Módulo 4
Desarrollo de interfaces interactivas con React

Elementos avanzados de ReactJS

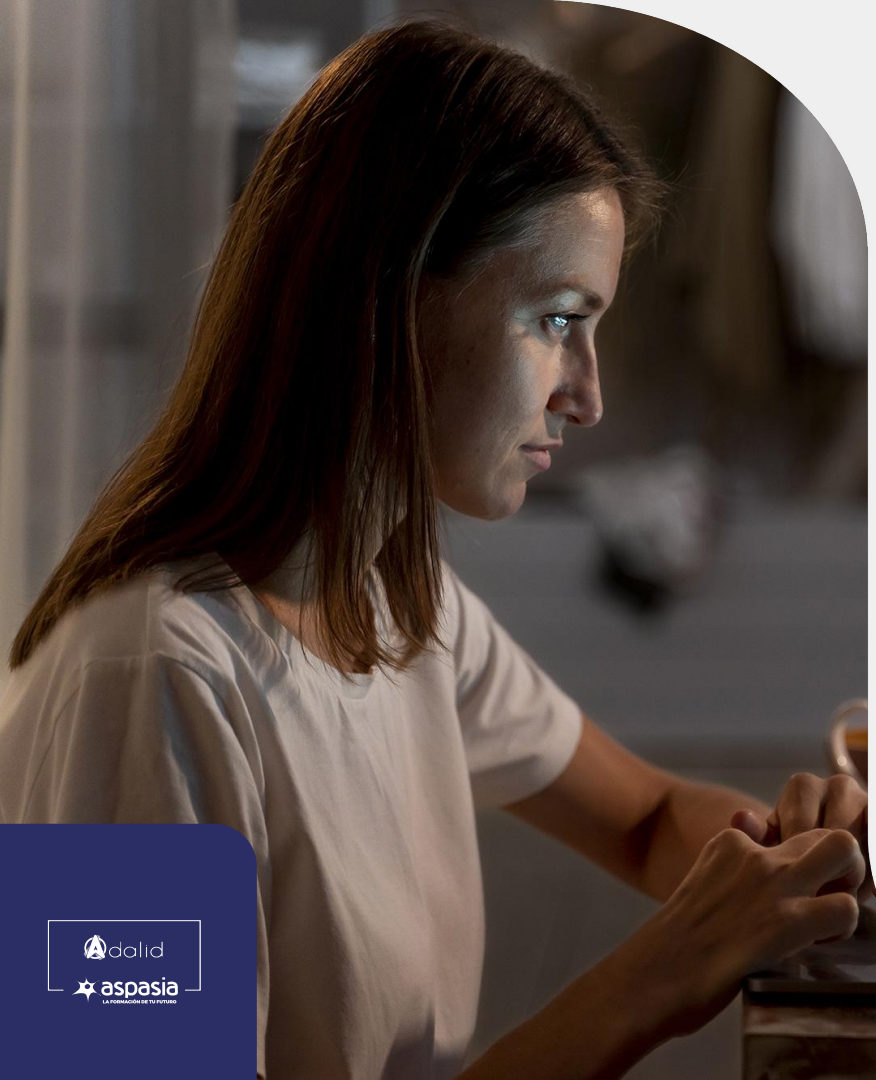
Módulo 4

AE 3.3

OBJETIVOS

Dominar portales, Profiler, reconciliación, validación de tipos, modo estricto, componentes no controlados y WebComponents en React para crear apps rápidas, robustas y modernas.

