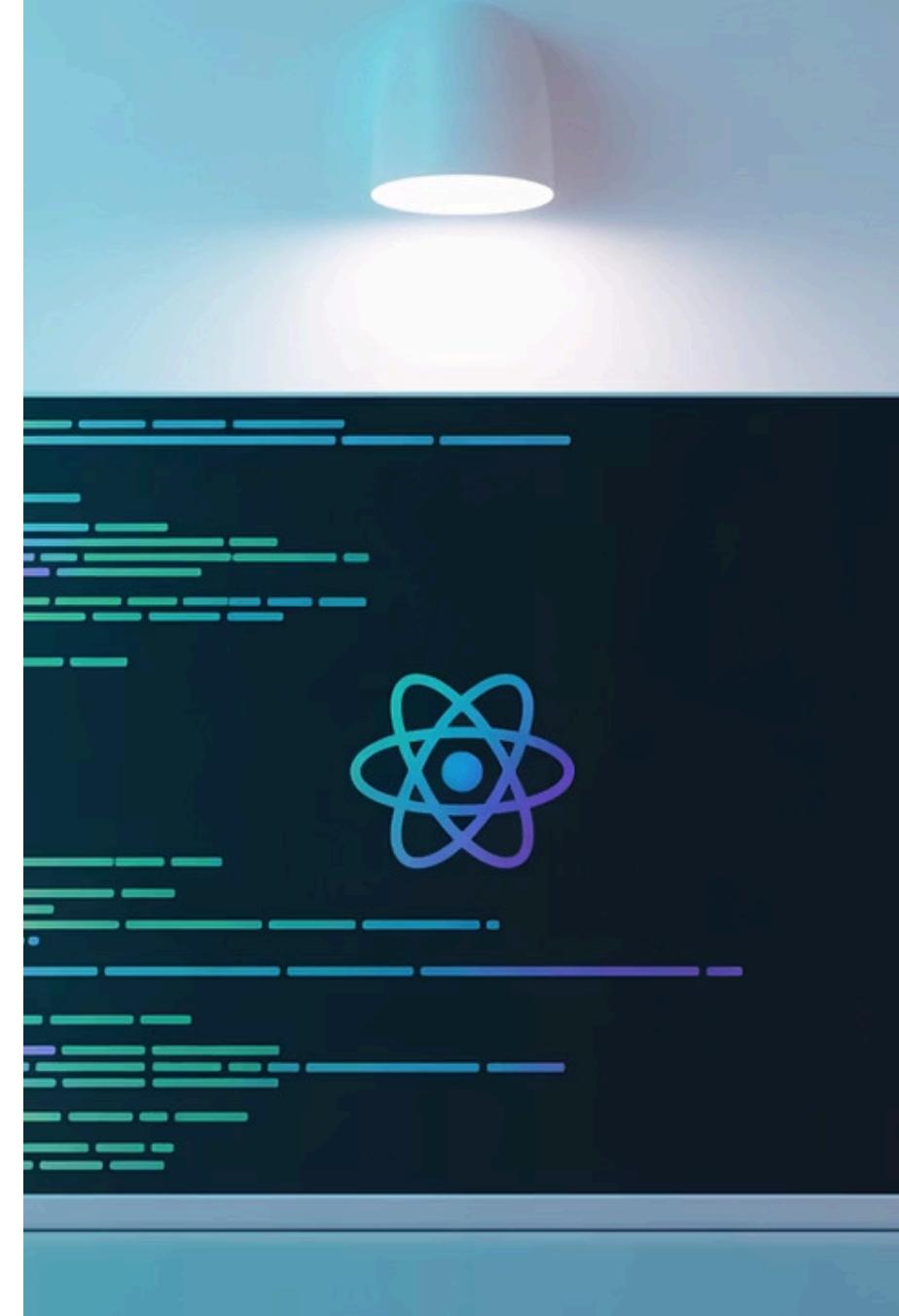


Introducción a React y Componentes



¿Qué veremos hoy?

- 1 Librerías y frameworks en frontend
- 2 React: ventajas y casos de uso
- 3 Componentes y reutilización
- 4 Creación de proyectos con Vite
- 5 Estructura de componentes
- 6 Props y ejemplos prácticos

Librerías y Frameworks Frontend

¿Por qué no seguir usando solo HTML, CSS y JavaScript?

Del desarrollo tradicional a las librerías

Desarrollo tradicional

- Archivos HTML, CSS y JS separados
- Manipulación directa del DOM
- Código difícil de mantener en proyectos grandes
- Mucho código repetido

Con librerías/frameworks

- Estructura organizada
- Manipulación abstracta y eficiente
- Mejor mantenimiento
- Reutilización de código
- Mayor productividad

Principales librerías y frameworks



React

Biblioteca para interfaces, desarrollada por Facebook. Usa un DOM virtual para renderizado eficiente.



