Módulo 4
Desarrollo de interfaces interactivas con React

Elementos avanzados de ReactJS

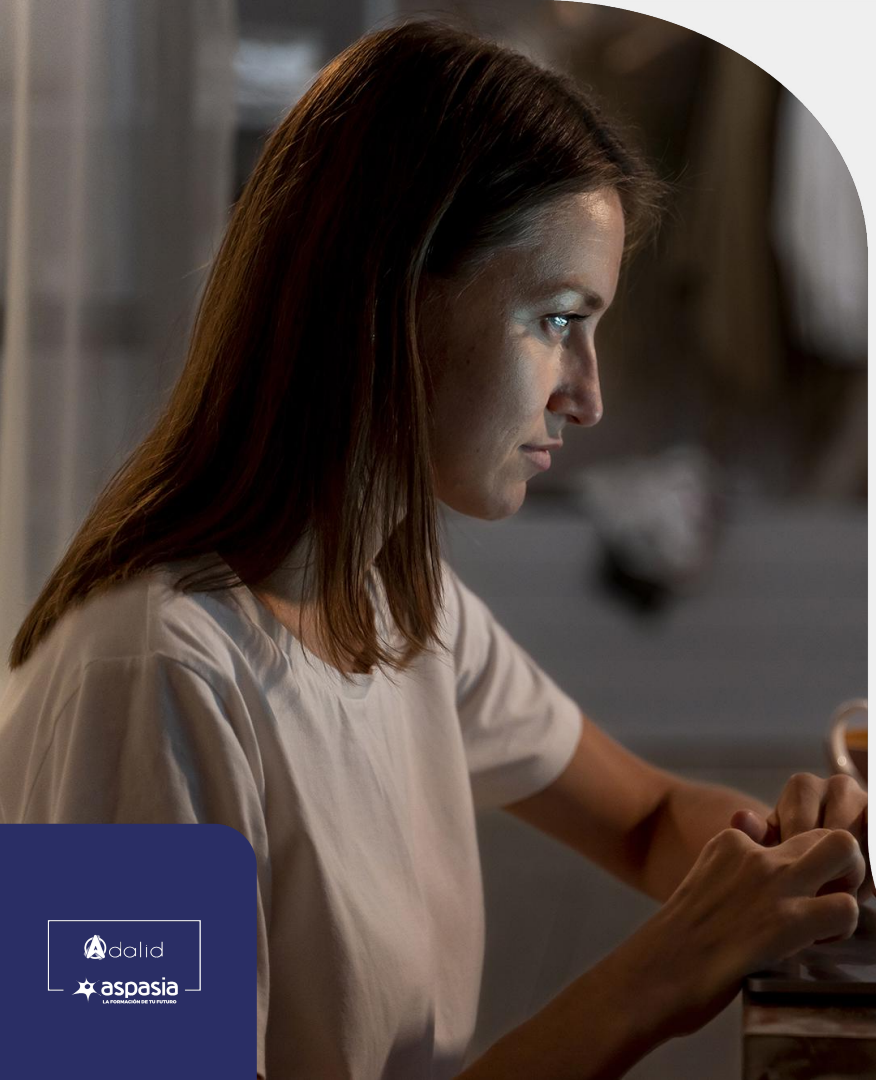
Módulo 4

AE 3.2

OBJETIVOS

Explorar y dominar elementos avanzados de ReactJS como Socket.io, Context API, Fragmentos, Transitions y Optimización de rendimiento.

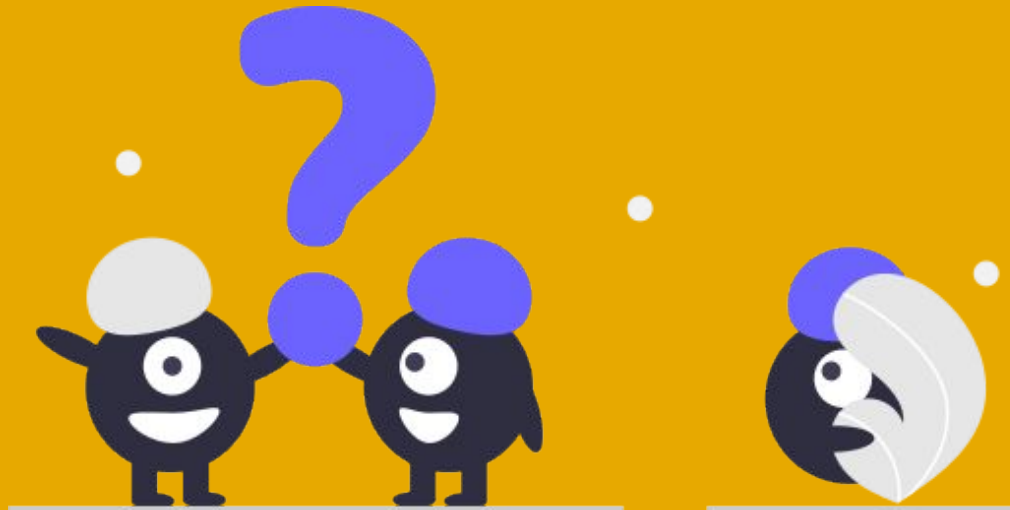
Chile
Adalid
aspasia
LA FORMACIÓN DE TU FUTURO

