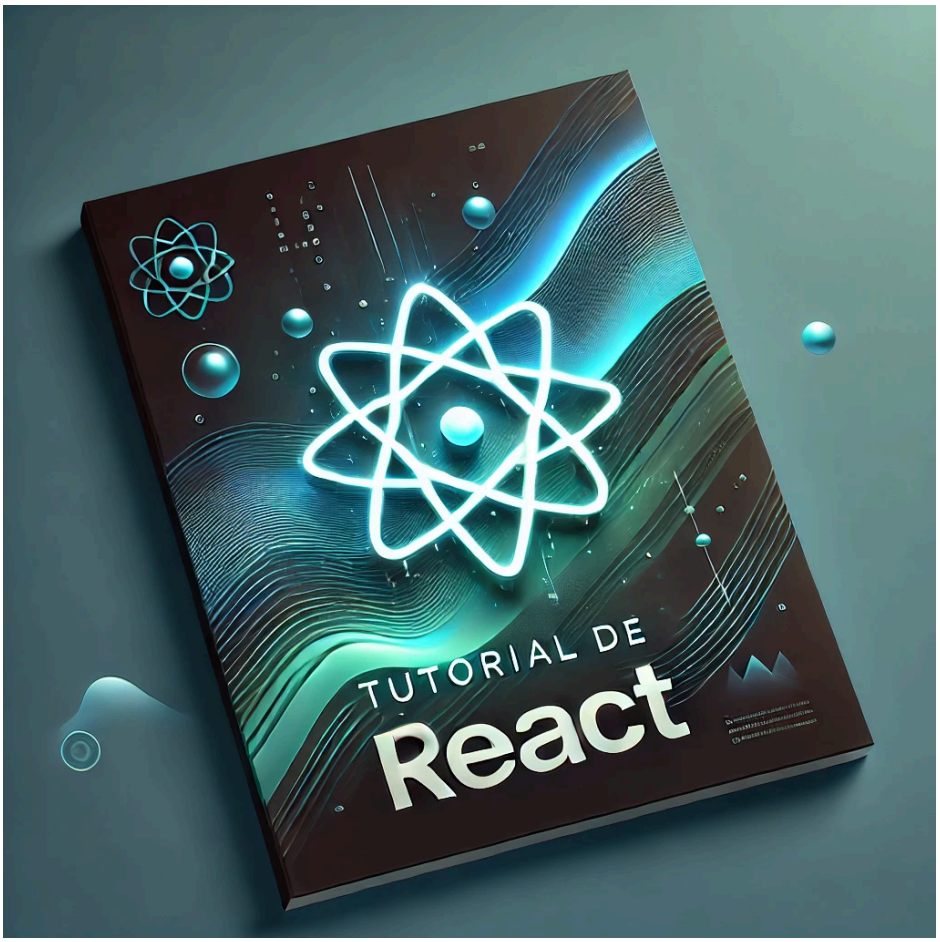
APLICACIONES DISTRIBUÍDAS EN INTERNET



Integrantes del grupo:

- Raúl López Arpa
- Adrián Bartel Prieto
- Paula Soriano Muñoz
- Rubén del Castillo Fuentes

1. Introducción a React.

React es una librería de JavaScript creada por Facebook en 2013, diseñada para desarrollar interfaces de usuario de manera eficiente y organizada. Desde su lanzamiento, ha ganado una gran popularidad entre los desarrolladores debido a su enfoque basado en componentes, su alto rendimiento y la facilidad con la que se integra en proyectos de diferentes tamaños.
El concepto principal de React se basa en la construcción de aplicaciones mediante componentes, que son bloques reutilizables de código encargados de representar diferentes partes de la interfaz. Esto facilita la organización del proyecto y el mantenimiento a lo largo del tiempo.

Principales características de React.

A continuación, se destacan las características principales de React:

1. Componentes:
   La estructura modular de React permite dividir una aplicación en componentes independientes. Cada componente representa una parte específica de la interfaz, como un formulario, un botón o una tarjeta, y puede reutilizarse en diferentes lugares del proyecto.
2. Virtual DOM:
   Para mejorar el rendimiento, React emplea un sistema llamado *DOM Virtual*, que actúa como una representación en memoria de la página web. Cuando algo cambia en la interfaz, React compara esta representación virtual con el *DOM real* y actualiza sólo los elementos que han cambiado, optimizando los tiempos de carga.
3. Flujo de datos unidireccional:
   En React, los datos se mueven en una sola dirección, desde los componentes “padre” hacia los “hijos”. Este enfoque ayuda a mantener un control claro sobre el flujo de información y reduce errores relacionados con la manipulación de datos.
4. JSX:
   Una de las características distintivas de React es JSX, una sintaxis que combina JavaScript y HTML en el mismo archivo. Esto permite escribir interfaces de forma más intuitiva y directa, integrando lógica y estructura en un mismo lugar.
5. Herramientas de desarrollo:
   React cuenta con extensiones oficiales para navegadores, como *React Developer Tools*, que facilitan la inspección y depuración de componentes. Esto simplifica la tarea de encontrar y corregir errores durante el desarrollo.
6. Comunidad activa:
   La comunidad de React es una de las más grandes y activas en el mundo del desarrollo web. Existen numerosos tutoriales, foros y recursos que facilitan el aprendizaje y la resolución de problemas.
7. Versatilidad:
   React no solo se limita a aplicaciones web. También puede utilizarse para desarrollar aplicaciones móviles mediante *React Native* o incluso aplicaciones de escritorio, lo que amplía sus posibilidades.