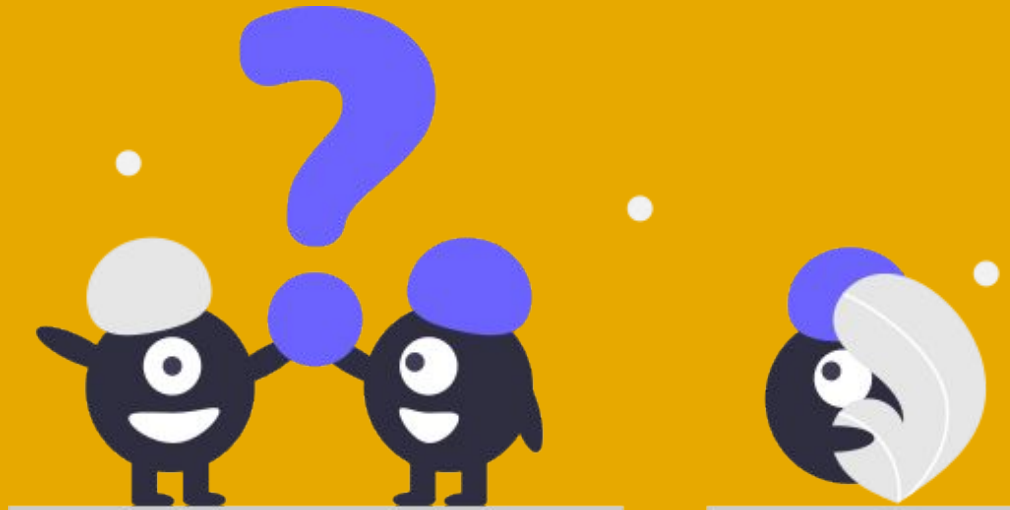
Chile
Adalid
aspasia
LA FORMACIÓN DE TU FUTURO

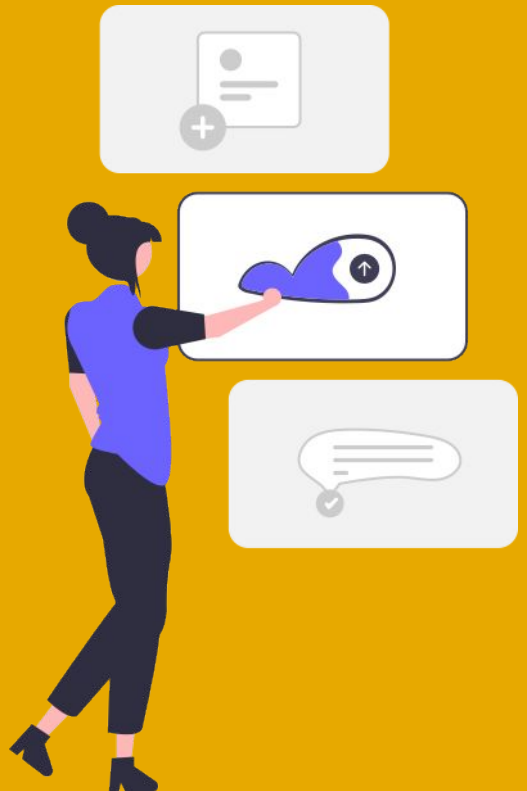


¿QUÉ VAMOS A VER?

- Portales.
- Profiler.
- Reconciliación.
- Comprobación de tipos estáticos.
- Modo estricto.
- Verificación de tipos con PropTypes.
- Componentes no controlados.
- WebComponents.

¿Cómo optimizamos el rendimiento en React?





Elementos avanzados de ReactJS

Adalid
aspasia
LA FORMACIÓN DE TU FUTURO

Continuación del Proyecto

Para este aprendizaje vamos a continuar con el desarrollo de nuestra aplicación que veníamos desarrollando en la clase pasada.

Alista tu proyecto 😁

Portales

Los portales permiten **renderizar componentes React fuera de la jerarquía DOM principal**, manteniendo al mismo tiempo la conexión con el estado y eventos del árbol React. Son útiles para casos como modales, tooltips, o notificaciones, donde el contenido debe estar desacoplado del flujo DOM estándar para evitar problemas de estilo o comportamiento.

Portales

Paso 1. Crear un componente para el modal

- Crea un archivo en src/components/Modal.jsx:

```
import ReactDOM from "react-dom";
import "./Modal.css";

function Modal({ children, onClose }) {
  return ReactDOM.createPortal(
    <div className="modal-overlay" onClick={onClose}>
      <div className="modal-content" onClick={(e) =>
e.stopPropagation()}>
        <button className="modal-close" onClick={onClose}>
          ✖
        </button>
        {children}
      </div>
    </div>,
    document.getElementById("modal-root")
  );
}

export default Modal;
```

Portales

Paso 2. Crear un contenedor para los portales en el HTML

- En el archivo **index.html**, añade el siguiente contenedor dentro del **<body>**:

```
<div id="modal-root"></div>
```