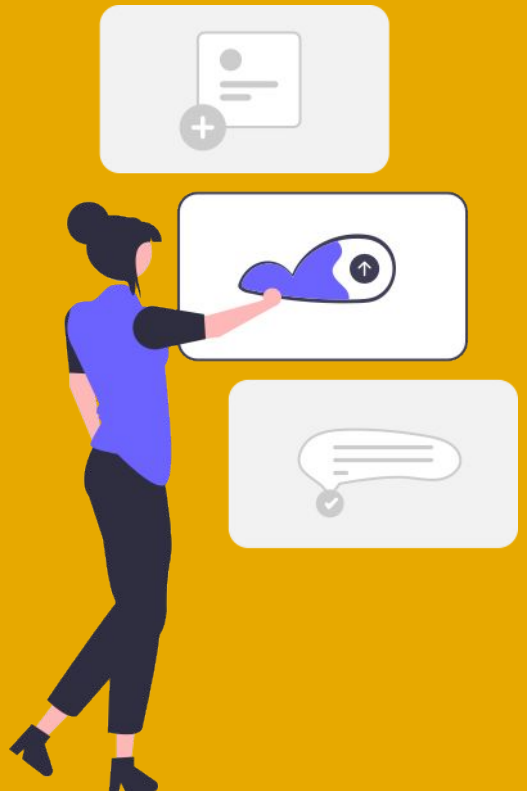


¿QUÉ VAMOS A VER?

- Elementos avanzados de ReactJS.
- Manejo de socket.io.
- División de código.
- Transformar elementos.
- Contexto.
- Fragmentos.
- Transitions.
- Componentes de orden superior.
- Optimizando el rendimiento.

¿Cómo optimizamos el rendimiento en React?





Elementos avanzados de ReactJS

Adalid
aspasia
LA FORMACIÓN DE TU FUTURO

Configuración Base del Proyecto

Para este aprendizaje vamos a usar como base un proyecto que está iniciado https://github.com/adalid-cl/event-manager, poco a poco cubriremos las temáticas mientras avanzamos en el desarrollo de nuestro proyecto.

Alista tu terminal 😁

Configuración Inicial del Proyecto

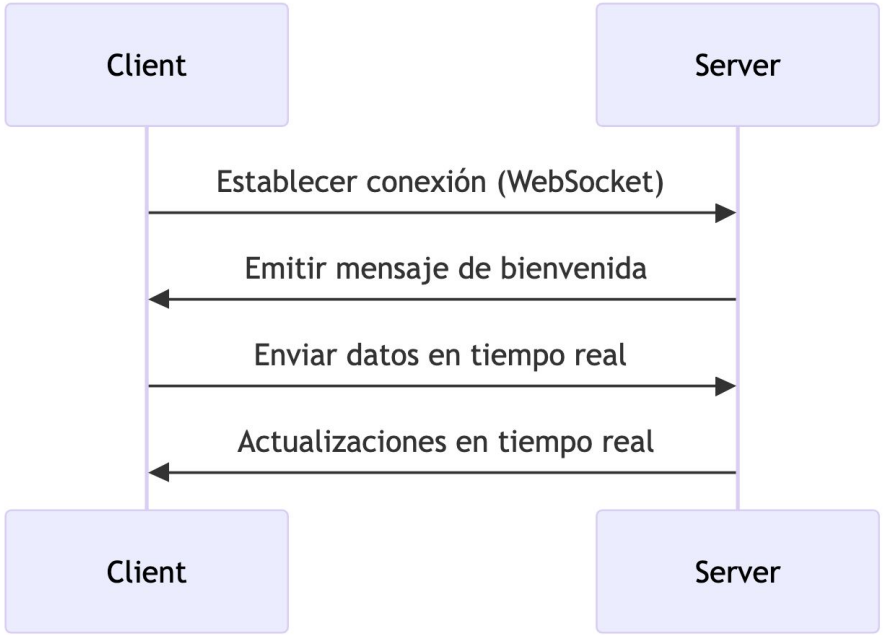
Sigue las instrucciones del **README** para poder clonarlo y tenerlo en tu equipo, revisa los archivos y carpetas que en él se encuentran.

```
/event-manager-app/
src/
├── assets/          // Carpeta para los Recursos estáticos
├── components/      // Carpeta para los Componentes
├── context/         // Carpeta para manejar Context API
├── hocs/            // Carpeta para los Componentes de Orden Superior
├── services/        // Carpeta para los Servicios externos (e.g., Socket.io)
├── utils/           // Carpeta para las Funciones auxiliares
├── views/           // Carpeta para las vistas
├── webcomponents/   // Carpeta para los Componentes nativos del navegador
├── App.css          // Archivo de Estilos globales
├── App.jsx          // Archivo de Componente raíz
└── main.jsx         // Archivo de Punto de entrada
```

```
/event-manager-server/
├── server.js
├── package.json
```

Manejo de Socket.IO

Socket.IO permite la comunicación en tiempo real entre **cliente** y **servidor**, ideal para chats, notificaciones, o aplicaciones colaborativas.



Manejo de Socket.IO

Dentro del proyecto **event-manager** existe una carpeta llamada **event-manager-server**, este es un servidor que usa **Socket.IO,** asegurate de acceder a la carpeta, instalar las dependencias y ejecutar el servidor. Revisa el archivo **server.js**.

```
cd event-manager-server
npm install
node server.js
```

```
const { Server } = require("socket.io");

// Permite conexiones desde cualquier origen
const io = new Server(4000, {
  cors: {
    origin: "*",
  },
});

// Array para almacenar los eventos
let events = [];

// Configuración de Socket.io
io.on("connection", (socket) => {
  console.log("Cliente conectado:", socket.id);

  // Enviar la lista inicial de eventos al cliente
  socket.emit("event:list", events);

  // Escuchar nuevos eventos desde el cliente
  socket.on("event:add", (newEvent) => {
    events.push(newEvent);

    // Enviar actualizaciones a todos los clientes conectados
    io.emit("event:update", events);
  });
});
```

Manejo de Socket.IO

Dentro del proyecto **event-manager** existe una carpeta llamada **event-manager-app**, esta es nuestra app para el frontend**,** asegurate de acceder a la carpeta, instalar las dependencias y ejecutar el servidor. Revisa su estructura de carpetas y archivos.

```
cd event-manager-app
npm install
npm run dev
```

Manejo de Socket.IO

Para poder usar Socket.IO en el frontend necesitamos una librería llamada **socket.io-client** esta nos permitirá la comunicación con el servidor que estamos ejecutando, posteriormente ejecutamos el servidor.

```
npm install socket.io-client
npm run dev
```

Manejo de Socket.IO

Paso 1. Crear el Contexto de Socket.io.

