Módulo 4. React (con NEXTJS). Componentes

CÓRDOBA CREA FUTURO

Introducción

En React con Next.js, los componentes son bloques reutilizables de interfaz de usuario (botones, barras laterales, páginas completas) que organizan el código. Next.js amplía React al añadir estructura (rutas, carpetas) y funciones como el renderizado en servidor (SSR). La principal novedad son los componentes de servidor (RSC), que no envían JavaScript al cliente, lo que mejora el rendimiento, y que coexisten con componentes de cliente, los cuales utilizan estado y *hooks* como `useState` para la interactividad.

FRONTEND Y BACKEND

NODE.JS

El *frontend* es la parte visible de una web o aplicación (diseño, botones, interactividad) con la que el usuario interactúa (HTML, CSS, JavaScript), mientras que el *backend* corresponde a la lógica oculta en el servidor (bases de datos, seguridad, procesamiento) que hace funcionar todo, gestionando datos y servidores. Se complementan: el *frontend* solicita datos al *backend*, y este se los devuelve para

mostrarlos al usuario. Se puede comparar con un coche: el *frontend* es la carrocería y el *backend*, la maquinaria interna.

Lenguajes de desarrollo

Antes de todo, conviene entender qué es un desarrollador web. Un desarrollador web no se limita a una sola función, sino que abarca múltiples conjuntos de habilidades que se traducen en diferentes especialidades. Los tres términos más comunes que se utilizan para nombrar dichas especialidades de manera genérica son *frontend*, *backend* y *full stack*.

Desarrollador *frontend*

Trabaja del lado del cliente, en el navegador, es decir, en la parte visible de la web o aplicación. Principalmente, se ocupa de los componentes externos del sitio o aplicación web. Por ello, es necesario dominar obligatoriamente HTML, CSS y JavaScript, además de otros lenguajes y herramientas.

- **HTML** (*HyperText Markup Language*): es el componente estructural de todas las páginas web. Sin él, las páginas no podrían existir.
- **CSS** (*Cascading Style Sheets*): proporciona estilo a HTML y se encarga de la capa de presentación. Dominar CSS y sus capacidades resulta más complejo de lo que parece a simple vista.
- **JavaScript**: mientras que HTML y CSS crean páginas estáticas, con JavaScript las páginas se vuelven interactivas.
- **React (**https://es.react.dev/**)**: biblioteca de JavaScript que utiliza JSX para permitir escribir código al estilo de HTML dentro de JavaScript, facilitando la detección de errores al compilarlo.
- **Vue.js** (https://vuejs.org/): *framework* de JavaScript de código abierto para la construcción de interfaces de usuario y

aplicaciones. La biblioteca central se centra en la capa de vista. Los componentes se pueden crear con dos estilos de API: API de opciones y API de composición.

- **Angular** (https://angular.dev/): *framework* para aplicaciones web desarrollado en TypeScript, de código abierto y mantenido por Google, utilizado para crear y mantener aplicaciones web de una sola página (SPA).
- **Backbone.js** (https://backbonejs.org/): herramienta de desarrollo y API para JavaScript, con una interfaz RESTful basada en JSON, siguiendo el patrón Modelo-Vista-Controlador. Está diseñada para desarrollar aplicaciones SPA y mantener sincronizadas las distintas partes de la aplicación (por ejemplo, múltiples clientes y un servidor).
- **Ember.js** (https://emberjs.com/): *framework* web de JavaScript de código abierto que utiliza un patrón componente-servicio.

Desarrollador *backend*

El desarrollador *backend* trabaja del lado del servidor, detrás del escenario, permitiendo que el usuario disfrute de su experiencia. Para programar del lado del servidor, existen numerosos lenguajes y *frameworks*, según la empresa en la que se trabaje. Actualmente, los más comunes son los siguientes:

- **ASP.NET**: plataforma de desarrollo web de Microsoft, muy utilizada en empresas. Tiene las variantes Web Forms y MVC, y actualmente también ASP.NET Core MVC.
- **PHP**: por ejemplo, el popular gestor de contenidos WordPress funciona con PHP. Laravel es uno de los *frameworks* más utilizados con este lenguaje.

