# React

"A Javascript library for building user interfaces"

KEEPCODING

Tech School

# Introducción

- Desarrollada por Facebook, liberada en 2013
- Documentación: https://reactjs.org/
- Documentación (NEW): https://react.dev/
- **Librería** javascript para crear interfaces de usuario (SPA, frameworks, mobile…)
- **NO ES UN FRAMEWORK,** pero se ha creado un amplio ecosistema de librerías alrededor para resolver aquellos aspectos que React no resuelve (estado, routing, estilos, formularios…)

# Principios básicos

- **Declarativo:** se diseña la vista para cada estado, React actualiza de modo eficiente el DOM cuando cambia el estado
- **Basado en componentes:** favorece construir componentes encapsulados y reutilizables, que se unen para componer interfaces más complejas
- **Aprender una vez, escribir todo tipo de aplicaciones**
  - **react:** librería core con todo el núcleo de funcionalidad
  - **react-dom:** aplicaciones web (browser)
  - **react-native:** aplicaciones mobile

# Declarativo

Código

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

+

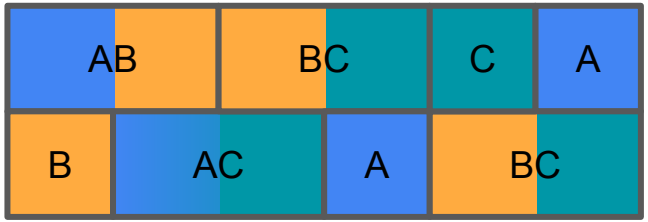
Datos

```
props = {
  name: 'KeepCoding',
};
```