- src/context/SocketContext.jsx:

```
import { createContext, useEffect, useState } from "react";
import { io } from "socket.io-client";

export const SocketContext = createContext();

// Cambia esta URL si está en producción
const socket = io("http://localhost:4000");
```

```
export default function SocketProvider({ children }) {
  const [events, setEvents] = useState([]);

  useEffect(() => {
    // Obtener lista inicial de eventos
    socket.on("event:list", (data) => setEvents(data));

    // Actualizar eventos en tiempo real
    socket.on("event:update", (data) => setEvents(data));

    return () => {
      socket.off("event:list");
      socket.off("event:update");
    };
  }, []);

  // Emitir nuevo evento al servidor
  const addEvent = (newEvent) => {
    socket.emit("event:add", newEvent);
  };

  return (
    <SocketContext.Provider value={{ events, addEvent }}>
      {children}
    </SocketContext.Provider>
  );
}
```

Manejo de Socket.IO

Paso 2: Crear el Formulario de Eventos

- src/components/SocketForm.jsx:

```
import { useState, useContext } from "react";
import { SocketContext } from "../context/SocketContext";

function SocketForm() {
  // Estado local para el título del evento
  const [title, setTitle] = useState("");
  // Obtener la función addEvent del contexto
  const { addEvent } = useContext(SocketContext);

  // Manejar el envío del formulario
  const handleSubmit = (e) => {
    e.preventDefault();
    // Crear un nuevo evento con el título, fecha actual y estado "Pendiente"
    const newEvent = { title, date: new Date().toLocaleString(), status:
"Pendiente" };
    // Añadir el nuevo evento usando la función del contexto
    addEvent(newEvent);
    // Limpiar el formulario
    setTitle("");
  };
```

```
  return (
    <form onSubmit={handleSubmit}>
      {/* Campo de entrada para el título del evento */}
      <input
        type="text"
        value={title}
        onChange={(e) => setTitle(e.target.value)}
        placeholder="Nuevo evento"
      />
      {/* Botón para enviar el formulario */}
      <button type="submit">Añadir Evento</button>
    </form>
  );
}

export default SocketForm;
```

Manejo de Socket.IO

Paso 3: Crear la Lista de Eventos

- src/components/SocketEventList.jsx:

```jsx
import { useContext } from "react";
import { SocketContext } from "../context/SocketContext";

// Componente que muestra la lista de eventos
function SocketEventList() {
  // Obtenemos los eventos del contexto
  const { events } = useContext(SocketContext);

  return (
    <div className="event-list">
      {/*  Por cada evento, mostramos el título, la fecha y el estado */}
      {events.map((event, index) => (
        <div key={index} className="event-item">
          <h3>{event.title}</h3>
          <p>Fecha: {event.date}</p>
          <p>Estado: {event.status}</p>
        </div>
      ))}
    </div>
  );
}

export default SocketEventList;
```

Manejo de Socket.IO

Paso 4: Crear una nueva vista para Socket.io.

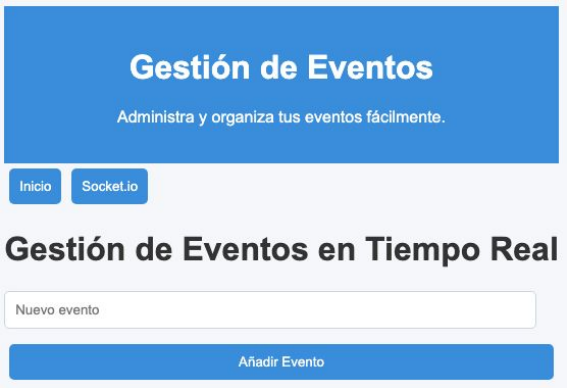
- src/views/SocketIOView.jsx:
  - Importa los componentes que hacen parte de la vista.
  - Agrega los componentes en la vista dentro de la función.

```jsx
import SocketProvider from "../context/SocketContext";
import SocketForm from "../components/SocketForm";
import SocketEventList from "../components/SocketEventList";

function SocketIOView() {
  return (
    <SocketProvider>
      <div>
        <h1>Gestión de Eventos en Tiempo Real</h1>
        <SocketForm />
        <SocketEventList />
      </div>
    </SocketProvider>
  );
}

export default SocketIOView;
```

