**📘 Guía Detallada para las Clases Grupales de React.js**

**🔹 Clase 1: Fundamentos de React**

**Objetivo:** Comprender qué es React y construir el primer componente funcional.

**Contenido paso a paso:**

1. **¿Qué es React?**
   * Biblioteca de JavaScript para construir interfaces de usuario.
   * Ventajas: modularidad, rendimiento, comunidad activa.
2. **JSX: la fusión entre HTML y JS**
   * Sintaxis: const elemento = <h1>Hola mundo</h1>
   * Reglas: usar una etiqueta raíz, camelCase, atributos como className, etc.
3. **Crear tu primer componente:**

jsx

function Bienvenida() {

return <h2>¡Hola, bienvenida!</h2>;

}

export default Bienvenida;

1. **Props vs State:**
   * props: datos que recibe el componente (inmutables).
   * state: datos internos que pueden cambiar.
2. **Ejercicio práctico:**
   * Crear un componente que reciba el nombre del usuario como props y lo salude.

**🔹 Clase 2: Componentes y estructura del editor**

**Objetivo:** Aprender sobre componentes reutilizables y cómo organizar la interfaz.

**Paso a paso:**

1. **Componentes funcionales**
   * Separar la interfaz en piezas independientes.
2. **Estructura tipo árbol**
   * Ejemplo visual del árbol de componentes para el editor.
3. **Ejercicio:**
   * Crear la barra lateral con botones (Add, Design, Pages) usando componentes.

**🔹 Clase 3: Hooks esenciales**

**Objetivo:** Manipular el estado y ciclo de vida con useState, useEffect, useRef.

**Desarrollo:**

1. **useState:** manejo de datos reactivos.
2. **useEffect:** ejecutar código al montar, actualizar o desmontar componentes.
3. **useRef:** acceder directamente al DOM.

**Ejercicio:**

* Crear una vista editable que se actualiza en tiempo real según lo que se escribe.

**🔹 Clase 4: Formularios y eventos**

**Contenido:**

1. **Formularios controlados:** uso de useState para capturar inputs.
2. **Validaciones básicas** (longitud, campos vacíos).
3. **Eventos:** onChange, onSubmit.

**Ejercicio:**

* Formulario de nuevo sitio web con nombre, categoría, plantilla.

**🔹 Clase 5: Routing y navegación**

**Herramientas:**

1. **React Router DOM:**
   * BrowserRouter, Route, Link, useNavigate.
2. **Ejercicio práctico:**
   * Crear una app con navegación entre “Editor” y “Configuración”.

**🔹 Clase 6: Context API y estado global**

**Conceptos:**

1. **Context API:** compartir datos entre componentes sin prop drilling.
2. **Ejercicio:**
   * Crear un tema global (colores, fuentes) que afecte todos los componentes.

**🔹 Clase 7: Renderizado condicional y listas**

1. **Render condicional con operadores lógicos.**
2. **Listas dinámicas con** .map()**.**
3. **Ejercicio:**
   * Galería de plantillas donde se ve la preview al hacer clic.

**🔹 Clase 8: Conexión con Backend**

1. **Fetch vs Axios:** ventajas y ejemplos.
2. **Consumo básico de datos externos.**
3. **Ejercicio:**
   * Guardar datos de sitios simulados usando JSON Server.

**📗 Clase Extra: Tailwind CSS + React.js**

**Objetivo:** Aprender a estilizar interfaces modernas con Tailwind en proyectos React.

**Paso a paso:**

1. **¿Qué es Tailwind CSS?**
   * Framework de utilidades que permite escribir estilos en línea.
2. **Instalación:**

bash

npm install -D tailwindcss

npx tailwindcss init

* + Configurar tailwind.config.js y index.css.

1. **Clases utilitarias clave:**
   * Layout: flex, grid, gap, justify-center
   * Tipografía: text-xl, font-bold, tracking-wide
   * Estilos: bg-indigo-500, rounded-md, shadow-lg
2. **Tematización:**
   * Personalizar colores, fuentes y breakpoints en el archivo de configuración.
3. **Ejercicio práctico:**
   * Construir una sección editable tipo “Hero” con título, imagen y botón CTA usando clases utilitarias.

**🧩 Organización del Grupo**

* Cada persona lidera una clase como “profe”, prepara ejemplos y guía al grupo.
* Se construye una mini app usando todo lo aprendido como base de la plataforma final.
* Se pueden grabar las sesiones y armar un repositorio con los ejemplos semanales.

**📘 Clase 1: Fundamentos de React.js**

**Título del capítulo:** *"React desde cero: tu primer componente con personalidad"*

**🧠 Introducción**

React.js es como el lienzo moderno para construir interfaces web. No es un lenguaje, es una **biblioteca** de JavaScript que te permite crear aplicaciones interactivas con facilidad y escalabilidad.

**¿Por qué usar React en tu proyecto?**

* 📦 Componentes reutilizables
* ⚡ Actualizaciones eficientes mediante Virtual DOM
* 🌍 Amplia comunidad y recursos
* 🔄 Flujo de datos claro: props → state

**🧩 ¿Qué es JSX?**

JSX es una extensión de JavaScript que permite escribir HTML dentro del código JS. Aunque el navegador no lo entiende directamente, React lo traduce en instrucciones para construir elementos del DOM.

jsx

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

**Reglas clave:**

* Todo el JSX debe estar dentro de una única etiqueta padre.
* Atributos como class se reemplazan por className.
* Estilos se escriben como objetos JS: style={{ color: 'red' }}

**🛠️ Crear tu primer componente**

Un componente es una función que retorna JSX. Es como una pieza del puzzle de tu interfaz.

jsx

function Bienvenida() {

return <h2>¡Bienvenida a tu primer componente React!</h2>;

}

export default Bienvenida;

**Uso del componente:**

jsx

import Bienvenida from './Bienvenida';

function App() {

return <Bienvenida />;

}

**📥 Props vs 📦 State**

**Props (Propiedades)**

* Son datos que se pasan **desde el componente padre al hijo**.
* Son **inmutables**, el componente no debe cambiarlos.

jsx

function Saludo({ nombre }) {

return <h3>Hola, {nombre}!</h3>;

}

**State (Estado)**

* Son datos **internos** que pueden cambiar.
* Se usan con useState.

jsx

import { useState } from 'react';

function Contador() {

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

return (

<div>

<p>Contador: {contador}</p>

<button onClick={() => setContador(contador + 1)}>Sumar</button>

</div>

);

}

**✏️ Ejercicio práctico: Componente “Bienvenida personalizada”**

**Objetivo:** Crear un componente que reciba el nombre del usuario como prop y muestre un mensaje personalizado.

**Paso a paso:**

1. Crear el archivo Bienvenida.jsx
2. Escribir la función con props
3. Usarlo en el archivo App.jsx

jsx

// Bienvenida.jsx

function Bienvenida({ nombre }) {

return <h2>¡Hola {nombre}, bienvenida a tu app React!</h2>;

}

export default Bienvenida;

// App.jsx

import Bienvenida from './Bienvenida';

function App() {

return <Bienvenida nombre="Verónica" />;

}

**📚 Bonus teórico: ¿Qué pasa detrás de escena?**

React usa un **Virtual DOM**, que es una versión ligera del DOM real. Cuando cambias el estado, React compara versiones y actualiza solo lo necesario. Esto es lo que lo hace tan rápido.

