



Gobierno del  
**CHACO**

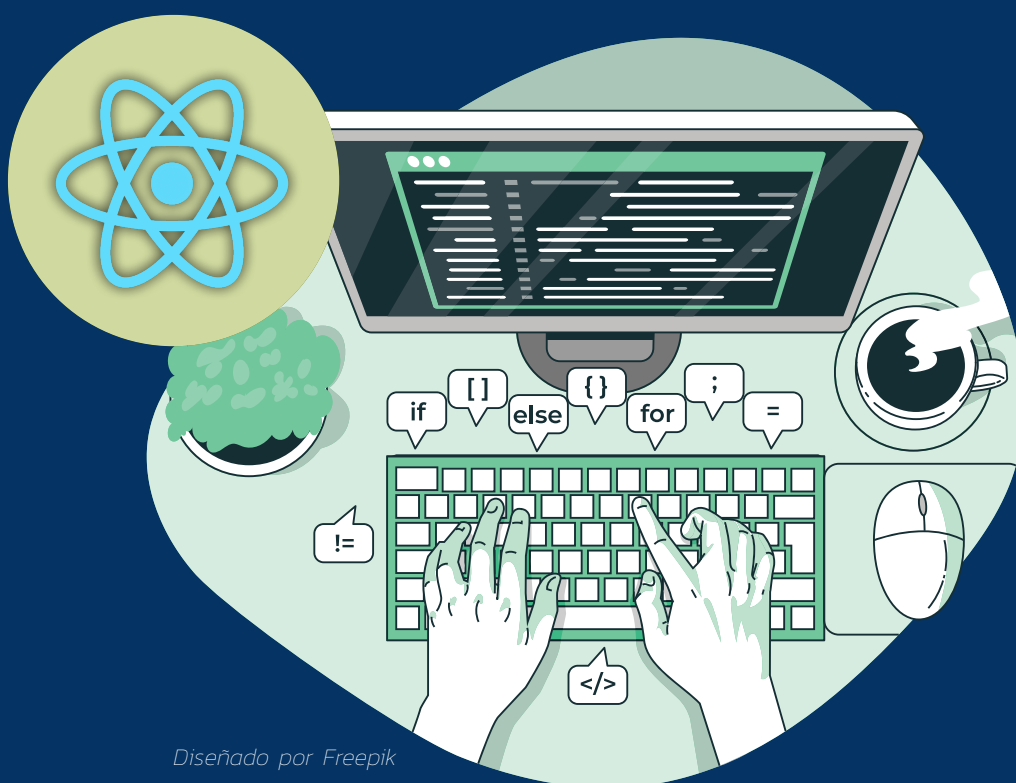
Ministerio  
de la Producción y el Desarrollo  
Económico Sostenible



**INFORMATARIO**

## ETAPA 3: ESPECIALIZACIONES

# REACT



*Diseñado por Freepik*

*Apunte N° 2*

# FUNDAMENTOS DE REACT

# 2

## Api de React

**React es una biblioteca de JavaScript desarrollada por Meta que permite construir interfaces de usuario.**

Está enfocada en la creación de componentes reutilizables, que representan piezas independientes de la interfaz. Estos componentes se combinan para construir aplicaciones dinámicas y modernas.




React se divide en paquetes para modularizar sus funcionalidades y adaptarse a diferentes contextos:

### react:

- Este paquete contiene las funciones esenciales de React, como `createElement`, y herramientas para construir y describir la interfaz de usuario. Es la base que abstrae cómo se estructuran los componentes.

### react-dom:

- `react-dom`: Este paquete es específico para aplicaciones web. Proporciona métodos para interactuar con el DOM real, como `ReactDOM.render`, que monta los componentes React en elementos HTML.

-  Web → Usa `react-dom`
-  Móvil → Usa `react-native`
-  VR → Usa `react-vr`

### ¿Por qué esta separación?

La separación existe para mantener la modularidad y adaptabilidad de React. Si querés u-

sar React en un entorno distinto al DOM, como React Native (para móviles) o React VR (para realidad virtual), el paquete `react` sigue siendo el mismo, pero cambia el paquete que interactúa con la plataforma específica (`react-dom`, `react-native`, etc.). Esto hace que React sea más flexible y fácil de extender.