Manejo de Socket.IO

Paso 5: Modificar App.jsx para incluir la nueva vista

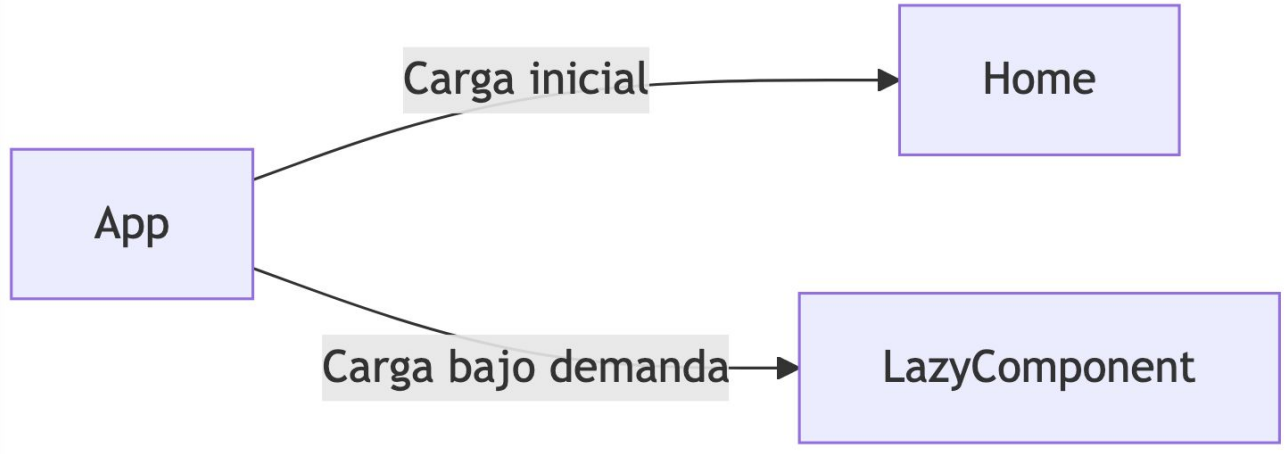
- Accede desde el navegador a la vista del socket por medio de los botones y añade un evento al formulario.
- Puedes detener el servidor **server.js** y ver qué ocurre.




```jsx
import { useState } from "react";
import Header from "./components/Header";
import Home from "./views/Home";
import SocketIOView from "./views/SocketIOView";
import "./App.css";

function App() {
  // Estado para controlar la vista actual
  const [view, setView] = useState("home");

  return (
    <div className="App">
      <Header />
      <nav>
        {/* Botones para cambiar la vista */}
        <button onClick={() => setView("home")}>Inicio</button>
        <button onClick={() => setView("socketio")}>Socket.io</button>
      </nav>
      {/* Mostramos la vista actual */}
      {view === "home" && <Home />}
      {view === "socketio" && <SocketIOView />}
    </div>
  );
}

export default App;
```

División de código (Code Splitting)

La división de código (code splitting) permite separar la aplicación en diferentes "paquetes" o "chunks" para **cargar solo las partes necesarias**. Esto optimiza el rendimiento al cargar solo el código requerido para la vista actual y posponer el resto hasta que sea necesario.



División de código (Code Splitting)

Paso 1: Crear un nuevo componente diferido

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

```
function LazyLoadedComponent() {
  return (
    <div>
      <h2>Componente Cargado Dinámicamente</h2>
      <p>Este componente se carga bajo demanda utilizando React.lazy.</p>
    </div>
  );
}

export default LazyLoadedComponent;
```

División de código (Code Splitting)

Paso 2: Crear una nueva vista para probar la carga diferida

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

```
import React, { Suspense, useState } from "react";

// Componente cargado dinámicamente
const LazyLoadedComponent = React.lazy(() =>
  import("../components/LazyLoadedComponent")
);
```

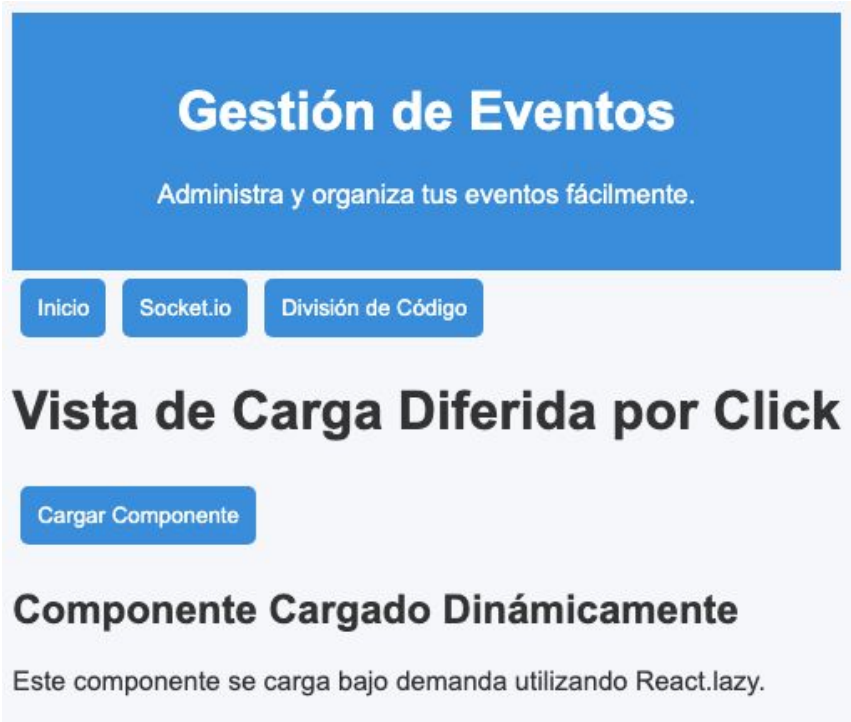
```
function LazyLoadingView() {
  // Estado para controlar la carga del componente
  const [loadComponent, setLoadComponent] = useState(false);

  return (
    <div>
      <h1>Vista de Carga Diferida por Click</h1>
      {/* Botón para cargar el componente */}
      <button onClick={() => setLoadComponent(true)}>
        Cargar Componente
      </button>
      {/* Carga del componente */}
      {loadComponent && (
        <Suspense fallback={<p>Cargando componente...</p>}>
          <LazyLoadedComponent />
        </Suspense>
      )}
    </div>
  );
}

export default LazyLoadingView;
```