Angular

Framework completo mantenido por Google. Incluye todo lo necesario para aplicaciones robustas.



Vue

Framework progresivo y accesible. Fácil de aprender y de integrar en proyectos existentes.

Cada uno tiene sus fortalezas, pero **React domina el mercado laboral** actualmente.

¿Por qué aprender React?



Ventajas de React



Enfoque declarativo

Describes **qué** quieres que se muestre, no **cómo** hacerlo. React se encarga de actualizar la interfaz de manera eficiente.



Virtual DOM

Representación en memoria del DOM real para minimizar las actualizaciones costosas.



Basado en componentes

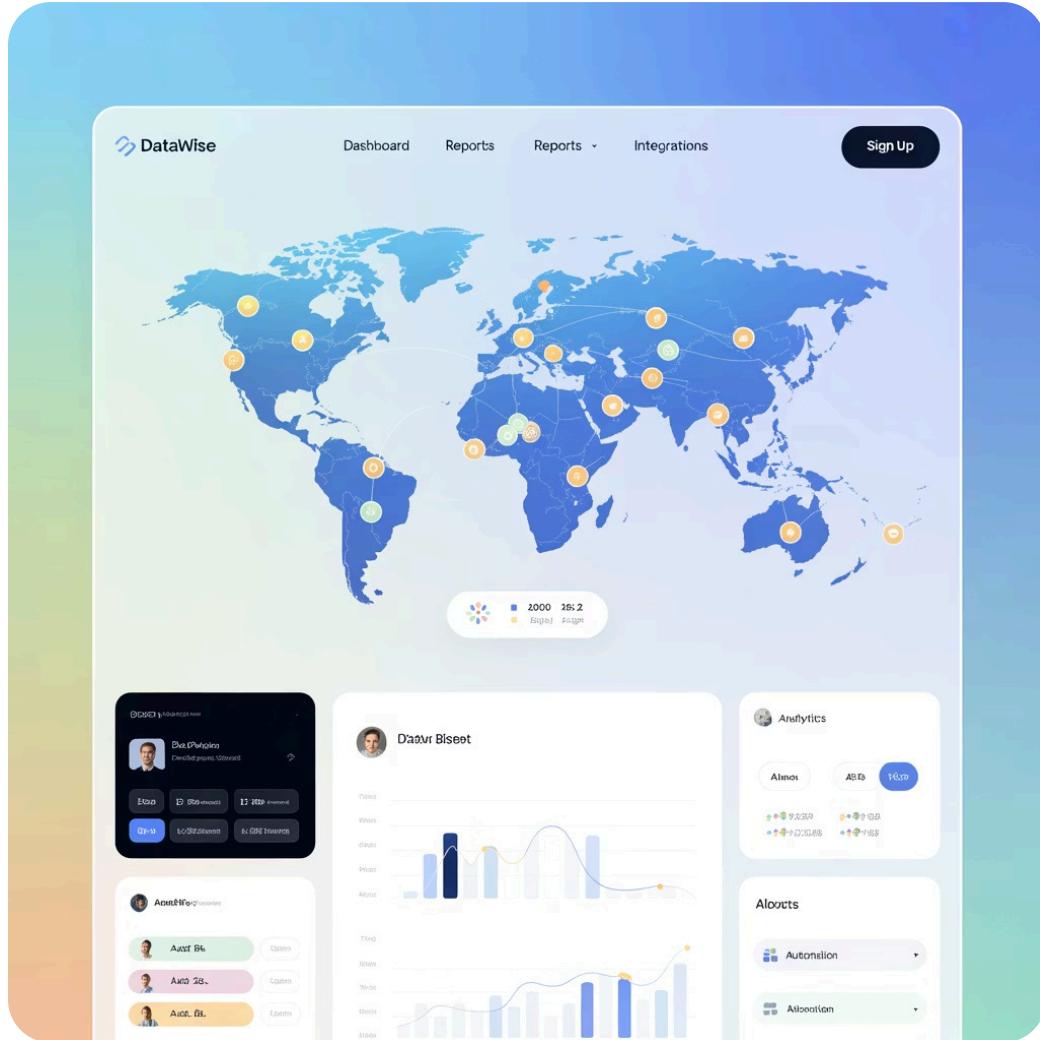
Crea piezas independientes y reutilizables que encapsulan su propio comportamiento.



Gran ecosistema

Amplia comunidad, numerosas bibliotecas complementarias y gran demanda laboral.