**🧪 Retos para practicar**

* Haz que el componente Bienvenida muestre también la fecha de ingreso (props.fecha).
* Crea otro componente que muestre un botón para cambiar el nombre con useState.

**🧑‍🏫 Práctica en Vivo – Clase 1: “React en Acción”**

**🎯 Objetivo de la práctica**

* Entender qué es React y cómo funciona JSX
* Crear y reutilizar componentes básicos
* Trabajar con props y useState para construir interactividad

**⏱️ Estructura de la clase**

| **Tiempo** | **Actividad** | **Detalles** |
| --- | --- | --- |
| 10 min | Introducción teórica | Breve repaso de qué es React, JSX, componentes |
| 15 min | Live coding grupal | Crear juntos el primer componente Bienvenida.jsx |
| 20 min | Desafío 1 | Cada miembro personaliza su componente con props |
| 20 min | Desafío 2 | Crear un componente Contador.jsx con useState |
| 10 min | Mini showcase | Cada integrante muestra su avance |
| 15 min | Cierre interactivo | Preguntas, reflexiones y anticipación de la clase 2 |

**🛠️ Live Coding – Paso a paso guiado**

**Primera parte:**

* Crear carpeta src/componentes
* Agregar archivo Bienvenida.jsx
* Mostrar cómo importar y usar componentes desde App.jsx

**Tip:** Compartí pantalla y que todos lo vayan replicando.

**🚀 Desafío 1: Componente personalizado**

Cada miembro del equipo debe:

* Crear su propio Bienvenida.jsx
* Recibir por props nombre y fecha
* Usar JSX para mostrar un saludo como:

jsx

¡Hola, Verónica! Te conectaste el 3 de agosto de 2025.

**Bonus:** Agregar estilos con Tailwind como text-xl text-blue-600

**🔄 Desafío 2: Componente interactivo**

Crear un archivo Contador.jsx con el siguiente flujo:

* Botón para incrementar
* Mostrar valor del contador
* Estilizar con Tailwind: ejemplo bg-green-500 hover:bg-green-700

jsx

<button onClick={() => setContador(contador + 1)} className="bg-green-500 hover:bg-green-700">

+1

</button>

**🌟 Actividad final: Mini show & feedback**

Cada integrante comparte su componente por pantalla o repo:

* ¿Qué te gustó de lo que hiciste?
* ¿Qué mejorarías?

**📘 Clase 2: Componentes y estructura del editor**

**Título del capítulo:** *"Construyendo bloques: la arquitectura visual con React"*

**🧠 Introducción**

En React, todo es un **componente**. Cada botón, cada panel, cada parte del editor es una pieza independiente que se **encapsula**, **reutiliza** y **organiza jerárquicamente**. Esta clase se enfoca en diseñar la estructura base del editor web con React.

**🏗️ ¿Qué es un componente funcional?**

Un **componente funcional** es una función que retorna JSX. Se usa para representar secciones de la UI que pueden recibir props y manejar estado local.

jsx

function BotonEditar() {

return <button className="btn">Editar</button>;

}

Se usa dentro de otros componentes como cualquier etiqueta HTML:

jsx

function BarraLateral() {

return (

<div>

<BotonEditar />

</div>

);

}

**🌲 Estructura tipo árbol**

Pensá el editor como un árbol donde cada nodo es un componente:

App.jsx

├── BarraLateral.jsx

│ ├── BotonAdd.jsx

│ ├── BotonDesign.jsx

│ └── BotonPages.jsx

├── VistaPrincipal.jsx

│ └── CanvasEdicion.jsx

└── PanelConfiguracion.jsx

Esto permite:

* Separar responsabilidades
* Reutilizar botones y bloques
* Escalar sin romper diseño

**🎯 Componentes reutilizables: qué y cómo**

Un componente reusable debe:

* Ser **autocontenido** (no depender de demasiados props externos)
* Tener **estilos desacoplados**
* Aceptar **props dinámicos** para cambiar textos, íconos, comportamiento

jsx

function Boton({ texto, onClick }) {

return (

<button onClick={onClick} className="px-4 py-2 bg-blue-500 text-white">

{texto}

</button>

);

}

Uso:

jsx

<Boton texto="Agregar módulo" onClick={handleAdd} />

<Boton texto="Diseñar página" onClick={handleDesign} />

**✏️ Ejercicio práctico: Crear la barra lateral del editor**

**Objetivo:** Armar una barra lateral con 3 botones: Add, Design y Pages

**Pasos:**

1. Crear Boton.jsx como componente reutilizable
2. Crear BarraLateral.jsx donde se importen tres instancias del botón
3. Mostrar la barra en App.jsx

jsx

// Boton.jsx

function Boton({ texto }) {

return (

<button className="w-full bg-indigo-600 text-white py-2 mb-2 rounded">

{texto}

</button>

);

}

export default Boton;

// BarraLateral.jsx

import Boton from './Boton';

function BarraLateral() {

return (

<aside className="w-64 p-4 bg-gray-100">

<Boton texto="Add" />

<Boton texto="Design" />

<Boton texto="Pages" />

</aside>

);

}

export default BarraLateral;

// App.jsx

import BarraLateral from './BarraLateral';

function App() {

return (

<div className="flex">

<BarraLateral />

{/\* Aquí irá VistaPrincipal \*/}

</div>

);

}

**🧪 Retos para practicar en grupo**

**Desafío 1:** Agregar un ícono SVG a cada botón y hacerlo configurable por props.

**Desafío 2:** Agregar efecto hover con Tailwind: hover:bg-indigo-700, transition, duration-300.

**Desafío 3:** Crear una sección VistaPrincipal.jsx con texto de bienvenida dinámico recibido por props.

**🔍 Bonus pedagógico: por qué es importante esto**

Pensar en componentes como bloques te permite:

* Crear una interfaz editable tipo Canva con áreas específicas
* Cambiar solo partes de la UI sin alterar todo
* Distribuir tareas del proyecto fácilmente (cada integrante trabaja en una rama del árbol)

**🧑‍🏫 Práctica en Vivo – Clase 2: “Diseñando el esqueleto del editor”**

**🎯 Objetivo de la práctica**

* Crear componentes funcionales y reutilizables
* Construir una **Barra Lateral** con botones dinámicos
* Aprender a organizar una estructura tipo árbol para apps React

**⏱️ Estructura sugerida de la clase**

| **Tiempo** | **Actividad** | **Detalles** |
| --- | --- | --- |
| 10 min | Intro teórica | ¿Qué son los componentes reutilizables y por qué son clave? |
| 15 min | Live coding guiado | Crear juntos Boton.jsx y BarraLateral.jsx |
| 25 min | Desafío en equipo | Cada integrante personaliza su versión con props dinámicos |
| 20 min | Práctica adicional | Crear VistaPrincipal.jsx y conectar con BarraLateral |
| 10 min | Showcase colaborativo | Compartir avances y comparativas |
| 10 min | Cierre y reflexión | Preguntas, mejoras y preview de la clase 3 |

