (
  <Layout>
   <Navbar />
   <Main />
   <Aside />
   <Footer />
  </Layout>
 );
}

ReactDOM.render(<App />, document.getElementById('root'));
```

Componentes dentro del principal

Esto es poderoso porque estamos usando la personalización de los componentes para describir lo que son (Ej: Layout) y su función en nuestra aplicación. Esto nos dice cómo deben ser utilizados con sólo mirar su nombre.

Además, estamos utilizando el poder de JSX para componer estos componentes. En otras palabras, utilizar la sintaxis de JSX similar a la de HTML para estructurarlos de una manera inmediatamente comprensible (como si la barra de navegación estuviera en la parte superior de la aplicación, el pie de página en la parte inferior, y así sucesivamente).

Se puede utilizar JavaScript en JSX utilizando claves

Al igual que podemos utilizar variables de JavaScript dentro de nuestros componentes, también podemos utilizarlas directamente dentro de nuestro JSX.

Sin embargo, hay algunas reglas básicas para usar valores dinámicos dentro de JSX:

- JSX puede aceptar cualquier valor primitivo (cadenas de texto, booleanos, números), pero no aceptará objetos planos.
- JSX también puede incluir expresiones que resuelvan estos valores.
- Por ejemplo, se pueden incluir condicionales dentro de JSX utilizando el operador ternario, ya que resuelve a un valor.

```
function Saludos() {
 const isAuthUser = true;

 return <div>{isAuthUser ? "Hola!" : null}</div>;
}
```

Props en React

Los componentes y variables, estados pueden ser pasados por valores usando props

Los datos que se pasan a los componentes en JavaScript se llaman **props**.
Los props tienen un aspecto idéntico al de los atributos de los elementos JSX/HTML normales, pero se puede acceder a sus valores dentro del propio componente.

Los props están disponibles en los parámetros del componente al que se le pasan. Los props siempre se incluyen como propiedades de un objeto.

```
ReactDOM.render(
 <Greeting usuario="John!" />,
 document.getElementById("root")
);

function Saludos(props) {
 return <h1>Hola {props.usuario}</h1>;
}
```

Componente Saludos

Props no deben ser cambiados directamente

Los props nunca deben ser modificados directamente dentro del componente hijo.
Otra forma de decir esto es que los props nunca deben ser **mutados**, ya que los props son un objeto JavaScript simple.

```
// No podemos modificar el objeto props:
function Header(props) {
 props.usuario = "Doug";
 return <h1>Hello {props.usuario}</h1>;
}
```

Modificar props

Los componentes se consideran funciones puras. Es decir, para cada entrada, deberíamos poder esperar la misma salida. Esto significa que no podemos mutar el objeto props, sólo leer de él.

Props especiales: prop children

Los props **children** son utiles si queremos pasar elementos / componentes como props a otros componentes.
Los props children son especialmente utiles, para cuando se quiere que el mismo componente (como un componente Layout) envuelva a todos los demás componentes.

```
function Layout(props) {
 return <div className="container">{props.children}</div>;
}
function IndexPage() {
 return (
  <Layout>
   <Header />
   <Hero />
   <Footer />
  </Layout>
 );
}
function AboutP() {
 return (
  <Layout>
   <About />
   <Footer />
  </Layout>
 );
}
```

Componente Layout, envolviendo

La ventaja de este patrón es que todos los estilos aplicados al componente Layout serán compartidos con sus componentes hijos.

Listas y claves en React

Cómo iterar sobre arreglos en JSX usando map

¿Cómo mostramos listas en JSX usando datos de arreglos? Utilizamos la función .map() para convertir listas de datos (arreglos) en listas de elementos.

```
const personas = ["John", "Bob", "Fred"];
const listaPersonas = personas.map((persona) => <p>{persona}</p>);
```

Iterar arreglos en React

Puede utilizar .map() para los componentes, asimismo para los elementos JSX simples.

```
function App() {
 const personas = ["John", "Bob", "Fred"];

 return (
  <ul>
   {personas.map((persona) => (
    <Persona name={persona} />
   ))}
  </ul>
 );
}

function Persona({ nombre }) {
 // Accedemos a la prop 'nombre' directamente usando la desestructuración de objetos
 return <p>This person's name is: {nombre}</p>;
}
```

Componente con desestructuración de objeto props

La importancia de key en las listas

Cada elemento React dentro de una lista de elementos necesita un **key prop** especial.
Las **claves** son esenciales, para que React pueda hacer un seguimiento de cada elemento sobre el que se está iterando la función .map().

React utiliza **claves** (**keys)** para actualizar elementos individuales, cuando sus datos cambian (en lugar de volver a renderizar toda la lista).

Las **claves** deben tener valores únicos, para poder identificar cada una de ellas según su valor clave.