Portales

Paso 3. Crear un componente para usar el modal

- Crea un archivo en src/components/PortalExample.jsx:

```jsx
import { useState } from "react";
import Modal from "./Modal";

function PortalExample() {
  const [isOpen, setIsOpen] = useState(false);

  const openModal = () => setIsOpen(true);
  const closeModal = () => setIsOpen(false);

  return (
    <div>
      <h2>Ejemplo de Portal</h2>
      <button onClick={openModal}>Abrir Modal</button>
      {isOpen && (
        <Modal onClose={closeModal}>
          <h3>Detalles del Evento</h3>
          <p>Aquí puedes añadir información adicional sobre el evento.</p>
        </Modal>
      )}
    </div>
  );
}

export default PortalExample;
```

Portales

Paso 4. Crear una vista para probar los portales

- Crea un archivo en src/views/PortalView.jsx:

```jsx
import PortalExample from "../components/PortalExample";

function PortalView() {
  return (
    <div>
      <h1>Portales en React</h1>
      <PortalExample />
    </div>
  );
}

export default PortalView;
```

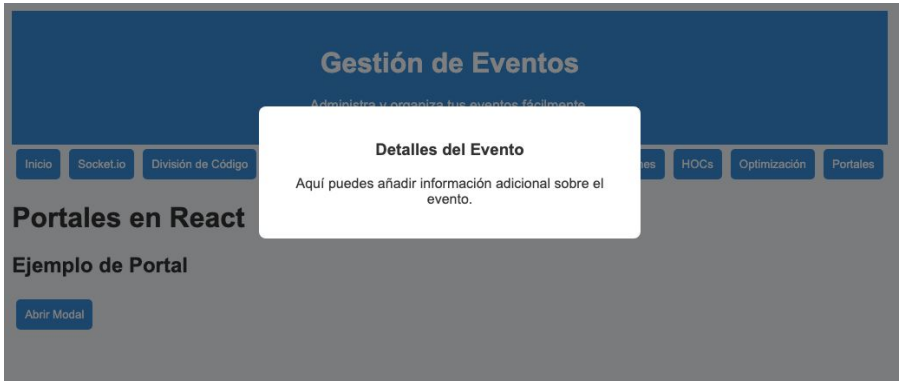
Portales

Paso 5: Añadir la vista al selector de vistas en App.jsx

- Modifica el archivo App.jsx:

```
import PortalView from "./views/PortalView";

// Añade al menú de vistas
<button onClick={() => setView("portals")}>Portales</button>
{view === "portals" && <PortalView />}
```



Profiler

El **React Profiler** es una herramienta para **medir el rendimiento de los componentes** en aplicaciones React. Ayuda a identificar cuántas veces se renderizan los componentes, cuánto tiempo tardan y qué eventos los desencadenan. Esta información es útil para optimizar la experiencia del usuario en aplicaciones grandes o complejas.

Profiler

Paso 1. Crear un componente para medir el rendimiento

- Crea un archivo en src/components/ProfilerExample.jsx:

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

function ProfilerExample() {
  const [count, setCount] = useState(0);

  const increment = () => setCount(count + 1);

  return (
    <div>
      <h3>Contador</h3>
      <p>Valor actual: {count}</p>
      <button onClick={increment}>Incrementar</button>
    </div>
  );
}

export default ProfilerExample;
```

Profiler

Paso 2. Integrar el Profiler en un componente

- Crea un archivo en src/views/ProfilerView.jsx:

```
import { Profiler } from "react";
import ProfilerExample from "../components/ProfilerExample";

function ProfilerView() {
  const onRenderCallback = (
    id, // Nombre del Profiler
    phase, // "mount" o "update"
    actualDuration // Duración del renderizado
  ) => {
    console.log(
      `${id} (${phase}) tomó ${actualDuration.toFixed(2)}ms para renderizar.`
    );
  };

  return (
    <div>
      <h1>React Profiler</h1>
      <Profiler id="ProfilerExample" onRender={onRenderCallback}>
        <ProfilerExample />
      </Profiler>
    </div>
  );
}