**🛠️ Live Coding grupal: armando la base**

**Actividad:**

* Crear Boton.jsx como componente que recibe texto, ícono y estilos por props
* Crear BarraLateral.jsx con 3 botones (Add, Design, Pages)
* Usar Tailwind para estilizar: w-full, p-2, bg-indigo-500, hover:bg-indigo-700

Ejemplo:

jsx

function Boton({ texto }) {

return (

<button className="w-full bg-indigo-600 text-white py-2 mb-2 rounded hover:bg-indigo-700 transition">

{texto}

</button>

);

}

**🚀 Desafío práctico para el grupo**

**Objetivo:** Cada integrante aporta un módulo personalizado al editor.

**Tareas:**

1. **Personalización con props:**
   * Que cada botón muestre un ícono SVG según su tipo (Add, Design, Pages)
   * Agregar prop onClick con un console.log() temporal
2. **Estilos dinámicos:**
   * Usar Tailwind para variar color, tamaño y animación entre botones
3. **Organización de componentes:**
   * Crear carpeta /componentes
   * Separar cada botón si tiene lógica propia (BotonAdd.jsx, etc.)
4. **Extra:** Crear VistaPrincipal.jsx con un mensaje dinámico de bienvenida

jsx

function VistaPrincipal({ usuario }) {

return (

<div className="p-6 text-xl">

Bienvenido/a, {usuario}. ¡Listo para editar tu sitio!

</div>

);

}

**🎓 Showcase final**

Cada integrante comparte:

* Su versión de BarraLateral
* Cómo usó props para modular su código
* Qué estilo visual eligió

Esto fomenta la exploración visual, el orgullo por el trabajo propio y la inspiración entre pares.

**📝 Material de apoyo sugerido**

* PDF visual con estructura del árbol de componentes
* Repo compartido con carpeta “Plantillas de Componentes”
* Tabla con props recomendados para cada módulo editable (ideal para el futuro)

**🧑‍🏫 Clase 3 – “React vivo: Estado y eventos para interfaces dinámicas”**

**🎯 Objetivos de la clase**

* Entender cómo funciona useState
* Crear interacciones con onClick, onChange, onSubmit
* Construir un componente de edición básico que responde a acciones del usuario

**⏱️ Estructura sugerida**

| **Tiempo** | **Actividad** | **Detalles** |
| --- | --- | --- |
| 10 min | Intro teórica | Qué es el estado y cómo se usa en React |
| 20 min | Live coding | Crear componente EditorTexto.jsx interactivo |
| 25 min | Desafío guiado | Agregar funcionalidades: input, botones, estilo dinámico |
| 15 min | Integración | Conectarlo con VistaPrincipal.jsx de la clase anterior |
| 10 min | Reflexión | ¿Qué aprendimos y cómo lo aplicamos en el proyecto final? |

**🧪 Teoría práctica: ¿Qué es el estado?**

React permite mantener **datos internos** en un componente con useState. Cada vez que ese “estado” cambia, **el componente se re-renderiza**.

jsx

import { useState } from 'react';

function Contador() {

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

return (

<div>

<p>Has hecho clic {contador} veces</p>

<button onClick={() => setContador(contador + 1)}>Click</button>

</div>

);

}

**✍️ Live Coding grupal: “Editor básico de texto”**

**Componentes:** EditorTexto.jsx

**Objetivo:** Crear un input y mostrar el valor dinámicamente

jsx

import { useState } from 'react';

function EditorTexto() {

const [texto, setTexto] = useState('');

return (

<div className="p-4 border rounded bg-gray-100">

<input

type="text"

value={texto}

onChange={(e) => setTexto(e.target.value)}

className="border px-2 py-1 w-full mb-2"

placeholder="Escribí algo..."

/>

<p className="text-lg">Vista previa: {texto}</p>

</div>

);

}

**🚀 Desafío práctico**

**Tareas para el grupo:**

1. **Input editables:**
   * Input para título
   * Input para texto principal
   * Selector de color de fondo
2. **Botón que cambia el color dinámicamente:**

jsx

const [color, setColor] = useState('white');

<button onClick={() => setColor('lightblue')}>Cambiar color</button>

1. **Mostrar preview del módulo editado:**
   * Mostrar el texto + título con estilos en VistaPreview.jsx
   * Usar props y useState combinados

**🧩 Integración con Clase 2**

* Importar EditorTexto.jsx en VistaPrincipal.jsx
* Pasar nombre de usuario como prop
* Crear una experiencia tipo "inicio de edición" estilo Canva

jsx

function VistaPrincipal({ usuario }) {

return (

<div className="p-6">

<h1 className="text-2xl mb-4">Bienvenido/a, {usuario}</h1>

<EditorTexto />

</div>

);

}

**🗂️ Repositorio sugerido**

Estructura base del proyecto:

/componentes

└── Boton.jsx

└── BarraLateral.jsx

└── VistaPrincipal.jsx

└── EditorTexto.jsx

└── VistaPreview.jsx

**📝 Recursos adicionales**

* Guía PDF: cómo usar useState, con ejemplos interactivos
* Plantilla Markdown: mini proyecto “Editor de texto en vivo”
* Ejercicio extra: agregar useEffect para guardar cambios en localStorage (puente a Clase 4)

**🧑‍🏫 Clase 4 – “Persistencia de datos: useEffect + localStorage”**

**🎯 Objetivos de la clase**

* Comprender el ciclo de vida con useEffect
* Guardar datos en localStorage
* Recuperar y sincronizar datos al cargar la app
* Preparar la estructura base para modular persistente (editor y vista previa)

**⏱️ Estructura sugerida**

| **Tiempo** | **Actividad** | **Detalles** |
| --- | --- | --- |
| 10 min | Teoría clave | ¿Qué es useEffect y cómo se relaciona con el renderizado? |
| 20 min | Live coding guiado | Guardar el contenido del editor en localStorage |
| 25 min | Desafío en equipo | Sincronizar datos con useEffect al abrir la app |
| 15 min | Extra: simulación de carga | Crear efecto visual de “cargando datos guardados” |
| 10 min | Cierre | Qué otros módulos podrían guardar su estado igual |

**📚 Teoría práctica: ¿Qué hace useEffect?**

* Es un **hook** que permite ejecutar lógica cuando el componente se monta, actualiza o desmonta
* Ideal para interactuar con APIs, localStorage, timers, etc.

jsx

import { useEffect } from 'react';

useEffect(() => {

console.log('Se ejecuta al montar el componente');

}, []);

**🔄 Ejemplo completo: Editor con persistencia**

jsx

import { useState, useEffect } from 'react';

function EditorTexto() {

const [texto, setTexto] = useState('');

// Al cargar el componente, recuperar datos guardados

useEffect(() => {

const textoGuardado = localStorage.getItem('editorTexto');

if (textoGuardado) {

setTexto(textoGuardado);

}

}, []);

// Guardar el texto en localStorage cada vez que cambia

useEffect(() => {

localStorage.setItem('editorTexto', texto);

}, [texto]);

return (

<div>

<input

type="text"

value={texto}

onChange={(e) => setTexto(e.target.value)}

placeholder="Escribí algo..."