Casos de uso ideales



- **Aplicaciones de una sola página (SPA)**, donde la experiencia de usuario es fluida
- **Interfaces complejas e interactivas** con muchos estados
- **Aplicaciones con actualizaciones frecuentes** de datos
- **Equipos grandes** donde varios desarrolladores trabajan en componentes separados
- **Proyectos escalables** que crecerán con el tiempo

El concepto de componentes

La piedra angular de React

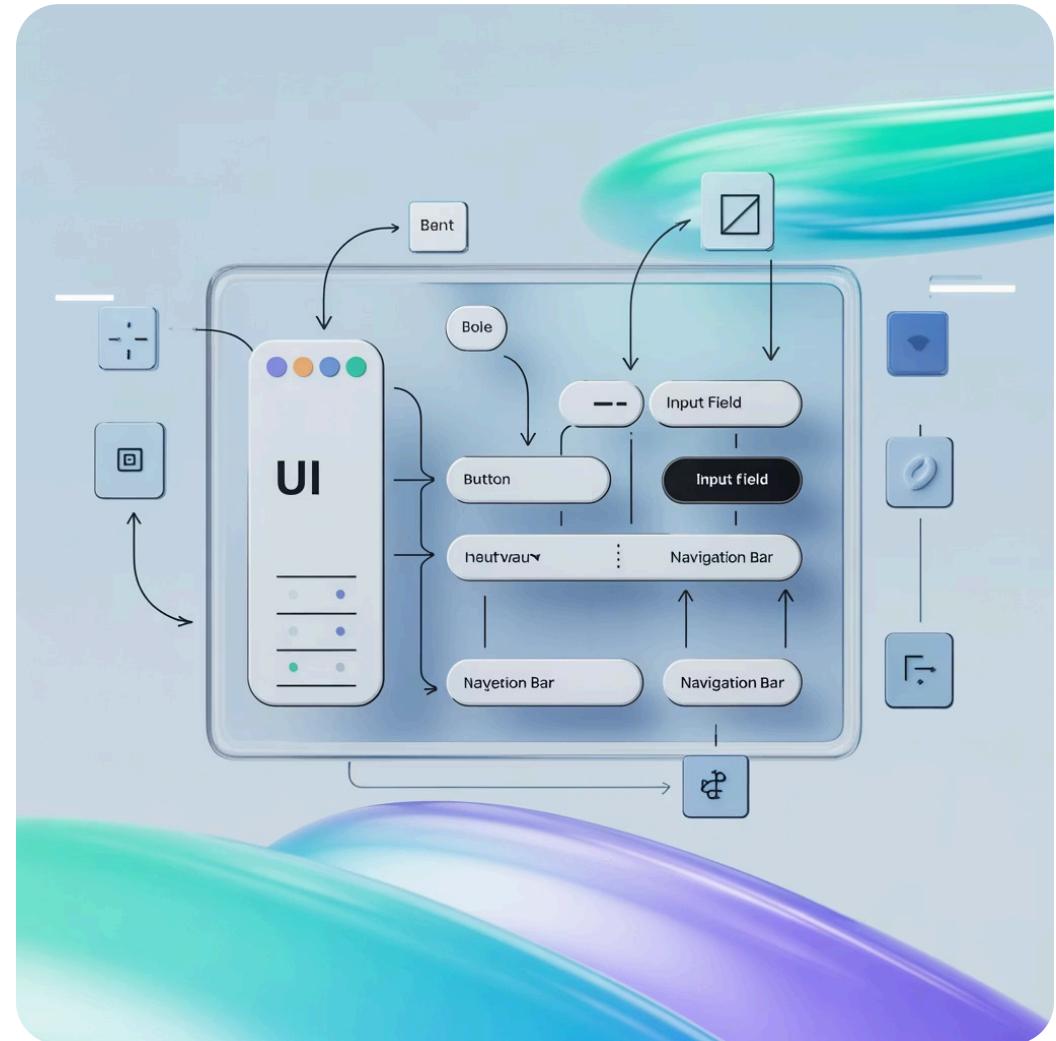
¿Qué es un componente?

Un componente es una pieza **independiente** y **reutilizable** de la interfaz de usuario.