# Basado en componentes

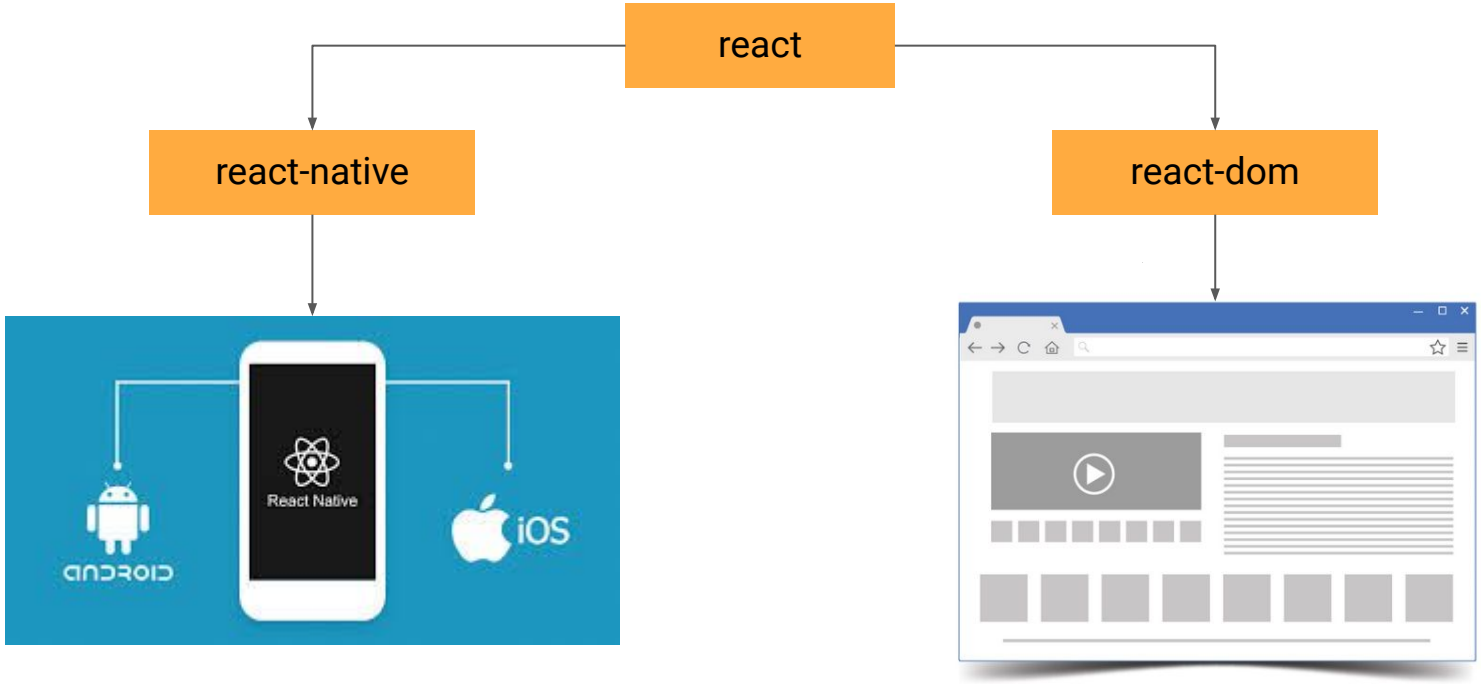

Componentes básicos


Componentes más complejos




Interfaces más complejas basadas en componentes

# Aprende de una vez



# React DOM: starters y frameworks

Podemos crear nuestra aplicación React de cero, pero implica manejo de varias herramientas, bundler (**webpack**), transpliador (**babel**), linter (**eslint**)... Lo mejor es usar alguna de estas herramientas.

- **Aplicaciones SPA (Single page application)**
  - Create React App
  - Vite JS React

- **Aplicaciones universales (Cliente Servidor, MPA)**
  - Next JS
  - Remix
  - Gatsby

# Elementos: createElement

**Elemento:** pieza más pequeña en aplicaciones React que describe un pedazo de interfaz

```
import { createElement } from 'react';
const element = createElement('div', {className: 'container'}, 'Hello, World!');
```

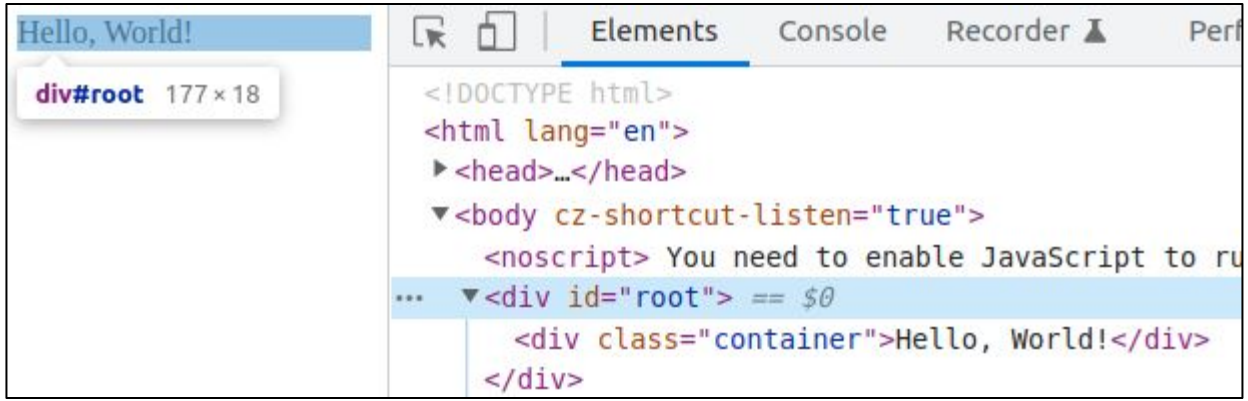
Un elemento es simplemente un objeto Javascript

# Elementos: renderizando en el DOM

**react-dom:** Renderiza el elemento en el DOM real (en el browser)

```
// necesitamos un elemento en nuestra página con id "root" (un div, por ejemplo)
// nuestro elemento se insertará en ese div
import { createRoot } from 'react-dom/client';
createRoot(document.getElementById('root')).render(element);
```



# Elementos: JSX

**JSX:** extensión Javascript con sintaxis tipo XML. Muy parecido al HTML, pero no es HTML, **es Javascript**

```
const element = <div className='container'>Hello, World!</div>;
```