className="border p-2 w-full mb-2"

/>

<p className="bg-blue-100 p-2 rounded">Vista previa: {texto}</p>

</div>

);

}

**🚀 Desafío práctico**

**Tareas:**

1. Crear EditorModulo.jsx para que guarde título, texto y color de fondo
2. Usar localStorage para guardar cada campo
3. Recuperar estado inicial desde localStorage en useEffect
4. Mostrar mensaje animado “Cargando datos guardados…” antes de mostrar el editor
5. Integrar con VistaPreview.jsx para mostrar persistencia visual

**🔗 Bonus avanzado**

* Simular carga con setTimeout() y estado loading
* Modularizar el guardado para todos los componentes futuros

jsx

const [loading, setLoading] = useState(true);

useEffect(() => {

const timer = setTimeout(() => setLoading(false), 1000);

return () => clearTimeout(timer);

}, []);

**🗂️ Estructura de componentes sugerida**

/componentes

└── EditorModulo.jsx

└── VistaPreview.jsx

└── helpers

└── localStorage.js

js

// helpers/localStorage.js

export function guardarDato(clave, valor) {

localStorage.setItem(clave, JSON.stringify(valor));

}

export function recuperarDato(clave) {

const dato = localStorage.getItem(clave);

return dato ? JSON.parse(dato) : null;

}

**📝 Recursos para el equipo**

* Guía PDF: resumen de useEffect con escenarios comunes
* Markdown con estructura “Editor persistente”
* Ejercicio opcional: guardar secciones como array de objetos
* Preview de Clase 5: levantar datos desde una API tipo Firebase o simulación REST

**📘 Clase 5 – “Routing y conexión con APIs: hacia una plataforma editable”**

**🧠 Objetivos clave**

* Crear rutas entre módulos de edición (React Router)
* Conectar tu app con una API para guardar y recuperar datos
* Preparar el flujo base de trabajo con backend: GET y POST
* Integrar edición, vista previa y navegación como una sola experiencia

**🔄 Parte 1: Routing con React Router DOM**

**📦 Instalación**

bash

npm install react-router-dom

**🛤️ Estructura base**

jsx

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

import Editor from './EditorModulo';

import Configuracion from './PanelConfiguracion';

function App() {

return (

<BrowserRouter>

<Routes>

<Route path="/" element={<Editor />} />

<Route path="/configuracion" element={<Configuracion />} />

</Routes>

</BrowserRouter>

);

}

**🔗 Navegación entre rutas**

jsx

import { Link } from 'react-router-dom';

<Link to="/configuracion" className="text-blue-500 underline">

Ir a Configuración

</Link>

**🌐 Parte 2: Conectando con una API externa**

Usaremos **JSON Server**, que crea una API falsa para guardar nuestros módulos.

**📦 Instalación**

bash

npm install -g json-server

**📄 Crear archivo db.json**

json

{

"sitios": []

}

Ejecutás el servidor:

bash

json-server --watch db.json --port 4000

**📡 Crear funciones de conexión**

**En** /services/api.js**:**

js

const API\_URL = 'http://localhost:4000/sitios';

export async function obtenerSitios() {

const response = await fetch(API\_URL);

return response.json();

}

export async function crearSitio(nuevoSitio) {

const response = await fetch(API\_URL, {

method: 'POST',

body: JSON.stringify(nuevoSitio),

headers: {

'Content-Type': 'application/json'

}

});

return response.json();

}

**🧩 Parte 3: Crear un editor conectado**

**En** EditorModulo.jsx**:**

jsx

import { useState } from 'react';

import { crearSitio } from '../services/api';

function EditorModulo() {

const [titulo, setTitulo] = useState('');

const [descripcion, setDescripcion] = useState('');

const guardarSitio = async () => {

const nuevoSitio = { titulo, descripcion, creadoEn: Date.now() };

await crearSitio(nuevoSitio);

alert('Sitio guardado en la API');

};

return (

<div className="p-6">

<input

type="text"

value={titulo}

onChange={(e) => setTitulo(e.target.value)}

placeholder="Título"

className="border mb-2 p-2 w-full"

/>

<textarea

value={descripcion}

onChange={(e) => setDescripcion(e.target.value)}

placeholder="Descripción"

className="border p-2 w-full h-24"

/>

<button

onClick={guardarSitio}

className="bg-indigo-600 text-white px-4 py-2 mt-2 rounded"

>

Guardar Sitio

</button>

</div>

);

}

**🧪 Desafío práctico en vivo**

Cada miembro del grupo debe:

1. Crear su EditorModulo.jsx que guarde título, texto y fecha en la API
2. Crear una ruta /sitios con VistaSitios.jsx que muestre todos los sitios creados
3. Estilizar con Tailwind para una galería visual de los módulos

**🚀 Bonus: hacia una plataforma editable multiusuario**

A partir de esta clase, podés:

* Crear múltiples sitios por usuario
* Cargar módulos desde un backend
* Implementar edición en tiempo real en clases futuras (PATCH, DELETE)
* Preparar conexión con bases de datos reales como Firebase o tu propio backend Java + Spring Boot

**📚 Extras sugeridos**

* 📁 Carpeta /services para todas las llamadas a la API
* 🗂️ Repositorio compartido con /db.json como base simulada
* 📄 PDF con explicación visual del flujo de datos (frontend ↔ backend)

**📘 Clase 6 – “Estado global con Context API: diseñá tu plataforma de forma inteligente”**

**🎯 Objetivos clave**

* Entender qué es prop drilling y cómo evitarlo
* Usar createContext y useContext para compartir información entre componentes
* Crear una lógica de configuración global (tema, usuario, idioma, etc.)
* Integrar un panel de edición donde los cambios se reflejan en toda la app

**🧩 Parte 1: ¿Por qué Context API?**

**❌ Problema: Prop Drilling**

Cuando pasás datos como props desde el componente padre al bisnieto o tataranieto, tu código se vuelve difícil de mantener:

jsx

<App>

<Config>

<Layout>

<Header tema={tema} /> // demasiadas capas

</Layout>

</Config>

</App>

**✅ Solución: Context API**

* Creás un **contexto global**
* Usás Provider para que todos los componentes puedan acceder
* Usás useContext() para obtener y modificar la información

**🔨 Parte 2: Crear contexto global – paso a paso**

**Creamos** TemaContext.jsx **en** /context**:**

jsx

import { createContext, useState } from 'react';

export const TemaContext = createContext();

export function TemaProvider({ children }) {

const [tema, setTema] = useState({

colorPrimario: '#4f46e5',

fuente: 'sans-serif'

});

return (

<TemaContext.Provider value={{ tema, setTema }}>

{children}

</TemaContext.Provider>

);

}

**Usamos el Provider en** App.jsx**:**

jsx

import { TemaProvider } from './context/TemaContext';

import VistaPrincipal from './componentes/VistaPrincipal';

function App() {

return (

<TemaProvider>

<VistaPrincipal />

</TemaProvider>

);

}

**🎛️ Parte 3: Modificar el estado global desde cualquier componente**

**Desde** PanelConfiguracion.jsx**:**

jsx

import { useContext } from 'react';

import { TemaContext } from '../context/TemaContext';

function PanelConfiguracion() {

const { tema, setTema } = useContext(TemaContext);

const cambiarColor = (nuevoColor) => {

setTema({ ...tema, colorPrimario: nuevoColor });

};

return (

<div className="p-4">

<input

type="color"

value={tema.colorPrimario}

onChange={(e) => cambiarColor(e.target.value)}

/>

<p className="mt-2">Color actual: {tema.colorPrimario}</p>

</div>

);

}

**🖼️ Parte 4: Aplicar el contexto en otros componentes**

**En** VistaPreview.jsx**:**

jsx

import { useContext } from 'react';

import { TemaContext } from '../context/TemaContext';

function VistaPreview() {

const { tema } = useContext(TemaContext);

return (

<div style={{ backgroundColor: tema.colorPrimario, fontFamily: tema.fuente }}>

<h1>Preview del sitio con tema seleccionado</h1>

</div>

);

}

🎯 ¡Boom! Cambiás el color en el panel y se actualiza automáticamente en toda la app.