División de código (Code Splitting)

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

- Modifica el archivo App.jsx:

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

// Añade al menú de vistas
<button onClick={() => setView("lazy")}>División de Código</button>
{view === "lazy" && <LazyLoadingView />}
```



Transformar elementos

Transformar elementos en React implica modificar cómo se renderizan los componentes en función de parámetros dinámicos, como cambiar un elemento HTML de div a section o agregar clases condicionalmente. Este enfoque permite crear **componentes altamente reutilizables y flexibles**, adaptándose a diferentes casos de uso.

Transformar elementos

Paso 1: Crear un nuevo componente para transformar eventos

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

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

function TransformableEvent({ event }) {
  // Estado para controlar el estado del evento
  const [status, setStatus] = useState(event.status);

  // Función para cambiar el estado del evento
  const toggleStatus = () => {
    setStatus((prevStatus) =>
      prevStatus === "Pendiente" ? "Completado" : "Pendiente"
    );
  };

  return (
    <div
      className={`event-item ${status === "Completado" ? "completed" : ""}`}
      onClick={toggleStatus}
    >
      <h3>{event.title}</h3>
      <p>Fecha: {event.date}</p>
      <p>Estado: {status}</p>
    </div>
  );
}

export default TransformableEvent;
```

Transformar elementos

Paso 2: Crear una nueva lista de eventos transformables

- Crea un archivo en src/components/TransformableEventList.jsx

```jsx
import TransformableEvent from "./TransformableEvent";

function TransformableEventList() {
  // Lista de eventos
  const events = [
    { id: 1, title: "Reunión de equipo", date: "2024-12-20", status: "Pendiente" },
    { id: 2, title: "Presentación de proyecto", date: "2024-12-22", status: "Pendiente" },
  ];

  return (
    <div className="event-list">
      {events.map((event) => (
        <TransformableEvent key={event.id} event={event} />
      ))}
    </div>
  );
}

export default TransformableEventList;
```

Transformar elementos

Paso 3: Crear una nueva vista para transformación de eventos

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

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

function TransformEventsView() {
  return (
    <div>
      <h1>Transformación de Elementos</h1>
      <p>Haz clic en un evento para cambiar su estado.</p>
      <TransformableEventList />
    </div>
  );
}

export default TransformEventsView;
```

Transformar elementos

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

- Modifica el archivo App.jsx:

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

// Añade al menú de vistas
<button onClick={() => setView("transform")}>Transformar Elementos</button>
{view === "transform" && <TransformEventsView />}
```

Contexto

React Context se utiliza para compartir datos globales (temas, idiomas, autenticación) **sin pasar props manualmente**.

- Elimina el paso de props innecesarias.
- Centraliza el manejo de datos globales.
- Simplifica la comunicación entre componentes no relacionados directamente.

Contexto

Paso 1: Crear el Contexto de Categorías

- Crea un archivo en src/context/CategoryContext.jsx:

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

// Creamos el contexto
export const CategoryContext = createContext();
```

```
// Creamos el provider
export default function CategoryProvider({ children }) {
  // Estado para almacenar las categorías
  const [categories, setCategories] = useState([
    "Reuniones",
    "Proyectos",
    "Social",
  ]);

  // Función para agregar una categoría
  const addCategory = (category) => {
    setCategories((prev) => [...prev, category]);
  };

  return (
    <CategoryContext.Provider value={{ categories, addCategory
}}>
      {children}
    </CategoryContext.Provider>
  );
}
```

Contexto

Paso 2: Crear un formulario para añadir categorías

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

```jsx
import { useState, useContext } from "react";
import { CategoryContext } from "../context/CategoryContext";

// Componente para el formulario de categorías
function CategoryForm() {
  // Estado para almacenar el valor
  const [newCategory, setNewCategory] = useState("");
  const { addCategory } = useContext(CategoryContext);

  // Función para manejar el envío del formulario
  const handleSubmit = (e) => {
    e.preventDefault();
    if (newCategory.trim()) {
      addCategory(newCategory);
      setNewCategory("");
    }
  };
```

```jsx
  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={newCategory}
        onChange={(e) => setNewCategory(e.target.value)}
        placeholder="Nueva Categoría"
      />
      <button type="submit">Añadir</button>
    </form>
  );
}

export default CategoryForm;
```

Contexto

Paso 3: Crear una lista para mostrar las categorías

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

```jsx
import { useContext } from "react";
import { CategoryContext } from "../context/CategoryContext";

function CategoryList() {
  // Obtenemos las categorías del contexto
  const { categories } = useContext(CategoryContext);

  return (
    <div>
      <h3>Categorías</h3>
      <ul>
        {/* Mostramos la lista de categorías */}
        {categories.map((category, index) => (
          <li key={index}>{category}</li>
        ))}
      </ul>
    </div>
  );
}

export default CategoryList;
```

Contexto

Paso 4: Crear una vista para manejar categorías

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

```
import CategoryProvider from "../context/CategoryContext";
import CategoryForm from "../components/CategoryForm";
import CategoryList from "../components/CategoryList";

function CategoryView() {
  return (
    <CategoryProvider>
      <div>
        <h1>Gestión de Categorías</h1>
        <CategoryForm />
        <CategoryList />
      </div>
    </CategoryProvider>
  );
}

export default CategoryView;
```

Contexto

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

- Modifica el archivo App.jsx:

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

// Añade al menú de vistas
<button onClick={() => setView("categories")}>Categorías</button>
{view === "categories" && <CategoryView />}
```

Fragmentos

Los fragmentos son una característica de React que permite **agrupar varios elementos JSX** sin añadir nodos innecesarios al DOM. Antes de los fragmentos, los desarrolladores solían envolver múltiples elementos dentro de un contenedor adicional como un div, lo que podía llevar a un DOM más complejo y difícil de manejar. Con los fragmentos, se pueden evitar estos contenedores innecesarios.