Necesitamos un proceso de **transpilación (Babel),** que transforme el JSX en llamadas a **createElement** que el browser entienda

```
<div className='container'>
  Hello, World!
</div>
```

BABEL

```
createElement(
  'div',
  {className: 'container'},
  'Hello, World!'
)
```

# Elementos: JSX - Atributos

Podemos pasar atributos a los elementos React. Los nombres de atributos corresponden a los atributos HTML, en **camelCase.**

```
const element = <img src="picture.jpg" alt="picture.jpg" />;
```

- Excepciones: class **(className)**, for **(htmlFor)**

Hacemos el JSX más dinámico introduciendo expresiones Javascript dentro del JSX (entre llaves { })

```
const name = 'John Doe';
const className = 'container';
const element = <div className={className}>Hello, {name}!</div>;
```

# Elementos: JSX - Spread atributos

Si tenemos un objeto con los atributos, podemos pasarlo haciendo **spread** del objeto en el JSX

```
const element = <img src="picture.jpg" alt="picture.jpg" />;

// Podemos hacerlo así
const props = {
 src: 'picture.jpg',
 alt: 'picture.jpg',
};

const element = <img {...props} />;
```

Es muy útil cuando queremos “dejar pasar” atributos

# Elementos: children

**children:** atributo especial que tienen todos los elementos. Es “lo que va dentro del tag”. Se puede especificar de dos modos:

```
// estas dos líneas son equivalentes
<div className="container">Hello, World!</div>
<div className="container" children="Hello, World!" />

// lo que va dentro del tag sobreescribe al atributo children (Goodbye World)
<div className="container" children="Hello, World!">Goodbye World</div>

// elementos que no tienen children dentro del tag, pueden cerrar directamente
<img src="picture.jpg" alt="picture.jpg" />
<input value="firstname" maxLength={10} />
```

# Componentes: propiedades (*aka* props)

**Componente:** Función **pura** que recibe un objeto de propiedades (props) y devuelve un elemento React que puede ser renderizado

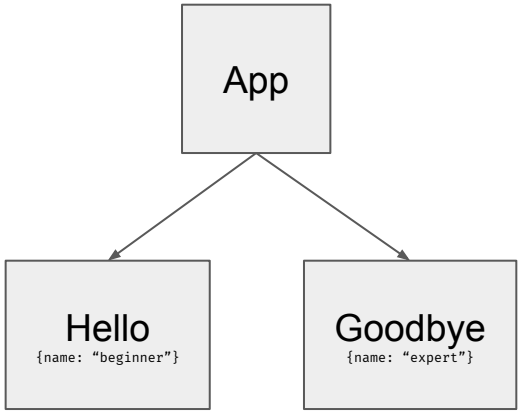
```
// se puede definir como una function clásica - Nombre empieza con MAYÚSCULA
function Welcome(props) {
 return <div>Hello, {props.name}</div>;
}
// o como una arrow function
const Welcome = (props) => <div>Hello, {props.name}</div>;
const Welcome = ({ name }) => <div>Hello, {name}</div>;

// podemos renderizar un componente con JSX, igual que los elementos HTML
// atributos === props
const element = <Welcome name="John" />;
```

⛔ **¡¡¡ Un componente NUNCA debe modificar las props que le llegan !!!**

# Componentes: anidando componentes

Podemos reutilizar nuestros componentes para crear nuevos componentes “más grandes”, creando una árbol jerárquico de padres/hijos tan complejo como necesitemos. Un padre puede **pasar props** a su(s) hijo(s)

```
const Hello = ({ name }) => <div>Hello, {name}</div>;
const Goodbye = ({ name }) => <div>Goodbye, {name}</div>;
const App = () => (
 <>
   <Hello name="beginner" />
   <Goodbye name="expert" />
 </>
);
ReactDOM.createRoot(
  document.getElementById('root')
).render(<App />);
```