2. Configuración

Antes de comenzar a desarrollar con React, es necesario preparar el entorno de trabajo. Esta configuración inicial incluye la instalación de herramientas clave y la creación del primer proyecto React. A continuación, se explican los pasos principales:

Instalación de Node.js y npm

Node.js es una plataforma que permite ejecutar JavaScript fuera del navegador, y npm (Node Package Manager) es su gestor de paquetes. Ambos son necesarios para instalar y gestionar las dependencias de React.

Para comprobar si están instalados, se pueden usar los siguientes comandos en la terminal:

- ➢ *"node -v"*
- ➢ *"npm -v"*

Si no están instalados, se pueden descargar e instalar desde la página oficial de Node.js: https://nodejs.org/.

Crear un proyecto React con Create React App

La forma más común y sencilla de empezar con React es usando *Create React App*, una herramienta que configura automáticamente un proyecto básico con todas las dependencias necesarias. Para crear un nuevo proyecto, se ejecuta el siguiente comando en la terminal:

- ➢ *"npx create-react-app nombre-del-proyecto"*

Algunos puntos importantes sobre este comando son:

- *npx* ejecuta paquetes sin instalarlos globalmente.
- *create-react-app* es la plantilla base para proyectos React.
- *nombre-del-proyecto* es el nombre que se le dará a la carpeta del proyecto.

Una vez finalizado el proceso, se crea una estructura de carpetas que incluye archivos como *src* (código fuente), *public* (archivos públicos) y un archivo *package.json* con las dependencias.

Iniciar el proyecto

Para comenzar a trabajar con el proyecto recién creado, se deben seguir los siguientes pasos:

1. Acceder a la carpeta del proyecto:
   - ➢ *"cd nombre-del-proyecto"*
2. Iniciar el servidor de desarrollo:
   - ➢ *"npm start"*

Esto abre automáticamente el navegador en *http://localhost:3000*, mostrando una página de bienvenida de React.

Configuración de un editor de texto

Aunque se puede usar cualquier editor de texto, Visual Studio Code (VS Code) es una de las opciones más populares para trabajar con React. Algunas extensiones útiles para React en VS Code incluyen:

- ES7+ React/Redux/React-Native snippets: para autocompletar código.
- Prettier: para formatear el código.

- Bracket Pair Colorizer: para visualizar mejor las parejas de llaves y paréntesis.

Dependencias adicionales (opcional)

En algunos proyectos, puede ser útil instalar dependencias adicionales, como bibliotecas para manejar rutas o estilos. Algunos ejemplos son:

- React Router: para gestionar la navegación entre páginas:
  - ➢ *"npm install react-router-dom"*
- Styled Components: para trabajar con estilos:
  - ➢ *"npm install styled-components"*

Con esta configuración básica, el entorno está listo para empezar a desarrollar con React. A partir de aquí, se puede modificar la estructura predeterminada del proyecto o añadir nuevos componentes según las necesidades.

3. JSX: Sintaxis de React.

¿Qué es JSX?

JSX, que significa JavaScript XML, es una extensión de la sintaxis de JavaScript que se utiliza en React para describir cómo debería verse la interfaz de usuario. Aunque parece HTML, en realidad es una mezcla de HTML y JavaScript, diseñada para que escribir componentes sea más intuitivo y fácil de leer.
El objetivo principal de JSX es permitir que se defina la estructura de la interfaz directamente en el código, integrando lógica y diseño en un mismo lugar.

¿Por qué usar JSX?

JSX no es obligatorio en React, pero es ampliamente utilizado porque simplifica mucho el desarrollo. Algunas de sus ventajas son:

- Permite combinar lógica y presentación en un mismo archivo, lo que facilita la gestión del código.
- Es más legible que el uso directo de funciones de JavaScript como *React.createElement*.
- Ofrece una experiencia similar a escribir HTML, lo que reduce la curva de aprendizaje para quienes ya conocen este lenguaje.

Cómo funciona JSX

JSX convierte el código que parece HTML en funciones de JavaScript. Esto lo hace posible gracias a herramientas como Babel, que traducen el JSX en código que el navegador entiende.
Por ejemplo, el siguiente JSX:

```jsx
const saludo = <h1>¡Hola, mundo!</h1>;
```

Se traduce internamente en:

```jsx
const saludo = React.createElement('h1', null, '¡Hola, mundo!');
```

Reglas básicas de JSX

1. Solo un elemento principal:
   En JSX, todo debe estar envuelto en un único elemento principal. Si se necesitan múltiples elementos, se pueden agrupar utilizando un contenedor como *'div'* o una etiqueta vacía (<>).
   Ejemplo:

```jsx
return (
    <>
      <h1>Título</h1>
      <p>Este es un párrafo</p>
    </>
  );
```

2. Clases y atributos;
   En JSX, la palabra *class* (usada en HTML) se reemplaza por *className* para evitar conflictos con las palabras clave de JavaScript.
   Ejemplo:

```
<h1 className="titulo">¡Hola, React!</h1>
```

3. Interpolación de JavaScript:
   Se pueden usar llaves *'{}'* para incluir código JavaScript dentro del JSX.
   Ejemplo:
   ```
   const nombre = "Juan";
   return <h1>¡Hola, {nombre}!</h1>;
   ```
4. Etiquetas cerradas:
   Todas las etiquetas deben estar cerradas, incluso aquellas que en HTML no lo necesitan, como *<img>* o *<input>*.
   Ejemplo:
   ```
   <img src="logo.png" alt="Logo" />
   ```
5. Expresiones válidas:
   Dentro de *'{}'* solo se pueden usar expresiones que devuelven un valor. Por ejemplo, no se pueden usar instrucciones *if*, pero sí operadores ternarios.
   Ejemplo:
   ```
   const esUsuario = true;
   return <h1>{esUsuario ? "Bienvenido" : "Por favor, regístrate"}</h1>;
   ```

<u>Ventajas de usar JSX</u>

- Legibilidad: Al combinar HTML y JavaScript, el código es más intuitivo y fácil de entender.
- Componentización: Facilita la creación de componentes reutilizables.
- Integración con la lógica: Permite agregar dinámicamente elementos o estilos basados en datos.