**🔧 Desafíos para el equipo**

1. Crear nuevos contextos: UsuarioContext, IdiomaContext, TemplateContext
2. Permitir elegir tipografía global con select (serif, sans-serif, etc.)
3. Mostrar la configuración activa en un resumen (PanelResumen.jsx)
4. Guardar el contexto en localStorage con useEffect para persistir

**🌐 Preview de Clase 7**

En la próxima clase aprenderán a **mostrar elementos condicionales y listas**, para construir una **galería de bloques** o templates (ideal para tu app tipo Canva/Wix). ¡Todo dinamizado por el estado global que acaban de crear!

🚀 Aquí tenés la **plantilla lista para subir al repo**, con estructura modular y preparada para escalar. Ideal para que tu equipo la use como base en sus clases y experimentos con Context API.

**🧰 Estructura del proyecto**

src/

├── context/

│ └── TemaContext.jsx

├── componentes/

│ ├── PanelConfiguracion.jsx

│ ├── VistaPrincipal.jsx

│ ├── VistaPreview.jsx

│ └── PanelResumen.jsx (opcional)

├── App.jsx

└── index.jsx

**📦 Archivos principales**

**✅ TemaContext.jsx (contexto global para tema)**

jsx

import { createContext, useState } from 'react';

export const TemaContext = createContext();

export function TemaProvider({ children }) {

const [tema, setTema] = useState({

colorPrimario: '#4f46e5',

fuente: 'sans-serif'

});

return (

<TemaContext.Provider value={{ tema, setTema }}>

{children}

</TemaContext.Provider>

);

}

**✅ App.jsx (envuelve todo con el proveedor)**

jsx

import { TemaProvider } from './context/TemaContext';

import VistaPrincipal from './componentes/VistaPrincipal';

function App() {

return (

<TemaProvider>

<VistaPrincipal />

</TemaProvider>

);

}

export default App;

**✅ VistaPrincipal.jsx (maquetado general)**

jsx

import PanelConfiguracion from './PanelConfiguracion';

import VistaPreview from './VistaPreview';

function VistaPrincipal() {

return (

<div className="flex gap-4 p-6">

<PanelConfiguracion />

<VistaPreview />

</div>

);

}

export default VistaPrincipal;

**✅ PanelConfiguracion.jsx (selector dinámico)**

jsx

import { useContext } from 'react';

import { TemaContext } from '../context/TemaContext';

function PanelConfiguracion() {

const { tema, setTema } = useContext(TemaContext);

const cambiarColor = (nuevoColor) =>

setTema({ ...tema, colorPrimario: nuevoColor });

const cambiarFuente = (nuevaFuente) =>

setTema({ ...tema, fuente: nuevaFuente });

return (

<div>

<h2>Configuración de Tema</h2>

<input

type="color"

value={tema.colorPrimario}

onChange={(e) => cambiarColor(e.target.value)}

/>

<select value={tema.fuente} onChange={(e) => cambiarFuente(e.target.value)}>

<option value="sans-serif">Sans Serif</option>

<option value="serif">Serif</option>

<option value="monospace">Monospace</option>

</select>

</div>

);

}

export default PanelConfiguracion;

**✅ VistaPreview.jsx (aplica los estilos del contexto)**

jsx

import { useContext } from 'react';

import { TemaContext } from '../context/TemaContext';

function VistaPreview() {

const { tema } = useContext(TemaContext);

return (

<div

style={{

backgroundColor: tema.colorPrimario,

fontFamily: tema.fuente,

padding: '2rem',

borderRadius: '8px',

color: '#fff',

}}

>

<h1>Preview del sitio</h1>

<p>Usando el tema seleccionado</p>

</div>

);

}

export default VistaPreview;

**🧪 Extras opcionales**

Si querés ir más allá, se puede agregar:

* PanelResumen.jsx: muestra la configuración activa
* Guardado en localStorage usando useEffect
* Componentes que cambian según el idioma (multi-contexto)

**📘 Clase 7 – “Condicionales y listas: construcción dinámica de bloques”**

**🎯 Objetivos clave**

* Comprender renderizado condicional (mostrar elementos según estado)
* Usar map() para renderizar listas de componentes
* Introducir el concepto de key en listas
* Aplicar edición, activación y eliminación de bloques dinámicos

**🧩 Parte 1: Renderizado condicional – mostrar solo si...**

**Ejemplo básico:**

jsx

function Alerta({ mostrar }) {

if (!mostrar) return null;

return <div className="bg-red-200 p-4">¡Esto es una alerta!</div>;

}

🔍 Si mostrar es falso, no se renderiza nada. Esto lo podés usar para mostrar configuraciones según el contexto.

**🧩 Parte 2: Listas con .map() – estructuras flexibles**

Supongamos que el usuario selecciona una lista de bloques que quiere mostrar:

jsx

const bloques = ['Header', 'Galería', 'Footer'];

**Renderización dinámica:**

jsx

function EditorBloques({ bloques }) {

return (

<div>

{bloques.map((bloque, index) => (

<div key={index} className="mb-4 p-2 border">

<h2>Bloque: {bloque}</h2>

</div>

))}

</div>

);

}

🎯 Cada bloque se renderiza como una sección. Ideal para un sistema de construcción modular tipo Wix.

**🛠️ Parte 3: Combinando condicional + lista + estado**

Vamos a hacer algo más interactivo:

jsx

import { useState } from 'react';

function ConstructorSitio() {

const [bloques, setBloques] = useState(['Header', 'Galería']);

const [activo, setActivo] = useState('Galería');

return (

<div>

{bloques.map((bloque, index) => (

<div

key={index}

className={`p-4 mb-2 border ${activo === bloque ? 'bg-green-200' : 'bg-gray-100'}`}

onClick={() => setActivo(bloque)}

>

<h3>{bloque}</h3>

{activo === bloque && <p>Este bloque está activo</p>}

</div>

))}

</div>

);

}

🚀 Al hacer clic, activás el bloque y se muestra contenido extra. Ideal para un modo edición donde el usuario navega por secciones del sitio.