- **Ruby**: junto con su *framework* Ruby on Rails.
- **Python**: fácil de aprender y usado a menudo con Django como *framework*.
- **Node.js**: cada vez más popular, ya que permite usar JavaScript en el lado del servidor.
- **Java**: lenguaje clásico y muy demandado. Para el desarrollo de aplicaciones web se utiliza frecuentemente junto a frameworks como Spring o Hibernate.

Sin embargo, no basta con dominar un lenguaje y un framework. Toda aplicación web necesita almacenar datos de alguna manera. Por ello, un desarrollador *backend* también debe estar familiarizado con bases de datos, por ejemplo:

- SQL Server
- MySQL
- Oracle
- PostgreSQL
- MongoDB, que es un almacén de datos no-relacional o NoSQL.

Como se mencionó antes, el entorno de trabajo puede obligar a especializarse en unas tecnologías u otras. Dominar estos conceptos es un paso importante para acercarse al perfil que el mercado conoce como *full stack developer*.

Node.js es un entorno de ejecución de JavaScript de código abierto que permite ejecutar código del lado del servidor (*backend*) y crear aplicaciones de red, usando el mismo lenguaje que se emplea en el navegador, gracias al motor V8 de Chrome. Su característica principal es la arquitectura orientada a eventos y el modelo de entrada/salida no bloqueante, lo que lo hace muy eficiente y rápido para manejar múltiples conexiones simultáneamente. Resulta ideal para aplicaciones en tiempo real, como chats, API y microservicios.

Características principales

Entre las principales características de Node.js se encuentran las siguientes:

- **Motor V8 de Google**. Compila JavaScript a código máquina para un alto rendimiento.
- **Asíncrono y no bloqueante**: maneja múltiples solicitudes sin esperar, usando un modelo basado en eventos.
- **Multiplataforma**: funciona en Windows, macOS y Linux.
- **NPM (Node Package Manager)**: gestor de paquetes que permite instalar bibliotecas y herramientas fácilmente.
- **Escalable**: capaz de manejar muchas conexiones concurrentes.

¿Para qué se usa?

Node.js se emplea en diversos contextos del desarrollo de *software*:

- Servidores web y API. Construcción del *backend* de aplicaciones web y servicios.
- Aplicaciones en tiempo real: chats, juegos en línea, servicios de *streaming* (mediante WebSockets).
- Aplicaciones de una sola página (SPA): gestión de la lógica del servidor.
- Microservicios: arquitecturas distribuidas y escalables.
- Herramientas de línea de comandos (CLI): *scripting* y automatización.

Node.js permite utilizar JavaScript para desarrollar aplicaciones completas, desde el *frontend* hasta el *backend*, y aprovechar su velocidad y eficiencia en aplicaciones modernas y de alta demanda.

CONTINUAR

1. Aspectos técnicos

Concurrencia

Node.js funciona con un modelo de ejecución de un único hilo, utilizando operaciones de entrada y salida asíncronas que pueden ejecutarse de forma concurrente en cantidades que alcanzan cientos de miles, sin incurrir en los costos asociados al cambio de contexto.

Si bien la concurrencia y su programación no forman parte de los contenidos abordados en este curso, se recomienda, una vez finalizado, explorar la programación concurrente en entornos de sistemas distribuidos y paralelos.

V8

V8 es el entorno de ejecución para JavaScript creado para Google Chrome. Es software libre desde 2008, está escrito en C++ y compila el código fuente JavaScript a código de

máquina, en lugar de interpretarlo en tiempo real. Node.js incluye la biblioteca libuv para gestionar eventos asíncronos.

Módulos —

Node.js incorpora varios módulos básicos compilados en su propio binario. Entre ellos se encuentra el módulo de red, que proporciona una capa para programación de red asíncrona, y otros módulos fundamentales, como los siguientes: «path», «filesystem», «buffer», «timers» y el de propósito general «stream». También es posible utilizar módulos desarrollados por terceros, ya sea como archivos «.node» precompilados o como archivos en JavaScript plano.