4. Componentes en React.

Introducción.

En React, los componentes constituyen la base para la construcción de las interfaces. Cada componente representa una parte de la interfaz de usuario (UI) que puede reutilizarse y personalizarse. Esto permite dividir una aplicación en bloques reutilizables y fáciles de mantener, favoreciendo la organización y escalabilidad del código.
Existen dos tipos principales de componentes:

- **Componentes funcionales:** Son funciones de JavaScript que devuelven una interfaz en JSX. Son más simples y se utilizan mucho en React actualmente gracias a los Hooks (más tarde hablaremos de ellos) que permiten manejar estado y efectos secundarios en este tipo de componentes.
- **Componentes de clase:** Son clases de JavaScript que extienden `React.Component`. Son menos comunes desde la introducción de los Hooks, y se utilizan para manejar estados y ciclos de vida en versiones previas de React.

Componente funcional.

Como bien hemos mencionado anteriormente, los componentes funcionales son funciones de JavaScript que retornan JSX.

prueba > src > components > Saludo.jsx > ...

```jsx
function Saludo() {
    return <h1>¡Hola, esta es una prueba de componente funcional!</h1>;
}

export default Saludo;
```

Este componente devuelve un encabezado con el texto: “Hola, esta es una prueba de componente funcional”, y puede ser reutilizada en cualquier lugar de la aplicación.

Componente de clase.

Un componente de clase tiene la siguiente estructura básica:

I. Define una clase que extiende `React.Component`.
II. Incluye un método obligatorio llamado `render()`, que devuelve el JSX a renderizar.
III. Puede recibir props, que se puede acceder usando `this.props`.

```jsx
prueba > src > components > SaludoClase.jsx > ...
import React from 'react';  6.9k (gzipped: 2.7k)
import PropTypes from 'prop-types';  1.4k (gzipped: 774)

class SaludoClase extends React.Component {
  render() {
    return <h2>Hola, {this.props.nombre}!</h2>;
  }
}

SaludoClase.propTypes = {
  nombre: PropTypes.string.isRequired,
};

export default SaludoClase;
```

Se ha definido una clase 'SaludoClase' que extiende `React.Component`, lo que le otorga todas las funcionalidades básicas de un componente de React.
Utiliza el método `render()`, que es obligatorio y se encarga de devolver el JSX que se muestra en la interfaz. En este caso, devuelve un encabezado <h2> que incluye el valor de `this.props.nombre`.
'PropTypes' es una librería que permite validar las props que un componente recibe. Aquí se comprueba que la prop nombre debe ser un string y es requerida.

Uso de Props en componentes.

Los componentes pueden recibir datos desde sus componentes padres mediante las props. Estas permiten personalizar los componentes y hacerlos más reutilizables. Ambos tipos de componentes (funcionales como de clase) pueden utilizar props.

Las props funcionan de la siguiente manera:

Estas se pasan al componente como si fueran atributos HTML.

Dentro del componente, las props pueden accederse para personalizar lo que se muestra.

Props en componentes funcionales.

En un componente funcional, las props se reciben como un argumento de una función.

```jsx
prueba > src > components > Bienvenida.jsx > ...
import PropTypes from 'prop-types';  1.4k (gzipped: 774)

function Bienvenida(props) {
  return <h1>¡Hola, {props.nombre}!</h1>;
}

// Validamos de las props
Bienvenida.propTypes = {
  nombre: PropTypes.string.isRequired,
};

export default Bienvenida;
```

Props es un objeto que contiene todas las propiedades pasadas al componente desde el componente padre. En este caso, `props.nombre` contiene el valor pasado a través de la prop 'nombre', y se accede al valor de 'nombre' usando `{props.nombre}` dentro del JSX.

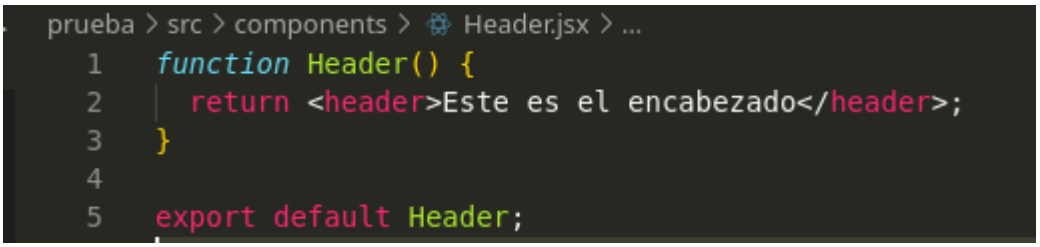
Props en componentes de clase.