# Fragments

Un componente no puede devolver varios elementos React a la vez (sería como tener varios retornos de **createElement**)
Podríamos envolver todo el retorno en un , pero podríamos romper los estilos, o la semántica del HTML
React nos ofrece el componente <Fragment />, o con (<>...</>)

```
import { Fragment } from 'react';
function Header() {
  return (
    <Fragment>
      <h1>Title</h1>
      <h2>Subtitle</h2>
    </Fragment>
  );
}
```

```
// con <>...</>
function Header() {
  return (
    <>
      <h1>Title</h1>
      <h2>Subtitle</h2>
    </>
  );
}
```

# Render condicional

Usamos Javascript para decidir si un elemento o componente renderiza.

```
// if else
let content;
if (firstTime) {
 content = <Hello />;
} else {
 content = <Goodbye />;
}
<div>{content}</div>;

// operador ternario
<div>
 {firstTime ? <Hello /> : <Goodbye />}
</div>
```

```
// operador lógico AND
// renderiza Hello si firstTime es
// “true”
<div>
 {firstTime && <Hello />}
</div>

// operador lógico OR
// renderiza Goodbye si firstTime es
// “false”
<div>
 {firstTime || <Goodbye />}
</div>
```

🔔 **React NO renderiza `true/false`, `“”`, `null`, `undefined`**

# Listas

Usamos Javascript para renderizar listas de elementos.

```
// tenemos un array con datos
const colors = [
 { name: 'Red', id: 1 },
 { name: 'Green', id: 2 },
 { name: 'Blue', id: 3 },
];
// lo transformamos en un “array” de elementos
<ul>
 {colors.map(color => (
   <li key={color.id}>{color.name}</li>
 ))}
</ul>
```

🔔 **No olvidar pasar un *key* única a cada elemento dentro de la lista**

# Eventos

Respondemos a eventos pasando funciones a los elementos.

```
const Button = () => {
 const handleClick = (event) => {
   alert('You clicked me!', event.target);
 }

 return <button onClick={handleClick}>Click me</button>;
}
```

La función recibe un *evento React* (aka synthetic event), que es un wrapper sobre el evento nativo del DOM, arreglando algunas inconsistencias cross-browser. react-event-object

🔔 **Pasar la función, NO EJECUTAR**

# Aplicando estilos

- Atributo **className**, funciona igual que el atributo **class**

```
// componente
<img className="avatar" />
```

```
/* CSS */
.avatar { border-radius: 50%; }
```

Classnames: Librería para trabajar con clases

🔔 Podemos usar cualquier framework CSS (tailwind, bootstrap…)

- Estilos inline, atributo **style**, propiedades en **camelCase**

```
<img
 style={{
   borderRadius: '50%'
 }}
/>
```

- Librerías CSS in JS: Styled components, Emotion

# Estado: useState

Definimos **estado** como datos “internos” de los componentes (memoria del componente). Su cambio provoca un nuevo render del mismo

```
// importamos el hook useState
import { useState } from 'react';

// el componente Button “recuerda” el valor de count
function Button() {
 // iniciamos el valor del estado
 const [count, setCount] = useState(0);
 // en cada click el estado se incrementa y el componente renderiza de nuevo
 return (
   <button onClick={() => setCount(count + 1)}>Clicked {count} times</button>
 );
}
```

🔔 **Si modificamos el estado directamente el componente no “reacciona”**

# Estado: useState

```
const [value, setValue] = useState(initialValue);
```

- **useState** devuelve un array con el valor del estado **(value)** y la función para modificarlo **(setValue)**
- Al destructurar el array podemos elegir los nombres que queramos
- Podemos guardar cualquier tipo de dato en el estado
- En el primer render **value** es **initialValue**
- Llamar a **setValue** con un **nuevo valor** implica un nuevo render del componente. En el nuevo render **useState** recordará ese valor y lo devuelve como **value**

🔔 **Si un valor puede ser calculado en función del estado, NO ES ESTADO**

# Estado: useState

Hay dos modos de utilizar la función devuelta por **useState**

- Pasando directamente el nuevo valor

```
setValue(newValue)
```

- Pasando una función que recibe el valor actual y devuelve el nuevo

```
setValue(currentValue => newValue)
```