En React, `createElement` es una función que permite crear elementos de manera declarativa, similar a cómo interactuamos con métodos tradicionales del DOM, pero con ciertas diferencias clave.

### Usando `createElement`

El método recibe tres parámetros principales:

- Tipo de elemento:** La etiqueta HTML (`'div'`, `'p'`, `'ul'`, etc.).
- Propiedades (props):** Un objeto que define atributos como `className`, eventos (`onClick`, etc.), o cualquier otra configuración. Si no hay, se pasa `null`.
- Hijos:** Puede ser un texto, otros elementos, o una lista de ellos.

Ejemplo:

```
const titulo = React.createElement(
  'h1',
  { className: 'titulo' },
  '¡Hola, Mundo!'
);
```

Esto define un encabezado `<h1>` con la clase `titulo` y el texto "¡Hola, Mundo!".

## Diferencias y similitudes con el DOM tradicional

Con los métodos del DOM como **document.createElement**:

### Creación de elementos:

```
const titulo = document.createElement("h1");
titulo.className = "titulo";
titulo.textContent = "¡Hola, Mundo!";
```

Aquí necesitás manipular el objeto directamente, asignando atributos y contenido de manera imperativa.

### Comparación clave:

- **React** gestiona elementos como estructuras inmutables, lo que significa que no modificás el objeto directamente. En cambio, se describe el estado inicial del elemento, y React se encarga de renderizarlo.
- **DOM** permite modificar el objeto resultante en cualquier momento, pero eso puede llevar a inconsistencias si no se gestiona bien.

## Elementos anidados

Con **createElement**, podés crear estructuras complejas de manera declarativa:

```
const lista = React.createElement(
  'ul',
  null,
  React.createElement('li', null, 'Elemento 1'),
  React.createElement('li', null, 'Elemento 2')
);
```

En el DOM tradicional, esto requiere múltiples pasos:

```
const lista = document.createElement('ul');
const li1 = document.createElement('li');
li1.textContent = 'Elemento 1';
lista.appendChild(li1);

const li2 = document.createElement('li');
li2.textContent = 'Elemento 2';
lista.appendChild(li2);
```

Con React, la estructura es más concisa, y el resultado es el mismo.

## Ventaja principal

**createElement** permite definir elementos de manera declarativa y estructurada, eliminando la necesidad de múltiples pasos o mutaciones, como ocurre en el DOM. Esto hace que el código sea más claro y mantenible, especialmente en proyectos grandes.

## Conclusión

El método **createElement** en React utiliza un enfoque imperativo, describiendo paso a paso la estructura de los elementos a crear. Esto se asemeja a cómo se opera con los métodos tradicionales del DOM, pero con la ventaja de integrarse en el flujo declarativo que React implementa en su proceso de renderizado. Aunque requiere especificar cada detalle, **createElement** es fundamental para comprender la base de cómo React construye y organiza elementos, ofreciendo flexibilidad y control al crear interfaces dinámicas.

## JSX

**JSX es una extensión de sintaxis que permite escribir HTML dentro de JavaScript. Aunque parece similar a HTML, tiene diferencias clave en cómo se maneja la estructura y las propiedades.**

### Características principales de JSX

**1. Sintaxis similar a HTML:** JSX permite crear elementos de manera intuitiva usando una sintaxis que recuerda a HTML. Esto facilita la creación de interfaces al poder escribir la estructura visual de los componentes dentro de los archivos JavaScript.

**2. Cierre de etiquetas:** En JSX, todas las etiquetas deben ser cerradas, incluso aquellas que en HTML pueden estar auto-cerradas, como `<img />` o `<input />`. Esta regla asegura que el código sea consistente y no cause errores de sintaxis.

**3. Atributos en camelCase:** En JSX, los atributos HTML como `class` y `for` se cambian por `className` y `htmlFor`, respectivamente. Esto es debido a que `class` y `for` son palabras reservadas en JavaScript.

**4. Expresiones en JSX:** Dentro de JSX, podés usar llaves `{}` para insertar expresiones de JavaScript. Esto incluye variables, funciones, operaciones matemáticas o incluso componentes.