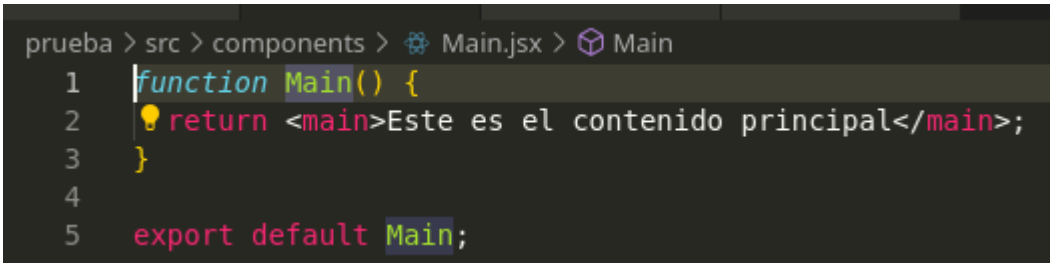
En los componentes de clase, las props también permiten pasar datos al componente, pero se acceden usando `this.props`.

Las props están disponibles como `this.props` dentro de un componente de clase. En este caso, `this.props.nombre` contiene el valor de la prop 'nombre'. Como se observa en el ejemplo, en los componentes de clase no es necesario pasarle el atributo props a la función render.

Anidación de componentes.

React facilita la creación de jerarquías mediante la anidación de componentes, es decir, un componente puede contener otros componentes dentro de su estructura, permitiendo la construcción de interfaces más complejas a partir de elementos simples.





prueba > src > components > Footer.jsx > ...

```jsx
function Footer() {
  return <footer>Este es el pie de página</footer>;
}

export default Footer;
```

prueba > src > App.jsx > default

```jsx
import './App.css';
import Header from './components/Header';
import Main from './components/Main';
import Footer from './components/Footer';


function App() {
  return (
    <div>
      <Header />
      <Main />
      <Footer />
    </div>
  );
}

export default App;
```

Este es el encabezado
Este es el contenido principal
Este es el pie de página

Cada componente (Header, Main y Footer) representa una parte específica de la interfaz. Esto hace que el código sea más organizado y fácil de entender.
En el componente App, los tres componentes definidos en las imágenes, están anidados, lo que crea una jerarquía clara y estructurada.

5. Estado y ciclo de vida en componentes de clase.

Introducción.

El estado (state) es una característica fundamental en los proyectos React, que permite que los componentes sean dinámicos y respondan a las interacciones del usuario. A diferencia de las props, que son valores que se reciben desde el componente padre y no pueden modificarse, el estado es interno al componente y puede ser modificado a lo largo del tiempo.

Por otro lado, el ciclo de vida de los componentes de clase describe las diferentes etapas por las que pasan desde que se crean, se actualizan, y finalmente, se eliminan. Estas etapas son esenciales para ejecutar acciones específicas, como limpiar recursos antes de que el componente desaparezca.

El estado y el ciclo de vida están estrechamente relacionados, ya que los cambios en el estado suelen provocar actualizaciones en el ciclo de vida del componente.

Manejo del estado.

El estado, como bien hemos mencionado anteriormente, es un objeto especial que los componentes de clase pueden usar para poder guardar y manejar datos internos que cambian con el tiempo. Estos datos pueden ser valores como un contador, el texto de un formulario o cualquier información que dependa de las acciones del usuario o de eventos externos.

Definir el estado.

El estado se define dentro del constructor de un componente de clase usando this.state. Este objeto contiene todas las variables que el componente necesita para funcionar.

El estado inicial contiene una variable ‘contador’, que inicia a 0. Este valor puede modificarse más adelante utilizando el método SetState.

SetState.

Para cambiar el estado de un componente, utilizamos el método SetState. Este método actualiza el estado y le indica a React que debe volver a renderizar el componente para reflejar los cambios en la interfaz.

```
incrementar = () => {
  this.setState((prevState) => ({
    contador: prevState.contador + 1,
  }));
};
```

Cuando el nuevo estado depende del estado anterior, es mejor utilizar una función. Esto nos asegura que `prevState` contiene el valor más reciente del estado, lo cual es útil cuando hay actualizaciones consecutivas.
Como el `setState` es asíncrono, las actualizaciones del estado no ocurrirán de manera inmediata. React agrupa las actualizaciones para optimizar el rendimiento, por ello, no debemos confiar en que `this.state` tenga el valor actualizado justo después de llamar a `SetState`.

Ciclo de vida de los componentes.

El ciclo de vida de un componente de clase en React describe las etapas por las que pasa un componente desde que se crea hasta que se elimina de la interfaz de usuario. Estas etapas permiten realizar acciones específicas en momentos concretos.
El ciclo de vida de un componente de clase se divide en tres fases principales:

I. **Montaje**.
II. **Actualización**.
III. **Desmontaje**.

Montaje.

Es cuando el componente se crea y se agrega al DOM por primera vez. Aquí es donde se inicializan el estado y las props, y se realizan acciones como cargar datos desde una API o configurar eventos.
Los métodos más utilizados en esta fase son:

1. **Constructor(props)**. Se llama al crear el componente, y se utiliza para inicializar el estado.
2. **ComponentDidMount()**. Se llama inmediatamente después de que el componente se monta en el DOM. Es ideal para realizar tareas como inicializar datos.

```
componentDidMount() {
  console.log('Componente montado');
}
```

Actualización.

Ocurre cada vez que cambian las props o el state del componente. Durante esta fase, React decide si debe volver a renderizar el componente para reflejar los cambios en la interfaz.
Los métodos más utilizados son:

1. **shouldComponentUpdate(nextProps, nextState).** Se llama antes de volver a renderizar el componente. Devuelve true en caso de que se deba actualizar, false en caso contrario.

```
shouldComponentUpdate(nextProps, nextState) {
  return nextState.contador !== this.state.contador;
}
```

2. **componentDidUpdate(prevProps, prevState).** Se llama después de que el componente se ha actualizado. Puede ser útil para realizar acciones basadas en los cambios hechos.
```
componentDidUpdate(prevProps, prevState) {
  console.log('El componente se actualizó');
}
```

Desmontaje.