Conceptualmente, los componentes son como **funciones de JavaScript** que:

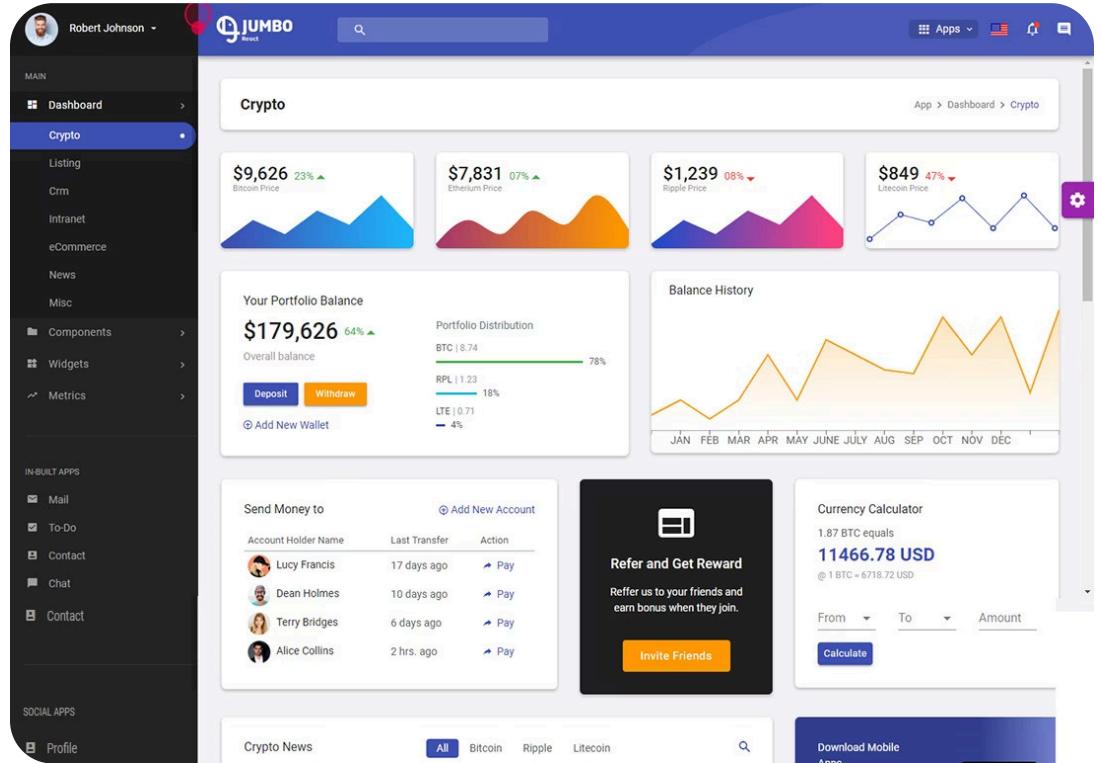
- Aceptan entradas (llamadas "props")
- Devuelven elementos React que describen lo que debe aparecer en pantalla

Puedes pensar en ellos como "piezas de LEGO" con las que construyes interfaces complejas.



Anatomía de una interfaz basada en componentes

En la imagen podemos ver cómo una página web se descompone en múltiples componentes reutilizables: navegación, tarjetas de producto, botones, etc.



Ventajas de los componentes



Reutilización

Crea un componente una vez y úsalos en múltiples lugares.

Modularidad

Divide problemas complejos en piezas más pequeñas y manejables.