Fragmentos

Paso 1: Crear un nuevo componente con fragmentos

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

```
function FragmentExample() {
  return (
    <>
      <h2>Título del Componente</h2>
      <p>Este componente utiliza fragmentos para agrupar elementos sin añadir nodos extra al DOM.</p>
    </>
  );
}

export default FragmentExample;
```

Fragmentos

Paso 2: Usar fragmentos en una lista

- Crea un archivo en src/components/FragmentList.jsx para mostrar una lista de elementos agrupados:

```jsx
import React from "react";

function FragmentList() {
  // Lista de elementos
  const items = [
    { id: 1, name: "Elemento 1" },
    { id: 2, name: "Elemento 2" },
    { id: 3, name: "Elemento 3" },
  ];

  return (
    <div>
      <h3>Lista con Fragmentos</h3>
      <ul>
        {/* Fragmento */}
        {items.map((item) => (
          <React.Fragment key={item.id}>
            <li>{item.name}</li>
            <p>Descripción del {item.name}</p>
          </React.Fragment>
        ))}
      </ul>
    </div>
  );
}

export default FragmentList;
```

Fragmentos

Paso 3: Crear una vista para probar fragmentos

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

```jsx
import FragmentExample from "../components/FragmentExample";
import FragmentList from "../components/FragmentList";

function FragmentView() {
  return (
    <div>
      <h1>Ejemplo de Fragmentos</h1>
      <FragmentExample />
      <FragmentList />
    </div>
  );
}

export default FragmentView;
```

Fragmentos

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

- Modifica el archivo App.jsx:

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

// Añade al menú de vistas
<button onClick={() => setView("fragment")}>Fragmentos</button>
{view === "fragment" && <FragmentView />}
```

Transitions

Las transiciones en React facilitan la creación de efectos visuales cuando los componentes entran, salen o cambian de estado en el DOM. Esto mejora la experiencia del usuario al hacer que las interacciones sean más atractivas y fluidas. La librería **react-transition-group** es una opción popular para manejar transiciones con React.

Transitions

Paso 1: Instalar React Transition Group

- Desde la carpeta event-manager-app, instala la librería:

```
npm install react-transition-group
```

Transitions

Paso 2: Crear un componente con transiciones

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

```
import { useState } from "react";
import { CSSTransition, TransitionGroup } from "react-transition-group";

function TransitionEventList() {
  // Lista de eventos
  const [events, setEvents] = useState([
    { id: 1, title: "Evento 1" },
    { id: 2, title: "Evento 2" },
  ]);
```

```
// Añadir evento
const addEvent = () => {
  const newEvent = {
    id: Math.random(),
    title: `Evento ${events.length + 1}`,
  };

  // Añadir evento a la lista
  setEvents([...events, newEvent]);
};

const removeEvent = (id) => {
  setEvents(events.filter((event) => event.id !== id));
};
```

Transitions

Paso 2.1: Crear un componente con transiciones

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

```
  return (
    <div>
      <h3>Lista con Transiciones</h3>
      <button onClick={addEvent}>Añadir Evento</button>
      {/* Lista de eventos */}
      <TransitionGroup component="ul" className="event-list">
        {/* Eventos */}
        {events.map((event) => (
          // Transición de entrada y salida
          <CSSTransition key={event.id} timeout={300} classNames="event">
            <li className="event-item" onClick={() => removeEvent(event.id)}>
              {event.title}
            </li>
          </CSSTransition>
        ))}
      </TransitionGroup>
    </div>
  );
}

export default TransitionEventList;
```

Transitions

Paso 3: Crear una vista para probar las transiciones

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

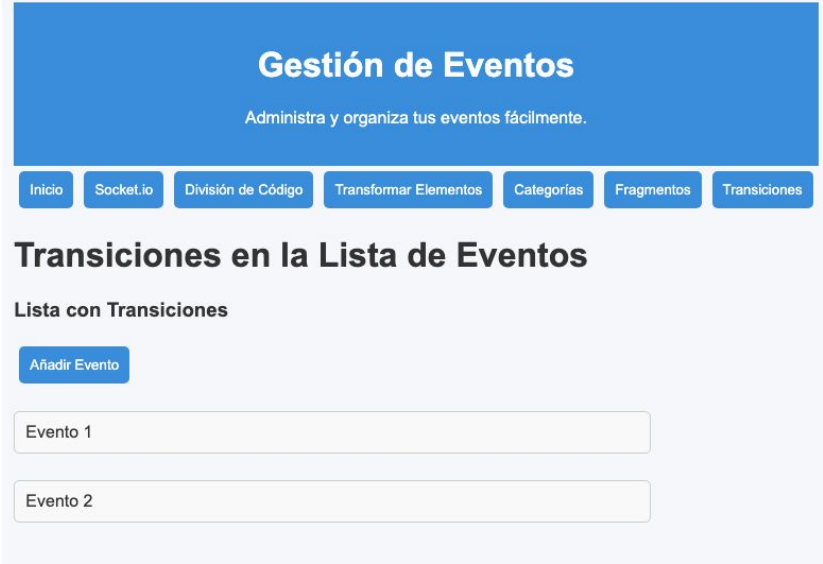
```jsx
import TransitionEventList from "../components/TransitionEventList";

function TransitionView() {
  return (
    <div>
      <h1>Transiciones en la Lista de Eventos</h1>
      <TransitionEventList />
    </div>
  );
}

export default TransitionView;
```

Transitions

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

- Modifica el archivo App.jsx:

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

// Añade al menú de vistas
<button onClick={() => setView("transitions")}>Transiciones</button>
{view === "transitions" && <TransitionView />}
```



Componentes de Orden Superior (HOC)

Un Componente de Orden Superior (Higher-Order Component, HOC) es una **función** que toma un **componente como argumento y retorna un nuevo componente** con funcionalidades adicionales. Los HOCs se usan para **reutilizar la lógica entre múltiples componentes** de manera eficiente.