Ocurre cuando el componente se elimina del DOM. Esta fase se utiliza para limpiar recursos, como eliminar eventos o detener procesos que ya no son necesarios.
El método más utilizado es:

1. **componentWillUnmount()**. Se llama justo antes de que el componente se elimine. Es útil para limpiar recursos, cancelar suscripciones..
```
componentWillUnmount() {
  console.log('Componente desmontado');
}
```

6. Hooks en React.

Introducción.

Los hooks son una de las funcionalidades más importantes en React. Su propósito es permitir que se utilicen el estado y otras características de React en componentes funcionales, algo que anteriormente solo era posible en los componentes de clase.
Antes de los Hooks, los componentes de clase eran necesarios para manejar el estado y métodos del ciclo de vida. Esto hacía que el código fuera más complejo y menos intuitivo. Con los Hooks, los componentes funcionales llegan a ser tan potentes como los de clase, pero con una sintaxis más simple y directa.

Ventajas de los Hooks.

1. **Simplificación de código**.
   Los hooks eliminan la necesidad de escribir clases, lo que reduce la complejidad y hace que el código sea más limpio y fácil de entender.
2. **Mejor reutilización de lógica**.
   Con los hooks, se pueden reutilizar piezas de lógica de estado entre componentes mediante funciones personalizadas (custom hooks).
3. **Compatibilidad con React moderno**.
   Los Hooks son la solución recomendada para nuevos proyectos y son compatibles con toda la API de React actual.

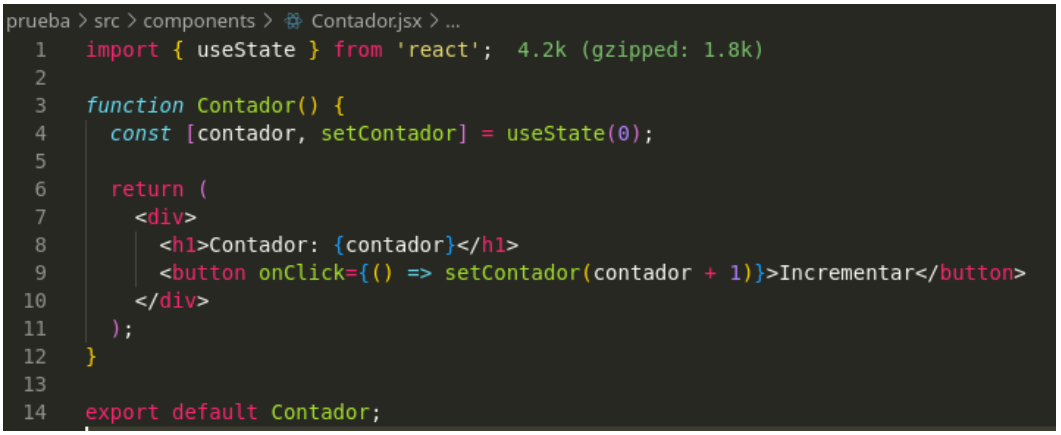
UseState.

El hook `useState` permite manejar el estado en componentes funcionales. Se pueden crear variables de estado y actualizarlas de manera reactiva, es decir, cada vez que cambien, React actualizará automáticamente la interfaz de usuario.

Utilización.

1. Importar el Hook useState.
   ```
   import { useState } from 'react';
   ```
2. Declarar el estado:
   Se llama a useState dentro del componente funcional, pasando como argumento el valor inicial del estado. Esto devuelve un array con dos elementos:
   - ➢ La variable de estado
   - ➢ Una función para actualizar el valor.

   ```
   const [contador, setContador] = useState(0);
   ```



El ejemplo es un componente funcional con un contador que se incrementa al hacer clic en un botón.

El contador se inicializa a 0, definido en `useState(0)`. Cada vez que se haga clic en el botón, se llama a `setContador` con un nuevo valor (contador + 1). Esto actualiza el estado y React vuelve a renderizar el componente mostrando el nuevo valor.

UseEffect.

Permite manejar efectos secundarios en componentes funcionales. Estos efectos secundarios son acciones que interactúan con el mundo exterior o realizan tareas adicionales como llamadas a APIs, configuraciones de eventos, limpiezas de recursos, manipulaciones directas del DOM...

En los componentes de clase, estas tareas se realizaban con métodos del ciclo de vida como `componentDidMount`, `componentDidUpdate` y `componentWillUnmount`. Con useEffect, estas tareas se manejan en un solo lugar.

Utilización.

1. Importar el Hook.
   `import { useEffect } from 'react';`
2. Declarar el efecto:
   Se define dentro del componente funcional. UseEffect toma como argumento una función que se ejecutará después de renderizar el componente.

```
useEffect(() => {
  console.log('El componente se ha renderizado');
});
```

   El segundo argumento de useEffect es un array de dependencias que determina cuando se ejecuta el efecto.
   - ➢ Sin dependencias: Si no se pasa un segundo argumento, el efecto se ejecuta en cada renderizado.
   - ➢ Con dependencias: Si se pasa un array vacío, el efecto solo se ejecuta una vez, y es después del montaje inicial del componente.
   - ➢ Dependiendo de variables: Si se incluyen variables en el array, el efecto se ejecutará cada vez que se modifiquen esas variables.

prueba > src > components > Contador.jsx > ...

```jsx
import  { useState, useEffect } from 'react';  4.3k (gzipped: 1.9k)

function ContadorConEfecto() {
  const [contador, setContador] = useState(0);

  useEffect(() => {
    document.title = `Contador: ${contador}`;
  }, [contador]);

  return (
    <div>
      <h1>Contador: {contador}</h1>
      <button onClick={() => setContador(contador + 1)}>Incrementar</button>
    </div>
  );
}

export default ContadorConEfecto;
```

Limpieza de efectos.