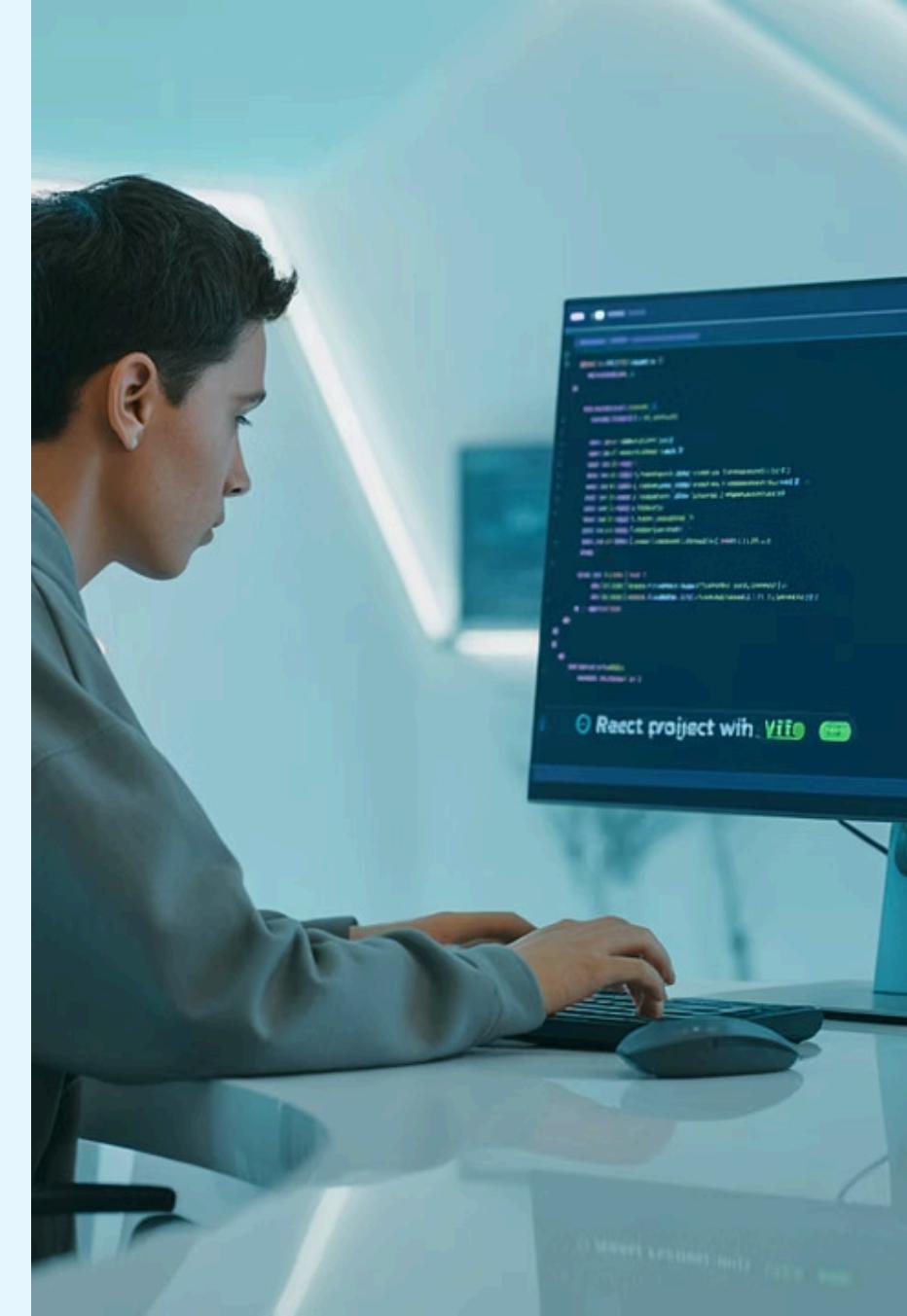
Mantenimiento

Actualiza un componente y el cambio se refleja en toda la aplicación.

- ⓘ Los componentes bien diseñados siguen el principio de **responsabilidad única**: cada componente debe hacer una sola cosa, pero hacerla bien.

Empezando con React

Creación de un proyecto con Vite



¿Qué es Vite?

Vite (pronunciado "vit") es una herramienta de compilación moderna que proporciona una experiencia de desarrollo más rápida y ágil.

Ventajas principales:

- Arranque instantáneo del servidor
- Actualizaciones rápidas al guardar cambios
- Optimizaciones integradas para producción
- Soporte para TypeScript, JSX, CSS, etc.



Creando un proyecto React con Vite

Abre tu terminal y ejecuta:

```
npm create vite@latest mi-app-react -- --template react  
cd mi-app-react  
npm install  
npm run dev
```

Esto creará una nueva aplicación React con la estructura básica necesaria para empezar.

- ❑ Asegúrate de tener Node.js (versión 14.18+ o 16+) instalado en tu sistema antes de ejecutar estos comandos.

Estructura de un proyecto React + Vite

```
mi-app-react/
├── node_modules/
├── public/
│   └── vite.svg
└── src/
    ├── assets/
    │   └── react.svg
    ├── App.css
    ├── App.jsx
    ├── index.css
    └── main.jsx
├── .gitignore
├── index.html
├── package.json
├── vite.config.js
└── README.md
```

main.jsx: Punto de entrada de la aplicación

App.jsx: Componente principal

assets/: Imágenes, iconos, etc.

public/: Archivos estáticos

package.json: Dependencias y scripts

vite.config.js: Configuración de Vite

Organización de componentes

Para un proyecto React escalable y fácil de mantener, organiza tus componentes de forma eficiente. Centraliza todos tus componentes reutilizables en una carpeta `components/` dentro de `src/`, y agrupa los archivos relacionados con cada componente en su propia subcarpeta. Esto mejora la organización y la reusabilidad.

```
src/
  └── components/
    ├── Header/
    │   ├── Header.jsx      # Código del componente
    │   ├── Header.css     # Estilos específicos
    │   └── index.js       # Archivo de barril para exportar
    ├── Button/
    │   ├── Button.jsx
    │   ├── Button.css
    │   └── index.js
    └── Card/
        ├── Card.jsx
        ├── Card.css
        └── index.js
  └── pages/          # Componentes de página/vista
  └── hooks/          # React Hooks personalizados
  └── utils/          # Funciones de utilidad
  └── App.jsx         # Componente raíz
  └── main.jsx        # Punto de entrada
```

