

PRÁCTICA EN VISUAL STUDIO CODE

- 1 Hello World
- 2 Folder Structure (Estructura de carpetas)
- 3 Components (Componentes)
 - 3.1 Functional Components (Componentes funcionales)
 - 3.2 Class Components (Componentes de clase)
- 4 Hooks Update (Actualización con Hooks)
- 5 JSX (Sintaxis de JavaScript XML)
- 6 Props y State
 - 6.1 Props
 - 6.2 State
 - 6.3 Set State
 - 6.4 Destructuring Props and State
- 7 Event Handling (Manejo de eventos)
 - 7.1 Binding Event Handlers
 - 7.2 Methods as Props
- 8 Conditional Rendering (Renderizado condicional)
- 9 List Rendering (Renderizado de listas)
 - 9.1 List and Keys
 - 9.2 Index as Key Anti-Pattern
- 10 Styling and CSS Basics (Estilos y CSS básicos)
- 11 Basics of Form Handling (Manejo de formularios)
- 12 Component Lifecycle Methods (Ciclo de vida del componente)
 - 12.1 Mounting Lifecycle Methods
 - 12.2 Updating Lifecycle Methods
- 13 Fragments (Fragmentos)
- 14 Pure Components y Memo
- 15 Refs (Referencias)
 - 15.1 Refs with Class Components
 - 15.2 Forwarding Refs
- 16 Portals (Portales)
- 17 Error Boundary (Límites de error)
- 18 Higher Order Components (Componentes de orden superior)
- 19 Render Props
- 20 Context (Contexto global)
- 21 HTTP and React (Comunicación con APIs)
 - 21.1 HTTP GET Request
 - 21.2 HTTP POST Request

1. Hello World

“Hello World” es el primer paso para comprobar que **React está correctamente instalado y funcionando.**

En este punto, simplemente mostramos un mensaje en pantalla utilizando un **componente funcional**.

Esto valida que Visual Studio Code, Node.js y React están configurados correctamente.

Ejemplo en React

En el archivo src/App.js, reemplaza su contenido con lo siguiente:

```
function App () {
  return (
    <div>
      <h1>Hola Mundo</h1>
    </div>
  );
}

export default App;
```

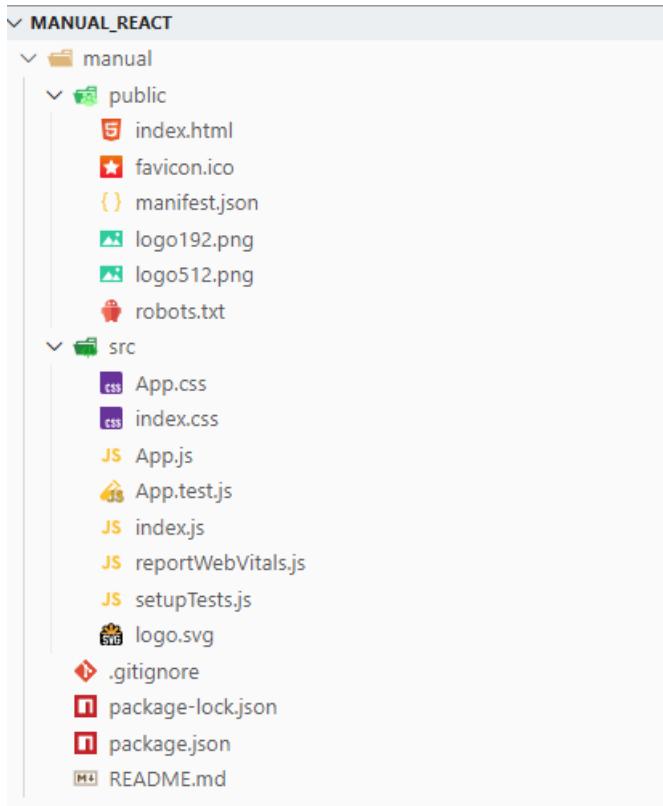
Hola Mundo

2. Folder Structure (Estructura de carpetas)

Cuando creas un proyecto React con create-react-app, se genera una estructura de carpetas estándar que organiza los archivos del proyecto.

Las más importantes son:

- **node_modules/** → contiene las dependencias instaladas.
- **public/** → archivos estáticos (como index.html, imágenes, íconos).
- **src/** → carpeta principal del código React (componentes, estilos, lógica).
- **package.json** → lista de dependencias y scripts del proyecto.



✳️ 3. Components (Componentes)

Los **componentes** son las piezas fundamentales en React.

Cada componente representa una parte visual o funcional de la aplicación (por ejemplo, un botón, una cabecera o una tarjeta).