Desarrollo homogéneo entre cliente y servidor

Node.js puede combinarse con bases de datos documentales —como MongoDB o CouchDB— y con bases de datos relacionales —como MySQL, PostgreSQL, entre otras—, lo que permite trabajar en un entorno unificado de desarrollo en JavaScript. Con la adaptación de patrones de desarrollo del lado del servidor, tales como MVC y sus variantes MVP, MVVM, entre otras, Node.js facilita la reutilización del mismo modelo de interfaz tanto en el cliente como en el servidor.

Bucle de eventos

Node.js se registra con el sistema operativo, y cada vez que un cliente establece una conexión se ejecuta un *callback*. Dentro del entorno de ejecución de Node.js, cada conexión recibe una pequeña asignación de memoria dinámica, sin necesidad de crear

un hilo de ejecución. El bucle de gestión de eventos finaliza cuando ya no quedan eventos pendientes de atender.

Arquitectura de Node.js y su funcionamiento

Node.js utiliza la arquitectura «single threaded event loop» para gestionar múltiples clientes de forma simultánea. Para comprender en qué se diferencia de otros entornos de ejecución, es necesario entender cómo se manejan los clientes concurrentes multihilo en lenguajes como Java.

En un modelo de solicitud-respuesta multihilo, varios clientes envían solicitudes que el servidor procesa una por una antes de devolver la respuesta. Para ello, se emplean múltiples hilos encargados de atender llamadas concurrentes. Estos hilos se definen en un *pool* de hilos, y cada vez que llega una petición, se asigna uno de ellos para gestionarla.

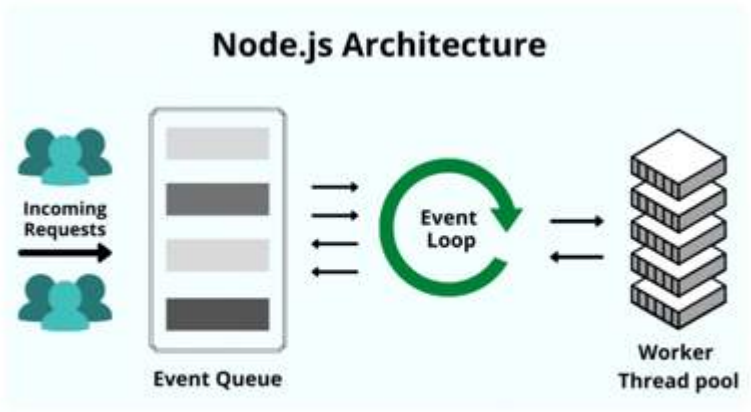
Node.js funciona de forma diferente. A continuación, se describe su funcionamiento:

- Node.js mantiene un *pool* de hilos limitado para atender las peticiones.
- Cada vez que llega una solicitud, Node.js la coloca en una cola.

- El bucle de eventos de un solo hilo —el componente principal— entra en acción. Este bucle espera peticiones de forma indefinida.
- Cuando llega una solicitud, el bucle la extrae de la cola y verifica si requiere una operación de entrada/salida (E/S) bloqueante. Si no es así, procesa la solicitud y envía la respuesta.
- Si la solicitud incluye una operación bloqueante, el bucle de eventos asigna un hilo del *pool* interno para gestionarla. Estos hilos auxiliares disponibles son limitados y conforman el llamado grupo de trabajadores.
- El bucle de eventos rastrea las solicitudes que quedan bloqueadas y las reintegra a la cola una vez que se completa la tarea correspondiente. De este modo, se mantiene el comportamiento no bloqueante de la arquitectura.