🔔 **Si el nuevo valor depende del anterior, mejor usar la forma “función”**

# Compartiendo estado

Elevar el estado a un “ancestro” común y pasar props hacia abajo

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

 function handleClick() {
   setCount((count) => count + 1);
 }
 return (
   <button onClick={handleClick}>
     Clicked {count} times
   </button>
 );
}

function App() {
 return (
   <div>
     <h1>Counters that update separately</h1>
     <Button />
     <Button />
   </div>
 );
}
```

```
function Button({ count, onClick }) {
 return (
   <button onClick={onClick}>
     Clicked {count} times
   </button>
 );
}

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

 function handleClick() {
   setCount((count) => count + 1);
 }
 return (
   <div>
     <h1>Counters that update together</h1>
     <Button count={count} onClick={handleClick} />
     <Button count={count} onClick={handleClick} />
   </div>
 );
}
```

# Formularios: inputs controlados

Un input controlado “refleja” el valor del estado

```
function Form() {
 // se define un estado donde guardar el valor del input
 const [firstName, setFirstName] = useState("");
 // ...
 return (
   <input
     // es buena práctica asociar un name a cada input
     name="firstName"
     // se fuerza al input a que muestre el valor del estado
     value={firstName}
     // en el evento onChange se actualiza el estado
     onChange={(e) => setFirstName(e.target.value)}
   />
 );
}
```

# Formularios: inputs no controlados

Un input no controlado es “manejado” por el DOM

```
function Form() {
 // ...
 return (
   <input
     name="firstName"
     // podemos pasar un defaultValue
     defaultValue={firstName}
   />
 );
}
```

# Formularios: checkboxes y radio

En lugar de **value** y **defaultValue**, se utiliza **checked** (controlados) y **defaultChecked** (no controlados) para mostrar la seleccion

```
// checkbox controlado
function Form() {
 const [checked, setChecked] = useState(false);
 return (
   <input
     type="checkbox"
     checked={checked}
     onChange={(e) => setChecked(e.target.checked)}
   />
 );
}

// checkbox NO controlado
function Form() {
  return <input type="checkbox" defaultChecked={checked} />;
}
```

# Formularios: submit

Podemos capturar el evento submit para enviar nuestros datos a un API. Debemos evitar el comportamiento por defecto del elemento form al hacer submit con **preventDefault**

Antes de enviar los datos podemos aplicar validaciones

```
function Form() {
 function handleSubmit(event) {
   event.preventDefault();
   // Los datos de nuestro formulario los tenemos en
   // 1. estado
   // 2. event.target.form
   // llamada al API para enviar datos
   callApi(data)
 }
 return <form onSubmit={handleSubmit}>{/* ... */}</form>;
}
```

# Efectos: useEffect

En los componentes necesitamos un lugar donde sincronizar con efectos externos, como por ejemplo:

- Llamadas a APIs para carga de datos
- Subscripciones a sockets, listeners…
- Inicialización de timeouts e intervalos
- Integraciones con librerías de terceros

El lugar donde hacer esto es el hook **useEffect**

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

useEffect(effectCallback, dependencyList);
```

🔔 [You Might Not Need an Effect]

# Efectos: anatomía de useEffect

- La función pasada a useEffect se ejecuta **después** de cada render, ¡**con los datos de ese render!**
- El array dependencias controla cuando se ejecuta el efecto
  - **undefined:** se ejecuta el efecto tras cada render
  - **[ ]**: se ejecuta el efecto **SOLO** después del primer render
  - **[a, b]**: se ejecuta **SOLO** si ha cambiado alguno de los valores del array (a, b…)
- Opcionalmente, el **callback** puede devolver una función de **clean up** o limpieza, que se ejecutará **antes** de la próxima llamada al efecto. Entre otras cosas, sirve para
  - Cancelar subscripciones
  - Liberar timeouts e intervalos
  - Abortar peticiones al API pendientes…