Hay **dos tipos principales: funcionales y de clase**.

✳️ 3.1 Functional Components (Componentes funcionales)

Son funciones de JavaScript que devuelven una interfaz (JSX).

Actualmente son los más usados porque son más simples y se combinan con Hooks.

💻 Ejemplo en React

Crea un archivo src/Welcome.js:

```
import Welcome from './Welcome';

function App() {
  return (
    <div>
      <h1>Manual React</h1>
      <Welcome />
    </div>
  );
}

export default App;
```

Y luego usa este componente dentro de App.js:

```
function Welcome() {
  return <h2>Welcome Component - Functional</h2>;
}

export default Welcome;
```

Manual React

Welcome Component - Functional

3.2 Class Components (Componentes de clase)

Son componentes basados en clases ES6.

Fueron el método original para manejar **estado y ciclo de vida**, antes de que existieran los Hooks.

Ejemplo en React

Crea un archivo src/Greeting.js:

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

class Greeting extends Component {
  render() {
    return <h2>Greeting Component - Class</h2>;
  }
}

export default Greeting;
```

Y modifica tu App.js:

```
import Greeting from './Greeting';
import Welcome from './Welcome';

function App() {
  return (
    <div>
      <h1>Manual React </h1>
      <Welcome />
      <Greeting />
    </div>
  );
}

export default App;
```

Manual React

Welcome Component - Functional

Greeting Component - Class

4. Hooks Update (Actualización con Hooks)

Los **Hooks** permiten usar características de React (como el estado o el ciclo de vida) dentro de componentes funcionales.

Antes, esto solo era posible con componentes de clase.

El Hook más usado es **useState**, que permite **almacenar y actualizar valores** dentro del componente.

Ejemplo en React

Crea un archivo src/Counter.js:

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

function Counter() {
  // Declaramos una variable de estado llamada "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <h2>Contador con Hook useState</h2>
      <p>Valor actual: {count}</p>
      <button onClick={() => setCount(count + 1)}>Aumentar</button>
      <button onClick={() => setCount(count - 1)}>Disminuir</button>
    </div>
  );
}

export default Counter;
```

Luego impórtalo en App.js:

```
import React from 'react';
import Counter from './Counter';

function App() {
  return (
    <div>
      <h1>Ejemplo de Hooks</h1>
      <Counter />
    </div>
  );
}

export default App;
```

Ejemplo de Hooks

Contador con Hook useState

Valor actual: 0

★ 5. JSX (JavaScript XML)

🧠 Explicación teórica

JSX es una extensión de JavaScript que permite escribir código similar a HTML dentro de los archivos .js.

React lo transforma internamente en llamadas a funciones JavaScript.

JSX permite **combinar lógica y diseño** en el mismo archivo.

💻 Ejemplo en React

Crea un archivo src/JSXExample.js:

```
import React from 'react';

function JSXExample() {
  const name = "React Developer";
  const year = new Date().getFullYear();

  return (
    <div>
      <h2>Hola, {name}</h2>
      <p>Estamos en el año {year}</p>
      <p>{2 + 3} es el resultado de una operación dentro de JSX</p>
    </div>
  );
}

export default JSXExample;
```

Modifica App.js para usarlo:

```
import JSXExample from './JSXExample';

function App() {
  return (
    <div>
      <h1>Ejemplo de JSX</h1>
      <JSXExample />
    </div>
  );
}

export default App;
```

Ejemplo de JSX

Hola, React Developer 

Estamos en el año 2026

5 es el resultado de una operación dentro de JSX

6. Props y State

Explicación teórica

- **Props:** son **propiedades** que se envían de un componente padre a uno hijo.
Sirven para **pasar información o personalizar** los componentes.
 - **State:** es un **estado interno** del componente, que puede cambiar con el tiempo.
Cuando el estado cambia, el componente se vuelve a renderizar automáticamente.
-

6.1 Props

Ejemplo en React

Crea src/User.js:

```
import React from 'react';

function User(props) {
  return <h2>Bienvenido, {props.name} 👋</h2>;
}

export default User;
```

Y modifícalo en App.js:

```
import React from 'react';
import User from './User';

function App() {
  return (
    <div>
      <h1>Ejemplo de Props</h1>
      <User name="Carlos" />
      <User name="María" />
    </div>
  );
}

export default App;
```

Ejemplo de Props

Bienvenido, Carlos 👋

Bienvenido, María 👋

6.2 State y SetState



Ejemplo en React

Crea src/Message.js:

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

function Message() {
  const [message, setMessage] = useState("Bienvenido a React");

  return (
    <div>
      <h2>{message}</h2>
      <button onClick={() => setMessage("Gracias por aprender React!")}>
        Cambiar mensaje
      </button>
    </div>
  );
}

export default Message;
```

Agrega en App.js:

```
import Message from './Message';

function App() {
  return (
    <div>
      <h1>Ejemplo de State</h1>
      <Message />
    </div>
  );
}

export default App;
```

Ejemplo de State

Bienvenido a React

Cambiar mensaje

6.3 Destructuring Props and State

El **destructuring** simplifica el acceso a los valores dentro de props o state.
En lugar de escribir props.name, puedes escribir directamente { name }.

Ejemplo en React

Crea src/Product.js:

```
function Product({ name, price }) {
  return <p>{name} - ${price}</p>;
}

export default Product;
```

Agrega en App.js:

```
import React from 'react';
import Product from './Product';

function App() {
  return (
    <div>
      <h1>Destructuring de Props</h1>
      <Product name="Teclado" price="25" />
      <Product name="Ratón" price="15" />
    </div>
  );
}

export default App;
```

Deconstructing de Props

Teclado - \$25

Ratón - \$15

7. Event Handling (Manejo de eventos)

En React, el **manejo de eventos** se realiza de forma muy similar a JavaScript, pero con pequeñas diferencias:

- Los nombres de los eventos se escriben en **camelCase** (por ejemplo, onClick, onChange).
- En lugar de cadenas, se pasan **funciones** directamente como manejadores.

Esto permite que los componentes respondan a interacciones del usuario, como clics, envíos de formularios o cambios en campos de texto.

Ejemplo en React

Crea un archivo src/EventExample.js:

```
function EventExample() {
  function handleClick() {
    alert("Has hecho clic en el botón!");
  }

  return (
    <div>
      <h2>Manejo de eventos en React</h2>
      <button onClick={handleClick}>Haz clic aquí</button>
    </div>
  );
}

export default EventExample;
```

Modifica App.js:

```
import EventExample from './EventExample';

function App() {
  return (
    <div>
      <h1>Ejemplo de manejo de eventos</h1>
      <EventExample />
    </div>
  );
}

export default App;
```

Ejemplo de manejo de eventos

Manejo de eventos en React

Haz clic aquí

✳ 7.1 Binding Event Handlers (Vinculación de eventos)

En los **componentes de clase**, es necesario vincular los métodos (bind) para que this funcione correctamente dentro del manejador.

En los componentes funcionales esto no es necesario gracias a las funciones flecha.

💻 Ejemplo en React

Crea src/ClassEvent.js:

```

import React, { Component } from 'react';

class ClassEvent extends Component {
  constructor(props) {
    super(props);
    this.state = { message: 'Haz clic para cambiar el texto' };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState({ message: 'Texto actualizado!' });
  }

  render() {
    return (
      <div>
        <h2>{this.state.message}</h2>
        <button onClick={this.handleClick}>Cambiar texto</button>
      </div>
    );
  }
}

export default ClassEvent;

```

Agrega en App.js:

```

import React from 'react';
import ClassEvent from './ClassEvent';

function App() {
  return (
    <div>
      <h1>Binding en Componentes de Clase</h1>
      <ClassEvent />
    </div>
  );
}

export default App;

```

Binding en Componentes de Clase

Haz clic para cambiar el texto

Cambiar texto

✿ 7.2 Methods as Props (Métodos como Props)

Podemos **pasar funciones** desde un componente padre a uno hijo como propiedades. Esto permite que el hijo **ejecute acciones del padre**, lo cual es útil para comunicación entre componentes.

💻 Ejemplo en React

Crea src/ChildButton.js:

```
import React from 'react';

function ChildButton(props) {
  return (
    <button onClick={props.handleClick}>
      Haz clic en el botón hijo
    </button>
  );
}

export default ChildButton;
```

Luego modifica App.js:

```
import React from 'react';
import ChildButton from './ChildButton';

function App() {
  function showMessage() {
    alert("Evento ejecutado desde el componente hijo");
  }

  return (
    <div>
      <h1>Methods as Props</h1>
      <ChildButton handleClick={showMessage}></ChildButton>
    </div>
  );
}

export default App;
```

Methods as Props

Haz clic en el botón hijo

✿ 8. Conditional Rendering (Renderizado condicional)

El **renderizado condicional** permite mostrar u ocultar elementos dependiendo del **estado o de una condición lógica**.

React evalúa las condiciones dentro del JSX usando operadores como if, &&, o el **operador ternario** (? :).

💻 Ejemplo en React

Crea src/ConditionalExample.js:

```

import { useState } from 'react';

function ConditionalExample() {
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  return (
    <div>
      <h2>Renderizado condicional</h2>
      {isLoggedIn ? <p>Bienvenido, usuario!</p> : <p>Por favor, inicia sesión.</p>}
      <button onClick={() => setIsLoggedIn(!isLoggedIn)}>
        {isLoggedIn ? 'Cerrar sesión' : 'Iniciar sesión'}
      </button>
    </div>
  );
}

export default ConditionalExample;

```

Modifica App.js:

```

import React from 'react';
import ConditionalExample from './ConditionalExample';

function App() {
  return (
    <div>
      <h1>Ejemplo de Renderizado Condicional</h1>
      <ConditionalExample />
    </div>
  );
}

export default App;

```

Ejemplo de Renderizado Condicional

Renderizado condicional

Por favor, inicia sesión.

[Iniciar sesión](#)

9. List Rendering (Renderizado de listas)

El **renderizado de listas** permite mostrar colecciones de datos (como usuarios o productos) de forma dinámica usando el método `.map()`.

Cada elemento debe tener una **propiedad única “key”** para que React pueda identificarlo correctamente.

Ejemplo en React

Crea `src/UserList.js`:

```
import React from 'react';

function UserList() {
  const users = ['Carlos', 'María', 'Ana', 'Pedro'];

  return (
    <div>
      <h2>Lista de usuarios</h2>
      <ul>
        {users.map((user, index) => (
          <li key={index}>{user}</li>
        ))}
      </ul>
    </div>
  );
}

export default UserList;
```

Modifica `App.js`:

```
import UserList from './UserList';

function App() {
  return (
    <div>
      <h1>Renderizado de Listas</h1>
      <UserList />
    </div>
  );
}

export default App;
```

Renderizado de Listas

Lista de usuarios

- Carlos
- Maria
- Ana
- Pedro

✳️ 9.1 List and Keys / 9.2 Index as Key Anti-Pattern

Las keys son valores únicos que se asignan a cada elemento de una lista.
Siempre que sea posible, se debe usar un **id único real**.

Un anti-pattern es una práctica que funciona, pero no es recomendable porque puede causar errores en el futuro.

Usar el índice del array (index) como key puede provocar:

- Errores visuales
- Renderizados incorrectos
- Pérdida de rendimiento

💻 Ejemplo correcto (con keys únicas)

Creamos src/UserListKeys

```
import React from 'react';

function UserListKeys () {
  const users = [
    { id: 1, name: 'Carlos' },
    { id: 2, name: 'Maria' },
    { id: 3, name: 'Ana' }
  ];

  return (
    <div>
      <h2>Lista con keys correctas</h2>
      <ul>
        {users.map(user => (
          <li key={user.id}>{user.name}</li>
        )));
      </ul>

      <h2>Lista usando index como key (Anti-Pattern)</h2>
      <ul>
        {users.map((user, index) => (
          <li key={index}>{user.name}</li>
        )));
      </ul>
    </div>
  );
}

export default UserListKeys;
```

Agrega en App.js:

```

import React from 'react';
import UserListKeys from './UserListKeys';

function App() {
  return (
    <div>
      <h1>List and Keys en React</h1>
      <UserListKeys />
    </div>
  );
}

export default App;

```

10. Styling and CSS Basics (Estilos y fundamentos de CSS en React)

En React, hay varias formas de aplicar estilos a los componentes:

CSS tradicional (archivos .css importados).

Inline styles (estilos en línea usando objetos JavaScript).

CSS Modules o Styled Components (formas más avanzadas y modernas).

El método más común es crear un archivo .css y asignar clases con className (en lugar de class, como en HTML).

Ejemplo en React

Crea src/StyleExample.js:

```

import React from 'react';
import './StyleExample.css';

function StyleExample() {
  return (
    <div className="container">
      <h2 className="title">Ejemplo de estilos en React</h2>
      <p className="paragraph">Este texto está estilizado desde un archivo CSS externo.</p>
    </div>
  );
}

export default StyleExample;

```

Crea un archivo CSS src/StyleExample.css:

```
.container {  
  border: 2px solid #4CAF50;  
  padding: 20px;  
  border-radius: 10px;  
  background-color: #F9F9F9;  
  text-align: center;  
}  
  
.title {  
  color: #4CAF50;  
  font-size: 24px;  
}  
  
.paragraph {  
  color: #333;  
  font-size: 16px;  
}
```

Agrega en App.js:

```
import React from 'react';  
import StyleExample from './StyleExample';  
  
function App() {  
  return (  
    <div>  
      <h1>Estilos en React</h1>  
      <StyleExample />  
    </div>  
  );  
}  
  
export default App;
```

Estilos en React

Ejemplo de estilos en React

Este texto está estilizado desde un archivo CSS externo.

11. Basics of Form Handling (Manejo básico de formularios)

React controla los formularios mediante **componentes controlados**, donde el valor del input se almacena en el **estado (state)** del componente.

Esto permite manejar los datos de los formularios de forma dinámica y validarlos fácilmente.

Ejemplo en React

Crea src/FormExample.js:

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

function FormExample() {
  const [name, setName] = useState('');

  function handleSubmit(e) {
    e.preventDefault();
    alert(`Hola, ${name}!`);
  }

  return (
    <div>
      <h2>Formulario Controlado</h2>
      <form onSubmit={handleSubmit}>
        <label>Nombre: </label>
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
          placeholder="Escribe tu nombre"
        />
        <button type="submit">Enviar</button>
      </form>
    </div>
  );
}

export default FormExample;
```

Modifica App.js:

```
import React from 'react';
import FormExample from './FormExample';

function App() {
  return (
    <div>
      <h1>Manejo de Formularios</h1>
      <FormExample />
    </div>
  );
}

export default App;
```

Manejo de Formularios

Formulario Controlado

Nombre: Enviar

✿ 12. Component Lifecycle Methods (Métodos del ciclo de vida del componente)

Los **métodos del ciclo de vida** existen en **componentes de clase** y permiten ejecutar código en momentos específicos:

- **Mounting (Montaje)**: cuando el componente aparece por primera vez.
- **Updating (Actualización)**: cuando cambia el estado o las props.
- **Unmounting (Desmontaje)**: cuando se elimina del DOM.

En componentes funcionales, este comportamiento se logra con **Hooks** como `useEffect()`.

12.1 Component Mounting Lifecycle Methods (Montaje)

Ejemplo en React

Crea `src/MountExample.js`:

```

import React, { Component } from 'react';

class MountExample extends Component {
  constructor(props) {
    super(props);
    this.state = { message: 'Montando el componente...' };
    console.log('Constructor: Se inicializa el estado');
  }

  componentDidMount() {
    console.log('componentDidMount: El componente se montó en el DOM');
    setTimeout(() => {
      this.setState({ message: 'Componente montado correctamente!' });
    }, 2000);
  }

  render() {
    console.log('Renderizando...');
    return <h2>{this.state.message}</h2>;
  }
}

export default MountExample;

```

Modifica App.js:

```

import React from 'react';
import MountExample from './MountExample';

function App() {
  return (
    <div>
      <h1>Ciclo de Vida: Montaje</h1>
      <MountExample />
    </div>
  );
}

export default App;

```

Ciclo de Vida: Montaje

¡Componente montado correctamente!

✳ 12.2 Component Updating Lifecycle Methods (Actualización)

💻 Ejemplo en React

Crea src/UpdateExample.js:

```
import React, { Component } from 'react';

class UpdateExample extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  componentDidUpdate(prevProps, prevState) {
    if (prevState.count !== this.state.count) {
      console.log('componentDidUpdate: El estado ha cambiado');
    }
  }

  render() {
    return (
      <div>
        <h2>Ejemplo de actualización</h2>
        <p>Valor actual: {this.state.count}</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Incrementar
        </button>
      </div>
    );
  }
}

export default UpdateExample;
```

Modifica App.js:

```
import React from 'react';
import UpdateExample from './UpdateExample';

function App() {
  return (
    <div>
      <h1>Ciclo de Vida: Actualización</h1>
      <UpdateExample />
    </div>
  );
}

export default App;
```

Ciclo de Vida: Actualización

Ejemplo de actualización

Valor actual: 0

Incrementar

✳️ 13. Fragments (Fragmentos en React)

Los **fragments** en React permiten agrupar varios elementos **sin añadir nodos extra al DOM**.

En lugar de envolver todo con un `<div>`, puedes usar `<React.Fragment>` o su forma corta `<></>`.

Esto mejora el rendimiento y mantiene un código más limpio.

 Ejemplo en React

Crea `src/FragmentExample.js`:

```
import React from 'react';

function FragmentExample() {
  return (
    <>
      <h2>Ejemplo de Fragment</h2>
      <p>Los fragmentos permiten devolver múltiples elementos sin un contenedor extra.</p>
      <p>Esto evita crear nodos innecesarios en el DOM.</p>
    </>
  );
}

export default FragmentExample;
```

Agrega en App.js:

```
import React from 'react';
import FragmentExample from './FragmentExample';

function App() {
  return (
    <div>
      <h1>Fragments en React</h1>
      <FragmentExample />
    </div>
  );
}

export default App;
```

Fragments en React

Ejemplo de Fragment

Los fragmentos permiten devolver múltiples elementos sin un contenedor extra.