Por ejemplo, puedes usar un HOC para manejar autenticación, permisos o cualquier lógica que deba ser compartida entre componentes.

Componentes de Orden Superior (HOC)

Paso 1: Crear un HOC para manejar clics

- Crea un archivo en src/hocs/withClickHandler.jsx:

```
function withClickHandler(WrappedComponent) {
  return function ClickHandlerComponent(props) {
    const handleClick = () => {
      console.log(`Componente clickeado: ${props.name}`);
    };

    return (
      <div onClick={handleClick}>
        <WrappedComponent {...props} />
      </div>
    );
  };
}

export default withClickHandler;
```

Componentes de Orden Superior (HOC)

Paso 2: Crear un HOC para mostrar datos adicionales

- Crea un archivo en src/hocs/withExtraInfo.jsx:

```
function withExtraInfo(WrappedComponent) {
  // Esta función retorna un componente funcional que envuelve al componente original
  return function ExtraInfoComponent(props) {
    const extraInfo = "Este componente tiene funcionalidad extendida.";

    // Se retorna el componente original envuelto en un div que muestra información adicional
    return (
      <div>
        {/* Se renderiza el componente original */}
        <WrappedComponent {...props} />
        <p style={{ fontStyle: "italic", color: "#666" }}>{extraInfo}</p>
      </div>
    );
  };
}

export default withExtraInfo;
```

Componentes de Orden Superior (HOC)

Paso 3: Crear un componente base para extender con HOCs

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

```
function BaseComponent({ name }) {
  return <h3>Hola, soy el componente {name}</h3>;
}

export default BaseComponent;
```

Componentes de Orden Superior (HOC)

Paso 4: Crear un componente extendido usando HOCs

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

```jsx
import BaseComponent from "./BaseComponent";
import withClickHandler from "../hocs/withClickHandler";
import withExtraInfo from "../hocs/withExtraInfo";

// Aplicar HOCs
const ClickableComponent = withClickHandler(BaseComponent);
const EnhancedComponent = withExtraInfo(ClickableComponent);

function HOCExample() {
  return (
    <div>
      <h2>Ejemplo de Componentes de Orden Superior</h2>
      <EnhancedComponent name="Extendido" />
    </div>
  );
}

export default HOCExample;
```

Componentes de Orden Superior (HOC)

Paso 5: Crear una vista para probar los HOCs

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

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

function HOCView() {
  return (
    <div>
      <h1>Componentes de Orden Superior (HOCs)</h1>
      <HOCExample />
    </div>
  );
}

export default HOCView;
```

Componentes de Orden Superior (HOC)

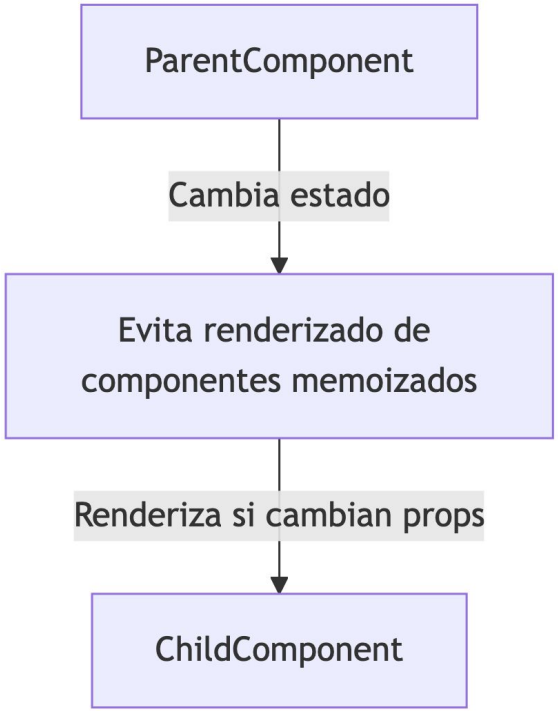
Paso 6: Añadir la vista al selector de vistas en App.jsx

- Modifica el archivo App.jsx:

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

// Añade al menú de vistas
<button onClick={() => setView("hocs")}>HOCs</button>
{view === "hocs" && <HOCView />}
```

Optimizando el rendimiento

Optimizar el rendimiento en React implica implementar técnicas y herramientas que **reduzcan los renderizados** innecesarios y mejoren la eficiencia general de la aplicación. Esto es especialmente crucial en aplicaciones grandes o aquellas con **componentes dinámicos** que manejan datos pesados.