Algunos efectos necesitan limpiarse cuando el componente se desmonta o antes de que se vuelva a ejecutar nuevamente el efecto. Para ello, se devuelve una función dentro de useEffect.

Ejemplo:

prueba > src > components > Temporizador.jsx > Temporizador > useEffect() callback

```jsx
import { useState, useEffect } from 'react';  4.3k (gzipped: 1.9k)

function Temporizador() {
  const [segundos, setSegundos] = useState(0);

  useEffect(() => {
    // Configura un temporizador que incrementa los segundos cada segundo
    const interval = setInterval(() => {
      setSegundos((prevSegundos) => prevSegundos + 1);
    }, 1000);

    // Limpieza del temporizador cuando el componente se desmonta
    return () => {
      clearInterval(interval);
      console.log('Temporizador limpiado');
    };
  }, []); // Se ejecuta solo al montar el componente

  return (
    <div>
      <h1>Segundos transcurridos: {segundos}</h1>
      <button onClick={() => setSegundos(0)}>Reiniciar</button>
    </div>
  );
}

export default Temporizador;
```

Otros Hooks útiles.

Además de useState y useEffect, React incluye otros Hooks que son muy útiles para manejar tareas específicas. Vamos a explicar algunos de los más importantes.

useContext.

Permite acceder a un contexto creado con `React.createContext`. Es ideal para compartir datos entre componentes sin tener que pasar props manualmente a través de cada nivel de la jerarquía.
Es decir, es como un atajo para compartir datos entre componentes que están en diferentes niveles de la jerarquía. Normalmente, para pasar datos desde un componente padre a un hijo, utilizamos props. Pero si tienes muchos niveles intermedios, pasar props a través de todos ellos puede ser un lío.
'useContext' lo simplifica, de manera que permite acceder directamente al dato que necesitas, sin importar en qué nivel está el componente.

useReducer.

Este Hook es como una versión mejorada de useState, pensado para manejar estados más complicados. Por ejemplo, si tienes varios valores en el estado que cambian dependiendo de diferentes acciones, 'useReducer' organiza todo esto con una función llamada reducer.

useRef.

Crea una referencia mutable que puede mantenerse entre renderizados. Se utiliza principalmente para acceder a elementos del DOM o almacenar valores que no activan una nueva renderización. También sirve para guardar valores que no cambian el diseño, como un temporizador o un contador que no afecta la interfaz.

7. Manejando Eventos

Una de las funcionalidades importantes para cualquier aplicación interactiva, es la capacidad de controlar acciones del usuario. Para ello, React proporciona una forma sencilla y concreta para detectar esas acciones como un click, desplazamientos, movimientos del mouse, etc...

React utiliza un sistema de eventos sintéticos que es una capa de abstracción sobre los eventos del navegador. Éstos, garantizan que los eventos se comporten de manera consistente en todos los navegadores, lo que facilita la escritura de código entre diferentes entornos.

Cómo manejar eventos en React

Un manejador de eventos es una función que aplica funcionalidades específicas a esas acciones anteriormente descritas por el usuario. Este framework usa camelCase para realizar el nombramiento de eventos, y como característica propia, se pasa una función como manejador en vez de un string.

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

En este caso, la función onClick, llama a la función activateLasers que realizará una acción concreta.

Binding de eventos en componentes de clase

En componentes de clase, se utiliza el método bind() para vincular una función a un evento. Esta práctica es la más antigua presente en React para vincular funciones con los eventos.

Por ejemplo, el siguiente código utiliza ese método para vincular una función a un evento click:

```
class Boton extends React.Component {
    constructor(props) {
      super(props);
      this.handleClick = this.handleClick.bind(this);
    }
    handleClick() {
      alert('Hola!');
    }
    render() {
      return (
        <button onClick={this.handleClick}>Click aquí</button>
      );
    }
  }
```

Uso de eventos en componentes funcionales

En componentes funcionales, podemos usar el operador flecha para vincular una función a un evento.

```
const Boton = () => {
    return (
      <button onClick={(e) => alert(e.target.id)}>Click aquí</button>
    );
  };
```

En el ejemplo anterior, la función alert se vincula al evento onClick del button. La función recibe un parámetro, que es el objeto que obtiene información sobre el evento. En este caso, se utiliza el parámetro para obtener el ID del elemento que ha desencadenado el evento.

Las funciones vinculadas a eventos en componentes funcionales se ejecutan en el contexto del componente. Esto significa que tienen acceso a todas las propiedades y métodos del componente.

Argumentos en eventos

En React, los manejadores de eventos pueden recibir argumentos adicionales además del propio evento. Esto es útil cuando se necesita pasar información adicional al manejador de eventos.

```jsx
// BotonEvento.jsx
export default function BotonEvento({ id }) {
  function handleClick(id, event) {
    console.log('Botón clickeado con id:', id);
    console.log('Tipo de evento:', event.type);
  }

  return (
    <button onClick={(event) => handleClick(id, event)}>
      Clicame
    </button>
  );
}
```

En este ejemplo, la función handleClick recibe tanto el argumento id como el objeto event. Al hacer clic en el botón, se imprimirá el id del botón y el tipo de evento en la consola.
Además de pasar argumentos, es fundamental en muchas situaciones prevenir el comportamiento predeterminado de los eventos.

