**ok, este es mi temario que me han dado de react y astro, he añadido el view transactions porque está muy bien aunque nunca lo use. también me comentaron el eventsource pero no me explicaron nada, así que si lo evito mejor, en su lugar el store zustand puede ayudar. quiero hacer una web de reserva de hoteles, usando la api y base de datos que cree en el otro proyecto como back. nunca lo he hehco y necesitaré que me ayudes mucho.**

**Store Zustand**

A diferencia de otros enfoques más complejos, como Redux, Zustand se destaca por su sencillez, flexibilidad y una curva de aprendizaje suave. Esta clase cubrirá los fundamentos de Zustand, cómo configurarlo y cómo integrarlo en tus componentes, además de algunos ejemplos prácticos.

**¿Qué es Zustand?**

**Zustand** es una librería de estado global para React que proporciona una API simple basada en funciones y Hooks, sin la complejidad de acciones, reducers o middlewares que se encuentran en soluciones más pesadas. Con Zustand, puedes:

* Crear un **store** de estado global usando una función.
* Acceder a ese estado desde cualquier componente mediante el Hook useStore.
* Actualizar el estado llamando a funciones directamente, sin acciones ni dispatchers.

**¿Por qué usar Zustand?**

* **Simplicidad**: Su API es más directa que la de Redux u otras librerías. No necesitas crear reducers ni tipos de acciones.
* **Rendimiento**: Zustand suscribe de forma fina a las partes del estado que cada componente necesita, minimizando re-renderizados innecesarios.
* **Flexibilidad**: Puedes definir el estado en una sola función, mezclar Hooks de React y lógica dentro de tu store, y estructurarlo a tu gusto.
* **Integración fácil**: No requiere una configuración compleja. Se instala y se usa rápidamente.

**Instalación**

Para agregar Zustand a tu proyecto, ejecuta en la terminal:

npm install zustand

o con Yarn:

yarn add zustand

**Creación de un Store**

En Zustand, creas el store definiendo una función que retorna un objeto con el estado y las funciones para actualizarlo. Luego, pasas esta función a create (importada de zustand) y obtienes un Hook que puedes utilizar en tus componentes.

**Ejemplo básico:**

import create from 'zustand';

const useStore = create((set) => ({

contador: 0,

incrementar: () => set((state) => ({ contador: state.contador + 1 })),

decrementar: () => set((state) => ({ contador: state.contador - 1 })),

}));

export default useStore;

**Explicación:**

* create((set) => ({})): set es una función para actualizar el estado.
* Definimos contador con valor inicial 0.
* Definimos incrementar y decrementar, que usan set para actualizar el estado.
* Esta función create devuelve el Hook useStore, que utilizaremos en los componentes para leer y actualizar el estado global.

**Uso del Store en los Componentes**

Para leer o actualizar el estado en tus componentes, importas el Hook useStore y seleccionas las partes del estado que necesitas.

**Ejemplo en un componente:**

import useStore from './store';

function Contador() {

const { contador, incrementar, decrementar } = useStore((state) => ({

contador: state.contador,

incrementar: state.incrementar,

decrementar: state.decrementar,

}));

return (

<div>

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

<button onClick={incrementar}>Incrementar</button>

<button onClick={decrementar}>Decrementar</button>

</div>

);

}

export default Contador;

**Explicación:**

* useStore((state) => ({})): Pasamos una función que recibe el estado completo y devuelve un objeto con las propiedades que queremos extraer.
* Obtenemos contador, incrementar y decrementar del estado global.
* Al presionar los botones, llamamos a las funciones para actualizar el estado global, y el componente se re-renderiza con el nuevo valor.

**Selección de Estado y Minimización de Renderizados**

Zustand recomienda seleccionar solo las partes del estado que necesitas, evitando renderizados innecesarios.

Ejemplo: Si tu estado es más grande y el componente solo necesita contador, haz:

import useStore from './store';

function SoloContador() {

const contador = useStore((state) => state.contador);

return <p>Valor: {contador}</p>;

}

De esta forma, el componente se actualizará solo cuando contador cambie.

**Ejemplo con Múltiples Componentes y Estado Compartido**

Imagina un sistema de carrito de compras:

import create from 'zustand';

const useCarritoStore = create((set) => ({

items: [],

agregarItem: (item) => set((state) => ({ items: [...state.items, item] })),

removerItem: (id) => set((state) => ({ items: state.items.filter((item) => item.id !== id) })),

}));

export default useCarritoStore;

Un componente ListaProductos que agrega productos al carrito:

import useCarritoStore from './carritoStore';

function ListaProductos() {

const agregarItem = useCarritoStore((state) => state.agregarItem);

const productos = [

{ id: 1, nombre: 'Camiseta', precio: 20 },

{ id: 2, nombre: 'Pantalón', precio: 40 },

];

return (

<div>

<h2>Productos</h2>

{productos.map((prod) => (

<div key={prod.id}>

<span>{prod.nombre} - ${prod.precio}</span>

<button onClick={() => agregarItem(prod)}>Agregar</button>

</div>

))}

</div>

);

}

Y un componente Carrito para mostrar y manipular el estado global:

import useCarritoStore from './carritoStore';

function Carrito() {

const items = useCarritoStore((state) => state.items);

const removerItem = useCarritoStore((state) => state.removerItem);

return (

<div>

<h2>Carrito</h2>

<ul>

{items.map((item) => (

<li key={item.id}>

{item.nombre} - ${item.precio}

<button onClick={() => removerItem(item.id)}>Eliminar</button>

</li>

))}

</ul>

</div>

);

}

Aquí ListaProductos y Carrito comparten el mismo estado global. ListaProductos modifica el estado agregando ítems, Carrito muestra el estado actual y permite eliminarlos. No es necesario pasar props entre ellos.

**Middleware y Persistencia del Estado**

Zustand soporta middleware para ampliar sus funcionalidades. Por ejemplo, zustand/middleware incluye persistencia en localStorage:

import create from 'zustand';

import { persist } from 'zustand/middleware';

const useStore = create(persist(

(set) => ({

contador: 0,

incrementar: () => set((state) => ({ contador: state.contador + 1 })),

}),

{ name: 'mi-app' } // Clave en localStorage

));

El contador ahora se guardará en localStorage y al recargar la página se mantendrá su valor.

**Comparación con Otras Soluciones**

* **Context + useReducer**: Zustand simplifica la lógica, evitando que pases Contextos por todos lados. Menos re-renderizados.
* **Redux**: Zustand es más ligero y requiere menos boilerplate. Redux, aunque más complejo, provee una estructura más estricta para apps grandes.
* **MobX, Recoil, Jotai**: Todas son buenas opciones. Zustand destaca por su sencillez y mínima infraestructura.

**Buenas Prácticas con Zustand**

1. **Mantener el store claro y pequeño**: Define el estado según las necesidades. Evita crear un store gigante.
2. **Separar por funcionalidad**: Para proyectos grandes, múltiples stores especializados (auth, carrito, config) pueden ser útiles.
3. **Inmutabilidad**: Aunque Zustand no lo exige estrictamente, mantener el estado inmutable es buena práctica.
4. **Uso de selectores**: Seleccionar solo las partes del estado necesarias evita renderizados excesivos.

**Ejercicio Propuesto**

Crea un store para manejar una lista de tareas (todos):

* Estado inicial: todos = [].
* Funciones:
  + agregarTarea(tarea) para agregar una nueva tarea.
  + toggleTarea(id) para alternar completada/no completada.

Crea dos componentes:

1. ListaTareas: Muestra las tareas y un botón para alternar su estado.
2. AgregarTarea: Contiene un input y un botón para agregar una tarea nueva.

Observa cómo ambos componentes pueden acceder y actualizar el mismo estado global sin pasar props entre sí.

**Conclusión**

En esta clase aprendimos:

* Qué es Zustand y por qué puede ser más sencillo que otras herramientas.
* Cómo crear un store y usarlo en componentes.
* Cómo actualizar y leer el estado global de forma sencilla.
* Ejemplos prácticos que muestran su uso en escenarios reales.

Zustand es una gran opción cuando quieres un estado global simple, sin la complejidad ni el boilerplate de Redux u otras soluciones más complejas. Así, tu código será más conciso, claro y fácil de mantener.

¡Experimenta con Zustand en tus proyectos y descubre cómo simplifica la gestión de estado global en React!

**Router**

Ahora nos adentraremos en el enrutamiento en aplicaciones React, utilizando **React Router**, una de las librerías más populares y flexibles para gestionar navegación y rutas en aplicaciones de una sola página (SPA). Veremos cómo instalar y configurar React Router, cómo definir rutas y componentes de ruta, cómo crear enlaces de navegación entre páginas y cómo trabajar con parámetros dinámicos y rutas anidadas.

**¿Por Qué React Router?**

En una aplicación React normal, tenemos un archivo index.html estático que sirve como punto de entrada. React utiliza el Virtual DOM para actualizar la vista, pero si necesitamos múltiples "pantallas" o "vistas" sin recargar la página, necesitamos un sistema de rutas. **React Router** nos permite:

* Definir diferentes "páginas" o vistas asociadas a determinadas URL.
* Navegar entre esas páginas sin recargar el navegador.
* Mantener una buena organización del código, separando lógicamente cada vista.

**Instalación y Configuración de React Router**

React Router se distribuye en múltiples paquetes, el más común para aplicaciones web es react-router-dom:

**Instalación:**

npm install react-router-dom

o con Yarn:

yarn add react-router-dom

Una vez instalado, podemos importar sus componentes y funciones en nuestros archivos. En React Router v6, la configuración es más directa.

**Configuración Básica:**

1. En el punto de entrada de tu aplicación, normalmente en App.jsx o en un componente contenedor, envuelve tu aplicación con <BrowserRouter>:
2. import { BrowserRouter } from 'react-router-dom';
3. function App() {
4. return (
5. <BrowserRouter>
6. {/\* Aquí van las rutas y componentes \*/}
7. </BrowserRouter>
8. );
9. }

export default App;

1. Dentro de <BrowserRouter>, utilizaremos <Routes> y <Route> para definir las distintas rutas.

**Definición de Rutas y Componentes de Ruta**

React Router trabaja con el concepto de "routing declarativo". Definimos las rutas y qué componente se renderizará para cada una.