Optimizando el rendimiento

Paso 1: Crear un componente optimizado con React.memo

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

```
import React from "react";

function OptimizedEvent({ event, onClick }) {
  console.log(`Renderizando evento: ${event.title}`);
  return (
    <div onClick={() => onClick(event.id)} className="event-item">
      <h3>{event.title}</h3>
      <p>Fecha: {event.date}</p>
    </div>
  );
}

export default React.memo(OptimizedEvent);
```

Optimizando el rendimiento

Paso 2: Crear una lista de eventos optimizada

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

```
import { useState, useCallback } from "react";
import OptimizedEvent from "./OptimizedEvent";

function OptimizedEventList() {
  const [events, setEvents] = useState([
    { id: 1, title: "Evento 1", date: "2024-12-20" },
    { id: 2, title: "Evento 2", date: "2024-12-22" },
  ]);
  const [selectedEvent, setSelectedEvent] = useState(null);

  const handleClick = useCallback((id) => {
    setSelectedEvent(id);
  }, []);

  const addEvent = () => {
    const newEvent = {
      id: Math.random(),
      title: `Evento ${events.length + 1}`,
      date: new Date().toLocaleDateString(),
    };
    setEvents([...events, newEvent]);
  };
```

```
  return (
    <div>
      <h3>Lista de Eventos Optimizada</h3>
      <button onClick={addEvent}>Añadir Evento</button>
      <div className="selected-event">
        {selectedEvent && <p>Evento seleccionado:
{selectedEvent}</p>}
      </div>
      <div className="event-list">
        {events.map((event) => (
          <OptimizedEvent key={event.id} event={event}
onClick={handleClick} />
        ))}
      </div>
    </div>
  );
}

export default OptimizedEventList;
```

Optimizando el rendimiento

Paso 3: Crear una vista para probar la optimización

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

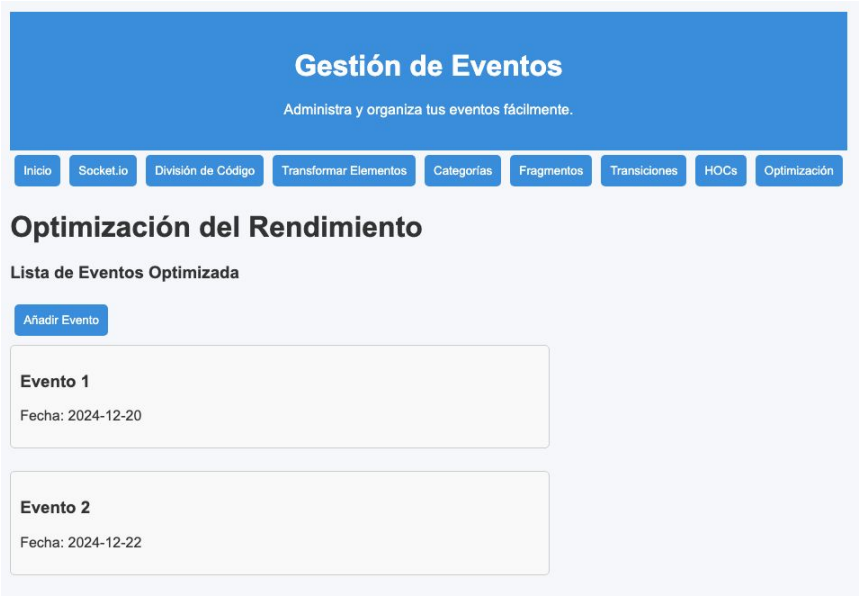
```jsx
import React, { Profiler } from "react";
import OptimizedEventList from "../components/OptimizedEventList";

function OptimizedView() {
  const onRenderCallback = (
    id, // Nombre del Profiler (OptimizedEventList)
    phase, // "mount" o "update"
    actualDuration // Tiempo que tardó en renderizar
  ) => {
    console.log(`${id} (${phase}) tomó ${actualDuration}ms para renderizar.`);
  };

  return (
    <div>
      <h1>Optimización del Rendimiento</h1>
      <Profiler id="OptimizedEventList" onRender={onRenderCallback}>
        <OptimizedEventList />
      </Profiler>
    </div>
  );
}

export default OptimizedView;
```

Optimizando el rendimiento

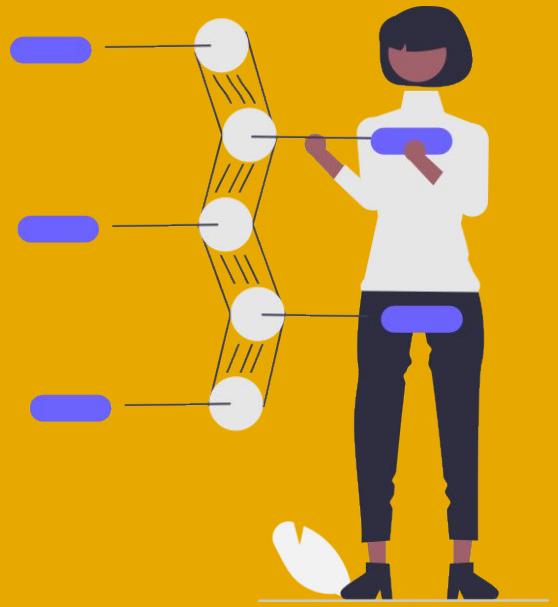
Paso 6: Añadir la vista al selector de vistas en App.jsx

- Modifica el archivo App.jsx:

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

// Añade al menú de vistas
<button onClick={() => setView("optimized")}>Optimización</button>
{view === "optimized" && <OptimizedView />}
```



Resumen de lo aprendido 🧑‍🎓



Adalid aspasia

Resumen de lo aprendido

- **Optimización:** Mejora del rendimiento con fragmentos, componentes de orden superior y división de código.
- **socket.io:** Implementación para comunicación en tiempo real.
- **Transformaciones y transiciones:** Animaciones y cambios de estado visual con React.
- **Contexto:** Manejo de estados globales sin pasar props manualmente.

GRACIAS POR TU ATENCIÓN

Nos vemos en la próxima clase

Chile
Adalid
aspasia
LA FORMACIÓN DE TU FUTURO