**🔧 Parte 4: Edición dinámica de bloques**

Podés permitir agregar o eliminar secciones de forma directa:

jsx

function Editor() {

const [bloques, setBloques] = useState(['Header', 'Footer']);

const agregar = () => setBloques([...bloques, 'Galería']);

const eliminar = (nombre) => setBloques(bloques.filter(b => b !== nombre));

return (

<div>

<button onClick={agregar}>Agregar Galería</button>

{bloques.map((bloque, index) => (

<div key={index} className="p-4 border mb-2">

<h4>{bloque}</h4>

<button onClick={() => eliminar(bloque)}>Eliminar</button>

</div>

))}

</div>

);

}

💡 Esto ya se acerca muchísimo a tu visión de plataforma editable, donde el usuario construye su sitio interactivamente.

**🧪 Desafío para el equipo**

1. Crear componente EditorBloques.jsx para recibir bloques desde un contexto
2. Hacer que el panel de configuración permita **agregar tipos de bloque**
3. Mostrar cada bloque con estilos y contenido personalizado
4. Guardar la lista en localStorage para mantenerla entre sesiones
5. Bonus: permitir reordenar los bloques con botones “↑ ↓”

**🌐 Preview Clase 8**

¡Agarrate! En la próxima clase entra **useEffect**, el hook mágico para manejar efectos secundarios, incluyendo sincronización con APIs, localStorage, o temporizadores. Vas a poder guardar configuraciones, cargar contenido inicial y hasta conectarte con servicios externos.

🔥 Este módulo eleva el Editor al siguiente nivel: vamos a implementar **drag-and-drop con** react-beautiful-dnd, ideal para permitir que el usuario organice secciones como en un Canva o Wix. Vas a ver cómo cambia la experiencia de edición al permitir arrastrar bloques dinámicamente.

**📘 Extensión Clase 7 – Drag-and-drop con react-beautiful-dnd**

**🎯 Objetivos clave**

* Integrar react-beautiful-dnd para mover bloques
* Usar Droppable y Draggable con DragDropContext
* Reordenar el estado (bloques) al soltar
* Combinar con edición dinámica (agregar/eliminar bloques)

**🧰 Instalación previa**

bash

npm install react-beautiful-dnd

**🔨 Paso a paso en EditorBloques.jsx**

jsx

import { DragDropContext, Droppable, Draggable } from 'react-beautiful-dnd';

import { useState } from 'react';

function EditorBloques() {

const [bloques, setBloques] = useState(['Header', 'Galería', 'Footer']);

const onDragEnd = (result) => {

if (!result.destination) return;

const nuevaLista = Array.from(bloques);

const [moved] = nuevaLista.splice(result.source.index, 1);

nuevaLista.splice(result.destination.index, 0, moved);

setBloques(nuevaLista);

};

return (

<DragDropContext onDragEnd={onDragEnd}>

<Droppable droppableId="bloques">

{(provided) => (

<div {...provided.droppableProps} ref={provided.innerRef}>

{bloques.map((bloque, index) => (

<Draggable key={bloque} draggableId={bloque} index={index}>

{(provided) => (

<div

ref={provided.innerRef}

{...provided.draggableProps}

{...provided.dragHandleProps}

className="p-4 mb-2 border bg-white shadow cursor-move"

>

<h3>{bloque}</h3>

</div>

)}

</Draggable>

))}

{provided.placeholder}

</div>

)}

</Droppable>

</DragDropContext>

);

}

export default EditorBloques;

**✨ Tips de implementación**

* Podés agregar un botón para **añadir bloques personalizados**
* Guardá los bloques en localStorage usando useEffect
* ¡Estilizalo con Tailwind para una experiencia top!

**💡 Ideas para tu plataforma editable**

* Reordenamiento de secciones visuales (landing, productos, contacto)
* Arrastrar módulos desde un panel lateral (como en Canva)
* Drag-and-drop entre columnas (usando múltiples Droppable)

💻💫 Te presento la **plantilla con** react-beautiful-dnd **integrada**, más el nuevo **BloquesContext** para manejar el estado global de secciones editables como si fueran módulos visuales tipo Canva.

**🧰 Estructura del Proyecto Extendida**

src/

├── context/

│ └── BloquesContext.jsx

├── componentes/

│ ├── EditorBloques.jsx

│ ├── PanelBloques.jsx

│ └── VistaPreview.jsx

├── App.jsx

└── index.jsx

**🧠 BloquesContext.jsx – Contexto global para la lista de bloques**

jsx

import { createContext, useState } from 'react';

export const BloquesContext = createContext();

export function BloquesProvider({ children }) {

const [bloques, setBloques] = useState(['Header', 'Galería', 'Footer']);

return (

<BloquesContext.Provider value={{ bloques, setBloques }}>

{children}

</BloquesContext.Provider>

);

}

**🧾 App.jsx – Encapsula con el contexto**

jsx

import { BloquesProvider } from './context/BloquesContext';

import EditorBloques from './componentes/EditorBloques';

function App() {

return (

<BloquesProvider>

<div className="p-6">

<h1 className="text-2xl font-bold mb-4">Editor Drag-and-Drop</h1>

<EditorBloques />

</div>

</BloquesProvider>

);

}

export default App;

**🎯 EditorBloques.jsx – Drag-and-drop con datos desde contexto**

jsx

import { useContext } from 'react';

import { BloquesContext } from '../context/BloquesContext';

import { DragDropContext, Droppable, Draggable } from 'react-beautiful-dnd';

function EditorBloques() {

const { bloques, setBloques } = useContext(BloquesContext);

const onDragEnd = (result) => {

if (!result.destination) return;

const nuevaLista = Array.from(bloques);

const [moved] = nuevaLista.splice(result.source.index, 1);

nuevaLista.splice(result.destination.index, 0, moved);

setBloques(nuevaLista);

};

return (

<DragDropContext onDragEnd={onDragEnd}>

<Droppable droppableId="bloques">

{(provided) => (

<div ref={provided.innerRef} {...provided.droppableProps}>

{bloques.map((bloque, index) => (

<Draggable key={bloque} draggableId={bloque} index={index}>

{(provided) => (

<div

ref={provided.innerRef}

{...provided.draggableProps}

{...provided.dragHandleProps}

className="p-4 mb-3 bg-white border rounded shadow cursor-move"

>

<h2 className="text-lg">{bloque}</h2>

</div>

)}

</Draggable>

))}

{provided.placeholder}

</div>

)}

</Droppable>

</DragDropContext>

);

}

export default EditorBloques;

**🛠️ PanelBloques.jsx – Agregar y eliminar bloques desde un panel**

jsx

import { useContext, useState } from 'react';

import { BloquesContext } from '../context/BloquesContext';

function PanelBloques() {

const { bloques, setBloques } = useContext(BloquesContext);

const [nuevoBloque, setNuevoBloque] = useState('');

const agregarBloque = () => {

if (!nuevoBloque || bloques.includes(nuevoBloque)) return;

setBloques([...bloques, nuevoBloque]);

setNuevoBloque('');

};

const eliminarBloque = (nombre) =>

setBloques(bloques.filter(b => b !== nombre));

return (

<div className="mb-6">

<input

type="text"

value={nuevoBloque}

onChange={(e) => setNuevoBloque(e.target.value)}

placeholder="Nombre del bloque"

className="p-2 border mr-2"