**5. Funciones y Condiciones en JSX:** Podés incluir lógicas condicionales o bucles en JSX usando las expresiones de JavaScript. Por ejemplo, para mostrar algo dependiendo de una condición, se utiliza el operador ternario:

### Características principales de JSX

JSX no es código JavaScript estándar y no puede ser interpretado directamente por el navegador. Por eso, necesita ser transformado a código JavaScript puro antes de ejecutarse. Esta transformación es gestionada por herramientas como Babel, que convierten el JSX en llamadas a `React.createElement`.

#### Ejemplo del proceso de compilación:

```
// El siguiente JSX
const elemento = <h1>Hola, mundo</h1>;

// Babel lo transforma a
const elemento = React.createElement('h1', null, 'Hola, mundo');
```

#### ¿Qué hace este código resultante?

- El primer argumento (`'h1'`) especifica el tipo de elemento.
- El segundo argumento (`null`) corresponde a las props del elemento.
- Los argumentos siguientes (`'Hola, mundo'`) son los hijos del elemento.

**Ventaja de la compilación:**

*Del ejemplo en la página anterior:*

*Esta abstracción permite que React sea independiente del entorno. Por ejemplo, para React Native, las llamadas a `createElement` generan componentes móviles en lugar de nodos del DOM.*

jsx

```
const Componente = () => (
  <>
    <h1>Título</h1>
    <p>Texto del párrafo.</p>
  </>
);
```

**JSX Anidado**

JSX permite crear estructuras complejas anidando elementos unos dentro de otros. Esto es esencial para construir interfaces más avanzadas, donde múltiples elementos se agrupan para formar componentes.

**Ejemplo de anidamiento:**

```
const componente = (
  <div>
    <h1>Título principal</h1>
    <p>Este es un párrafo dentro de un contenedor.</p>
    <ul>
      <li>Elemento 1</li>
      <li>Elemento 2</li>
    </ul>
  </div>
);
```

En este ejemplo, todos los elementos dentro del `div` están anidados, y React lo procesa como un árbol jerárquico..

**Fragments**

Los Fragments permiten devolver múltiples elementos desde un componente sin necesidad de envolverlos en un contenedor adicional, como un `div`. Esto ayuda a mantener el DOM limpio y evita la creación de nodos innecesarios.

**Uso básico de Fragments:****¿Qué hace el Fragment?**

React agrupa el `h1` y el `p` sin agregar un contenedor extra al DOM. Esto es útil en componentes que necesitan devolver múltiples elementos.

También puedes usar `React.Fragment` en lugar de la sintaxis corta.

```
const Componente = () => (
  <>
    <li>Elemento 1</li>
    <li>Elemento 2</li>
  </>
);

const Componente = (
  <React.Fragment>
    <h1>Título</h1>
    <p>Texto del párrafo.</p>
  </React.Fragment>
);
```

**Conclusión**

JSX es una parte fundamental de React, ya que proporciona una manera de escribir la interfaz de usuario de forma declarativa y estructurada dentro de JavaScript. Las características como



la interpolación de expresiones, el uso de operadores como el spread de props, y la capacidad de anidar y agrupar elementos a través de Fragments hacen que la escritura de componentes sea más flexible y eficiente. Aunque se ve como HTML, JSX tiene reglas específicas que lo hacen más potente y fácil de integrar con JavaScript.

## Componentes

***En React, los componentes son funciones que te permiten dividir la interfaz de usuario en piezas reutilizables.***

Cada componente es una función que acepta un objeto llamado **props** y devuelve algo que React puede renderizar: más elementos de React, cadenas de texto, números o incluso **null**.

### Definición simple de un componente

Un componente es una función que toma props como entrada y devuelve un elemento renderizable.

Aquí tienes un ejemplo básico:

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

Puedes usar este componente como cualquier otro elemento de React, pasándole datos a través de **props**:

```
<Saludo nombre="Iván" />
```

### Props: Dinamismo en los componentes

Las **props** (abreviatura de propiedades) son la

forma en que los componentes obtienen datos. Se pasan desde el componente "padre" al "hijo". Esto hace que los componentes sean flexibles y reutilizables.