Propagación de eventos

En React, la propagación de eventos sigue un modelo similar al del DOM nativo, donde los eventos pueden propagarse a través de una fase de captura (desde el elemento raiz hacia el objetivo) y una fase de burbujeo (desde el objetivo hacia la raiz).

Por defecto, React maneja los eventos de burbujeo, pero es posible controlarlo usando el prefijo on...Capture para concretar la funcionalidad.

```jsx
// MiEventoCaptureBurbujeo.jsx
export default function MiEventoCaptureBurbujeo() {
    function manejarClickCaptura(event) {
        console.log('Fase de captura: ', event.target);
    }

    function manejarClickBurbujeo(event) {
        console.log('Fase de burbujeo: ', event.target);
    }

    return (
        <div onClickCapture={manejarClickCaptura} onClick={manejarClickBurbujeo}>
            <button>Clic aquí</button>
        </div>
    );
}
```

Por otro lado, también es posible detener la propagación de un evento en cualquier fase utilizando el método event.stopPropagation(). Esto es útil para evitar que un evento manejado en un componente hijo se propague a un componente padre.

8. Renderizado condicional y listas

En react se pueden utilizar diferentes técnicas para renderizar condicionalmente componentes. Una técnica común es utilizar la sentencia if. Por ejemplo el siguiente código utiliza la sentencia if para renderizar un componente <P> solo si la variable mostrar tiene el valor true. Además, podemos hacer uso de && para añadir más condicionados a la renderización del componente:

```
const ComponenteCondicional = () => {
    const mostrar = true;
    return (
      <div>
        {mostrar && <P>Este componente se renderizará</P>}
      </div>
    );
  };
```

Por otro lado, existe el hook map() el cual permite renderizar una lista de elementos de forma iterativa. De esta forma, podemos crear páginas web dinámicas, por ejemplo:

```
const ListaNumeros = () => {
    const numeros = [1, 2, 3, 4, 5];
    return (
      <div>
        {numeros.map((numero, index) => (
          <p key={index}>{numero}</p>
        ))}
      </div>
    );
  };
```

9. Formularios en React

Para crear un formulario, podemos hacer uso de un componente funcional que haga uso del hook useState, con intención de almacenar los datos del formulario de forma más sencilla y que así podamos rastrear y actualizar los valores de los campos del formulario.

Para capturar los cambios en los campos del formulario, podemos definir un manejador de cambio handleChange que actualice el estado del formulario cada vez que un usuario introduzca información. Además, podemos definir un manejador de envío handleSubmit que nos permita realizar acciones como enviar los datos a un servidor.

```jsx
import React, { useState } from "react";

const Formulario = () => {
  // Estado para almacenar los datos del formulario
  const [formData, setFormData] = useState({
    nombre: "",
    email: "",
  });

  // Manejador de cambios en los inputs
  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData({ ...formData, [name]: value });
  };

  // Manejador de envio del formulario
  const handleSubmit = (e) => {
    e.preventDefault();
    // Aquí se puede realizar acciones con los datos del formulario
    console.log("Datos del formulario:", formData);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Nombre:
        <input
          type="text"
          name="nombre"
          value={formData.nombre}
          onChange={handleChange}
        />
      </label>
      <br />
      <label>
        Email:
        <input
          type="email"
          name="email"
          value={formData.email}
          onChange={handleChange}
        />
      </label>
      <br />
      <button type="submit">Enviar</button>
    </form>
  );
};
```

10. Estilos y CSS en React

Aplicar estilos en los componentes en React es sencillo, ya que podemos usar el atributo **style** en un componente y poner estilos sencillos. En caso de querer utilizar estilos más complejos, que empeoren la legibilidad del código, se pueden importar también clases de CSS externas para usarlos, y así seguir teniendo un código limpio. Veamos un ejemplo con ambas formas:

- Ejemplo con el atributo style:

```
import React from 'react';

const MiComponente = () => {
    return (
        <div style={{ color: 'yellow', backgroundColor: 'black', fontSize: '30px' }}>
            Contenido de mi componente con estilos sencillos
        </div>
    );
};
```

- Ejemplo con la clase CSS importada:

```
import React from 'react';
import "../styles/estilos.css";

const MiComponente = () => {
    return (
        <div className="mi-componente">
            Contenido de mi componente con estilos sencillos
        </div>
    );
};
```

También podemos usar otras dos formas de importar estilos que ofrecen soluciones óptimas para manejar estilos en React: CSS Modules o la librería **styled-components**.

CSS Modules es una técnica de encapsulamiento de estilos en archivos CSS para componentes específicos, evitando posibles conflictos entre los nombres de las clases, ya que se "especifica" el fichero desde el que queremos sacar el estilo. Suponiendo que tenemos un estilo llamado miComponente en *estilos.css* y en *otros-estilos.css*:

```
import React from 'react';
import styles from '../styles/estilos.css';
import '../styles/otros-estilos.css';

const MiComponente = () => {
    return (
        <div className={styles.miComponente}>
            Contenido de mi componente con estilos sencillos
        </div>
    );
};
```

Por otro lado, styled-components es una popular biblioteca para escribir componentes con JavaScript en los archivos de React, con un enfoque más dinámico al crear las etiquetas como si fueran constantes del código con un nombre personalizado (y pudiendo reutilizarlo a lo largo del código). Para poder ejecutarlo tendremos que instalar la biblioteca con el comando **npm install styled-components**.

```
import React from 'react';
import styled from 'styled-components';

const MiDiv = styled.div`
    color: yellow;
    background-color: black;
    fontSize: 30px;
`;

const MiComponente = () => {
    return (
        <MiDiv>Contenido de mi componente con estilos sencillos</MiDiv>
    );
};
```

11. Rutas y navegación usando ReactRouter

Como en muchos otros frameworks, React tiene una librería muy popular que facilita la gestión de rutas y navegación en las aplicaciones, consiguiendo hacer una SPA (Single Page Application). Para poder usarlo tendremos que instalar la librería con el comando **npm install react-router-dom**, luego tendremos que crear un fichero llamado, por ejemplo, *index.js*.