**Ejemplo básico de rutas:**

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

import Home from './pages/Home';

import About from './pages/About';

function App() {

return (

<BrowserRouter>

<Routes>

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

<Route path="/about" element={<About />} />

</Routes>

</BrowserRouter>

);

}

**Explicación:**

* <Routes>: Contenedor para todas las rutas de la aplicación.
* <Route path="/" element={<Home />} />: Define que la URL raíz (/) mostrará el componente Home.
* <Route path="/about" element={<About />} />: Define que la URL /about mostrará el componente About.

**Comportamiento:**

* Cuando el usuario accede a http://localhost:3000/, verá <Home />.
* Cuando accede a http://localhost:3000/about, verá <About />.

**Navegación entre Páginas con Link y NavLink**

Para navegar entre rutas sin recargar la página, utilizamos el componente <Link> proporcionado por React Router.

**Ejemplo con <Link>:**

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

function Navbar() {

return (

<nav>

<Link to="/">Inicio</Link>

<Link to="/about">Acerca</Link>

</nav>

);

}

* to especifica la ruta a la que queremos navegar.
* Al hacer clic en los enlaces, la aplicación cambia de vista sin recargar el navegador.

**<NavLink> para enlaces activos:**

<NavLink> es similar a <Link>, pero permite estilos o clases especiales cuando la ruta está activa.

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

function Navbar() {

return (

<nav>

<NavLink

to="/"

style={({ isActive }) => ({ fontWeight: isActive ? 'bold' : 'normal' })}

>

Inicio

</NavLink>

<NavLink

to="/about"

className={({ isActive }) => isActive ? 'link-activo' : ''}

>

Acerca

</NavLink>

</nav>

);

}

* NavLink recibe una función para style o className que entrega un objeto { isActive: boolean } para determinar si la ruta actual coincide con el to.
* Esto facilita indicar visualmente al usuario la página en la que se encuentra.

**Parámetros en las Rutas**

A menudo necesitamos rutas dinámicas, por ejemplo, /users/123 para mostrar el perfil del usuario con ID 123.

**Definición de ruta con parámetro:**

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

import UserProfile from './pages/UserProfile';

function App() {

return (

<BrowserRouter>

<Routes>

<Route path="/users/:id" element={<UserProfile />} />

</Routes>

</BrowserRouter>

);

}

* :id define un parámetro dinámico llamado id.

**Acceder al parámetro en el componente:**

En UserProfile, podemos obtener el valor del parámetro usando useParams():

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

function UserProfile() {

const { id } = useParams();

// 'id' contendrá el valor pasado en la URL, p. ej. '123'

return <div>Perfil de usuario con ID: {id}</div>;

}

export default UserProfile;

* Si navegamos a http://localhost:3000/users/123, id será '123'.

**Uso múltiple de parámetros:**

<Route path="/products/:category/:id" element={<ProductDetail />} />

En ProductDetail, podríamos hacer:

const { category, id } = useParams();

// category e id contendrán los valores de la URL.

**Rutas Anidadas**

React Router permite crear rutas "hijas" o anidadas, útiles para secciones de la app con subpáginas. Por ejemplo, una sección de administración con múltiples subrutas.

**Definición de rutas anidadas:**

import AdminHome from './pages/AdminHome';

import AdminUsers from './pages/AdminUsers';

import AdminSettings from './pages/AdminSettings';

function App() {

return (

<BrowserRouter>

<Routes>

<Route path="/admin" element={<LayoutAdmin />}>

<Route index element={<AdminHome />} />

<Route path="users" element={<AdminUsers />} />

<Route path="settings" element={<AdminSettings />} />

</Route>

</Routes>

</BrowserRouter>

);

}

**Explicación:**

* <Route path="/admin" element={<LayoutAdmin />}>: Define una ruta padre /admin.
* Rutas hijas:
  + <Route index element={<AdminHome />} />: Sin especificar path, esta ruta será /admin y mostrará AdminHome. La ruta index indica la ruta predeterminada del padre.
  + <Route path="users" element={<AdminUsers />} />: Equivale a /admin/users.
  + <Route path="settings" element={<AdminSettings />} />: Equivale a /admin/settings.

**Componente Padre (LayoutAdmin):**

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

function LayoutAdmin() {

return (

<div>

<h1>Panel de Administración</h1>

<nav>

<Link to="/admin">Inicio</Link>

<Link to="/admin/users">Usuarios</Link>

<Link to="/admin/settings">Configuración</Link>

</nav>

<hr />

{/\* Outlet es el "placeholder" donde se renderizan las rutas hijas \*/}

<Outlet />

</div>

);

}

export default LayoutAdmin;

* <Outlet> renderiza el componente de la ruta hija que coincida con la URL actual.
* Si la URL es /admin, se mostrará AdminHome.
* Si la URL es /admin/users, se mostrará AdminUsers en el <Outlet>.

**Navegación Programática**

Además de <Link> y <NavLink>, puedes navegar programáticamente usando el Hook useNavigate().

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

function MiComponente() {

const navigate = useNavigate();

const irAAbout = () => {

navigate('/about');

};

return <button onClick={irAAbout}>Ir a About</button>;

}

* useNavigate devuelve una función que permite cambiar la ruta imperativamente.
* Esto es útil, por ejemplo, después de guardar datos en un formulario, se puede redirigir al usuario a otra página.

**Redirecciones y Rutas Inexistentes**

**Redirecciones:**  
Para redirigir se puede usar navigate dentro de un efecto (useEffect) o como respuesta a una acción del usuario.

**Rutas inexistentes (404):**  
Es común definir una ruta "catch-all" para manejar URLs desconocidas:

<Routes>

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

<Route path="/about" element={<About />} />

<Route path="\*" element={<NotFound />} />

</Routes>

* path="\*" atrapa cualquier ruta no definida, mostrando NotFound.

**Buenas Prácticas con React Router**

1. **Organizar las rutas en un solo archivo**: Muchos proyectos separan la definición de rutas en un archivo dedicado para facilitar el mantenimiento.
2. **Cargas Dinámicas (Code Splitting)**: Puedes usar React.lazy y Suspense para cargar componentes de rutas bajo demanda, mejorando el performance.
3. **Protección de Rutas**: Para restringir acceso a ciertas páginas (por ejemplo, panel de administración solo para usuarios logueados), puedes crear rutas protegidas que verifiquen la autenticación antes de renderizar el componente.
4. **Usar NavLink para mejorar la navegación**: Mostrar enlaces activos ayuda a la usabilidad.

**Ejercicio Propuesto**

1. Crea una aplicación con las siguientes rutas:
   * /: Muestra un componente Home.
   * /about: Muestra un componente About.
   * /users/:id: Muestra un componente UserDetail que cargue datos de un usuario basado en el ID de la URL y los muestre.
2. Añade un menú de navegación con <NavLink> para ir a Home y About.
3. Añade un botón en Home que use useNavigate para redirigir a /about.
4. Implementa una ruta por defecto con path="\*" que muestre un componente NotFound.

Este ejercicio te ayudará a poner en práctica la definición de rutas, parámetros, enlaces y navegación programática.

**Conclusión**

* Cómo instalar y configurar React Router.
* Cómo definir rutas y renderizar componentes según la URL.
* Cómo crear enlaces de navegación con <Link> y <NavLink>.
* Cómo manejar parámetros dinámicos en las rutas con useParams().
* Cómo crear rutas anidadas y usar <Outlet> para componentes de layout.
* Cómo navegar programáticamente con useNavigate().

React Router proporciona una infraestructura sólida para construir aplicaciones React complejas con navegación intuitiva y sin recargas de página, aportando una excelente experiencia de usuario.

**Manejo de Eventos**

Nos centraremos en cómo manejar eventos en React, entendiendo la diferencia entre los **eventos sintéticos** proporcionados por React y los eventos nativos del navegador, así como las formas de prevenir comportamientos por defecto y detener la propagación de eventos.

**Eventos Sintéticos en React**

React utiliza un sistema interno para manejar eventos conocido como **"eventos sintéticos"**. Estos eventos sintéticos son objetos que se construyen en base a los eventos nativos del navegador, pero con compatibilidad asegurada entre diferentes navegadores y una API consistente.

* Los eventos sintéticos funcionan de manera similar a los eventos nativos del DOM.
* Sus nombres siguen la convención **camelCase**: onClick, onChange, onSubmit, onMouseEnter, etc.
* Los eventos se asocian a componentes de React como atributos.

**Ejemplo básico:**

function Boton() {

const manejarClick = () => {

console.log('Botón clickeado');

};

return <button onClick={manejarClick}>Clic aquí</button>;

}

* El evento onClick recibe una función, no el resultado de una función. Por eso se pasa manejarClick sin paréntesis.
* Cuando el botón se hace clic, se ejecuta la función manejarClick.

**Enlace y Manejo de Eventos en Componentes**

En React, el manejo de eventos se hace con funciones. Puedes definir las funciones para manejar eventos dentro del mismo componente o importarlas de otro lugar.

**1. Manejadores inline vs externos:**

// Manejador inline

<button onClick={() => console.log('Clic inline!')}>Clic Inline</button>

// Manejador en función

function manejarClick() {

console.log('Clic en función!');

}

<button onClick={manejarClick}>Clic Función</button>

**2. Pasando parámetros a eventos:**

function Lista({ items }) {

const manejarClick = (item) => {

console.log('Item clickeado:', item);

};

return (

<ul>

{items.map((item, index) => (

<li key={index}>

{/\* Usar una función flecha para pasar parámetros \*/}

<button onClick={() => manejarClick(item)}>Seleccionar {item}</button>

</li>

))}

</ul>

);

}

* Al usar una función flecha dentro del evento, puedes pasar argumentos al manejador.
* Evita definir funciones inline muy complejas, ya que se crean en cada render. Sin embargo, para casos simples, está bien.