En el ejemplo anterior, **nombre** es una prop. Puedes cambiar su valor cada vez que uses el componente:

```
<Saludo nombre="Maxi" />
<Saludo nombre="Diego" />
```

### Comportamiento renderizable

El resultado de un componente puede ser cualquier cosa que React considere "renderizable". Esto incluye:

- **Elementos React:** Lo más común, como **<h1>** o **<div>**.
- **Texto o números:** Útiles para renderizar valores dinámicos.
- **null:** Si no hay nada que mostrar, puedes devolver **null**.

### Ejemplo práctico:



```
function MostrarTexto(props) {
  if (!props.texto) {
    return null; // No renderiza nada si texto es falso
  }
  return <p>{props.texto}</p>;
}
```

Usando este componente:

```
<MostrarTexto texto="Este es un mensaje" />
<MostrarTexto texto="" /> { /* No se renderiza nada */ }
```

## Conclusión

Los componentes son el núcleo de React. Al aceptar **props** y devolver resultados renderizables, te permiten construir interfaces dinámicas, modulares y predecibles. Esta simplicidad en su diseño es lo que hace que React sea tan poderoso y fácil de usar en nuestros proyectos.

## TypeScript

**TypeScript mejora la experiencia con React al proporcionar seguridad en los tipos y detección de errores en tiempo de desarrollo.**

### 1. Cómo tipar props

Las props en React son los datos que un componente recibe. Con TypeScript, puedes definirlas explícitamente para evitar errores.

Ejemplo básico:

En este ejemplo:

- **nombre** es obligatorio y tiene que ser de tipo **string**.
- **edad** puede ser **undefined** o **number**.

```
type SaludoProps = {
  nombre: string;
  edad?: number; // Opcional
};

function Saludo(props: SaludoProps) {
  return (
    <h1>
      Hola, {props.nombre} {props.edad && ` (${props.edad})` }
    </h1>
  );
}
```





## 2. Narrow types (Acotación de tipos)

TypeScript puede “acotar” un tipo amplio a uno más específico basado en verificaciones.

Ejemplo:

Aquí, TypeScript entiende que dentro del bloque `if`, `mensaje` es `null`, y fuera del bloque es una `string`. Esto evita errores al trabajar con tipos más amplios.

```
type SaludoProps = {
  mensaje: string | null;
};

function MostrarMensaje({ mensaje }: SaludoProps) {
  if (mensaje === null) {
    return <p>No hay mensaje.</p>;
  }
  return <p>{mensaje}</p>;
}
```

## 3. Tipos derivados (Derived types)

Los tipos derivados son útiles para evitar redundancia en la definición de tipos.

Ejemplo:

Aquí, el tipo `Datos` se deriva directamente de la estructura de `datos`, eliminando duplicación y errores en la sincronización de tipos.

```
const datos = {
  nombre: "Iván",
  edad: 25,
};

type Datos = typeof datos;

function MostrarDatos(props: Datos) {
  return (
    <p>
      Nombre: {props.nombre}, Edad: {props.edad}
    </p>
  );
}
```





#### 4. Default props (Props predeterminadas)

Puedes definir valores predeterminados para props en componentes con TypeScript.

Ejemplo:

El valor **"Haz clic aquí"** es la prop predeterminada si no se pasa **label**. Esto garantiza que el componente funcione sin requerir todas las props explícitamente.

```
type BotonProps = {
  label?: string;
};

function Boton({ label = "Haz clic aquí" }: BotonProps) {
  return <button>{label}</button>;
}
```

#### 5. Reducir duplicación

Con TypeScript, puedes evitar redundancia al reutilizar tipos y estructuras.

Usando **Pick**, extraemos solo las propiedades necesarias del tipo **Usuario**.

Ejemplo con **Pick** y **Omit**:

```
type Usuario = {
  id: number;
  nombre: string;
  email: string;
};

type UsuarioPreview = Pick<Usuario, "nombre" | "email">;

function MostrarUsuario({ nombre, email }: UsuarioPreview) {
  return (
    <p>
      {nombre} ({email})
    </p>
  );
}
```

## 6. El operador satisfies

El operador **satisfies** asegura que una estructura de datos cumpla con un tipo específico, sin imponer restricciones en su uso posterior. Este operador es especialmente útil cuando defines objetos literales y deseas que TypeScript valide solo las partes necesarias.