# Efectos: ejemplo de useEffect (1)

Ejemplo de **useEffect** para iniciar un **intervalo** al montar el componente y cancelarlo al desmontar

```
useEffect(() => {
 // Después del render se inicia el intervalo
 const interval = setInterval(() => {
   console.log('tic: ', new Date());
 }, 1000);

 // Antes de desmontar el componente se libera el intervalo
 return () => {
   clearInterval(interval);
 };
 // []: El efecto se ejecuta SOLO después del primer render
}, []);
```

# Efectos: ejemplo de useEffect (2)

Ejemplo de **useEffect** para suscribir a un chat y cancelar la suscripción cuando cambia *chatId*

```
useEffect(() => {
  const handler = message => {
    console.log(chatId, message);
  };
  // Suscribir al chat con chatId
  ChatAPI.subscribe(chatId, handler);
  // Antes de suscribir a un nuevo chat, se cancela la suscripción al antiguo
  return () => {
    ChatAPI.unsubscribe(chatId, handler);
  };
  // []: El efecto se ejecuta cada vez que cambia chatId
}, [chatId]);
```

# Efectos: StrictMode

<StrictMode> permite encontrar bugs durante el desarrollo

- Chequeo de uso de métodos obsoletos en componentes
- Re-render de componentes para detectar bugs
- Re-ejecución de efectos para detectar bugs

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

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
 <React.StrictMode>
   <App />
 </React.StrictMode>
);
```

🔔 **Se recomienda no quitar StrictMode en desarrollo**

# Hooks: algunas reglas

React se basa en el orden en que los hooks están escritos para mantener internamente los valores (estado, efectos, …)
Por ello, debemos seguir ciertas reglas cuando usamos hooks:

- Sólo pueden ser usados dentro de componentes o de otros hooks. No pueden ser usados en funciones “normales” o clases
- Dentro del componente (o de otro hook), no pueden ser usados dentro de bucles, condiciones o funciones internas

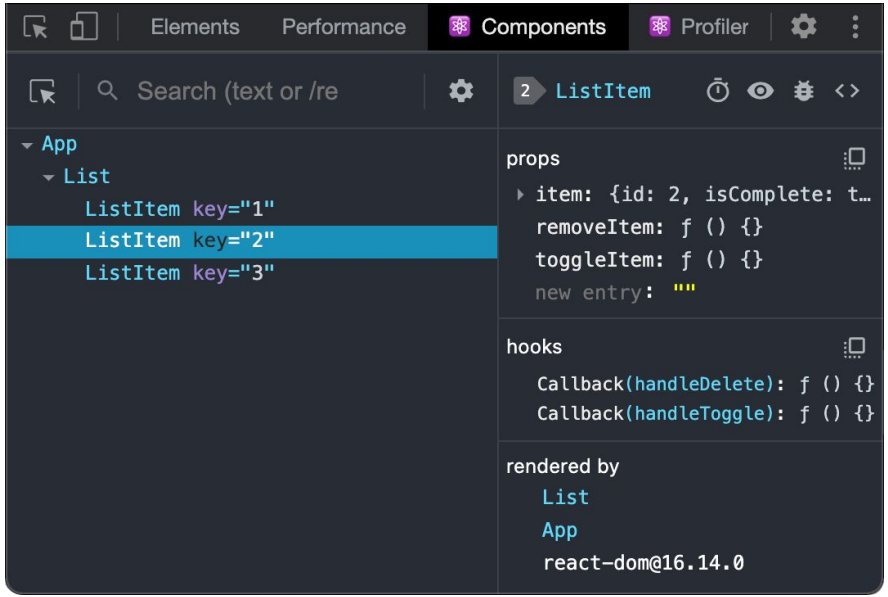
Para asegurar que usamos hooks correctamente, existe un plugin de **eslint** `eslint-plugin-react-hooks`, que nos avisa de la violación de estas reglas, así como del correcto uso de las dependencias en hooks como useEffect

🔔 **Este plugin ya viene configurado en create-react-app**

# Developer Tools (aka devtools)

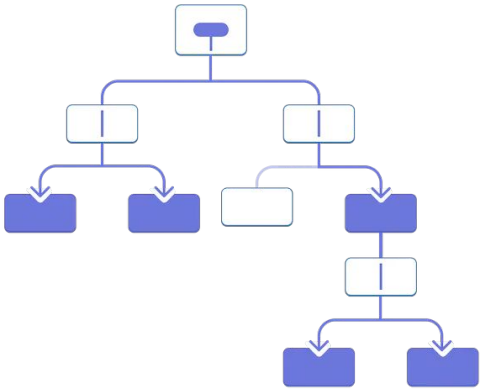
Extensión del browser donde podemos inspeccionar componentes, props y estado.

- Instalar en **Chrome**
- Instalar en **Firefox**
- Instalar en **Edge**



# Context

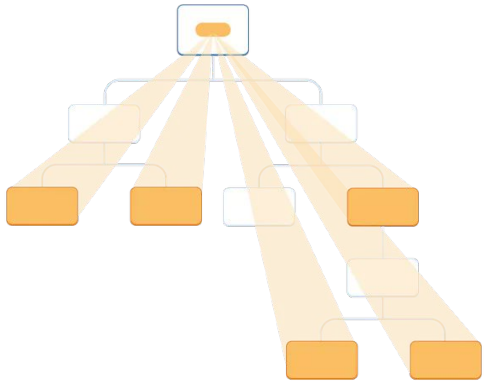
- Pasar **props** de padres a hijos se complica cuando el árbol crece y es propenso a errores (*prop drilling)*
- Componentes intermedios tienen que dejar pasar **props**
- **React context** permite a un componente compartir datos *directamente* con todos los componentes que están debajo de él.



PROPS
DRILLING

vs

CONTEXTO



# Context: paso 1 - crear contexto

Creamos un contexto con **createContext** pasando un default value
Este valor por defecto es un valor “estático” que React usará cuando no encuentre el *provider* del contexto