Estructura de un componente React

Los bloques fundamentales

Componentes en React

Existen dos tipos principales de componentes en React:

Componentes funcionales

```
function Saludo(props) {  
  return (  
    <h1>¡Hola, {props.nombre}!</h1>  
    <p>Bienvenido a React</p>  
  );  
}
```

Funciones JavaScript que devuelven **elementos JSX**

Componentes de clase

```
class Saludo extends React.Component {  
  render() {  
    return (  
      <h1>¡Hola, {this.props.nombre}!</h1>  
      <p>Bienvenido a React</p>  
    );  
  }  
}
```

Clases ES6 que extienden React.Component

Actualmente se recomienda usar **componentes funcionales** con Hooks y dejar los de clase solo para mantenimiento de código antiguo.

¿Qué es JSX?

JSX (JavaScript XML) es una extensión de sintaxis para JavaScript que parece HTML pero permite incluir toda la potencia de JavaScript.

Características importantes:

- Permite escribir "HTML" dentro de JavaScript
- Es transformado a llamadas a `React.createElement()`
- Puedes incluir expresiones JavaScript entre llaves {}
- Usa camelCase para propiedades HTML (`className` en vez de `class`)

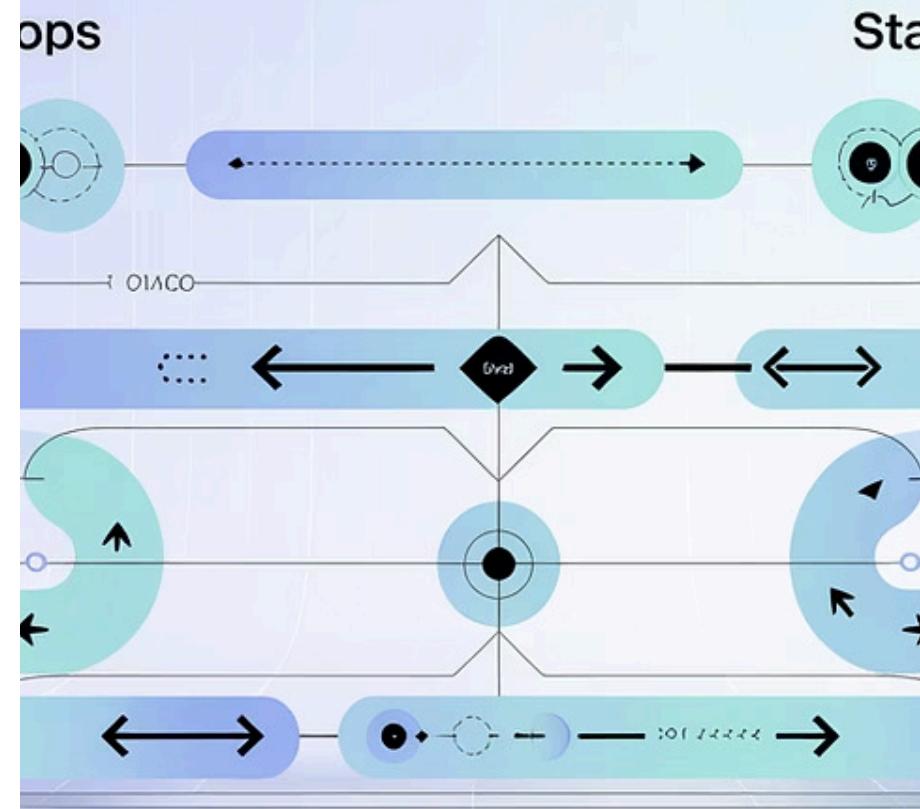
```
// Esto:  
const elemento = (  
  <h1>Hola {nombre}!</h1>  
);
```

```
// Se convierte en:  
const elemento = React.createElement(  
  'h1',  
  { className: 'titulo' },  
  'Hola ',  
  nombre,  
  '!'  
);
```

Anatomía de un componente funcional

1. **Declaración:** Función JavaScript/Arrow function
2. **Props:** Parámetros que recibe el componente
3. **Return:** Elementos JSX que se renderizarán
4. **Expresiones:** Código JavaScript entre llaves {}