Ejemplo:

```
const configuracion = {
  modo: "oscuro",
  duracion: 300,
} satisfies { modo: "oscuro" | "claro"; duracion: number };

// configuracion es un objeto que cumple con las reglas del tipo,
// pero TypeScript permite que el uso del objeto sea más flexible.
```

## Conclusión

Usar TypeScript en React eleva la calidad del código al prevenir errores y proporcionar una documentación viva a través de los tipos. Desde tipar props hasta técnicas avanzadas como **satisfies**, TypeScript te permite escribir componentes más seguros y mantenibles, reduciendo la duplicación y facilitando la colaboración en equipo.

## Estilos

***En React, existen diversas formas de aplicar estilos a una aplicación. Cada enfoque tiene ventajas y desventajas dependiendo del tamaño del proyecto, la complejidad de los estilos y las preferencias del equipo.***

### 1. Inline Styles (Estilos en línea)

Los estilos en línea se definen directamente en los elementos como objetos de JavaScript. Esta técnica es simple y adecuada para estilos rápidos y específicos. (Ver ejemplo siguiente).

#### Ventajas:

- Escopo garantizado (no hay colisiones de nombres).
- Directo y fácil de usar.

**Desventajas:**

- Difícil de reutilizar y mantener.
- Carece de pseudoclases como `:hover`.

```
function Boton() {
  const estilo = { backgroundColor: "blue", color: "white", padding: "10px" };
  return <button style={estilo}>Haz clic aquí</button>;
}
```

## 2. CSS Tradicional

Puedes usar hojas de estilo externas y simplemente importarlas en tus componentes.

Veamos los siguientes ejemplos:

**Archivo `styles.css`:**

```
.boton {
  background-color: blue;
  color: white;
  padding: 10px;
}
```

**Componente React:**

```
import "./styles.css";

function Boton() {
  return <button className="boton">Haz clic aquí</button>;
}
```

**Ventajas:**

- Soporte para todas las características de CSS como animaciones, pseudoclases, y media queries.

**Desventajas:**

- No hay aislamiento de estilos; puede haber colisiones de clases.

## 3. CSS Modules

Los módulos CSS garantizan que los estilos sean locales a un componente, evitando colisiones de nombres:

**Archivo `Boton.module.css`:**

```
.boton {
  background-color: blue;
  color: white;
  padding: 10px;
}
```

**Componente React:**

```
import styles from "./Boton.module.css";

function Boton() {
  return <button className={styles.boton}>Haz clic aquí</button>;
}
```

**Ventajas:**

- Estilos aislados automáticamente.
- Perfecto para componentes reutilizables.

**Desventajas:**

- Aunque aíslan los estilos y reducen la probabilidad de colisiones, pueden hacer más difícil aplicar estilos globales consistentes.

## 4. Styled Components (CSS-in-JS)

Styled Components es una biblioteca que permite escribir CSS directamente dentro de componentes utilizando sintaxis de template literals. Ejemplo:

```
import styled from "styled-components";

const Boton = styled.button`
  background-color: blue;
  color: white;
  padding: 10px;
`;

function App() {
  return <Boton>Haz clic aquí</Boton>;
}
```

**Ventajas:**

- Aislamiento total de estilos.
- Posibilidad de usar lógica de JavaScript en los estilos.

**Desventajas:**

- Generar estilos dinámicos en tiempo de ejecución puede impactar negativamente el rendimiento, especialmente en aplicaciones grandes o complejas
- Puede complicarse en proyectos grandes. Manejar una gran cantidad de componentes estilizados puede resultar complicado, ya que los estilos quedan distribuidos en múltiples archivos JavaScript, dificultando la navegación y el mantenimiento.

## 5. Tailwind CSS

Tailwind es un framework de utilidades que te permite escribir estilos directamente en los atributos **className** usando clases predefinidas. Ejemplo:

```
function Boton() {
  return <button className="bg-blue-500 text-white p-2">Haz clic aquí</button>;
}
```

**Ventajas:**

- Las clases utilitarias predefinidas permiten construir interfaces rápidamente sin necesidad de escribir CSS personalizado.
- El sistema de diseño integrado (colores, tamaños, espaciados) asegura que todos los estilos sigan un esquema uniforme.