**Prevención del Comportamiento por Defecto y Propagación de Eventos**

Algunos elementos HTML tienen comportamientos por defecto que puede que desees prevenir. Por ejemplo, los enlaces recargan la página y los formularios envían datos al servidor.

**1. Prevenir comportamiento por defecto (preventDefault)**

function Formulario() {

const manejarSubmit = (e) => {

e.preventDefault(); // Evita que el navegador recargue la página

console.log('Formulario enviado lógicamente');

};

return (

<form onSubmit={manejarSubmit}>

<input type="text" placeholder="Nombre" />

<button type="submit">Enviar</button>

</form>

);

}

* El objeto del evento sintético se pasa automáticamente a la función manejadora, por lo que e.preventDefault() funciona como en los eventos nativos.
* e es un **SyntheticEvent** de React, con una interfaz similar al evento nativo.

**2. Detener la propagación del evento (stopPropagation)**

A veces, querrás evitar que un evento que ocurre en un elemento hijo se propague a su elemento padre.

function CajaPadre() {

const manejarClickPadre = () => {

console.log('Clic en el padre');

};

return (

<div onClick={manejarClickPadre} style={{ border: '1px solid blue', padding: '20px' }}>

Padre

<CajaHija />

</div>

);

}

function CajaHija() {

const manejarClickHija = (e) => {

e.stopPropagation(); // Evita que el evento llegue al padre

console.log('Clic en la hija');

};

return (

<div onClick={manejarClickHija} style={{ border: '1px solid red', margin: '10px' }}>

Hija

</div>

);

}

* Al hacer clic en la caja hija, sin stopPropagation, el evento se propagaría hasta la caja padre, ejecutando ambos manejadores.
* Con stopPropagation(), el clic se detiene en la hija, no llegando al padre.

**Ejemplos Prácticos**

**1. Lista de tareas con eliminación**

function ListaTareas({ tareas, eliminarTarea }) {

return (

<ul>

{tareas.map((tarea, index) => (

<li key={index}>

{tarea}

<button onClick={() => eliminarTarea(index)}>Eliminar</button>

</li>

))}

</ul>

);

}

* Cada botón "Eliminar" llama a una función que recibe el índice de la tarea a eliminar.

**2. Navegación personalizada**

function LinkPersonalizado({ href, children }) {

const manejarClick = (e) => {

e.preventDefault();

console.log(`Navegando a ${href} sin refrescar la página`);

// Aquí podrías utilizar, por ejemplo, react-router para cambiar de ruta

};

return <a href={href} onClick={manejarClick}>{children}</a>;

}

* Se evita la navegación tradicional y se controla manualmente la acción en el manejador de eventos.

**Buenas Prácticas**

1. **Nombrar las funciones manejadoras descriptivamente**:  
   Por ejemplo, manejarSubmit, manejarClick, manejarCambioInput.  
   Esto mejora la legibilidad del código.
2. **Evitar lógica compleja inline**:  
   Es mejor definir las funciones fuera del JSX, mejorando la claridad.
3. **Evitar el uso excesivo de stopPropagation()**:  
   Intenta diseñar tus componentes para no necesitar interrumpir la propagación a menudo.  
   Usa este método solo cuando realmente se requiera.
4. **Manejo de eventos compatible con accesibilidad**:  
   Considera la accesibilidad (por ejemplo, usando onKeyDown además de onClick para permitir interacción con el teclado).

**Ejercicio Propuesto**

Crea un componente MenuDesplegable que:

* Renderice un botón "Abrir Menú".
* Al hacer clic en el botón, muestre una lista de opciones.
* Al hacer clic en una opción, se ejecute una función que reciba la opción elegida.
* Si se hace clic fuera del menú (fuera del botón y de la lista), el menú se cierra.

**Pistas:**

* Usa el estado para mostrar/ocultar el menú.
* Usa eventos como onClick en el document para detectar clics fuera del menú.
* Usa e.stopPropagation() para evitar que el clic en el menú se propague al documento.

import { useState, useEffect, useRef } from 'react';

function MenuDesplegable({ opciones, onOpcionSeleccionada }) {

const [abierto, setAbierto] = useState(false);

const menuRef = useRef(null);

useEffect(() => {

const manejarClickFuera = (e) => {

if (menuRef.current && !menuRef.current.contains(e.target)) {

setAbierto(false);

}

};

document.addEventListener('click', manejarClickFuera);

return () => {

document.removeEventListener('click', manejarClickFuera);

};

}, []);

const manejarClickBoton = (e) => {

e.stopPropagation();

setAbierto(!abierto);

};

const manejarOpcionClick = (opcion) => {

onOpcionSeleccionada(opcion);

setAbierto(false);

};

return (

<div ref={menuRef}>

<button onClick={manejarClickBoton}>Abrir Menú</button>

{abierto && (

<ul onClick={(e) => e.stopPropagation()}>

{opciones.map((opcion, index) => (

<li key={index} onClick={() => manejarOpcionClick(opcion)}>

{opcion}

</li>

))}

</ul>

)}

</div>

);

}

export default MenuDesplegable;

**useState**

**Clase 5: Estado en Componentes**

En esta clase, exploraremos el concepto de **estado (state)** en React, centrándonos en cómo manejarlo en **componentes funcionales** utilizando el Hook useState. Aprenderemos qué es el estado, cómo actualizarlo y cómo utilizarlo para crear componentes interactivos y dinámicos.

**Introducción al estado (state)**

**Estado** es un objeto que representa información interna y dinámica de un componente que puede cambiar a lo largo del tiempo. A diferencia de las **props**, que son inmutables y se utilizan para pasar datos de un componente padre a un hijo, el **estado** es mutable y se utiliza para manejar datos que pueden cambiar en respuesta a acciones del usuario, eventos del sistema o cualquier otra interacción.

**Características del estado:**

* **Interno al componente**: Cada componente puede tener su propio estado.
* **Mutable**: Puede cambiar usando funciones específicas.
* **Reactividad**: Cuando el estado cambia, React vuelve a renderizar el componente para reflejar el nuevo estado.

**Manejo del estado en componentes funcionales**

Con la introducción de los **Hooks** en React 16.8, es posible manejar el estado en componentes funcionales utilizando el Hook useState.

**Uso básico de useState**

El Hook useState permite agregar estado a componentes funcionales. Se importa desde React y se utiliza de la siguiente manera:

import { useState } from 'react';

function MiComponente() {

const [estado, setEstado] = useState(valorInicial);

// ...

return (

// ...

);

}

* **estado**: Es la variable que contiene el valor actual del estado.
* **setEstado**: Es una función que actualiza el valor del estado.
* **useState(valorInicial)**: Inicializa el estado con valorInicial.

**Ejemplo sencillo: Contador**

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)}>

Incrementar

</button>

</div>

);

}

export default Contador;

**Explicación:**

* Iniciamos el estado contador en 0.
* Al hacer clic en el botón, llamamos a setContador para actualizar el estado incrementándolo en 1.
* Al cambiar el estado, el componente se vuelve a renderizar y muestra el nuevo valor de contador.

**Actualización y mutación del estado**

**Reglas importantes al actualizar el estado:**

1. **No mutar directamente el estado**: Siempre usar la función de actualización proporcionada por useState.
2. **Las actualizaciones de estado pueden ser asincrónicas**: No dependa del valor actual del estado inmediatamente después de llamarlo.
3. **Las actualizaciones de estado son reemplazos, no fusiones**: Al actualizar objetos o arrays, asegúrese de copiar y actualizar correctamente.

**Ejemplo: No mutar directamente el estado**

**Incorrecto:**

function Lista() {

const [items, setItems] = useState([1, 2, 3]);

const agregarItem = () => {

items.push(4); // No hacer esto

setItems(items);

};

// ...

}

**Correcto:**

function Lista() {

const [items, setItems] = useState([1, 2, 3]);

const agregarItem = () => {

setItems([...items, 4]);

};

// ...

}

* Usamos el operador **spread (...)** para crear una nueva copia del array con el nuevo elemento.

**Actualización basada en el estado anterior**

Cuando la actualización del estado depende del valor anterior, es recomendable utilizar la función de actualización que recibe el estado previo.

**Ejemplo:**

function Contador() {

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

const incrementar = () => {

setContador(prevContador => prevContador + 1);

};

return (

<div>

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

<button onClick={incrementar}>Incrementar</button>

</div>

);

}

* setContador(prevContador => prevContador + 1);: Recibe el valor previo de contador y devuelve el nuevo valor.

**Ejemplos prácticos con estado local**

**Ejemplo 1: Formulario controlado**

Crear un formulario donde el valor del input se maneja con el estado.

import { useState } from 'react';

function Formulario() {

const [nombre, setNombre] = useState('');

const manejarCambio = (e) => {

setNombre(e.target.value);

};

const manejarSubmit = (e) => {

e.preventDefault();

alert(`Hola, ${nombre}`);

};

return (

<form onSubmit={manejarSubmit}>

<label>

Nombre:

<input type="text" value={nombre} onChange={manejarCambio} />

</label>

<button type="submit">Enviar</button>

</form>

);

}

export default Formulario;

**Explicación:**

* El estado nombre almacena el valor del input.
* Al cambiar el valor del input, se actualiza el estado con setNombre.
* Al enviar el formulario, se muestra un mensaje de saludo con el nombre ingresado.

**Ejemplo 2: Mostrar/Ocultar elemento**

Crear un botón que muestre u oculte un elemento en la interfaz.

import { useState } from 'react';

function MostrarOcultar() {

const [visible, setVisible] = useState(true);

const alternarVisibilidad = () => {

setVisible(!visible);

};

return (

<div>

<button onClick={alternarVisibilidad}>

{visible ? 'Ocultar' : 'Mostrar'} Mensaje

</button>

{visible && <p>Este es un mensaje que puede ser ocultado.</p>}

</div>

);

}

export default MostrarOcultar;

**Explicación:**

* El estado visible determina si el mensaje se muestra o no.
* Al hacer clic en el botón, se alterna el valor de visible.
* Se utiliza renderizado condicional para mostrar u ocultar el mensaje.