Dado que Node.js utiliza menos hilos, consume menos recursos y memoria, lo que permite una ejecución más ágil de las tareas. Para los fines propuestos, esta arquitectura de un solo hilo resulta equivalente a la arquitectura multihilo. Sin embargo, cuando se requiere procesar grandes volúmenes de datos, puede ser más adecuado recurrir a lenguajes multihilo como Java. Para aplicaciones en tiempo real, Node.js representa una opción apropiada.

Figura 1. Arquitectura de Node.js



Fuente: Khare, 2025, https://goo.su/7VSsKr

Aplicaciones de Node.js

Node.js no es un lenguaje de programación ni un marco de trabajo (*framework*); es un entorno para ambos. Se utiliza en una amplia variedad de aplicaciones, entre las que se encuentran las siguientes:

- **"Chats en tiempo real**. Debido a su arquitectura asíncrona de un solo hilo, resulta especialmente adecuado para gestionar la comunicación en tiempo real. Puede emplearse en la creación de chatbots y en el envío de notificaciones *push*.
- **Aplicaciones de Internet de las cosas (IoT)**: este tipo de aplicaciones suele estar compuesto por múltiples sensores que envían pequeños fragmentos de datos, lo que genera un gran volumen de peticiones. Node.js es una opción adecuada, ya que permite gestionarlas de manera concurrente y eficiente.
- **Transmisión de datos (*streaming*)**: empresas como Netflix utilizan Node.js para servicios de *streaming*, principalmente por su ligereza, velocidad y porque proporciona una API de

streaming nativa. Estos flujos permiten canalizar solicitudes directamente entre sí, lo que facilita la entrega directa de datos a su destino final.

- **Aplicaciones complejas de una sola página (SPA)**: en las SPA, toda la aplicación se carga en una sola página, lo que implica la realización de solicitudes en segundo plano para componentes específicos. En este contexto, el bucle de eventos de Node.js permite procesarlas sin bloqueo" (Khare, 2025, https://goo.su/7VSsKr).

Dado que JavaScript se emplea tanto en el *frontend* como en el *backend*, es posible establecer una comunicación fluida entre el servidor y la interfaz mediante API REST utilizando Node.js. Además, este entorno ofrece paquetes como «Express.js» y «Koa», que simplifican la creación de aplicaciones web.

Figura 2. Aplicaciones que pueden usar Node.js

Fuente: Khare, 2025, https://goo.su/7VSsKr

¿Cómo instalar Node.js?

Para instalar Node.js en Windows, se deben seguir los siguientes pasos:

1. Ingresar al sitio oficial: https://nodejs.org/en.
2. Descargar el instalador correspondiente al sistema operativo Windows.
3. Ejecutar el archivo descargado y seguir las instrucciones del instalador para completar la instalación.
4. Verificar que la instalación se haya realizado correctamente. Para ello, se debe abrir la terminal o símbolo del sistema (*cmd*) y ejecutar uno de los siguientes comandos:

- `node -v`

- `node --version`

  Si el resultado muestra un número de versión, significa que Node.js fue instalado con éxito.

Paquetes populares

Node.js cuenta con una gran cantidad de paquetes que amplían sus funcionalidades. A continuación, se mencionan algunos de los más utilizados en la actualidad:

- **Express**. Marco de desarrollo web inspirado en Sinatra. Es el estándar de facto para la mayoría de las aplicaciones desarrolladas con Node.js.
- **MongoDB**: controlador oficial para conectarse a bases de datos MongoDB desde Node.js. Proporciona una API para la gestión de datos.
- **Socket.io**: permite la comunicación en tiempo real, bidireccional y basada en eventos.
- **Lodash**: facilita el trabajo con arreglos, números, objetos, cadenas, entre otros elementos de JavaScript.
- **Moment**: biblioteca para analizar, validar, manipular y formatear fechas en JavaScript.
- **Commander.js**: permite crear interfaces de línea de comandos de forma sencilla y estructurada.
- **Forever**: herramienta de línea de comandos que garantiza que un script de Node.js se mantenga en ejecución continua, incluso ante fallos.
- **Async**: módulo de utilidades que proporciona funciones simples y eficaces para trabajar con JavaScript asíncrono.