### Desventajas:

- Puede llevar tiempo acostumbrarse a las clases.
- Al incluir muchas clases en los atributos **className**, los archivos pueden volverse difíciles de leer y mantener.

## Conclusión

React ofrece múltiples formas de aplicar esti-

los, desde enfoques simples como estilos en línea y CSS tradicional, hasta soluciones avanzadas como CSS Module y Styled Components. La elección del método depende de las necesidades de tu proyecto y las preferencias del equipo. Para proyectos pequeños, los métodos tradicionales pueden ser suficientes, mientras que para proyectos grandes, herramientas como CSS Modules o CSS-in-JS son ideales para evitar colisiones.

## Formularios

### Manejo de Formularios en React

En React, trabajar con formularios no es muy distinto de lo que se hace con las APIs estándar del DOM y JavaScript. Sin embargo, React facilita el manejo y personalización de formularios con algunos conceptos clave.

### Manejo de eventos de envío (onSubmit)

El evento onSubmit se utiliza para manejar el

envío de formularios. Cuando un formulario se envía, React pasa un evento que contiene un atributo **currentTarget**, el cual hace referencia al nodo `<form>` en el DOM. Esto permite acceder a los elementos del formulario y extraer sus valores de diversas maneras, como mediante el objeto **FormData**.

Ejemplo básico:

```
<form
  onSubmit={(event) => {
    event.preventDefault(); // Evita el comportamiento predeterminado de recargar la página
    const formData = new FormData(event.currentTarget);
    console.log(formData.get("name")); // Obtener valores por nombre
  }}
>
  <label>
    Name:
    <input name="name" type="text" />
  </label><button type="submit">Submit</button>
</form>
```



*Esto es útil para evitar recargas de página, un comportamiento que era común en las aplicaciones web tradicionales pero que ya no se ajusta a las expectativas de aplicaciones modernas, donde se busca una experiencia fluida y eficiente.*

## Accesibilidad: Etiquetado de campos

Para mejorar la accesibilidad y experiencia del usuario, es importante etiquetar los campos correctamente. React proporciona dos enfoques:

### Envolver el campo con la etiqueta (**label**):

```
<label>
  Name:
  <input name="name" type="text" />
</label>
```

### Asociar el campo y la etiqueta mediante **htmlFor**:

```
<label htmlFor="nameInput">Name:</label>
<input id="nameInput" name="name" type="text" />
```

El atributo **htmlFor** en React corresponde al atributo **for** en HTML, y asocia la etiqueta con el campo.

## Prevención del comportamiento pre-determinado

En aplicaciones modernas, los formularios no deben causar recargas completas de página. Usando **event.preventDefault()** dentro del manejador de **onSubmit**, evitamos este comportamiento, asegurándonos de que la lógica personalizada, como la validación o el envío asíncronico, pueda ejecutarse sin interrupciones.

## Tipos de Input

HTML y React ofrecen múltiples tipos de **<input>** que mejoran la experiencia del usuario con validaciones y comportamientos específicos:

1. **text**: Entrada básica para texto.
2. **email**: Validación automática para correos electrónicos.
3. **password**: Oculta el texto ingresado.
4. **number**: Acepta solo números, con controles opcionales de incremento.
5. **date**, **time**, **datetime-local**: Selección de fechas y horas.
6. **file**: Subida de archivos.
7. **checkbox**: Entrada booleana (marcado o no).
8. **radio**: Opciones mutuamente exclusivas dentro de un grupo.
9. **range**: Control deslizante con valores numéricos.
10. **color**: Selector visual de colores.

[Acá encontrarás un listado con una explicación detallada de cada tipo de input](#)

## Conclusión

El manejo de formularios en React es directo gracias a su integración con las APIs estándar del DOM. Proporciona control sobre el comportamiento de los formularios, asegurando que las aplicaciones sean accesibles y eficientes. React no impone reglas estrictas, sino que amplía las capacidades del DOM nativo con una experiencia más controlada y fluida.



## Renderizando listas

***En React, renderizar listas es un patrón común cuando necesitas mostrar una colección de datos de forma dinámica en tu aplicación. Este proceso implica iterar sobre un conjunto de elementos (como un array) y crear componentes o elementos para cada uno.***

### Cómo Renderizar Listas

Para renderizar una lista, se utiliza típicamente el método `.map()` de los arrays. Este método permite transformar cada elemento en un componente o un elemento JSX.