/>

<button onClick={agregarBloque} className="bg-green-500 text-white px-3 py-2">

Agregar

</button>

<ul className="mt-4">

{bloques.map((b) => (

<li key={b} className="flex justify-between p-2 border mt-2">

<span>{b}</span>

<button onClick={() => eliminarBloque(b)} className="text-red-500">Eliminar</button>

</li>

))}

</ul>

</div>

);

}

export default PanelBloques;

**🖼️ VistaPreview.jsx – Visualiza los bloques renderizados**

jsx

import { useContext } from 'react';

import { BloquesContext } from '../context/BloquesContext';

function VistaPreview() {

const { bloques } = useContext(BloquesContext);

return (

<div className="border rounded p-6 bg-gray-50 mt-6">

<h2 className="text-xl mb-4">Preview</h2>

{bloques.map((bloque, index) => (

<div key={index} className="p-4 bg-white mb-2 shadow rounded">

<p>Render: {bloque}</p>

</div>

))}

</div>

);

}

export default VistaPreview;

**🚀 Extensiones posibles**

* Guardar los bloques en localStorage (usando useEffect)
* Guardar posición personalizada de cada bloque con estilos individuales
* Permitir duplicar bloques o aplicar estilos específicos desde el contexto

**📘 Clase 8 – “Integración con Backend: guardar y editar el sitio desde una API”**

**🎯 Objetivos**

* Usar fetch o axios para comunicar con un backend (simulado)
* Crear un formulario de edición
* Guardar contenido en la API (POST)
* Recuperar contenido guardado (GET)
* Simular una experiencia de carga real

**🧰 Herramientas necesarias**

* Backend simulado con **JSON Server**
* Alternativa: Mock API tipo https://jsonplaceholder.typicode.com
* También se puede usar Firebase en fases futuras

**🔧 Parte 1: Instalación de JSON Server**

bash

npm install -g json-server

**📂 Estructura del archivo db.json**

json

{

"sitios": []

}

**🖥️ Ejecutar la API simulada**

bash

json-server --watch db.json --port 4000

**🔗 Parte 2: Crear módulo de servicio – /services/sitios.js**

js

const BASE\_URL = 'http://localhost:4000/sitios';

export async function obtenerSitios() {

const res = await fetch(BASE\_URL);

return res.json();

}

export async function guardarSitio(sitioNuevo) {

const res = await fetch(BASE\_URL, {

method: 'POST',

body: JSON.stringify(sitioNuevo),

headers: { 'Content-Type': 'application/json' },

});

return res.json();

}

**🛠️ Parte 3: Formulario editable – EditorSitio.jsx**

jsx

import { useState } from 'react';

import { guardarSitio } from '../services/sitios';

function EditorSitio() {

const [titulo, setTitulo] = useState('');

const [contenido, setContenido] = useState('');

const enviar = async () => {

const nuevoSitio = { titulo, contenido, creadoEn: new Date().toISOString() };

await guardarSitio(nuevoSitio);

alert('¡Sitio guardado en la API!');

setTitulo('');

setContenido('');

};

return (

<div className="p-6 bg-white shadow rounded">

<input

type="text"

value={titulo}

onChange={(e) => setTitulo(e.target.value)}

placeholder="Título"

className="w-full p-2 border mb-3"

/>

<textarea

value={contenido}

onChange={(e) => setContenido(e.target.value)}

placeholder="Contenido"

className="w-full p-2 border mb-3 h-24"

/>

<button onClick={enviar} className="bg-indigo-600 text-white px-4 py-2 rounded">

Guardar sitio

</button>

</div>

);

}

export default EditorSitio;

**🖼️ Parte 4: Vista de sitios guardados – VistaSitios.jsx**

jsx

import { useEffect, useState } from 'react';

import { obtenerSitios } from '../services/sitios';

function VistaSitios() {

const [sitios, setSitios] = useState([]);

const [cargando, setCargando] = useState(true);

useEffect(() => {

const cargar = async () => {

const data = await obtenerSitios();

setSitios(data);

setCargando(false);

};

cargar();

}, []);

if (cargando) return <p>Cargando sitios guardados...</p>;

return (

<div className="mt-6">

<h2 className="text-xl font-bold mb-4">Sitios guardados</h2>

{sitios.map((s) => (

<div key={s.id} className="p-4 border mb-2 rounded bg-gray-50">

<h3 className="text-lg font-semibold">{s.titulo}</h3>

<p>{s.contenido}</p>

<p className="text-sm text-gray-500">Creado el: {s.creadoEn}</p>

</div>

))}

</div>

);

}

export default VistaSitios;

**🔄 Parte 5: Conexión entre módulos (App con Routing)**

jsx

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

import EditorSitio from './componentes/EditorSitio';

import VistaSitios from './componentes/VistaSitios';

function App() {

return (

<BrowserRouter>

<Routes>

<Route path="/" element={<EditorSitio />} />

<Route path="/sitios" element={<VistaSitios />} />

</Routes>

</BrowserRouter>

);

}

export default App;

**🧪 Desafío en vivo**

1. Que cada miembro edite y guarde un “sitio”
2. Agregar opción de “categoría” y “template usado”
3. Mostrar los sitios por filtro o búsqueda (básico)
4. Persistencia con localStorage si no hay conexión
5. Bonus: modificar (PATCH) y eliminar (DELETE) sitios

**🔮 Avances para tu plataforma editable**

Desde esta clase podés:

* Guardar secciones por usuario
* Cargar módulos en base a configuración previa
* Usar Firebase o tu backend en Java Spring Boot para reemplazar JSON Server
* Hacer la app colaborativa y multiusuario con seguridad e inicio de sesión

**📗 Clase Extra – “Tailwind CSS + React: diseño moderno sin dolor”**

**🎯 Objetivo**

* Entender qué es Tailwind CSS y cómo se usa con React
* Integrar estilos utilitarios en componentes
* Tematizar tu app (paleta de colores, tipografía)
* Crear una sección editable tipo “Hero” con imagen, texto y CTA

**🧩 Parte 1: ¿Qué es Tailwind CSS?**

Tailwind es un **framework de utilidades CSS**. A diferencia de Bootstrap, no te da componentes prefabricados sino clases pequeñas como p-4, bg-indigo-500, rounded, flex que vas combinando para construir tu propia interfaz.