Dentro del fichero *index.js* tenemos que implementar el router:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter as Router } from 'react-router-dom';
import App from './App';

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

Ahora, en nuestro componente principal, podemos definir rutas a los componentes de React con las funciones que proporciona React Router **Router**, **Routes** y **Route**, además de usar **Link** para añadir unos links de navegación (esto habría que editarlo en un futuro).

```
import React from 'react';
import Inicio from './Inicio';
import Sobre from './Sobre';
import Contacto from './Contacto';

import { BrowserRouter as Router, Routes, Route, Link } from 'react-router-dom';

const App = () => {
    return (
        <Router>
            {/* <!-- Navegación --> */ }
            <nav>
                <ul>
                    <li><Link to="/">Inicio</Link></li>
                    <li><Link to="/sobre">Sobre</Link></li>
                    <li><Link to="/contacto">Contacto</Link></li>
                </ul>
            </nav>

            {/* <!-- Rutas --> */ }
            <Routes>
                <Route path="/" element={<Inicio />} />
                <Route path="/sobre" element={<Sobre />} />
                <Route path="/contacto" element={<Contacto />} />
            </Routes>
        </Router>
    );
};

export default App;
```

También se podría crear un Header para los links de navegación, lo que dejaría un archivo principal mucho más limpio. Quedaría de la siguiente forma:

- Código del Header:

```jsx
import React from 'react';
import { Link } from 'react-router-dom';

const Header = () => {
    return (
        <header>
            <h1>Header tutorial de React</h1>
            <nav>
                <ul><Link to="/">Inicio</Link></ul>
                <ul><Link to="/sobre">Sobre</Link></ul>
                <ul><Link to="/contato">Contato</Link></ul>
            </nav>
        </header>
    );
};

export default Header;
```

- Nuevo código del fichero principal:

```jsx
import React from 'react';
import Inicio from './Inicio';
import Sobre from './Sobre';
import Contacto from './Contacto';
import Header from './Header';

import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

const App = () => {
    return (
        <Router>
            {/* <!-- Navegación --> */ }
            <Header />

            {/* <!-- Rutas --> */ }
            <Routes>
                <Route path="/" element={<Inicio />} />
                <Route path="/sobre" element={<Sobre />} />
                <Route path="/contacto" element={<Contacto />} />
            </Routes>
        </Router>
    );
};

export default App;
```

12. Llamadas a APIs

Algo muy útil a la hora de desarrollar código es hacer llamadas a APIs, ya sea tanto propias como externas, y en React hay varias formas de hacer esto. Las más comunes son **fetch** y **axios**. Igualmente, con ambas formas tenemos que manejar el estado del componente durante diferentes fases del ciclo de vida, para eso se usa **loading** para cuando están cargando los datos, **error** para cuando hay un error cargando los datos y **data** para cuando se han obtenido los datos con éxito.

fetch

Es una función nativa de JavaScript usada para realizar peticiones HTTP y que es compatible con la mayoría de navegadores modernos. La ventaja que tiene es que proporciona una interfaz simple y promisificada (transformación de una función que acepta un callback a un función que devuelve una promesa). Veamos un código sencillo de cómo se utiliza:

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

const 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(err){
                console.error(err);
            }
        };

        fetchData();
    }, []);

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

Axios

Es una biblioteca externa a React, pero simplifica las solicitudes HTTP y proporciona una sintaxis limpia. Para poder usarla primero hay que instalarla con **npm install axios**. Veamos el mismo ejemplo de antes pero con Axios:

```jsx
import React, { useState, useEffect } from 'react';
import axios from 'axios';

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

    useEffect(() => {
        const fetchData = async () => {
            try{
                const response = await axios.get('https://api.example.ciom/data');
                setData(response.data);
            }
            catch(err){
                console.error(err);
            }
        };

        fetchData();
    }, []);

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

Como vemos, es muy muy parecido a como se usa fetch, pero un poco más claro y sin tener que exportar la respuesta obtenida a JSON.

13. Bibliografía

- Manual de React: https://desarrolloweb.com/manuales/manual-de-react.html
- Proyecto de React: https://www.digitalocean.com/community/tutorials/how-to-set-up-a-react-project-with-create-react-app-es
- JSX en React: https://elblogdelprogramador.com/posts/jsx-en-react-aprende-como-usarlo-con-ejemplos-practicos/#gsc.tab=0
- React: https://es.legacy.reactjs.org/docs/
- UseState: https://react.dev/reference/react/useState
- UseEffect: https://react.dev/reference/react/useEffect
- Hooks:https://es.react.dev/reference/react/hooks
- Estilos en React: https://platzi.com/blog/react-css/
- React Router: https://reactrouter.com/home
- Función JS fetch: https://lenguajejs.com/javascript/peticiones-http/fetch/
- Axios en React: https://www.freecodecamp.org/espanol/news/como-usar-axios-con-react/
- Tutorial React: https://certidevs.com/curso-react