Ejemplo básico:

```
const frutas = ["Manzana", "Banana", "Cereza"];

function ListaDeFrutas() {
  return (
    <ul>
      {frutas.map((fruta) => (
        <li key={fruta}>{fruta}</li>
      ))}
    </ul>
  );
}
```

Aquí:

- `frutas.map()`: Itera sobre el array `frutas`.
- `key={fruta}`: Cada elemento debe tener una clave única para que React pueda gestionar cambios de forma eficiente.

### La Propiedad key

Las claves (`key`) son un concepto crucial en React. Permiten a React identificar de manera única los elementos que han cambiado, añadido o eliminado.

Por qué `key` es importante:

- **Mejora el rendimiento:** Ayuda a React a evitar renderizados innecesarios.
- **Identificación precisa:** React usa la clave para rastrear el estado y el orden de los elementos.

Ejemplo de uso correcto de `key`:

```
const usuarios = [
  { id: 1, nombre: "Iván" },
  { id: 2, nombre: "Maxi" },
  { id: 3, nombre: "Mati" },
];

function ListaDeUsuarios() {
  return (
    <ul>
      {usuarios.map((usuario) => (
        <li key={usuario.id}>{usuario.nombre}</li>
      ))}
    </ul>
  );
}
```

Evita usar índices como claves, especialmente si el orden de los elementos puede cambiar, ya que puede generar comportamientos inesperados. Para más información sobre esto te invito a leer [un artículo que escribí hace un tiempo](#)

### Ejemplo con Componentes

Puedes usar componentes para renderizar lis-





tas cuando cada elemento requiere lógica o estructura adicional.

Por ejemplo:

```
function Usuario({ nombre }) {
  return <li>{nombre}</li>;
}

function ListaDeUsuarios() {
  const usuarios = ["Iván", "Maxi", "Mati"];
  return (
    <ul>
      {usuarios.map((nombre, index) => (
        <Usuario key={index} nombre={nombre} />
      ))}
    </ul>
  );
}
```

## Listas anidadas

Para listas jerárquicas, puedes anidar llamadas a `.map()`:

Ver debajo ejemplo básico de lista anidada:

```
const categorias = [
  { nombre: "Frutas", items: ["Manzana", "Banana", "Cereza"] },
  { nombre: "Verduras", items: ["Zanahoria", "Lechuga"] },
];

function ListaDeCategorias() {
  return (
    <div>
      {categorias.map((categoria) => (
        <div key={categoria.nombre}>
          <h2>{categoria.nombre}</h2>
          <ul>
            {categoria.items.map((item) => (
              <li key={item}>{item}</li>
            ))}
          </ul>
        </div>
      ))}
    </div>
  );
}
```

## Errores comunes al renderizar listas

- **Falta de key:** No usar claves o usar claves incorrectas puede generar errores o comportamientos inesperados.
- **Índices como key:** Usar índices puede ser problemático si los elementos cambian de orden o se eliminan.
- **Mapeo incorrecto:** Asegúrate de que el array que estás mapeando existe y tiene valores válidos.

## Conclusión

Renderizar listas en React es una técnica esencial y flexible que permite mostrar colecciones dinámicas de datos. Usar claves únicas con la propiedad `key` es crucial para mantener el rendimiento y el manejo eficiente del DOM. Además, separar la lógica en componentes reutilizables puede mejorar la organización y legibilidad del código.

## Vite

***Vite es una build tool diseñada para simplificar y acelerar el desarrollo y construcción de aplicaciones web modernas.***

La principal diferencia de Vite con herramientas tradicionales radica en que adopta un enfoque basado en módulos ES (ES Modules) nativos del navegador, lo que elimina gran parte de la sobrecarga inicial en proyectos grandes. A diferencia de empaquetadores como Webpack o Parcel, Vite no procesa todos los archivos de tu aplicación de inmediato. En cambio:

- **Durante el desarrollo:** Sirve directamente los módulos ES al navegador.
- **En producción:** Empaqueta los archivos para maximizar la eficiencia.