export default ProfilerView;
```

Profiler

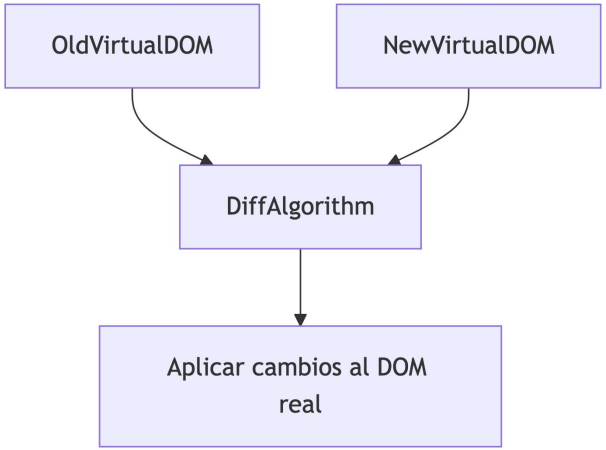
Paso 4: Añadir la vista al selector de vistas en App.jsx

- Modifica el archivo App.jsx:

```
import ProfilerView from "./views/ProfilerView";

// Añade al menú de vistas
<button onClick={() => setView("profiler")}>React
Profiler</button>
{view === "profiler" && <ProfilerView />}
```

Reconciliación



La reconciliación es el proceso interno de React para determinar qué **cambios debe realizar en el DOM**. Cuando se actualiza un componente, React compara el nuevo árbol de elementos virtuales (virtual DOM) con el anterior, identifica las diferencias (diffing), y aplica los cambios necesarios al DOM real de forma eficiente.

Reconciliación

Paso 1. Crear un componente con lista dinámica

- Crea un archivo en src/components/ReconciliationList.jsx:

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

function ReconciliationList() {
  const [items, setItems] = useState([
    { id: 1, text: "Elemento 1" },
    { id: 2, text: "Elemento 2" },
    { id: 3, text: "Elemento 3" },
  ]);

  const addItem = () => {
    const newItem = { id: Math.random(), text: `Elemento ${items.length + 1}` };
    setItems([...items, newItem]);
  };

  const shuffleItems = () => {
    setItems((prevItems) => [...prevItems].sort(() => Math.random() - 0.5));
  };

  return (
    <div>
      <h3>Lista de Reconciliación</h3>
      <button onClick={addItem}>Añadir Elemento</button>
      <button onClick={shuffleItems}>Mezclar Elementos</button>
      <ul>
        {items.map((item) => (
          <li key={item.id}>{item.text}</li>
        ))}
      </ul>
    </div>
  );
}

export default ReconciliationList;
```

Reconciliación

Paso 2. Crear un componente sin claves únicas (mal uso de reconciliación)

- Crea un archivo en src/components/BadReconciliationList.jsx:

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

function BadReconciliationList() {
  const [items, setItems] = useState(["Elemento A", "Elemento B",
"Elemento C"]);

  const addItem = () => {
    const newItem = `Elemento ${String.fromCharCode(65 + items.length)}`;
    setItems([...items, newItem]);
  };

  const shuffleItems = () => {
    setItems((prevItems) => [...prevItems].sort(() => Math.random() - 0.5));
  };

  return (
    <div>
      <h3>Lista sin Claves Únicas</h3>
      <button onClick={addItem}>Añadir Elemento</button>
      <button onClick={shuffleItems}>Mezclar Elementos</button>
      <ul>
        {items.map((item, index) => (
          <li key={index}>{item}</li> // Mal uso del índice como clave
        ))}
      </ul>
    </div>
  );
}

export default BadReconciliationList;
```

Reconciliación

Paso 3. Crear una vista para probar la reconciliación

- Crea un archivo en src/views/ReconciliationView.jsx:

```
import ReconciliationList from
"../components/ReconciliationList";
import BadReconciliationList from
"../components/BadReconciliationList";

function ReconciliationView() {
  return (
    <div>
      <h1>Reconciliación en React</h1>
      <p>Ejemplo de una lista bien implementada y una
lista con claves incorrectas.</p>
      <ReconciliationList />
      <BadReconciliationList />
    </div>
  );
}

export default ReconciliationView;
```

Reconciliación

Paso 4: Añadir la vista al selector de vistas en App.jsx

- Modifica el archivo App.jsx:

```
import ReconciliationView from "./views/ReconciliationView";

// Añade al menú de vistas
<button onClick={() =>
setView("reconciliation")}>Reconciliación</button>
{view === "reconciliation" && <ReconciliationView />}
```

Observa la consola del navegador

Comprobación de tipos estáticos

La comprobación de tipos estáticos implica **validar los tipos de datos** en tiempo de desarrollo para garantizar que se usen correctamente dentro de la aplicación. En React, herramientas como **TypeScript** o **Flow** permiten definir tipos para props, estados y funciones.

A diferencia de PropTypes, que funciona en tiempo de ejecución, las herramientas de tipos estáticos realizan las validaciones antes de ejecutar el código, detectando errores en tiempo de desarrollo.

Modo estricto (Strict Mode)

El modo estricto en React (StrictMode) es una herramienta para **detectar posibles problemas en la aplicación**. No renderiza nada visualmente, pero activa advertencias en el entorno de desarrollo para ayudarte a identificar prácticas obsoletas, problemas con ciclos de vida, o cualquier comportamiento inesperado.

Modo estricto (Strict Mode)

Paso 1. Verifica que tienes el modo estricto activado

- Revisa tu archivo main.jsx:

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import App from './App.jsx'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <App />
  </StrictMode>,
)
```

Modo estricto (Strict Mode)

Paso 2. Crear un componente para mostrar problemas comunes

- Crea un archivo en src/components/StrictModeExample.jsx:

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

function StrictModeExample() {
  const [count, setCount] = useState(0);

  // Simulación de un problema común: efectos secundarios ejecutados dos veces en modo desarrollo
  useEffect(() => {
    console.log("Efecto ejecutado");
  }, []);

  return (
    <div>
      <h3>Ejemplo de Modo Estricto</h3>
      <p>El valor actual del contador es: {count}</p>
      <button onClick={() => setCount(count + 1)}>Incrementar</button>
    </div>
  );
}

export default StrictModeExample;
```

Modo estricto (Strict Mode)

Paso 3. Crear una vista para probar el Modo Estricto

- Crea un archivo en src/views/StrictModeView.jsx:

```
import StrictModeExample from
"../components/StrictModeExample";

function StrictModeView() {
  return (
    <div>
      <h1>Modo Estricto</h1>
      <p>
        Este ejemplo demuestra cómo `React.StrictMode`
detecta problemas
        potenciales como efectos secundarios o métodos
obsoletos.
      </p>
      <StrictModeExample />
    </div>
  );
}

export default StrictModeView;
```

Modo estricto (Strict Mode)

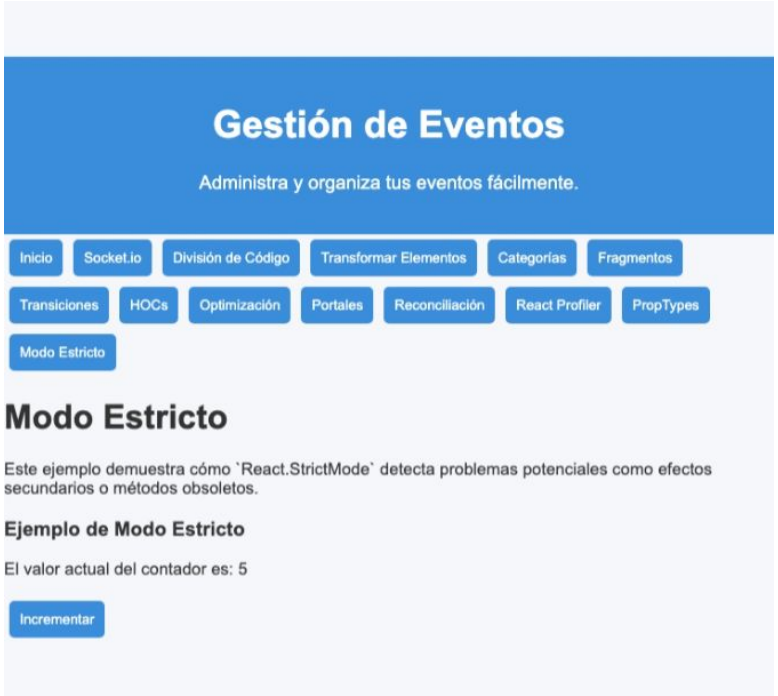
Paso 4: Añadir la vista al selector de vistas en App.jsx

- Modifica el archivo App.jsx:

```
import StrictModeView from "./views/StrictModeView";

// Añade al menú de vistas
<button onClick={() => setView("strictmode")}>Modo
Estricto</button>
{view === "strictmode" && <StrictModeView />}
```

Observa la consola del navegador



Verificación de tipos con PropTypes

PropTypes es una herramienta de React para verificar los **tipos de props en componentes**. Es especialmente útil en aplicaciones grandes para evitar errores al pasar datos incorrectos entre componentes.

- Detecta errores en desarrollo relacionados con tipos.
- Mejora la robustez del código.
- Documenta explícitamente las props esperadas.

Verificación de tipos con PropTypes

Paso 1. Instalar PropTypes

- Aunque PropTypes está integrado en React, se debe instalar como paquete independiente:

```
npm install prop-types
```

Verificación de tipos con PropTypes

Paso 2. Añadir PropTypes a un componente simple

- Crea un archivo en src/components/PropTypeExample.jsx:

```jsx
import PropTypes from "prop-types";

function PropTypeExample({ title, count }) {
  return (
    <div>
      <h3>{title}</h3>
      <p>El valor del contador es: {count}</p>
    </div>
  );
}

// Validación de tipos de propiedades
PropTypeExample.propTypes = {
  title: PropTypes.string.isRequired, // Propiedad obligatoria de tipo string
  count: PropTypes.number, // Propiedad opcional de tipo number
};

// Valores predeterminados para las propiedades
PropTypeExample.defaultProps = {
  count: 0, // Si no se pasa `count`, usará 0 por defecto
};

export default PropTypeExample;
```

Verificación de tipos con PropTypes

Paso 3. Crear un componente que pase las propiedades

- Crea un archivo en src/components/PropTypeParent.jsx:

```
import PropTypeExample from "./PropTypeExample";

function PropTypeParent() {
  return (
    <div>
      <h2>Ejemplo de PropTypes</h2>
      <PropTypeExample title="Título válido" count={10} />
      <PropTypeExample title="Solo título" />
      {/* <PropTypeExample count={20} />  Esto lanzará una advertencia en la consola */}
    </div>
  );
}

export default PropTypeParent;
```

Verificación de tipos con PropTypes

Paso 4. Crear una vista para probar PropTypes

- Crea un archivo en src/views/PropTypesView.jsx:

```jsx
import PropTypeParent from
"../components/PropTypeParent";

function PropTypesView() {
  return (
    <div>
      <h1>Validación con PropTypes</h1>
      <PropTypeParent />
    </div>
  );
}

export default PropTypesView;
```

Verificación de tipos con PropTypes

Paso 5: Añadir la vista al selector de vistas en App.jsx

- Modifica el archivo App.jsx:

```
import PropTypesView from "./views/PropTypesView";

// Añade al menú de vistas
<button onClick={() =>
setView("proptypes")}>PropTypes</button>
{view === "proptypes" && <PropTypesView />}
```

Componentes no controlados

Un componente no controlado en React es aquel que no gestiona su estado mediante el **estado interno de React (useState)**. En cambio, el estado del componente se administra directamente en el DOM utilizando **referencias (ref)**. Esto es útil para manejar formularios o elementos del DOM donde no es necesario un control complejo.

Componentes no controlados

Paso 2. Crear una vista para probar el formulario no controlado

- Crea un archivo en src/views/UncontrolledView.jsx:

```jsx
import UncontrolledForm from "../components/UncontrolledForm";

function UncontrolledView() {
  return (
    <div>
      <h1>Componentes No Controlados</h1>
      <p>
        Este ejemplo muestra cómo manejar datos de formularios directamente
        desde el DOM usando referencias (`ref`).
      </p>
      <UncontrolledForm />
    </div>
  );
}

export default UncontrolledView;
```

Componentes no controlados

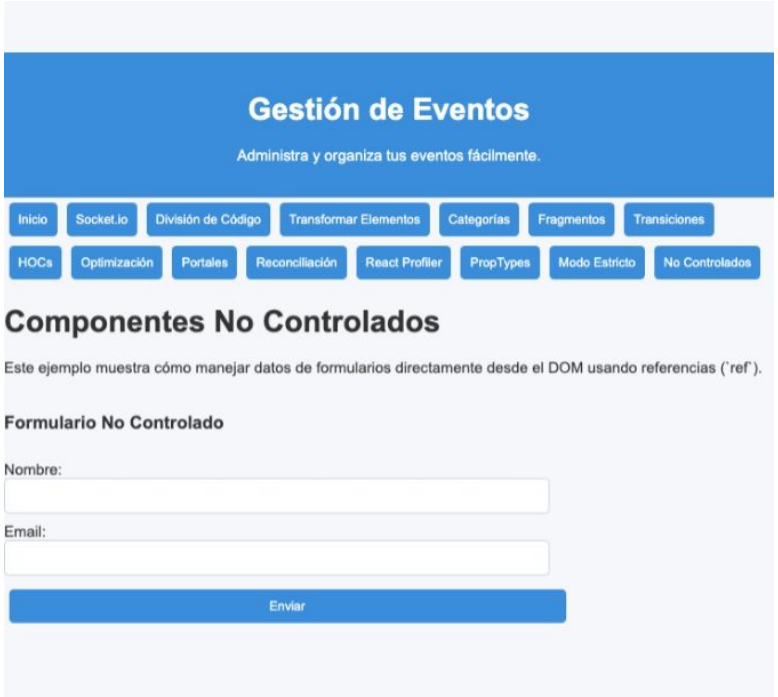
Paso 3: Añadir la vista al selector de vistas en App.jsx

- Modifica el archivo App.jsx:

```
import UncontrolledView from "./views/UncontrolledView";

// Añade al menú de vistas
<button onClick={() => setView("uncontrolled")}>No
Controlados</button>
{view === "uncontrolled" && <UncontrolledView />}
```

Observa la consola del navegador



WebComponents

Los WebComponents son una tecnología nativa de los navegadores para crear componentes reutilizables y encapsulados utilizando HTML, CSS y JavaScript estándar. Estos componentes **funcionan de manera independiente** de cualquier framework, como React o Angular.

WebComponents

Paso 1. Crear un WebComponent personalizado

- Crea un archivo en src/webcomponents/MyCustomComponent.js:

```
class MyCustomComponent extends HTMLElement {
  constructor() {
    super();
    const shadow = this.attachShadow({ mode: "open" });

    const wrapper = document.createElement("div");
    wrapper.setAttribute("class", "custom-component");

    const style = document.createElement("style");
    style.textContent = `
      .custom-component {
        display: inline-block;
        padding: 10px;
        background-color: #4caf50;
        color: white;
        font-size: 14px;
        border-radius: 5px;
        text-align: center;
        cursor: pointer;
      }

      .custom-component:hover {
        background-color: #45a049;
      }
    `;

    const content = document.createElement("span");
    content.textContent = this.getAttribute("text") || "Componente Personalizado";

    wrapper.appendChild(content);
    shadow.appendChild(style);
    shadow.appendChild(wrapper);
  }
}

// Registrar el WebComponent
customElements.define("my-custom-component", MyCustomComponent);
```

WebComponents

Paso 2. Registrar el WebComponent en React

- En el archivo src/main.jsx, importa y registra el WebComponent:

```
import "./webcomponents/MyCustomComponent";
```

WebComponents

Paso 3. Usar el WebComponent en un componente React

- Crea un archivo en src/components/WebComponentExample.jsx:

```
function WebComponentExample() {
  return (
    <div>
      <h3>Ejemplo de WebComponent</h3>
      <my-custom-component text="¡Hola desde WebComponent!" />
    </div>
  );
}

export default WebComponentExample;
```

WebComponents

Paso 4. Crear una vista para probar el WebComponent

- Crea un archivo en src/views/WebComponentView.jsx:

```
import WebComponentExample from "../components/WebComponentExample";

function WebComponentView() {
  return (
    <div>
      <h1>WebComponents</h1>
      <p>
        Este ejemplo muestra cómo crear e integrar un WebComponent en una aplicación React.
      </p>
      <WebComponentExample />
    </div>
  );
}

export default WebComponentView;
```

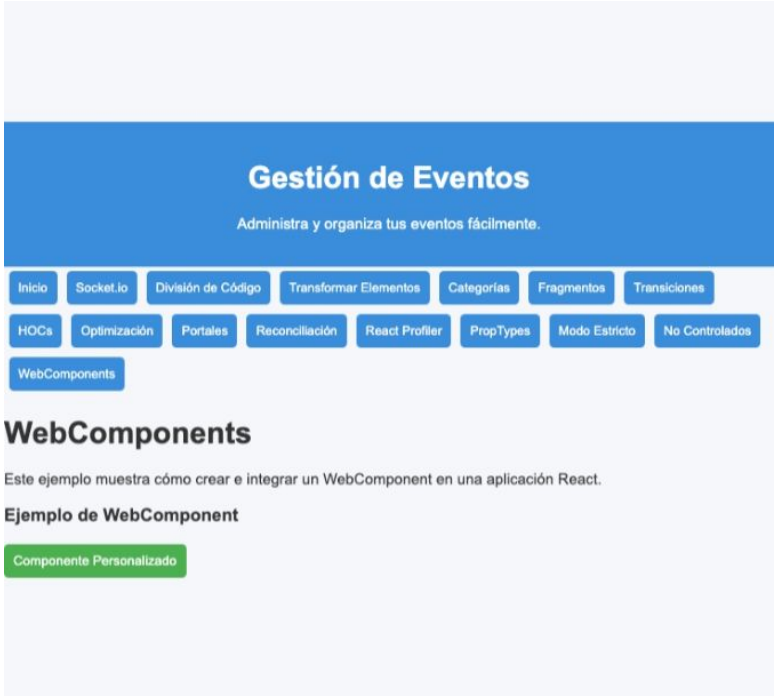
Componentes no controlados

Paso 3: Añadir la vista al selector de vistas en App.jsx

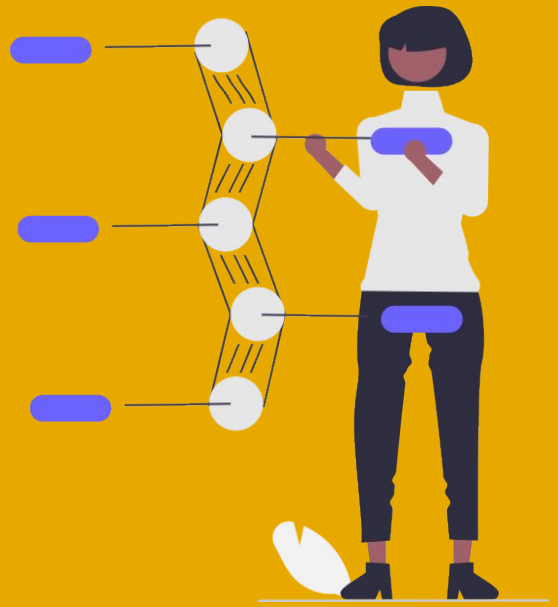
- Modifica el archivo App.jsx:

```
import WebComponentView from "./views/WebComponentView";

// Añade al menú de vistas
<button onClick={() =>
setView("webcomponents")}>WebComponents</button>
{view === "webcomponents" && <WebComponentView />}
```



Resumen de lo aprendido 🧑‍🎓



Adalid aspasia

Resumen de lo aprendido

- **Interactividad avanzada:** Usar Portales para componentes fuera del DOM principal y Profiler para optimizar rendimiento.
- **Eficiencia y estructura:** Comprender la reconciliación para actualizaciones rápidas y validar tipos con TypeScript o Flow.
- **Calidad en desarrollo:** Detectar problemas con Modo Estricto y verificar props con PropTypes.
- **Flexibilidad:** Manejar datos con componentes no controlados y crear WebComponents reutilizables y encapsulados. 🚀

GRACIAS POR TU ATENCIÓN

Nos vemos en la próxima clase

Chile
Adalid
aspasia
LA FORMACIÓN DE TU FUTURO