**Ejemplo 3: Lista dinámica de tareas**

Crear una aplicación simple de lista de tareas donde puedas agregar y eliminar tareas.

import { useState } from 'react';

function ListaTareas() {

const [tareas, setTareas] = useState([]);

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

const agregarTarea = (e) => {

e.preventDefault();

if (texto.trim() !== '') {

setTareas([...tareas, texto]);

setTexto('');

}

};

const eliminarTarea = (index) => {

setTareas(tareas.filter((\_, i) => i !== index));

};

return (

<div>

<h2>Lista de Tareas</h2>

<form onSubmit={agregarTarea}>

<input

type="text"

value={texto}

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

placeholder="Nueva tarea"

/>

<button type="submit">Agregar</button>

</form>

<ul>

{tareas.map((tarea, index) => (

<li key={index}>

{tarea}{' '}

<button onClick={() => eliminarTarea(index)}>Eliminar</button>

</li>

))}

</ul>

</div>

);

}

export default ListaTareas;

**Explicación:**

* **Estados utilizados:**
  + tareas: Array que almacena las tareas.
  + texto: Almacena el texto ingresado en el input.
* **Funciones:**
  + agregarTarea: Añade una nueva tarea al array tareas.
  + eliminarTarea: Elimina una tarea específica del array.
* **Renderizado:**
  + Se muestra una lista (ul) con las tareas actuales.
  + Cada tarea tiene un botón para eliminarla.

**Manejo de estado con objetos y arrays**

Cuando el estado es un objeto o array, es importante actualizarlo correctamente para mantener la inmutabilidad.

**Actualizando objetos en el estado**

**Ejemplo:**

function PerfilUsuario() {

const [usuario, setUsuario] = useState({

nombre: 'Ana',

edad: 28,

ciudad: 'Madrid',

});

const actualizarCiudad = () => {

setUsuario({ ...usuario, ciudad: 'Barcelona' });

};

return (

<div>

<p>Nombre: {usuario.nombre}</p>

<p>Edad: {usuario.edad}</p>

<p>Ciudad: {usuario.ciudad}</p>

<button onClick={actualizarCiudad}>Cambiar Ciudad</button>

</div>

);

}

**Explicación:**

* Usamos el operador **spread** para copiar el objeto usuario y actualizar solo la propiedad ciudad.
* Esto evita mutar directamente el estado y asegura que React detecte el cambio.

**Actualizando arrays en el estado**

**Ejemplo:**

function ListaFrutas() {

const [frutas, setFrutas] = useState(['Manzana', 'Banana']);

const agregarFruta = () => {

setFrutas([...frutas, 'Naranja']);

};

return (

<div>

<ul>

{frutas.map((fruta, index) => (

<li key={index}>{fruta}</li>

))}

</ul>

<button onClick={agregarFruta}>Agregar Naranja</button>

</div>

);

}

**Explicación:**

* Al agregar una nueva fruta, creamos un nuevo array con el operador **spread**.
* Esto mantiene la inmutabilidad del estado.

**Buenas prácticas al manejar el estado**

1. **No mutar directamente el estado**: Siempre crear nuevas copias de objetos o arrays al actualizar el estado.
2. **Usar funciones de actualización basadas en el estado anterior cuando sea necesario**: Especialmente en actualizaciones concurrentes.
3. **Mantener el estado lo más simple posible**: Evita tener estados innecesariamente complejos.
4. **Evitar efectos secundarios en las funciones de actualización del estado**: Las funciones de actualización deben ser puras.

**Ejercicio propuesto**

Crea un componente Calculadora que permita sumar, restar, multiplicar y dividir dos números ingresados por el usuario.

**Paso 1: Crear el componente**

import { useState } from 'react';

function Calculadora() {

const [numero1, setNumero1] = useState('');

const [numero2, setNumero2] = useState('');

const [resultado, setResultado] = useState(null);

const manejarCambio1 = (e) => setNumero1(e.target.value);

const manejarCambio2 = (e) => setNumero2(e.target.value);

const sumar = () => setResultado(Number(numero1) + Number(numero2));

const restar = () => setResultado(Number(numero1) - Number(numero2));

const multiplicar = () => setResultado(Number(numero1) \* Number(numero2));

const dividir = () => setResultado(Number(numero1) / Number(numero2));

return (

<div>

<h2>Calculadora Simple</h2>

<input type="number" value={numero1} onChange={manejarCambio1} />

<input type="number" value={numero2} onChange={manejarCambio2} />

<div>

<button onClick={sumar}>Sumar</button>

<button onClick={restar}>Restar</button>

<button onClick={multiplicar}>Multiplicar</button>

<button onClick={dividir}>Dividir</button>

</div>

{resultado !== null && <p>Resultado: {resultado}</p>}

</div>

);

}

export default Calculadora;

**Explicación:**

* **Estados utilizados:**
  + numero1 y numero2: Almacenan los números ingresados.
  + resultado: Almacena el resultado de la operación.
* **Funciones:**
  + sumar, restar, multiplicar, dividir: Actualizan el estado resultado con la operación correspondiente.
* **Renderizado:**
  + Inputs para ingresar números.
  + Botones para seleccionar la operación.
  + Muestra el resultado si está disponible.

**Componentes**

**Clase 4: Componentes Funcionales en React**

En esta clase, nos centraremos en los **componentes funcionales** de React, que son la forma moderna y recomendada de crear componentes. Aprenderemos qué son, cómo crearlos, cómo pasarles propiedades (props) y cómo componer y reutilizar componentes para construir aplicaciones más complejas.

**Definición de Componentes en React**

Un **componente** en React es una pieza reutilizable de la interfaz de usuario que puede manejar su propio estado y lógica. Los componentes permiten dividir la interfaz en partes independientes, facilitando su desarrollo y mantenimiento.

Existen dos tipos principales de componentes en React:

* **Componentes Funcionales**: Son funciones de JavaScript que pueden recibir propiedades (props) y retornan elementos JSX que representan la interfaz de usuario.
* **Componentes de Clase** (No los usaremos en este curso): Eran la forma tradicional de crear componentes con estado antes de la introducción de los Hooks en React 16.8. Ahora, los componentes funcionales son la opción preferida.

// Componente funcional

function Saludo() {

return <h1>¡Hola, mundo!</h1>;

}

// Componente de clase

class Saludo extends React.Component {

render() {

return <h1>¡Hola, mundo!</h1>;

}

}

**Creación de Componentes Funcionales**

Un **componente funcional** es simplemente una función de JavaScript que retorna JSX.

**Ejemplo básico:**

function Saludo() {

return <h1>¡Hola, mundo!</h1>;

}

export default Saludo;

Puedes también utilizar una **función flecha**:

const Saludo = () => {

return <h1>¡Hola, mundo!</h1>;

};

export default Saludo;

**Uso del componente en tu aplicación:**

import Saludo from './Saludo';

function App() {

return (

<div>

<Saludo />

</div>

);

}

export default App;

**Props y cómo pasarlas a los componentes**

Las **props** (propiedades) son una forma de pasar datos de un componente padre a un componente hijo. Son similares a los parámetros de una función.

**Paso 1: Definir props en el componente hijo**

function Saludo(props) {

return <h1>¡Hola, {props.nombre}!</h1>;

}

Alternativamente, puedes usar **desestructuración** para acceder directamente a las props:

function Saludo({ nombre }) {

return <h1>¡Hola, {nombre}!</h1>;

}

**Paso 2: Pasar props desde el componente padre**

function App() {

return (

<div>

<Saludo nombre="María" />

</div>

);

}

**Ejemplo completo:**

// Saludo.jsx

function Saludo({ nombre }) {

return <h1>¡Hola, {nombre}!</h1>;

}

export default Saludo;

// App.jsx

import Saludo from './Saludo';

function App() {

return (

<div>

<Saludo nombre="Carlos" />

<Saludo nombre="Ana" />

<Saludo nombre="Luis" />

</div>

);

}

export default App;

**Pasar múltiples props:**

function Usuario({ nombre, edad }) {

return (

<div>

<p>Nombre: {nombre}</p>

<p>Edad: {edad}</p>

</div>

);

}

// Uso en App.jsx

<Usuario nombre="Laura" edad={25} />

**Composición y reutilización de componentes**

La **composición** es una técnica que permite construir componentes complejos a partir de componentes más simples. Esto promueve la reutilización de código y una estructura más organizada.

**Ejemplo: Componentes anidados**

Supongamos que queremos crear una tarjeta de perfil de usuario.

**Paso 1: Crear componentes simples**

// Avatar.jsx

function Avatar({ urlImagen, alt }) {

return <img src={urlImagen} alt={alt} />;

}

// InformacionUsuario.jsx

function InformacionUsuario({ nombre, bio }) {

return (

<div>

<h2>{nombre}</h2>

<p>{bio}</p>

</div>

);

}

**Paso 2: Componer componentes en uno más complejo**

// TarjetaPerfil.jsx

import Avatar from './Avatar';

import InformacionUsuario from './InformacionUsuario';

function TarjetaPerfil({ usuario }) {

return (

<div className="tarjeta-perfil">

<Avatar urlImagen={usuario.imagen} alt={usuario.nombre} />

<InformacionUsuario nombre={usuario.nombre} bio={usuario.bio} />

</div>

);

}

export default TarjetaPerfil;

**Paso 3: Usar el componente compuesto en la aplicación**

// App.jsx

import TarjetaPerfil from './TarjetaPerfil';

function App() {

const usuario = {

nombre: 'Sofia',

bio: 'Desarrolladora web entusiasta.',

imagen: 'https://ejemplo.com/sofia.jpg',

};

return (

<div>

<TarjetaPerfil usuario={usuario} />

</div>

);

}

export default App;

**Ejemplo práctico: Lista de productos**

Vamos a crear una lista de productos utilizando componentes funcionales y props.