### Cómo funciona Vite en desarrollo

Cuando se ejecuta el servidor de desarrollo de Vite:

- Usa ESBUILD, un motor extremadamente rápido escrito en Go, para transformar el código TypeScript y JSX.
- Solo analiza y convierte el código que realmente se solicita en el navegador.
- Implementa un sistema de Hot Module Replacement (HMR) para que los cambios se reflejen instantáneamente.

**Ejemplo:** Cuando editas un archivo React y guardas, Vite reemplaza dinámicamente solo ese módulo afectado, en lugar de reconstruir todo el proyecto.

### Ventajas de Vite en el desarrollo

- **Inicio casi instantáneo:**  
Con ES Modules nativos, no necesitas construir un gran gráfico de dependencias al inicio. Incluso en proyectos complejos, el servidor está listo en segundos.
- **Hot Module Replacement (HMR):**  
Los cambios en CSS, componentes o lógica son visibles de inmediato sin recargar toda la página.
- **Soporte moderno por defecto:**  
Ofrece configuración mínima para TypeScript, JSX y frameworks populares.
- **Simplicidad:**  
Muchas decisiones están predeterminadas, pero puedes sobrescribirlas fácilmente.
- **Compatibilidad con herramientas actuales:**  
Compatible con Rollup plugins, CSS preprocessors (SASS, LESS), y más.

#### Ventajas de Vite

Aspecto	Vite	Webpack	Parcel
 Tiempo de inicio	Inmediato	Lento en proyectos grandes	Rápido inicial
 HMR	Ultra rápido	Moderado	Moderado
 Configuración inicial	Simple	Compleja	Sencilla
 Soporte ES Modules	Nativo	Limitado	Limitado



## Cómo funciona Vite en producción

Para construir una aplicación optimizada, Vite utiliza Rollup. Esto incluye:

- División de código (code splitting): Divide tu código en partes reutilizables para mejorar el rendimiento.
- Minificación eficiente: Comprime CSS y JS para reducir el peso final.
- Árbol de dependencias optimizado: Solo incluye en el bundle lo que realmente se utiliza.

## Configuración básica para iniciar un proyecto con Vite

### Instalación:

```
bash

npm create vite@latest my-app
cd my-app
npm install
npm run dev
```

### Estructura de Archivos:

```
├─ public/           # Archivos estáticos
├─ src/              # Código fuente
│   └─ main.js       # Entrada principal
│       └─ components/
├─ index.html        # Página principal
└─ vite.config.js    # Configuración
```

### Configuración básica (vite.config.js):

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';

export default defineConfig({
  plugins: [react()],
  server: {
    port: 4000,
    open: true,
  },
  build: {
    outDir: 'dist',
  },
});
```

## Comparativa de Vite con Webpack y Parcel

Aspecto	Vite	Webpack	Parcel
⌚ Tiempo de inicio	Inmediato	Lento en proyectos grandes	Rápido inicial
🔥 HMR (Hot Module Replacement)	Ultra rápido	Moderado	Moderado
🔧 Configuración inicial	Simple	Compleja	Sencilla
🌐 Soporte ES Modules	Nativo	Limitado	Limitado
🚀 Performance en Desarrollo	Alta	Media	Alta
🌱 Soporte de Plugins	Integración con Rollup	Gran ecosistema de plugins	Plugins integrados



## Ventajas y desventajas

### Ventajas:

1. **Rapidez extrema:** Ideal para proyectos con muchas dependencias.
2. **Facilidad de uso:** Configuración mínima.
3. **Escalabilidad:** Aunque está optimizado para proyectos pequeños y medianos, funciona bien en aplicaciones grandes.

### Desventajas:

1. **Dependencia de ESBUILD:** Aunque es rápido, está menos probado que Babel para transformaciones complejas

2. **Compatibilidad limitada con navegadores antiguos:** Necesitas herramientas como Polyfill para soporte completo.

3. **Ecosistema joven:** Aunque crece rápidamente, puede no tener tantos recursos como Webpack.

## Conclusión

Vite no es solo una herramienta para proyectos modernos; es un estándar emergente que redefine cómo desarrollamos y construimos aplicaciones web. Su velocidad, simplicidad y enfoque modular lo convierten en una opción ideal para desarrolladores que buscan maximizar su productividad sin sacrificar funcionalidad.