```
function App() {
 const personas = [
  { id: "Ksy7py", nombre: "John" },
  { id: "6eAdl9", nombre: "Bob" },
  { id: "6eAdl9", nombre: "Fred" },
 ];

 return (
  <ul>
   {personas.map((persona) => (
    <Person key={persona.id} nombre={persona.nombre} />
   ))}
  </ul>
 );
}
```

key prop in una lista de componentes

Estados y manejo de datos en React

¿Qué es el estado en React?

El **estado** es un concepto que se refiere a cómo cambian los datos de nuestra aplicación a lo largo del tiempo.
La importancia de los estados en React, es que es una forma de hablar de nuestros datos por separado de la interfaz de usuario (lo que el usuario ve)..

Hablamos del manejo de estados, porque necesitamos una forma eficaz de mantener un registro y actualizar los datos de nuestros componentes a medida que nuestro usuario interactúa con ellos.

Para cambiar nuestra aplicación de elementos HTML estáticos a una dinámica con la que el usuario pueda interactuar, necesitamos estados.

Ejemplos de cómo utilizar el estado en React

Necesitamos gestionar el estado a menudo, cuando nuestro usuario quiere interactuar con nuestra aplicación.

Cuando un usuario escribe en un formulario, mantenemos el estado del formulario en ese componente.

Cuando obtenemos datos de una API para mostrarlos al usuario (como las entradas de un blog), necesitamos guardar esos datos en estados.

Cuando queremos cambiar los datos que un componente recibe de props, usamos el estado para cambiarlo en lugar de mutar el objeto props.

Introducción a los hooks de React con useState

La forma de "crear" estado en React dentro de un componente particular es con el hook useState. ¿Qué es un hook? Es muy parecido a una función de JavaScript, pero sólo se puede utilizar en un componente de función de React en la parte superior del componente.

Utilizamos hooks para hacer ciertas características y useState nos da la capacidad de crear y gestionar el estado.
useState es un ejemplo de un React hook, que proviene directamente de la biblioteca de React:React.useState.

```
import React from 'react';

function Saludos() {
 const estado = React.useState("Hola React");

 return <div>{estado[0]}</div> // Muestra "Hola React"
}
```

Hola React

¿Cómo funciona useState? Como una función normal, podemos pasarle un valor inicial (como "Hola React").

Lo que se devuelve de useState es un arreglo. Para acceder a la variable state y su valor, podemos utilizar el primer valor de ese arreglo: estado[0].

Sin embargo, hay una manera de mejorar la forma en que escribimos esto. Podemos utilizar la desestructuración de arreglos para acceder directamente a esta variable de estado y llamarla como queramos, por ejemplo, título.

```
import React from 'react';

function Saludo() {
 const [titulo] = React.useState("Hola React");

 return <div>{titulo}</div> // Muestra "Hola React"
}
```


¿Qué pasa si queremos permitir que nuestro usuario actualice el saludo que ve? Si incluimos un formulario, el usuario puede escribir un nuevo valor. Sin embargo, necesitamos una forma de actualizar el valor inicial de nuestro título.

```
import React from "react";

function Saludo() {
 const [titulo] = React.useState("Hola React");

 return (
  <div>
   <h1>{titulo}</h1>
   <input placeholder="Actualizar titulo" />
  </div>
 );
}
```

Texto por default en el input
Podemos hacerlo con la ayuda del segundo elemento del **arreglo** que devuelve **useState**. Es una función setter, a la que podemos pasar el valor que queramos que sea el nuevo estado.
En nuestro caso, queremos obtener el valor que se escribe en el input cuando un usuario está en el proceso de escribir. Podemos obtenerlo con la ayuda de los **eventos** de React.

¿Qué son los eventos en React?

Los eventos son formas de obtener datos sobre una determinada acción que un usuario ha realizado en nuestra app.

Los props más comunes utilizados para manejar eventos son onClick (para eventos de click), onChange (cuando un usuario escribe en una entrada) y onSubmit (cuando se envía un formulario).
Los datos de los eventos se nos dan conectando una función a cada uno de estos puntales enumerados (Cabe anotar que hay muchos más para elegir que estos tres).

Para obtener datos sobre el evento cuando nuestra entrada es cambiada, podemos añadir onChange en la entrada y conectarlo a una función que manejará el evento. Esta función se llamará handleInputChange:

```
import React from "react";

function Saludo() {
 const [titulo] = React.useState("Hola React");

 function handleInputChange(event) {
  console.log("input cambiado!", event);
 }

 return (
  <div>
   <h1>{titulo}</h1>
   <input
          placeholder="Actualización titulo"
          onChange={handleInputChange}
           />
  </div>
 );
}
```

Evento onChange

Tenga en cuenta que en el código anterior, registrará un nuevo evento en la consola del navegador, cada vez que el usuario escriba en la entrada.

Los datos del evento se nos proporcionan como un objeto con muchas propiedades que dependen del tipo de evento.

Cómo actualizar el estado en React con useState

Para actualizar el estado con useState, podemos utilizar el segundo elemento que useState nos devuelve de su matriz.

Este elemento es una función que nos permitirá actualizar el valor de la variable de estado (el primer elemento). Lo que pasemos a esta función **setter** cuando la llamemos modificara el estado.

```
import React from "react";

function Saludo() {
 const [titulo, setTitulo] = React.useState("Hola React");

 function handleInputChange(event) {
  setTitle(event.target.value);
 }

 return (
  <div>
   <h1>{titulo}</h1>
   <input
          placeholder="Actualizar titulo"
          onChange={handleInputChange}
           />
  </div>
 );
}
```

Actualizar input

Usando el código anterior, lo que el usuario escriba en el input (el texto viene de event.target.value) será puesto en estado usando setTitulo y mostrado dentro del elemento h1.

Lo que es especial sobre el estado y por qué debe ser gestionado con un hook dedicado como useState, es porque una actualización del estado (como cuando llamamos a setTitle) provoca un re-renderizado.

Un re-renderizado es cuando un determinado componente se renderiza o se muestra de nuevo basándose en los nuevos datos. Si nuestros componentes no se volvieran a renderizar cuando los datos cambian, ¡nunca veríamos que la apariencia de la aplicación cambia en absoluto!