Esto evita crear nodos innecesarios en el DOM.

✳ 14. Pure Components (Componentes puros)

Un **PureComponent** es una versión especial de los componentes de clase que **evita renderizados innecesarios**.

Solo se vuelve a renderizar si las **props o el state cambian realmente**.
Esto mejora el rendimiento en aplicaciones grandes.

Ejemplo en React

Crea src/PureExample.js:

```
import React, { PureComponent } from 'react';

class PureExample extends PureComponent {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  render() {
    console.log("Renderizando componente puro...");
    return (
      <div>
        <h2>Pure Component</h2>
        <p>Valor: {this.state.count}</p>
        <button onClick={() => this.setState({ count: this.state.count })}>
          No cambiar valor
        </button>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Aumentar
        </button>
      </div>
    );
  }

  export default PureExample;
```

Agrega en App.js:

```
import React from 'react';
import PureExample from './PureExample';

function App() {
  return (
    <div>
      <h1>Componente Puro</h1>
      <PureExample />
    </div>
  );
}

export default App;
```

✳️ 15. Memo (Optimización con React.memo)

React.memo() es una función que **optimiza componentes funcionales**, haciendo que se vuelvan a renderizar **solo si cambian sus props**.

Funciona igual que PureComponent pero para **componentes funcionales**.

💻 Ejemplo en React

Crea src/MemoExample.js:

```

import React, { PureComponent } from 'react';

class PureExample extends PureComponent {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  render() {
    console.log("Renderizando componente puro...");
    return (
      <div>
        <h2>Pure Component</h2>
        <p>Valor: {this.state.count}</p>
        <button onClick={() => this.setState({ count: this.state.count })}>
          No cambiar valor
        </button>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Aumentar
        </button>
      </div>
    );
  }
}

export default PureExample;

```

Agrega en App.js:

```

import MemoExample from './MemoExample';

function App() {
  return (
    <div>
      <h1>React.memo</h1>
      <MemoExample />
    </div>
  );
}

export default App;

```

React.memo

Ejemplo con React.memo

Hola desde React.memo

Contador: 0

[Incrementar contador](#)

✳ 16. Refs (Referencias)

Los **Refs** en React permiten acceder directamente a elementos del DOM o a componentes hijos.

Son útiles cuando necesitas manipular elementos de forma directa (por ejemplo, enfocar un input).

💻 Ejemplo en React

Crea src/RefExample.js:

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

function RefExample() {
  const inputRef = useRef(null);

  function focusInput() {
    inputRef.current.focus();
  }

  return (
    <div>
      <h2>Uso de useRef</h2>
      <input ref={inputRef} type="text" placeholder="Escribe algo..." />
      <button onClick={focusInput}>Tocar input</button>
    </div>
  );
}

export default RefExample;
```

Agrega en App.js:

```
import React from 'react';
import RefExample from './RefExample';

function App () {
  return (
    <div>
      <h1>Ejemplo de useRef</h1>
      <RefExample />
    </div>
  );
}

export default App;
```

Ejemplo de useRef

Uso de useRef

Escribe algo... Tocar input

✳️ 16.1 Refs with Class Components (Refs en Componentes de Clase)

💻 Ejemplo en React

Crea src/ClassRefExample.js:

```

import React, { Component } from 'react';

class ClassRefExample extends Component {
  constructor(props) {
    super(props);
    this.inputRef = React.createRef();
  }

  focusInput = () => {
    this.inputRef.current.focus();
  };

  render() {
    return (
      <div>
        <h2>Refs en Componentes de Clase</h2>
        <input ref={this.inputRef} type="text" placeholder="Clase con ref" />
        <button onClick={this.focusInput}>Tocar input</button>
      </div>
    );
  }
}

export default ClassRefExample;

```

Agrega en App.js:

```

import ClassRefExample from './ClassRefExample';

function App() {
  return (
    <div>
      <h1>Refs con Componentes de Clase</h1>
      <ClassRefExample />
    </div>
  );
}

export default App;

```

Refs con Componentes de Clase

Refs en Componentes de Clase

Clase con ref Tocar input

★ 16.2 Forwarding Refs (Reenvío de referencias)

El **reenvío de refs** (Forwarding Refs) permite que un componente hijo **reciba una ref del padre** para que el padre pueda acceder directamente al DOM del hijo.

Ejemplo en React

Crea src/ChildInput.js:

```
import React from 'react';

const ChildInput = React.forwardRef((props, ref) => {
  return <input ref={ref} type="text" placeholder="Soy el hijo con ref"/>;
});

export default ChildInput;
```

Crea

src/ForwardRefExample.js:

```

import React, { useRef } from 'react';
import ChildInput from './ChildInput';

function ForwardRefExample() {
  const inputRef = useRef(null);

  const focusChildInput = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <h2>Forwarding Ref Example</h2>
      <ChildInput ref={inputRef} />
      <button onClick={focusChildInput}>Tocar input del hijo</button>
    </div>
  );
}

export default ForwardRefExample;

```

Agrega
en App.js:

```

import React from 'react';
import ForwardRefExample from './ForwardRefExample';

function App() {
  return (
    <div>
      <h1>Forwarding Refs</h1>
      <ForwardRefExample />
    </div>
  );
}

export default App;

```

Forwarding Refs

Forwarding Ref Example

Soy el hijo con ref	Tocar input del hijo
---------------------	----------------------

17. Portals

Los **Portals** permiten renderizar un componente **frente al árbol DOM principal** de React.

Se usan mucho para **modales, diálogos, tooltips**, etc., que deben mostrarse “por encima” del resto de la app.

Aunque el componente esté frente al div#root, sigue siendo parte de React.

Ejemplo en React

En public/index.html, añade debajo de root:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width,
      initial-scale=1" />
    <title>React Portals</title>
  </head>
  <body>
    <div id="root"></div>
    <div id="portal-root"></div>
  </body>
</html>
```

Crea src/PortalExample.js:

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

function PortalExample() {
  return ReactDOM.createPortal(
    <div style={{ background: '#eee', padding: '20px' }}>
      <h2>Esto es un Portal</h2>
      <p>Se renderiza fuera del div root</p>
    </div>,
    document.getElementById('portal-root')
  );
}

export default PortalExample;
```

En App.js:

```
import React from 'react';
import PortalExample from './PortalExample';

function App() {
  return (
    <div>
      <h1>Ejemplo de Portals</h1>
      <PortalExample />
    </div>
  );
}

export default App;
```

Ejemplo de Portals

Esto es un Portal

Se renderiza fuera del div root

✿ 18. Error Boundary

Un **Error Boundary** es un componente que **captura errores** de otros componentes y evita que toda la app se rompa.

Solo existen como **componentes de clase**.

Se usan para mostrar mensajes de error personalizados.

💻 Ejemplo en React

Crea src/ErrorBoundary.js:

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

class ErrorBoundary extends Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    console.error("Error capturado:", error);
  }

  render() {
    if (this.state.hasError) {
      return <h2>Algo salió mal. 😱</h2>;
    }
    return this.props.children;
  }
}

export default ErrorBoundary;
```

Crea un componente con error src/BuggyComponent.js:

```
function BuggyComponent() {
  throw new Error("Error intencional");
}

export default BuggyComponent;
```

En App.js:

```

import React from 'react';
import ErrorBoundary from './ErrorBoundary';
import BuggyComponent from './BuggyComponent';

function App() {
  return (
    <div>
      <h1>Error Boundary</h1>
      <ErrorBoundary>
        <BuggyComponent />
      </ErrorBoundary>
    </div>
  );
}

export default App;

```

19. Higher Order Components (HOC)

Un **HOC** es una función que **recibe un componente y devuelve otro componente** con funcionalidades extra.

Se usa para reutilizar lógica (autenticación, permisos, logging, etc.).

Ejemplo en React

Crea src/withLogger.js:

```

import React from 'react';

const withLogger = (WrappedComponent) => {
  return (props) => {
    console.log("Componente renderizado");
    return <WrappedComponent {...props} />;
  };
}

export default withLogger;

```

Crea src/SimpleComponent.js:

```
import React from 'react';
import withLogger from './withLogger';

function SimpleComponent() {
  return <h2>Componente con HOC</h2>;
}

export default withLogger(SimpleComponent);
```

En App.js:

```
import React from 'react';
import SimpleComponent from './SimpleComponent';

function App() {
  return (
    <div>
      <h1>Higher Order Component</h1>
      <SimpleComponent />
    </div>
  );
}

export default App;
```

Higher Order Component

Componente con HOC

20. Render Props

El patrón **Render Props** consiste en pasar una función como prop para decidir **qué se renderiza**.

Permite reutilizar lógica sin herencia.