- **Redis**: cliente para integrar la base de datos Redis en proyectos desarrollados con Node.js.
- **Mocha**: marco de pruebas limpio y flexible para ejecutar test en JavaScript, tanto en Node.js como en el navegador.
- **Passport**: biblioteca de autenticación sencilla y modular. Su propósito principal es autenticar solicitudes en aplicaciones Node.js.

A continuación, se muestra un ejemplo simple de cómo crear un servidor HTTP utilizando Node.js. Este servidor responderá con un mensaje de texto cuando se acceda desde el navegador:

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 1337;

http.createServer((req, res) => {
 res.writeHead(200, { 'Content-Type': 'text/plain' });
 res.end('Hola Mundo de Córdoba\n');
}).listen(port, hostname, () => {
   console.log(Server ejecutándose en http://${hostname}:${port}/);
});
```

Este código debe guardarse en un archivo llamado server.js. Luego, desde la terminal, se ejecuta el siguiente comando para iniciar el servidor:

Si todo funciona correctamente, al ingresar en http://localhost:1337 desde un navegador, debería mostrarse el siguiente mensaje:

Hola Mundo de Córdoba

Este ejemplo utiliza el módulo incorporado «http», que permite a Node.js gestionar solicitudes y respuestas a través del Protocolo de Transferencia de Hipertexto. Primero se importa el módulo, luego se crea el servidor con «createServer», y finalmente se lo pone en escucha mediante el método «listen».

Creación de un servidor mediante Express

servidor mediante Express

Antes de comenzar con el uso de Express, es útil repasar brevemente qué es un servidor. Se trata de un componente responsable de recibir solicitudes de los clientes, procesarlas y enviar una respuesta adecuada. Tradicionalmente, esto se realiza mediante software como Apache o Nginx. En el entorno de Node.js, Express permite crear un servidor de forma sencilla y eficiente.

Express.js es un *framework* minimalista y flexible para aplicaciones web y móviles en Node.js. Proporciona un conjunto sólido de funcionalidades que incluye el manejo de rutas y la configuración de *middleware*. Para instalarlo, se utiliza el siguiente comando:

npm install express –save

En el ejemplo anterior se utilizó el módulo incorporado «http» para crear un servidor básico. Ahora, se verá cómo crear un servidor equivalente usando Express:

```
// server-express.js

const express = require('express')

const app = express() // initialize app

const port = 3000



// GET callback function returns a response message

app.get('/', (req, res) => {

res.send('Hola mundo de Córdoba')

})



app.listen(port, () => {
```

```
console.log(`Server ejecutándose en http://localhost:${port}`)

})
```

Este archivo debe guardarse como `server-express.js`. Para ejecutar el servidor, se utiliza el siguiente comando en la terminal:

```
node server-express.js
```

Luego, se puede abrir el navegador y acceder a http://localhost:3000. Si todo está configurado correctamente, debería visualizarse el mensaje:

Hola Mundo de Córdoba

Una vez comprendidos los conceptos teóricos, es momento de poner en práctica lo aprendido.

CONTINUAR

2. Integrando conceptos con ejemplos

A continuación, se presentan distintos tipos de componentes en React y una estructura básica de implementación, con el objetivo de afianzar los conceptos aprendidos.

Componentes funcionales

Los componentes funcionales son funciones que devuelven una interfaz JSX. Pueden definirse de forma tradicional o mediante funciones flecha:

```
// Componente como función

function Saludo() {

 return <h1>¡Hola Mundo!</h1>;

}
```

```
// O con arrow function (flecha)

const Boton = () => {

 return <button>Click aquí</button>;

};
```

Componentes de clase

Otra forma de definir componentes es mediante clases que extienden de `React.Component`:

```
class Saludo extends React.Component {

 render() {

   return <h1>¡Hola desde una clase!</h1>;

 }

}
```

El siguiente ejemplo muestra cómo se organiza un componente típico en React:

```
import React from 'react';