**Paso 1: Definir el componente Producto**

function Producto({ nombre, precio, descripcion }) {

return (

<div className="producto">

<h3>{nombre}</h3>

<p>{descripcion}</p>

<p>Precio: ${precio}</p>

</div>

);

}

export default Producto;

**Paso 2: Crear el componente ListaProductos**

import Producto from './Producto';

function ListaProductos({ productos }) {

return (

<div>

{productos.map((producto) => (

<Producto

key={producto.id}

nombre={producto.nombre}

precio={producto.precio}

descripcion={producto.descripcion}

/>

))}

</div>

);

}

export default ListaProductos;

**Paso 3: Usar ListaProductos en App**

import ListaProductos from './ListaProductos';

function App() {

const productos = [

{

id: 1,

nombre: 'Camiseta',

precio: 20,

descripcion: 'Camiseta 100% algodón.',

},

{

id: 2,

nombre: 'Pantalones',

precio: 40,

descripcion: 'Pantalones cómodos y elegantes.',

},

{

id: 3,

nombre: 'Zapatos',

precio: 60,

descripcion: 'Zapatos de cuero genuino.',

},

];

return (

<div>

<h1>Lista de Productos</h1>

<ListaProductos productos={productos} />

</div>

);

}

export default App;

**Buenas prácticas con componentes funcionales**

* **Nombrar componentes con PascalCase**: Por convención, los componentes se nombran con la primera letra en mayúscula.
* function MiComponente() {
* return <div>Contenido</div>;

}

* **Usar desestructuración en las props**: Facilita el acceso a las propiedades y hace el código más legible.
* function Saludo({ nombre }) {
* return <h1>Hola, {nombre}</h1>;

}

* **Evitar la mutación de props**: Las props son de solo lectura. No intentes modificar sus valores dentro del componente.
* **Dividir componentes grandes en más pequeños**: Si un componente crece demasiado, considera dividirlo en componentes más pequeños y manejables.

**Uso de props.children**

props.children es una propiedad especial que representa los elementos hijos que se pasan a un componente.

**Ejemplo: Componente de cuadro**

function Cuadro({ titulo, children }) {

return (

<div className="cuadro">

<h2>{titulo}</h2>

<div>{children}</div>

</div>

);

}

export default Cuadro;

**Uso en App**

function App() {

return (

<div>

<Cuadro titulo="Información importante">

<p>Este es el contenido del cuadro.</p>

</Cuadro>

</div>

);

}

export default App;

**Hooks en componentes funcionales**

Los **Hooks** permiten agregar estado y otras características a los componentes funcionales.

**Uso de useState para manejar estado**

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)}>Incrementar</button>

</div>

);

}

export default Contador;

**Ejercicio**

Crea una aplicación de lista de tareas donde puedas agregar y mostrar tareas utilizando componentes funcionales.

**Paso 1: Crear el componente FormularioTarea para agregar nuevas tareas**

import { useState } from 'react';

function FormularioTarea({ agregarTarea }) {

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

const manejarSubmit = (e) => {

e.preventDefault();

agregarTarea(texto);

setTexto('');

};

return (

<form onSubmit={manejarSubmit}>

<input

type="text"

value={texto}

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

placeholder="Nueva tarea"

/>

<button type="submit">Agregar</button>

</form>

);

}

export default FormularioTarea;

**Paso 2: Crear el componente ListaTareas para mostrar las tareas**

function ListaTareas({ tareas }) {

return (

<ul>

{tareas.map((tarea, index) => (

<li key={index}>{tarea}</li>

))}

</ul>

);

}

export default ListaTareas;

**Paso 3: Integrar en App**

import { useState } from 'react';

import FormularioTarea from './FormularioTarea';

import ListaTareas from './ListaTareas';

function App() {

const [tareas, setTareas] = useState([]);

const agregarTarea = (tarea) => {

setTareas([...tareas, tarea]);

};

return (

<div>

<h1>Lista de Tareas</h1>

<FormularioTarea agregarTarea={agregarTarea} />

<ListaTareas tareas={tareas} />

</div>

);

}

export default App;

**Resumen**

* Los **componentes funcionales** son la forma moderna y recomendada de crear componentes en React.
* Los componentes funcionales son simplemente funciones de JavaScript que retornan elementos JSX.
* Las **props** permiten pasar datos de un componente padre a un componente hijo.

function Saludo({ nombre }) {

return <h1>¡Hola, {nombre}!</h1>;

}

* La **composición** es una técnica para construir componentes complejos a partir de componentes más simples.

function TarjetaPerfil({ usuario }) {

return (

<div>

<Avatar urlImagen={usuario.imagen} alt={usuario.nombre} />

<InformacionUsuario nombre={usuario.nombre} bio={usuario.bio} />

</div>

);

}

* **props.children** es una propiedad especial que representa los elementos hijos que se pasan a un componente.

function Cuadro({ titulo, children }) {

return (

<div className="cuadro">

<h2>{titulo}</h2>

<div>{children}</div>

</div>

);

}

**Ciclo de Vida y Hooks**

Con la llegada de los Hooks, el manejo del ciclo de vida en componentes funcionales ha cambiado de forma significativa. Antes, en los componentes de clase, teníamos métodos de ciclo de vida como componentDidMount, componentDidUpdate y componentWillUnmount. Ahora, con componentes funcionales, utilizamos el Hook useEffect para manejar efectos secundarios y simular estos ciclos de vida.

**¿Qué es el Ciclo de Vida?**

El **ciclo de vida** de un componente se refiere a las diferentes etapas por las que pasa un componente desde que se monta en el DOM, se actualiza cuando cambian sus props o estado, hasta que se desmonta del DOM. En componentes funcionales, no existen métodos explícitos de ciclo de vida, sino que todo se maneja mediante **Hooks**, principalmente con useEffect.

**¿Qué son los Hooks y por qué usarlos?**

**Hooks** son funciones especiales de React que permiten utilizar estado y otras características de React sin escribir componentes de clase. Antes de los Hooks, era necesario usar componentes de clase para tener estado y ciclo de vida. Con Hooks:

* **Menos código y más legibilidad**: Reducen la complejidad que tenían los componentes de clase.
* **Reutilización de lógica de estado**: Puedes compartir lógica entre componentes sin cambiar su jerarquía.
* **Transición sencilla a componentes funcionales**: Ya no necesitas clases para manejar estado o efectos secundarios.

Los Hooks más comunes son:

* **useState**: Para manejar el estado local en componentes funcionales.
* **useEffect**: Para manejar efectos secundarios (ciclo de vida).
* **useContext**: Para consumir contextos sin necesidad de prop drilling.

Y hay otros Hooks nativos como useReducer, useCallback, useMemo, así como la posibilidad de crear tus propios Hooks personalizados.

**Reglas de los Hooks**

1. **Usar Hooks solo en el nivel superior del componente**: No los llames dentro de bucles, condicionales o funciones anidadas. Deben estar siempre en la raíz de la función del componente.
2. **Usar Hooks solo en componentes funcionales**: No los llames en clases o fuera de componentes de React. Puedes llamarlos en tus propios Hooks personalizados, que también son funciones.
3. **Respetar el orden de llamadas a Hooks**: Cada render del componente debe llamar a los Hooks en el mismo orden para que React pueda preservar el estado adecuadamente.

Estas reglas aseguran que React pueda seguir el estado de manera confiable entre renderizados.

**useEffect: El Hook para efectos secundarios**

useEffect nos permite ejecutar efectos secundarios en componentes funcionales. Los efectos secundarios son cualquier operación que no se limite al cálculo o renderizado puro, por ejemplo:

* Llamadas a APIs (fetching de datos)
* Suscripciones a eventos del navegador o WebSockets
* Manipulaciones directas del DOM
* Temporizadores o intervalos

**Sintaxis básica:**

import { useEffect } from 'react';

function MiComponente() {

useEffect(() => {

// Código del efecto

});

return <div>Hola</div>;

}

**¿Cuándo se ejecuta el efecto?**  
Por defecto, el efecto se ejecuta después de cada renderizado del componente (incluyendo el montaje y cada actualización).

**Controlando Cuándo se Ejecuta el Efecto**

Podemos controlar cuándo se ejecuta el efecto pasando un segundo parámetro a useEffect: un array de dependencias.

**Array de dependencias vacío ([]):**  
Si pasas un array vacío, el efecto se ejecutará **solo una vez** cuando el componente se monte.

useEffect(() => {

console.log('Componente montado');

}, []);

**Dependencias específicas:**  
Si pasas variables en el array, el efecto se ejecutará cada vez que alguna de esas variables cambie.

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

useEffect(() => {

console.log('El contador cambió:', contador);

}, [contador]);

* Este efecto se ejecuta al montar el componente y cada vez que contador cambie.

**Limpiar los Efectos**

Algunos efectos, como suscripciones o intervalos, deben limpiarse (cancelarse) cuando el componente se desmonte o antes de ejecutar el efecto siguiente. Para ello, useEffect puede retornar una función de limpieza.

**Ejemplo: Temporizador con limpieza**

function Reloj() {

const [hora, setHora] = useState(new Date());

useEffect(() => {

const intervalId = setInterval(() => {

setHora(new Date());

}, 1000);

// Retorna una función de limpieza que se ejecuta al desmontar o antes de re-renderizar

return () => clearInterval(intervalId);

}, []);

return <div>Hora actual: {hora.toLocaleTimeString()}</div>;

}

* El efecto crea un intervalo que actualiza la hora cada segundo.
* La función retornada por useEffect se encarga de limpiar el intervalo cuando el componente se desmonte, evitando fugas de memoria.