```
// ThemeContext.js
import { createContext } from 'react';

export const ThemeContext = createContext('light');
```

# Context: paso 2 - consumir contexto

Consumimos el contexto en cualquier componente mediante el hook **useContext**

Un cambio en el contexto forzará un render del componente

```
// MyButton.js
import { useContext } from 'react';
import { ThemeContext } from './ThemeContext.js';

const MyButton = () => {
 const theme = useContext(ThemeContext);

 // el componente puede utilizar theme
};
```

# Context: paso 3 - proveer contexto

Proveer el valor del contexto con **Context.Provider**

Todos los componentes que queden dentro del provider, tendrán acceso al valor del contexto

```
// ThemeProvider.js
import { ThemeContext } from './ThemeContext.js';
export default function ThemeProvider({ children, theme }) {
 return (
   <ThemeContext.Provider value={theme}>{children}</ThemeContext.Provider>
 );
}
// App.js o cualquier componente donde queramos poner el provider del contexto
// Cualquier componente “dentro” del provider tendrá acceso al value
import ThemeProvider from './ThemeProvider.js';
export default function App() {
 return (
   <ThemeProvider value='light'><MyButton /></ThemeProvider>
 );
}
```

# Refs

Valores que los componentes recuerdan pero sin provocar render
Es como un estado (useState) pero sin provocar render al cambiar

Se definen con el hook **useRef** y permiten guardar:

- timeout o interval ids
- referencias a elementos del DOM
- cualquier valor que necesitemos guardar en un componente y que no queramos que cause render

```
import { useRef } from "react";
const count = useRef(0) // devuelve { current: 0 }
// Si queremos mutar el valor de la ref
count.current = count.current + 1;
```

# Refs: ejemplo de ref para acceder al DOM

Ejemplo de ref para acceder a un elemento del DOM y tener acceso a toda su API nativa (métodos y propiedades)

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

export default function Form() {
 const inputRef = useRef(null);
 function handleClick() {
   // Tenemos acceso a todo el API nativo del input, por ejemplo al método “focus”
   inputRef.current.focus();
 }
 return (
   <>
     {/* Pasamos la ref al elemento input */}
     <input ref={inputRef} />
     <button onClick={handleClick}>
       Focus the input
     </button>
   </>
 );
}
```