// 2. Crear el componente
function TarjetaUsuario() {
 // 3. Lógica aquí (opcional)
 const nombre = "Ana";

 // 4. Retornar JSX (lo que se ve en pantalla)
 return (
   <div className="tarjeta">
     <h2>{nombre}</h2>
     <p>Desarrolladora web</p>
     <button>Contactar</button>
   </div>
 );
}

// 5. Exportar para usarlo en otros lugares
export default TarjetaUsuario;
```

ESTRUCTURA BÁSICA DE UN COMPONENTE | CARACTERÍSTICAS DE LOS COMPONENTES | COMPONENTES EN NEXT.JS

Como explicamos, los componentes en React tienen varias características que los hacen versátiles y potentes. A continuación,

se presentan algunas de las más relevantes, acompañadas de ejemplos.

Reutilizables

Un mismo componente puede usarse múltiples veces dentro de una misma interfaz:

```
// Usamos el mismo componente varias veces
function App() {
 return (
   <div>
     <TarjetaUsuario />
     <TarjetaUsuario />
     <TarjetaUsuario />
   </div>
 );
}
```

Reciben *props* (datos de entrada)

Las *props* permiten enviar datos desde un componente padre a uno hijo:

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

// Uso:
<Saludo nombre="Carlos" />
<Saludo nombre="María" />
```

Pueden tener estado interno

Los componentes también pueden manejar su propio estado, es decir, conservar valores en memoria y actualizarlos dinámicamente:

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

function Contador() {
 const [numero, setNumero] = useState(0);

 return (
   <div>
     <p>Contador: {numero}</p>
     <button onClick={() => setNumero(numero + 1)}>
       Sumar
     </button>
   </div>
 );
}
```


Next.js incorpora tipos especiales de componentes según el entorno en el que se ejecutan. Estos se dividen del siguiente modo:

- **Componentes del lado del servidor (*server components*)**

Estos componentes se ejecutan exclusivamente en el servidor y se utilizan por defecto. Son ideales para acceder a bases de datos o

realizar operaciones que no requieren interacción directa con el usuario:

```
// Por defecto en Next.js 13+
// Se ejecutan en el servidor
async function ListaProductos() {
 const productos = await obtenerProductos(); // Desde BD

 return (
  <ul>
   {productos.map(producto => (
    <li key={producto.id}>{producto.nombre}</li>
   ))}
  </ul>
 );
}
```

- **Componentes del cliente (*client components*)**

Para indicar que un componente se ejecuta en el cliente, se debe incluir la siguiente línea al comienzo del archivo:

```
'use client'; // Esta línea indica que es del cliente

import { useState } from 'react';

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

 return <button onClick={() => setContador(contador + 1)}>
  Clics: {contador}
 </button>;
```

```
}
```

Recomendaciones —

A continuación, se presentan algunas buenas prácticas para trabajar con componentes en React.

1. Nombrar los componentes con mayúscula inicial:

```
function ComponenteBueno() {/* ... */}
```

2. Crear componentes pequeños y específicos, que cumplan una única función:

```
function BotonPrincipal() {/* ... */}
function ListaTareas() {/* ... */}
```

3. Organizar los componentes en una carpeta específica, por ejemplo, `components`:

```
// 📂 app/
//   📂 components/
//     📄 Header.jsx
//     📄 Footer.jsx
//     📄 Card.jsx
```

¿Qué evitar y qué aplicar?

1. No mezclar lógica de negocio con la presentación:

```
// MAL:
function ComponenteMalo() {
 // Mucha lógica de negocio aquí...
 return <div>...</div>;
}

// BIEN:
function Presentacion() {
 return <div>...</div>;
}
```

2. Evitar componentes excesivamente extensos. Si un componente supera las 100 líneas, es recomendable dividirlo en partes más manejables.

Ejemplo completo de componente en Next.js —

A continuación, se muestra un ejemplo completo de cómo estructurar y utilizar un componente en Next.js. Este componente se ejecuta en el cliente y permite simular una acción de seguimiento (*seguir*) mediante el uso de estado.
Archivo: app/components/TarjetaPerfil.jsx