**Simulando Métodos del Ciclo de Vida de Clases con Hooks**

* **componentDidMount**: Se simula utilizando useEffect con un array de dependencias vacío ([]). El código dentro del efecto se ejecuta una vez que el componente se monta.
* useEffect(() => {
* console.log('Componente montado');

}, []);

* **componentDidUpdate**: Se simula usando useEffect especificando las dependencias. Cada vez que alguna dependencia cambie, se ejecuta el efecto.
* const [contador, setContador] = useState(0);
* useEffect(() => {
* console.log('Componente actualizado, contador:', contador);

}, [contador]);

* **componentWillUnmount**: Se simula retornando una función de limpieza. Esta función se ejecuta cuando el componente se va a desmontar.
* useEffect(() => {
* console.log('Componente montado');
* return () => {
* console.log('Componente desmontado');
* };

}, []);

**Ejemplos Prácticos**

**1. Fetching de datos**

Cargar datos de una API al montar el componente.

function ListaUsuarios() {

const [usuarios, setUsuarios] = useState([]);

useEffect(() => {

fetch('https://api.example.com/usuarios')

.then(response => response.json())

.then(data => setUsuarios(data))

.catch(error => console.error(error));

}, []); // Se ejecuta una sola vez al montar

return (

<ul>

{usuarios.map(usuario => (

<li key={usuario.id}>{usuario.nombre}</li>

))}

</ul>

);

}

* Al montar el componente, se realiza la solicitud a la API y se guardan los usuarios en el estado.
* Cuando usuarios cambie, el componente se vuelve a renderizar mostrando los datos actualizados.

**2. Escuchar y limpiar eventos del navegador**

Suscribirse a un evento del navegador al montar el componente y desuscribirse al desmontar.

function VentanaResize() {

const [ancho, setAncho] = useState(window.innerWidth);

useEffect(() => {

const manejarResize = () => setAncho(window.innerWidth);

window.addEventListener('resize', manejarResize);

return () => {

// Limpiar el evento al desmontar

window.removeEventListener('resize', manejarResize);

};

}, []);

return <p>El ancho de la ventana es: {ancho}px</p>;

}

* El efecto se suscribe al evento resize al montar.
* La función de limpieza se encarga de remover el evento al desmontar.

**Buenas Prácticas con useEffect**

1. **Efectos específicos por tarea**: Intenta mantener los efectos enfocados a una sola tarea (fetch de datos, temporizador, etc.).
2. **Evitar hacer demasiadas cosas en un solo efecto**: Si el efecto se vuelve muy complejo, considera dividirlo en varios efectos.
3. **Dependencias correctas**: Asegúrate de incluir todas las variables que el efecto utiliza en el array de dependencias. Esto asegura que React sepa cuándo volver a ejecutarlo.
4. **Cuidado con los bucles infinitos**: Si olvidas agregar el array de dependencias, el efecto se ejecutará después de cada render, potencialmente creando bucles infinitos.

**Ejercicio Propuesto**

Crea un componente Buscador que:

* Tenga un input para ingresar un término de búsqueda.
* Use useEffect para ejecutar una búsqueda (simulada con setTimeout) cada vez que el término cambie.
* Muestra un mensaje mientras se "cargan" los resultados y luego muestra el resultado simulado.

**Pista:**

import { useState, useEffect } from 'react';

const products = [

'Apple',

'Banana',

'Cherry',

'Date',

'Elderberry',

'Fig',

'Grape',

'Honey'

]

function Buscador() {

const [termino, setTermino] = useState('');

const [resultado, setResultado] = useState(null);

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

const [items, setItems] = useState([])

useEffect(() => {

if (termino === '') {

setResultado(null);

return;

}

setResultado(`Resultados para "${termino}"`);

setItems(products.filter(product => product.toLowerCase().includes(termino)))

}, [termino]);

return (

<div>

<input

type="text"

value={termino}

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

placeholder="Escribe algo para buscar"

/>

{cargando && <p>Cargando...</p>}

{resultado && <p>{resultado}</p>}

<ul>

{items.map(item => (

<li><a href="#">{item}</a></li>

))}

</ul>

</div>

);

}

export default Buscador;

* Cuando el usuario escribe en el input, se actualiza termino.
* El efecto se vuelve a ejecutar, mostrando "Cargando..." y luego, tras 1 segundo, simula la entrega de resultados.
* Si se cambia el término durante la carga, la función de limpieza cancela el temporizador anterior antes de iniciar uno nuevo.

**Conclusión**

* Cómo el ciclo de vida de los componentes se maneja en componentes funcionales a través de Hooks, principalmente con useEffect.
* Cómo useEffect puede simular componentDidMount, componentDidUpdate y componentWillUnmount dependiendo de cómo lo utilicemos.
* Cómo limpiar efectos y manejar dependencias para controlar cuándo se ejecutan.

Comprender el uso de useEffect es esencial para manejar efectos secundarios, suscripciones, temporizadores y cualquier operación que ocurra fuera del renderizado puro en React.

**Formularios y Componentes Controlados**

Aprenderemos a manejar el input del usuario en React, comprenderemos la diferencia entre **componentes controlados** y **no controlados**, y veremos cómo implementar validaciones básicas. Además, exploraremos cómo manejar múltiples inputs en un mismo formulario.

**Manejo de Entradas de Usuario**

En React, las entradas de usuario (inputs, selects, textareas) generalmente se manejan mediante el estado. De esta forma, la interfaz refleja el estado interno del componente, asegurando un flujo de datos unidireccional y predecible.

**Ejemplo básico:**

import { useState } from 'react';

function FormularioSimple() {

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

const manejarCambio = (e) => {

setTexto(e.target.value);

};

const manejarSubmit = (e) => {

e.preventDefault();

alert(`Has escrito: ${texto}`);

};

return (

<form onSubmit={manejarSubmit}>

<input type="text" value={texto} onChange={manejarCambio} />

<button type="submit">Enviar</button>

</form>

);

}

* Cada cambio en el input dispara onChange, actualiza el estado y el valor del input se sincroniza con el estado.
* Este patrón convierte al input en un **componente controlado**.

**Componentes Controlados vs No Controlados**

**Componentes Controlados:**  
Son aquellos cuyo valor se gestiona completamente a través del estado de React. El valor del input se establece desde el estado, y cualquier cambio se refleja actualizando el estado.

* Ventajas:
  + Fuente de la verdad única: El estado determina el valor del input.
  + Fácil validación y transformación de datos.
  + Fácil de depurar y razonar sobre el flujo de datos.

**Ejemplo:**

function InputControlado() {

const [valor, setValor] = useState('');

return <input type="text" value={valor} onChange={(e) => setValor(e.target.value)} />;

}

**Componentes No Controlados:**  
Son aquellos en los que no se gestiona el valor del input mediante el estado de React, sino que se accede al valor directamente desde el DOM cuando es necesario (por ejemplo, usando ref).

* Ventajas:
  + Menos código y sobrecarga para inputs muy simples.
* Desventajas:
  + Puede ser más difícil mantener sincronizados los valores.
  + Menos control sobre el flujo de datos y validación.

**Ejemplo:**

import { useRef } from 'react';

function InputNoControlado() {

const inputRef = useRef(null);

const manejarSubmit = (e) => {

e.preventDefault();

alert(`Has escrito: ${inputRef.current.value}`);

};

return (

<form onSubmit={manejarSubmit}>

<input type="text" ref={inputRef} />

<button type="submit">Enviar</button>

</form>

);

}

* Aquí, value no depende del estado de React, sino directamente del DOM.

**Validación Básica de Formularios**

La validación se puede manejar fácilmente cuando usamos componentes controlados, ya que el estado refleja el valor actual del input y podemos verificar su validez en cada cambio o al enviar el formulario.

**Ejemplo: Validar que un campo no esté vacío**

import { useState } from 'react';

function FormularioValidado() {

const [email, setEmail] = useState('');

const [error, setError] = useState('');

const manejarSubmit = (e) => {

e.preventDefault();

if (email.trim() === '') {

setError('El email no puede estar vacío');

return;

}

setError('');

alert(`Email enviado: ${email}`);

};

return (

<form onSubmit={manejarSubmit}>

<input

type="email"

value={email}

onChange={(e) => {

setEmail(e.target.value);

setError('');

}}

placeholder="Ingresa tu email"

/>

{error && <p style={{ color: 'red' }}>{error}</p>}

<button type="submit">Enviar</button>

</form>

);

}

* Cada vez que el usuario escribe, actualizamos el estado email.
* Al enviar, validamos si el campo está vacío. Si lo está, mostramos un error.
* Si no hay errores, procedemos con la acción deseada.

**Mejoras en la validación:**

* Podemos validar otros criterios (formato de email, longitud mínima, etc.).
* Podemos validar en tiempo real o solo al enviar.
* Podemos mostrar errores específicos bajo cada input.

**Manejo de Múltiples Inputs**

Para formularios con múltiples campos, podemos mantener todos los valores en un solo estado como un objeto, o tener estados separados por cada input. Manejar un objeto es una estrategia común, ya que facilita agregar y eliminar campos.

**Ejemplo con objeto de estado:**

function FormularioMultiple() {

const [formData, setFormData] = useState({

nombre: '',

email: '',

edad: '',

});

const manejarCambio = (e) => {

const { name, value } = e.target;

setFormData({

...formData,

[name]: value,

});

};

const manejarSubmit = (e) => {

e.preventDefault();

console.log(formData);

};

return (

<form onSubmit={manejarSubmit}>

<input

type="text"

name="nombre"

value={formData.nombre}

onChange={manejarCambio}

placeholder="Nombre"

/>

<input

type="email"

name="email"

value={formData.email}

onChange={manejarCambio}

placeholder="Email"

/>

<input

type="number"

name="edad"

value={formData.edad}

onChange={manejarCambio}

placeholder="Edad"

/>

<button type="submit">Enviar</button>

</form>

);

}

**Explicación:**

* Cada input tiene un name coincidiendo con una propiedad en el objeto formData.
* En manejarCambio, actualizamos el campo específico usando [name]: value.
* Esto permite escalar fácilmente cuando el formulario crece.