React Functional Component



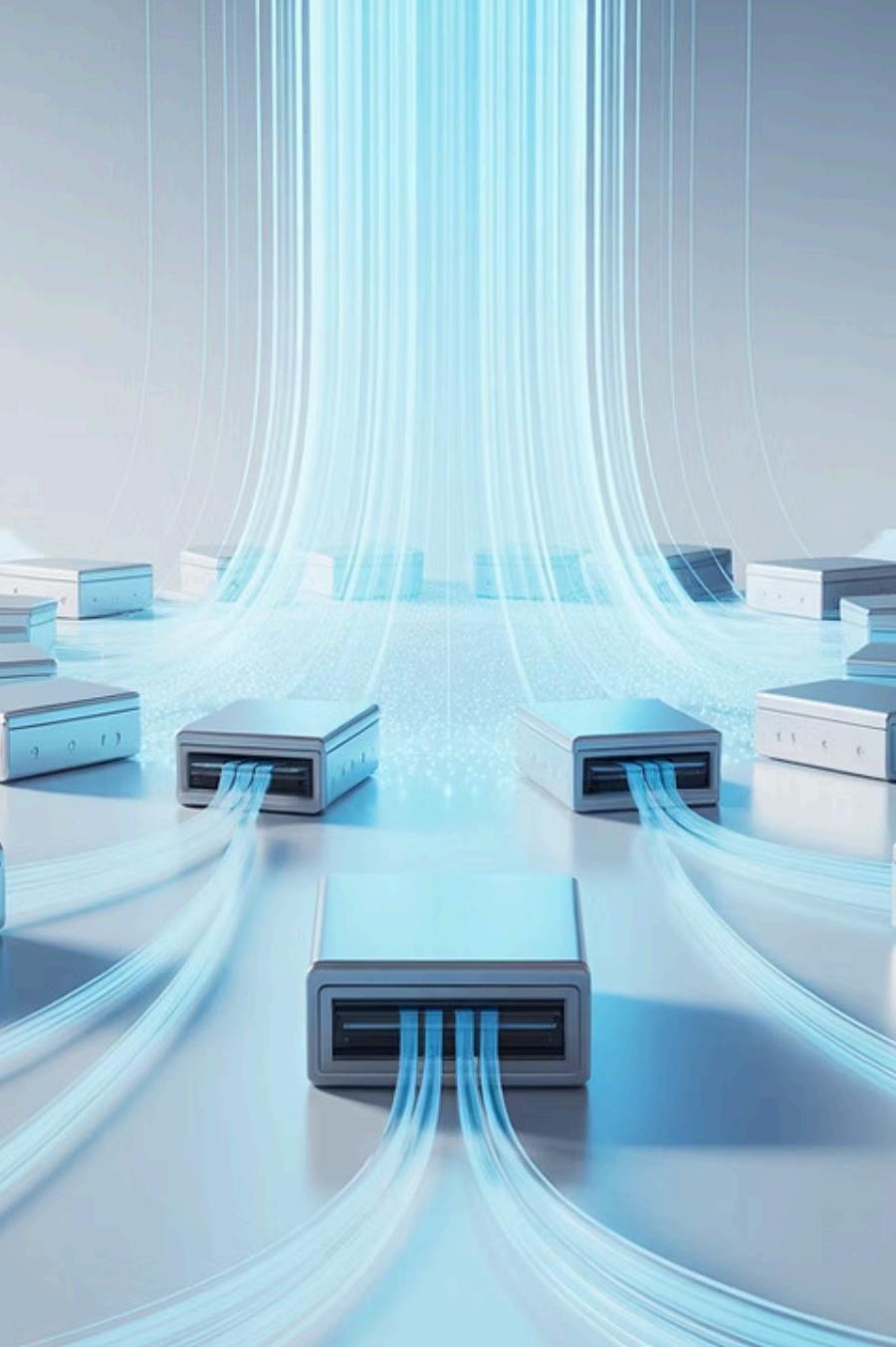
Ejemplo de componente funcional

```
import React from 'react';
import './Tarjeta.css';

function Tarjeta(props) {
  // Lógica JavaScript
  const { titulo, descripcion, imagen } = props;
  const claseCSS = titulo.length > 20 ? 'titulo-largo' : 'titulo-normal';

  // Devuelve JSX
  return (
    <div className="tarjeta">
      <img src={imagen} alt={titulo} />
      <h2 className={claseCSS}>{titulo}</h2>
      <p>{descripcion}</p>
      <buttons>
        <button variant="solid" onClick={() => console.log('Clic en', titulo)}>
          Ver más
        </button>
      </buttons>
    </div>
  );
}

export default Tarjeta;
```