🔧 Ventajas clave:

* Estilos rápidos y consistentes
* No tenés que salir de JSX para modificar el diseño
* Ideal para aplicaciones editables y personalizables

**⚙️ Parte 2: Instalación y configuración**

**En tu proyecto React:**

bash

npm install -D tailwindcss postcss autoprefixer

npx tailwindcss init

**Configuración mínima:**

tailwind.config.js

js

module.exports = {

content: ['./src/\*\*/\*.{js,jsx}'],

theme: {

extend: {},

},

plugins: [],

}

src/index.css

css

@tailwind base;

@tailwind components;

@tailwind utilities;

🔌 Importá index.css en main.jsx o App.jsx

**🎨 Parte 3: Clases utilitarias clave**

| **Categoría** | **Ejemplos** | **Descripción** |
| --- | --- | --- |
| Espaciado | p-4, m-2, gap-6 | Padding, margin y separación |
| Layout | flex, grid, justify-between | Diseño en filas y columnas |
| Tipografía | text-xl, font-bold, uppercase | Tamaño y estilo de texto |
| Colores | bg-blue-500, text-gray-700 | Fondo, texto, bordes |
| Estilo | rounded-lg, shadow-md | Bordes redondeados, sombras |

**🖼️ Parte 4: Crear una sección editable tipo “Hero”**

jsx

function HeroEditable() {

return (

<section className="bg-indigo-600 text-white p-8 rounded-lg text-center">

<img src="/hero.png" alt="Hero" className="mx-auto mb-4 w-32 h-32 rounded-full" />

<h1 className="text-3xl font-bold">Bienvenido a tu sitio</h1>

<p className="mt-2 text-lg">Editá esta sección como quieras</p>

<button className="mt-4 bg-white text-indigo-600 px-4 py-2 rounded hover:bg-gray-100 transition">

Empezar

</button>

</section>

);

}

🧩 Cada clase representa un estilo: colores, espacios, tipografía, comportamiento. Podés editarlos en tiempo real con useState y aplicarlos dinámicamente (ideal para tu plataforma editable).

**🎛️ Parte 5: Tematización personalizada**

En tailwind.config.js podés extender tu tema:

js

module.exports = {

theme: {

extend: {

colors: {

vitex: '#7645d8',

},

fontFamily: {

vitex: ['"Open Sans"', 'sans-serif'],

},

},

},

};

🎨 Usás tus colores/tipografías personalizados:

jsx

className="bg-vitex font-vitex text-white p-6"

**🧪 Práctica en vivo para la clase**

**💻 Mini app editable con Tailwind**

| **Tarea** | **Detalles** |
| --- | --- |
| Crear HeroEditable.jsx | Sección con imagen, texto y botón |
| Usar useState | Permitir que el usuario cambie el texto desde inputs |
| Cambiar estilos dinámicos | Fondo y color de texto con select (select bg, select text) |
| Bonus | Agregar animación con transition, duration-300, hover |

🎯 Cada integrante personaliza su Hero y lo muestra al equipo. Pueden construir una galería con los distintos diseños creados.

**🌐 Avances que habilita Tailwind**

Con Tailwind ahora podés:

* Diseñar componentes visuales editables sin tocar CSS
* Permitir que el usuario modifique el estilo visual por contexto (TemaContext)
* Crear una experiencia visual fluida como Canva
* 🚀 Vamos a armar una plantilla educativa lista para usar, que le permita a tu equipo editar una sección tipo "Hero" en React, con estilos dinámicos de Tailwind CSS y configuración visual desde un panel personalizado. Esto les va a enseñar cómo usar estados, props y configuración de diseño editable al estilo Canva, que es clave para tu futura plataforma.
* **🛠️ Proyecto educativo – “Hero configurado por el usuario”**
* **📁 Estructura del proyecto**
* bash
* src/
* │
* ├─ components/
* │ └─ HeroConfigurable.jsx
* │
* ├─ config/
* │ └─ heroSettings.json
* │
* ├─ App.jsx
* └─ index.css
* **🧠 1. Archivo de configuración visual (JSON)**
* heroSettings.json
* json
* {
* "title": "Bienvenido a VITEX",
* "subtitle": "Diseñá libremente tu sitio web",
* "buttonText": "Comenzar",
* "bgColor": "bg-violet-600",
* "textColor": "text-white",
* "image": "/hero.png"
* }
* 🎯 Este archivo puede ser modificado por el usuario final desde una interfaz tipo panel o formulario. Lo cargás con fetch() o import.
* **💡 2. Componente HeroConfigurable.jsx**
* jsx
* import React, { useState, useEffect } from 'react';
* import heroSettings from '../config/heroSettings.json';
* export function HeroConfigurable() {
* const [settings, setSettings] = useState(heroSettings);
* return (
* <section className={`${settings.bgColor} ${settings.textColor} p-8 rounded-lg text-center`}>
* <img
* src={settings.image}
* alt="Hero"
* className="mx-auto mb-4 w-32 h-32 rounded-full"
* />
* <h1 className="text-3xl font-bold">{settings.title}</h1>
* <p className="mt-2 text-lg">{settings.subtitle}</p>
* <button className="mt-4 bg-white text-violet-700 px-4 py-2 rounded hover:bg-gray-100 transition">
* {settings.buttonText}
* </button>
* </section>
* );
* }
* ✅ Si querés que se actualice dinámicamente, podés agregar un formulario con inputs para editar cada valor.
* **🧪 3. Ejercicio bonus: panel visual editable por el usuario**
* **Crea HeroEditorPanel.jsx**
* Un panel con campos para editar color, texto, imagen y botón, que actualiza el componente en tiempo real:
* jsx
* <input
* type="text"
* value={settings.title}
* onChange={(e) => setSettings({ ...settings, title: e.target.value })}
* />
* 🎨 También podés hacer un selector de color con clases de Tailwind (select, option) y un preview en vivo del diseño.
* **📦 4. Repo listo para clonar**

estructura completa tipo plantilla educativa para que tu equipo lo clone, edite y practique. También agregaré instrucciones README en español y un modo de publicación en Vercel. Así todos podrán tener su propia “Hero Page” editada 100% con React y Tailwind ✨

**🔗 Link directo al repositorio**

Podés acceder al proyecto educativo HeroEditable-VITEX en GitHub desde este enlace: 👉 Repositorio HeroEditable-VITEX en GitHub

**📘 README completo – Guía paso a paso**

markdown

# HeroEditable-VITEX 🚀

Una plantilla educativa para crear una sección Hero editable con React y Tailwind CSS. Ideal para aprender diseño dinámico, configuración visual y publicación en Vercel.

---

## 🧰 Tecnologías usadas

- React.js

- Tailwind CSS

- JSON para configuración visual

- Vite como bundler

- Vercel para despliegue

---

## 📁 Estructura del proyecto

src/ ├─ components/ │ └─ HeroConfigurable.jsx ├─ config/ │ └─ heroSettings.json ├─ App.jsx └─ index.css

---

## ⚙️ Instalación

```bash

git clone https://github.com/tu-usuario/HeroEditable-VITEX.git

cd HeroEditable-VITEX

npm install

npm run dev

**🧪 Personalización**

Editá el archivo heroSettings.json para cambiar:

* Título (title)
* Subtítulo (subtitle)
* Texto del botón (buttonText)
* Color de fondo (bgColor)
* Color de texto (textColor)
* Imagen (image)

También podés agregar un panel visual (HeroEditorPanel.jsx) para que el usuario edite en tiempo real.

**🌐 Publicación en Vercel**

1. Subí el repo a tu cuenta de GitHub
2. Iniciá sesión en vercel.com
3. Importá el proyecto desde GitHub
4. ¡Listo! Tu app estará online en segundos

**🎓 Ideal para**

* Clases de React + Tailwind
* Proyectos de diseño editable
* Prácticas de frontend moderno
* Prototipos tipo Canva

**✨ Créditos**

Creado por FRONTGROUP visionarios 💜