**Ejemplo de Validación Múltiple**

Podemos integrar validación a cada campo en un formulario con múltiples inputs:

function FormularioConValidacion() {

const [formData, setFormData] = useState({ nombre: '', email: '' });

const [errores, setErrores] = useState({});

const validar = () => {

const nuevosErrores = {};

if (formData.nombre.trim() === '') {

nuevosErrores.nombre = 'El nombre no puede estar vacío';

}

if (formData.email.trim() === '') {

nuevosErrores.email = 'El email no puede estar vacío';

} else if (!formData.email.includes('@')) {

nuevosErrores.email = 'El email debe contener @';

}

return nuevosErrores;

};

const manejarSubmit = (e) => {

e.preventDefault();

const nuevosErrores = validar();

if (Object.keys(nuevosErrores).length > 0) {

setErrores(nuevosErrores);

return;

}

alert('Formulario enviado correctamente');

setErrores({});

};

const manejarCambio = (e) => {

const { name, value } = e.target;

setFormData({

...formData,

[name]: value,

});

setErrores({

...errores,

[name]: '',

});

};

return (

<form onSubmit={manejarSubmit}>

<div>

<input

type="text"

name="nombre"

value={formData.nombre}

onChange={manejarCambio}

placeholder="Nombre"

/>

{errores.nombre && <span style={{ color: 'red' }}>{errores.nombre}</span>}

</div>

<div>

<input

type="email"

name="email"

value={formData.email}

onChange={manejarCambio}

placeholder="Email"

/>

{errores.email && <span style={{ color: 'red' }}>{errores.email}</span>}

</div>

<button type="submit">Enviar</button>

</form>

);

}

* La función validar verifica cada campo y genera un objeto nuevosErrores con mensajes por cada campo inválido.
* Si hay errores, se establecen en el estado errores y se muestran debajo de cada input.
* Si no hay errores, se procede con la lógica de envío (por ejemplo, guardar datos o llamar a una API).

**Ejercicio Propuesto**

Crea un formulario de registro con los siguientes campos: usuario, email, password. Al enviar:

* Validar que todos los campos estén completos.
* Validar que email tenga un formato correcto (puedes usar una validación simple, como que contenga un @).
* Validar que password tenga al menos 6 caracteres.
* Si hay errores, mostrarlos debajo de cada campo.
* Si no hay errores, mostrar un mensaje de éxito.

Este ejercicio te ayudará a practicar el manejo de múltiples inputs, componentes controlados y validaciones básicas.

**Fetch**

**Clase 13: Fetching de Datos**

Aprenderemos a obtener datos desde APIs u otros recursos externos dentro de React. Utilizarás funciones como fetch (nativo del navegador) o librerías como axios para hacer solicitudes HTTP, manejarás estados de carga y error, y mostrarás los resultados en tu interfaz. Además, aprenderás a usar useEffect para realizar el fetching al montar el componente, y a mostrar mensajes de "Cargando..." o errores dependiendo del estado de la petición.

**Ejemplo:**

import { useState, useEffect } from 'react';

function ListaUsuarios() {

const [usuarios, setUsuarios] = useState([]);

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

const [error, setError] = useState(null);

useEffect(() => {

fetch('https://api.example.com/usuarios')

.then((res) => {

if (!res.ok) throw new Error('Error al obtener usuarios');

return res.json();

})

.then((data) => {

setUsuarios(data);

setCargando(false);

})

.catch((err) => {

setError(err.message);

setCargando(false);

});

}, []);

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

if (error) return <p>Error: {error}</p>;

return (

<ul>

{usuarios.map((u) => (

<li key={u.id}>{u.nombre}</li>

))}

</ul>

);

}

export default ListaUsuarios;

**ES6**

**Clase 2: JavaScript Moderno (ES6+)**

En esta clase, veremos las características modernas de JavaScript introducidas en ES6 y versiones posteriores, que son fundamentales para trabajar eficientemente con React.

**let, const y alcance de variables**

**let** y **const** son formas de declarar variables con alcance de bloque, reemplazando en muchos casos a var.

* **let**: Permite declarar variables que pueden ser reasignadas.
* let nombre = 'Juan';

nombre = 'María'; // Reasignación permitida

* **const**: Declara variables constantes que no pueden ser reasignadas.
* const PI = 3.1416;

PI = 3; // Error: no se puede reasignar una constante

**Alcance de bloque**:

if (true) {

let mensaje = 'Hola';

console.log(mensaje); // 'Hola'

}

console.log(mensaje); // Error: mensaje no está definido

**Funciones flecha**

Sintaxis más concisa para escribir funciones anónimas. No tienen su propio this.

// Función tradicional

function sumar(a, b) {

return a + b;

}

// Función flecha

const sumar = (a, b) => a + b;

console.log(sumar(2, 3)); // 5

Con un solo parámetro y cuerpo de función más complejo:

const cuadrado = n => {

const resultado = n \* n;

return resultado;

};

console.log(cuadrado(4)); // 16

**Template strings**

Permiten insertar expresiones dentro de cadenas de texto usando backticks (`) y la sintaxis ${expresión}.

const nombre = 'Ana';

const saludo = `Hola, ${nombre}!`;

console.log(saludo); // 'Hola, Ana!'

const a = 5;

const b = 10;

console.log(`La suma de ${a} y ${b} es ${a + b}.`); // 'La suma de 5 y 10 es 15.'

**Desestructuración de objetos y arrays**

Extrae valores de objetos o arrays en variables separadas.

**Objetos**:

const persona = {

nombre: 'Carlos',

edad: 30,

ciudad: 'Madrid'

};

const { nombre, edad } = persona;

console.log(nombre); // 'Carlos'

console.log(edad); // 30

**Arrays**:

const colores = ['rojo', 'verde', 'azul'];

const [primero, segundo] = colores;

console.log(primero); // 'rojo'

console.log(segundo); // 'verde'

**Operadores spread y rest**

**Spread (...)**: Expande elementos de un iterable (como un array) donde se esperan cero o más argumentos.

**Arrays**:

const numeros1 = [1, 2, 3];

const numeros2 = [4, 5, 6];

const combinados = [...numeros1, ...numeros2];

console.log(combinados); // [1, 2, 3, 4, 5, 6]

**Objetos**:

const obj1 = { a: 1, b: 2 };

const obj2 = { c: 3 };

const objCombinado = { ...obj1, ...obj2 };

console.log(objCombinado); // { a: 1, b: 2, c: 3 }

**Rest (...)**: Agrupa múltiples elementos en una sola variable.

function sumar(...numeros) {

return numeros.reduce((acc, curr) => acc + curr, 0);

}

console.log(sumar(1, 2, 3, 4)); // 10

**Módulos (import y export)**

Permiten dividir el código en archivos y reutilizarlo mediante la importación y exportación de funciones, objetos o variables.

**Exportar**:

// archivo matematicas.js

export const PI = 3.1416;

export function sumar(a, b) {

return a + b;

}

**Importar**:

// archivo principal.js

import { PI, sumar } from './matematicas.js';

console.log(PI); // 3.1416

console.log(sumar(4, 5)); // 9

**Exportación e importación por defecto**:

// archivo saludo.js

export default function saludar(nombre) {

return `Hola, ${nombre}!`;

}

// archivo principal.js

import saludar from './saludo.js';

console.log(saludar('Luis')); // 'Hola, Luis!'

**Promesas y async/await**

**Promesas**: Manejan operaciones asíncronas en JavaScript.

const promesa = new Promise((resolve, reject) => {

const exito = true;

if (exito) {

resolve('Operación exitosa');

} else {

reject('Ocurrió un error');

}

});

promesa

.then(resultado => console.log(resultado))

.catch(error => console.error(error));

**async/await**: Sintaxis más clara para trabajar con promesas.

function obtenerDatos() {

return new Promise(resolve => {

setTimeout(() => {

resolve('Datos obtenidos');

}, 1000);

});

}

async function mostrarDatos() {

const datos = await obtenerDatos();

console.log(datos);

}

mostrarDatos(); // 'Datos obtenidos' después de 1 segundo

**JSX**

**Clase 3: JSX y Renderización de Elementos**

En esta clase, exploraremos **JSX** y cómo se utiliza para construir interfaces de usuario en React. Nos enfocaremos en ejemplos prácticos para ilustrar cada concepto.

**¿Qué es JSX?**

**JSX** (JavaScript XML) es una extensión de la sintaxis de JavaScript que permite escribir código similar a HTML dentro de archivos JavaScript. Facilita la creación de componentes de React al permitirnos combinar lógica y presentación en una sintaxis familiar.

**Ejemplo básico de JSX:**

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

**Sintaxis básica de JSX**

**1. Elementos JSX**

Los elementos JSX pueden ser etiquetas HTML o componentes personalizados.

**Ejemplo con etiquetas HTML:**

const titulo = <h1>Bienvenido a mi sitio web</h1>;

**Ejemplo con componentes personalizados:**

function Saludo() {

return <p>¡Hola a todos!</p>;

}

const elemento = <Saludo />;

**2. Atributos en JSX**

Los atributos en JSX se escriben de forma similar a HTML, pero algunas palabras clave cambian.

* **class** se convierte en **className**
* **for** se convierte en **htmlFor**

**Ejemplo:**

const enlace = <a href="https://www.ejemplo.com">Visitar sitio</a>;

const boton = <button className="btn btn-primary">Clic aquí</button>;

**3. Cerrar todas las etiquetas**

En JSX, todas las etiquetas deben cerrarse, incluso las que en HTML son auto-cerradas.

**Ejemplo:**

// Correcto

const imagen = <img src="ruta/imagen.jpg" alt="Descripción" />;

// Incorrecto (falta cerrar la etiqueta)

const imagen = <img src="ruta/imagen.jpg" alt="Descripción">;

**4. Fragmentos**

Los fragmentos te permiten devolver múltiples elementos sin agregar nodos extra al DOM.

**Ejemplo:**

function Lista() {

return (

<>

<li>Elemento 1</li>

<li>Elemento 2</li>

<li>Elemento 3</li>

</>

);

}

**Renderización de elementos en el DOM**

Para renderizar elementos en el DOM con React, necesitas un archivo HTML que actúe como punto de entrada y un archivo JavaScript donde definirás tus componentes.

**Paso 1: Crear el archivo HTML**

Crea un archivo index.html con la siguiente estructura:

<!DOCTYPE html>

<html lang="es">

<head>

<meta charset="UTF-8">

<title>Mi Aplicación React</title>

</head>

<body>

<!-- Elemento raíz donde React montará la aplicación -->

<div id="root"></div>

<!-- Importar el script principal de React -->

<script type="module" src="main.jsx"></script>

</body>

</html>

* **<div id="root"></div>**: Es el contenedor donde React renderizará tu aplicación.
* **<script type="module" src="main.jsx"></script>**: Importa tu código React escrito en main.jsx.