# Refs: forwardRef

Mecanismo para pasar refs a nuestros componentes, como si fuesen elementos del DOM

```
import { forwardRef, useRef } from 'react';
const MyInput = forwardRef(function MyInput(props, ref) {
 return (
   <label>
     {props.label}
     <input ref={ref} />
   </label>
 );
});
// consumiendo el componente MyInput
const ref = useRef(null);
// ref nos da acceso al elemento input interior
<MyInput ref={ref} />
```

# Performance

Un componente React renderiza cada vez que:

- Cambia algún estado del componente
- Cambia algún valor en un contexto consumido por el componente
- Renderiza su padre, independientemente de que cambien las props que éste envíe, cambien estas o no

Esto último, puede hacer que un componente renderice **innecesariamente** para devolver el mismo resultado que en el render anterior
Normalmente, esto no es un problema para React, pero en aplicaciones más grandes puede provocar problemas de performance

🔔 **Preocuparse por rerenders cuando sea un problema**

# Performance: memo

**memo** permite evitar el render de un componente cuando sus props no han cambiado

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

const MemoizedComponent = memo(SomeComponent, arePropsEqual?)
// SomeComponent evitará renderizar si recibe las mismas props

// arePropsEqual permite definir con una función en qué casos el componente
// renderiza de nuevo
// recibe las props nuevas y viejas y podemos comparar entre ellas para decidir
// evitar nuevo render: devolver true
// permitir nuevo render: devolver false
```

🔔 **memo no evita renders por cambio de estado o contexto**

# Performance: useCallback y useMemo

Cuando usamos **memo** debemos de tener cuidado al comparar props que sean objetos o funciones (nuevas referencias)

- **useCallback** permite fijar una referencia de una función cuando la tenemos que pasar como prop hacia abajo
- **useMemo** permite fijar la referencia de un objeto (o array) cuando lo vamos a pasar como prop hacia abajo
  También permite evitar cálculos complejos cuando los valores de los que depende ese cálculo no han cambiado

# Performance: useCallback y useMemo

## useCallback

```
import { useCallback } from 'react';
const handler = useCallback(() => {
 //...
}, [a, b]);
// la referencia de handler se mantendrá a menos que cambie algún valor del array de
dependencias (a, b…)
// de ese modo podemos asegurar que la referencia no se renueva innecesariamente
```

## useMemo

```
import { useMemo } from 'react';
const value = useMemo(() => someExpensiveCalc(a, b) , [a, b]);
// la referencia de value se mantendrá a menos que cambie algún valor del array de
dependencias (a, b…)
// se evita llamar a someExpensiveCalc si ya “sabemos” lo que va a devolver
```

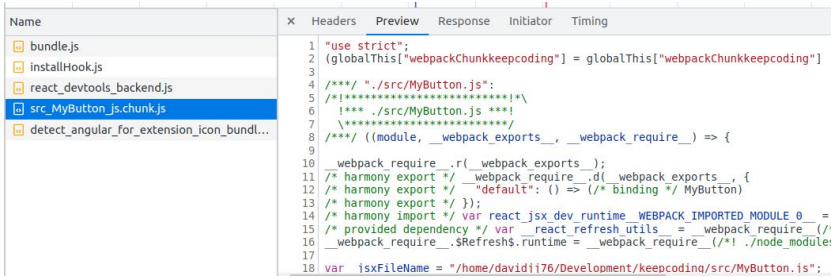
# Lazy loading y code splitting

**lazy** permite retrasar la carga del código de un componente hasta que es realmente necesario

Cuando cargamos un componente con **lazy** el código del componente (src_MyButton_js.chunk.js) es separado del bundle principal (bundle.js)

Con **Suspense** podemos definir una interfaz alternativa que será presentada mientras descarga el código del componente (loader, spinner…)

```
import { Suspense, lazy } from 'react';
const MyButton = lazy(() => import('./MyButton'));

export default function App() {
 return (
   <Suspense fallback={<div>Loading...</div>}>
     <MyButton>Click me</MyButton>
   </Suspense>
 );
}
```



# KEEPCODING

Tech School

Madrid | Barcelona | Bogotá

Datos de contacto

y