```
'use client'; // Componente del cliente

import { useState } from 'react';

export default function TarjetaPerfil({ nombre, edad }) {
 const [seguidores, setSeguidores] = useState(0);

 const seguir = () => {
  setSeguidores(seguidores + 1);
```

```
  };

  return (
    <div className="tarjeta-perfil">
      <h2>{nombre}</h2>
      <p>Edad: {edad}</p>
      <p>Seguidores: {seguidores}</p>
      <button onClick={seguir}>
        Seguir
      </button>
    </div>
  );
}
```

Archivo: app/page.js

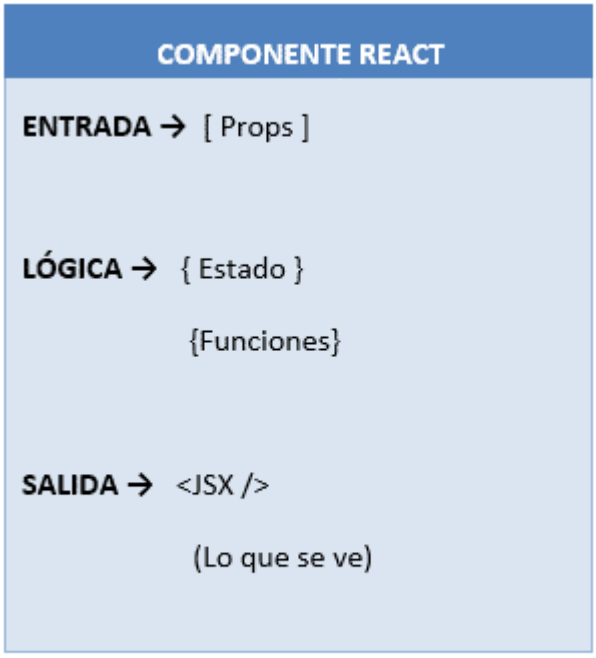
```
import TarjetaPerfil from './components/TarjetaPerfil';

export default function HomePage() {
  return (
    <main>
      <h1>Nuestra Comunidad</h1>
      <TarjetaPerfil nombre="Luis" edad={25} />
      <TarjetaPerfil nombre="Sofía" edad={30} />
    </main>
  );
}
```

Para cerrar esta sección, se presenta un esquema que resume de forma visual la estructura básica de un componente en React:

Este esquema muestra cómo un componente recibe *props* como entrada, gestiona su lógica interna mediante estado y funciones, y finalmente produce una salida visual a través de JSX.

Figura 3. Esquema general de un componente en React



Fuente: elaboración propia

Consejos para empezar

Antes de comenzar a desarrollar componentes más complejos, conviene tener en cuenta algunas recomendaciones iniciales que pueden facilitar el proceso:

- Comenzar con ejemplos simples, como botones o tarjetas.
- Dividir los componentes grandes en partes más pequeñas y manejables.

- Pensar en la posibilidad de reutilización: «¿podría usar este componente en otro lugar?».
- Seguir convenciones de escritura: usar mayúsculas iniciales en los nombres de componentes y optar por nombres descriptivos.
- Practicar con frecuencia: crear distintos tipos de componentes ayuda a afianzar los conceptos.

Se debe tener presente que cada componente es una pieza independiente que puede combinarse con otras para construir interfaces complejas, de manera similar a cómo se arman estructuras con bloques de LEGO.

CONTINUAR

Referencias

Khare, M. (2025). Qué es Node.js y por qué debería usarlo. *Kinsta*. https://kinsta.com/es/blog/que-es-node-js/

Referencias bibliográficas de consulta

Deitel, H. y Deitel, P. (2013). *Cómo programar en Java*. Pearson

Joyanes Aguilar, L. (2014). *Fundamentos de programación*. McGraw-Hill

Moreno, P. J. (2015). *Programación orientada a objetos*. RA-MA Editorial

CONTINUAR