Ejemplo en React

Crea src/CounterRenderProps.js:

```

import React, { useState } from 'react';

function CounterRenderProps({ render }) {
  const [count, setCount] = useState(0);

  return (
    <div>
      {render(count)}
      <button onClick={() => setCount(count + 1)}>Incrementar</button>
    </div>
  );
}

export default CounterRenderProps;

```

En App.js:

```

import React from 'react';
import CounterRenderProps from './CounterRenderProps';

function App() {
  return (
    <div>
      <h1>Render Props</h1>
      <CounterRenderProps
        render={(count) => <p>Valor actual: <span>{count}</span></p>}
      />
    </div>
  );
}

export default App;

```

Render Props

Valor actual: 0

[Incrementar](#)

✳️ 20. Context

Context permite compartir datos globales (tema, idioma, usuario) sin pasar props manualmente por todos los componentes.

💻 Ejemplo en React

Crea src/UserContext.js:

```
import React from 'react';

const UserContext = React.createContext("Invitado");
export default UserContext;
```

Crea src/ContextExample.js:

```
import React, { useContext } from 'react';
import UserContext from './UserContext';

function ContextExample() {
  const user = useContext(UserContext);
  return <h2>Usuario: {user}</h2>;
}

export default ContextExample;
```

En App.js:

```
import React from 'react';
import UserContext from './UserContext';
import ContextExample from './ContextExample';

function App() {
  return (
    <UserContext.Provider value="Carlos">
      <h1>Context API</h1>
      <ContextExample />
    </UserContext.Provider>
  );
}

export default App;
```

Context API

Usuario: Carlos

21. HTTP and React (GET y POST)

Las aplicaciones React suelen comunicarse con **servidores y APIs REST** para obtener o enviar información.

Para ello se utilizan **peticiones HTTP**, siendo las más comunes **GET y POST**.

React no incluye un sistema propio para hacer peticiones HTTP, pero se apoya en herramientas como:

- fetch (incluida en el navegador)
- Librerías externas como axios

Estas peticiones se realizan normalmente dentro de componentes React.

21.1 HTTP GET Request

Una petición GET se utiliza para obtener información de un servidor.

No modifica los datos del servidor, solo los consulta.

En React, las peticiones GET suelen ejecutarse cuando el componente se carga por primera vez, utilizando el Hook useEffect.

Ejemplo en React

Crea src/ GetExample.js:

```

import React, { useEffect, useState } from 'react';

function GetExample() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/users')
      .then(res => res.json())
      .then(data => setUsers(data));
  }, []);
}

return (
  <div>
    <h2>Lista de usuarios (GET)</h2>
    <ul>
      {users.map(user => (
        <li key={user.id}>
          {user.name}
        </li>
      ))}
    </ul>
  </div>
);

export default GetExample;

```

En App.js:

```

import React from 'react';
import GetExample from './GetExample';

function App() {
  return (
    <div>
      <h1>HTTP GET en React</h1>
      <GetExample />
    </div>
  );
}

export default App;

```

HTTP GET en React

Listado de usuarios (GET)

- Leanne Graham
 - Ervin Howell
 - Clementine Bauch
 - Patricia Lebsack
 - Chelsey Dietrich
 - Mrs. Dennis Schulist
 - Kurtis Weissnat
 - Nicholas Runolfsdottir V
 - Glenna Reichert
 - Clementina DuBuque
-

21.2 HTTP POST Request

Una petición POST se utiliza para enviar datos al servidor.

Normalmente se usa para crear nuevos registros, enviar formularios o guardar información.

A diferencia de GET, POST sí modifica datos en el servidor.

Ejemplo en React

Creamos src/PostExample.js:

```
import React from 'react';

function PostExample() {
  function sendPost() {
    fetch('https://jsonplaceholder.typicode.com/posts', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({
        title: 'React POST',
        body: 'Ejemplo de envío',
        userId: 1,
      }),
    })
      .then(response => response.json())
      .then(data => {
        console.log('Respuesta del POST:', data);
        alert('POST enviado correctamente (ver consola)');
      });
  }

  return (
    <div>
      <h2>HTTP POST Request</h2>
      <button onClick={sendPost}>Enviar POST</button>
    </div>
  );
}

export default PostExample;
```

Para App.js

```
import React from 'react';
import GetExample from './GetExample';
import PostExample from './PostExample';

function App() {
  return (
    <div>
      <h1>HTTP en React</h1>
      <GetExample />
      <PostExample />
    </div>
  );
}

export default App;
```

HTTP en React

Lista de usuarios (GET)

- Leanne Graham
- Ervin Howell
- Clementine Bauch
- Patricia Lebsack
- Chelsey Dietrich
- Mrs. Dennis Schulist
- Kurtis Weissnat
- Nicholas Runolfsdottir V
- Glenna Reichert
- Clementina DuBuque

HTTP POST Request

Código del GET