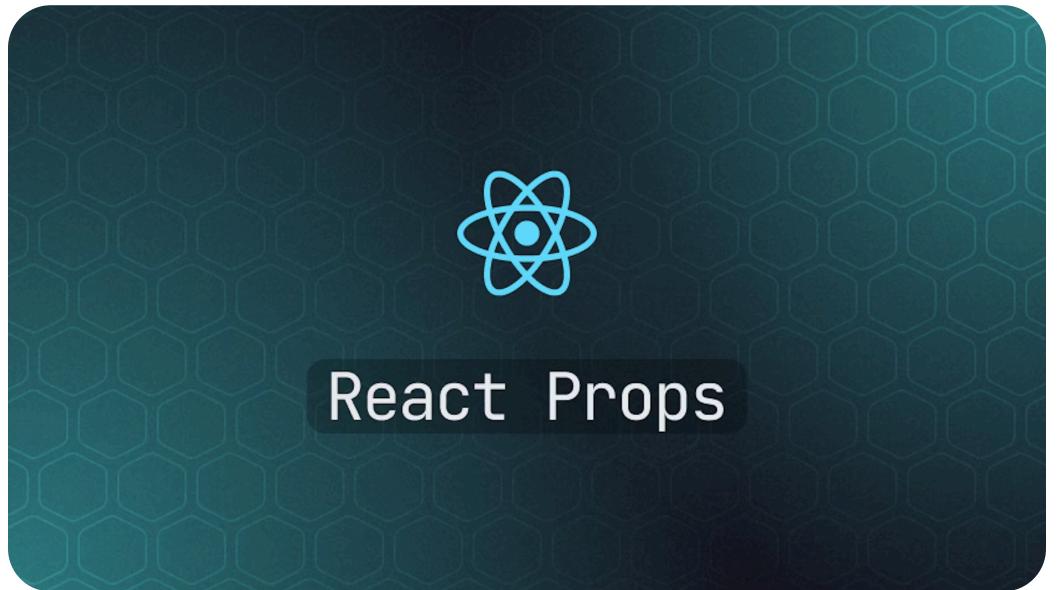
Props: Comunicación entre componentes

¿Qué son las props?

Las **props** (propiedades) son el mecanismo que permite pasar datos de un componente padre a un componente hijo.

Características principales:

- Son **inmutables** (de solo lectura)
- Fluyen en una sola dirección (de padre a hijo)
- Pueden ser de cualquier tipo: strings, números, objetos, funciones, etc.
- Permiten la reutilización de componentes con diferentes datos



Uso de props

Componente Padre:

```
function App() {  
  return (  
    <div>  
      <Perfil  
        nombre="Juan"  
        profesion="Desarrollador"  
        edad={30}  
        estaActivo={true}  
        tecnologias={['React', 'Node.js', 'Python']}  
      />  
      <Perfil  
        nombre="Maria"  
        profesion="Diseñadora"  
        edad={25}  
        estaActivo={false}  
        tecnologias={['Figma', 'Sketch', 'Photoshop']}  
      />  
    </div>  
  );  
}
```

Componente Hijo:

```
function Perfil(props) {  
  return (  
    <div>  
      <h2>{props.nombre}</h2>  
      <p>Profesión: {props.profesion}</p>  
      <p>Edad: {props.edad}</p>  
      <p>Estado: {props.estaActivo ? 'Activo' : 'Inactivo'}</p>  
      <h3>Tecnologías:</h3>  
      <ul>  
        {props.tecnologias.map((tech, index) => (  
          <li key={index}>{tech}</li>  
        ))}  
      </ul>  
    </div>  
  );  
}
```

Desestructuración de props

Es una práctica común desestructurar las props para mejorar la legibilidad:

En lugar de:

```
function Perfil(props) {  
  return (  
    <div>  
      <h2>{props.nombre}</h2>  
      <p>Profesión: {props.profesion}</p>  
    </div>  
  );  
}
```

Mejor así:

```
function Perfil({ nombre, profesion, tecnologias = [] }) {  
  return (  
    <div>  
      <h2>{nombre}</h2>  
      <p>Profesión: {profesion}</p>  
      <h3>Tecnologías:</h3>  
      <ul>  
        {tecnologias.map(tech => (  
          <li key={tech}>{tech}</li>  
        ))}  
      </ul>  
    </div>  
  );  
}
```

- También podemos asignar valores por defecto para cuando una prop no se proporciona.

Reflexión final

¿Qué diferencia hay entre un componente y una página HTML tradicional?

HTML tradicional

- Estático y descriptivo
- Difícil de reutilizar partes específicas
- Mezcla estructura, contenido y comportamiento
- Requiere manipulación manual del DOM
- No mantiene un "estado" de la interfaz

Componentes React

- Dinámicos y reactivos
- Diseñados para la reutilización
- Encapsulan estructura, lógica y estilo
- Actualizaciones automáticas del DOM
- Mantienen y gestionan su propio estado

Los componentes nos permiten pensar en términos de **interfaces** en lugar de páginas.