**Paso 2: Crear el archivo JavaScript**

Crea un archivo main.jsx con el siguiente contenido:

import React from 'react';

import { createRoot } from 'react-dom/client';

function App() {

return <h1>¡Hola, React!</h1>;

}

const contenedor = document.getElementById('root');

const root = createRoot(contenedor);

root.render(<App />);

* **Importaciones**:
  + import React from 'react';: Importa la biblioteca React.
  + import { createRoot } from 'react-dom/client';: Importa la función para renderizar en el DOM.
* **Definir un componente**:
  + function App() { return <h1>¡Hola, React!</h1>; }: Crea un componente que retorna un elemento JSX.
* **Renderizar en el DOM**:
  + const contenedor = document.getElementById('root');: Obtiene el elemento donde se montará la aplicación.
  + const root = createRoot(contenedor);: Crea una raíz de React.
  + root.render(<App />);: Renderiza el componente App dentro del contenedor.

Con estos pasos, has creado una aplicación React básica que renderiza un componente en el DOM utilizando JSX. Este ejemplo muestra cómo JSX facilita la combinación de JavaScript y HTML para construir interfaces de usuario de manera eficiente.

**Expresiones y lógica en JSX**

Puedes incorporar expresiones de JavaScript dentro de JSX usando llaves {}.

**1. Insertar variables**

**Ejemplo:**

const nombre = 'María';

function Saludo() {

return <h1>Hola, {nombre}!</h1>;

}

**2. Operaciones y funciones**

**Ejemplo:**

function obtenerHora() {

return new Date().toLocaleTimeString();

}

function Reloj() {

return <p>La hora es: {obtenerHora()}</p>;

}

**3. Renderizado condicional**

Puedes usar operadores ternarios o el operador lógico && para mostrar contenido condicionalmente.

**Ejemplo con operador ternario:**

function Mensaje({ esUsuario }) {

return (

<div>

{esUsuario ? <p>Bienvenido de nuevo!</p> : <p>Por favor, regístrate.</p>}

</div>

);

}

**Ejemplo con operador lógico &&:**

function Notificacion({ nueva }) {

return <div>{nueva && <p>Tienes una nueva notificación.</p>}</div>;

}

**4. Listas y keys**

Al renderizar listas, es importante asignar una key única a cada elemento para ayudar a React a identificar cambios.

**Ejemplo:**

const frutas = ['Manzana', 'Banana', 'Cereza'];

function ListaFrutas() {

return (

<ul>

{frutas.map((fruta, index) => (

<li key={index}>{fruta}</li>

))}

</ul>

);

}

**5. Estilos en línea y clases**

Puedes aplicar estilos en línea usando objetos y asignar clases con className.

**Ejemplo de estilos en línea:**

const estilo = {

color: 'blue',

fontSize: '20px',

};

function TextoEstilizado() {

return <p style={estilo}>Este texto está estilizado.</p>;

}

**Ejemplo de clases CSS:**

function Boton() {

return <button className="btn btn-success">Aceptar</button>;

}

**Ejemplos prácticos**

**Ejemplo 1: Contador simple**

import React, { useState } from 'react';

function Contador() {

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

return (

<div>

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

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

</div>

);

}

**Ejemplo 2: Lista de tareas**

const tareas = [

{ id: 1, texto: 'Aprender React', completada: true },

{ id: 2, texto: 'Practicar ejercicios', completada: false },

{ id: 3, texto: 'Leer documentación', completada: false },

];

function ListaTareas() {

return (

<ul>

{tareas.map((tarea) => (

<li key={tarea.id} style={{ textDecoration: tarea.completada ? 'line-through' : 'none' }}>

{tarea.texto}

</li>

))}

</ul>

);

}

**Buenas prácticas con JSX**

* **Siempre cierra las etiquetas**: Tanto las etiquetas que contienen contenido como las auto-cerradas (<img />, <br />).
* **Usa className en lugar de class**: Para asignar clases CSS.
* **Envuelve múltiples elementos en un solo elemento padre**: Puede ser un <div> o un fragmento <> </>.
* **Asigna keys únicas al renderizar listas**: Idealmente, utiliza un identificador único como id.

**Ejercicio propuesto**

Crea un componente PerfilUsuario que reciba props con la información del usuario y muestre:

* Su nombre y apellido.
* Su edad.
* Una lista de hobbies.
* Si es mayor de edad, muestra un mensaje adicional.

**Ejemplo de uso:**

function App() {

const usuario = {

nombre: 'Juan',

apellido: 'Pérez',

edad: 20,

hobbies: ['Fútbol', 'Lectura', 'Programación'],

};

return <PerfilUsuario {...usuario} />;

}

**Pista para el componente PerfilUsuario:**

function PerfilUsuario({ nombre, apellido, edad, hobbies }) {

return (

<div>

<h2>

{nombre} {apellido}

</h2>

<p>Edad: {edad}</p>

{edad >= 18 && <p>Eres mayor de edad.</p>}

<h3>Hobbies:</h3>

<ul>

{hobbies.map((hobbie, index) => (

<li key={index}>{hobbie}</li>

))}

</ul>

</div>

);

}

**Fundamentos de React**

**Clase 1: Fundamentos de React**

**¿Qué es React y por qué usarlo?**

**React** es una biblioteca de JavaScript de código abierto desarrollada por Facebook para construir interfaces de usuario de manera eficiente y declarativa. Permite crear aplicaciones web rápidas y escalables mediante el uso de componentes reutilizables.

**¿Por qué usar React?**

* **Eficiencia**: Gracias al Virtual DOM, React actualiza y renderiza solo los componentes necesarios cuando los datos cambian.
* **Reutilización de Componentes**: Fomenta la creación de componentes reutilizables, lo que acelera el desarrollo y facilita el mantenimiento.
* **Comunidad Activa**: Cuenta con una gran comunidad que contribuye con herramientas, bibliotecas y soporte.
* **Aprendizaje Único**: Una vez que entiendes los conceptos básicos, puedes aplicarlos en diferentes plataformas como web, móvil (React Native) y realidad virtual (React VR).

**Historia y evolución de React**

* **2011**: React fue desarrollado internamente en Facebook para manejar el creciente tráfico y complejidad de su aplicación.
* **2013**: Facebook lanza React como proyecto de código abierto en la conferencia JSConf US.
* **2015**: Introducción de React Native, permitiendo desarrollar aplicaciones móviles nativas con React.
* **2016**: Lanzamiento de la versión 15, mejoras en el rendimiento y manejo de errores.
* **2017**: Presentación de Fiber, una reescritura del core de React para mejorar la reconciliación.
* **2019**: Introducción de los Hooks en la versión 16.8, permitiendo manejar estado y efectos en componentes funcionales.
* **2020 en adelante**: Mejoras continuas en rendimiento, desarrollo de Concurrent Mode y Suspense para manejar carga asincrónica.

**Ventajas y características clave**

* **Virtual DOM**: React utiliza un DOM virtual para minimizar manipulaciones directas del DOM real, mejorando el rendimiento.
* **JSX**: Una extensión de sintaxis que permite escribir código similar a HTML dentro de JavaScript.
* **Unidirectional Data Flow**: El flujo de datos en una sola dirección hace que la aplicación sea más predecible y fácil de depurar.
* **Ecosistema Rico**: Amplia gama de bibliotecas y herramientas para manejo de rutas, estado, pruebas y más.
* **SEO Amigable**: Con herramientas como Next.js o Astro, es posible renderizar React en el servidor para mejorar el SEO.
* **Aprendizaje Transferible**: Los conocimientos adquiridos en React pueden aplicarse en otras bibliotecas y frameworks que utilizan conceptos similares.

**Instalación del entorno de desarrollo (Node.js, npm, Vite)**

**1. Instalación de Node.js y npm**

* **Descarga Node.js**: Visita la [página oficial de Node.js](https://nodejs.org/) y descarga la versión LTS (Long Term Support) para tu sistema operativo.
* **Instalación**: Sigue las instrucciones del instalador para completar la instalación.
* **Verificación**: Abre una terminal y ejecuta:
* node -v

npm -v

Deberías ver los números de versión instalados.

**2. Crear una nueva aplicación React con Vite**

**¿Qué es Vite?**

Vite es una herramienta de construcción rápida y ligera para proyectos web modernos. Ofrece un entorno de desarrollo ultrarrápido y un bundle de producción altamente optimizado.

**Pasos para crear una aplicación React con Vite:**

1. **Crear el proyecto**

Abre la terminal y ejecuta:

npm create vite@latest my-react-app -- --template react

* + my-react-app es el nombre de tu proyecto.
  + El flag --template react indica que usarás el template de React.

1. **Ingresar al directorio del proyecto**

cd my-react-app

1. **Instalar las dependencias**

npm install

1. **Iniciar el servidor de desarrollo**

npm run dev

* + Verás una salida similar a:
  + VITE vX.X.X ready in X ms
  + ➜ Local: http://localhost:5173/
  + ➜ Network: use --host to expose
  + Abre <http://localhost:5173/> en tu navegador para ver la aplicación en funcionamiento.

1. **Explorar el proyecto**
   * Abre el proyecto en tu editor de código preferido (por ejemplo, Visual Studio Code).
   * Observa la estructura de archivos:
   * my-react-app/
   * ├── index.html
   * ├── package.json
   * ├── vite.config.js
   * └── src/
   * ├── App.jsx
   * ├── main.jsx
   * └── assets/
   * App.jsx: Componente principal de tu aplicación.
   * main.jsx: Punto de entrada donde React renderiza App en el DOM.
2. **Beneficios de usar Vite**
   * **Velocidad**: Vite utiliza ESM (Módulos de JavaScript Nativos) para ofrecer un arranque del servidor de desarrollo y actualizaciones en caliente extremadamente rápidas.
   * **Configuración mínima**: Funciona fuera de la caja sin necesidad de configuraciones complejas.
   * **Soporte para React y JSX**: Incluye soporte para JSX y otros estándares modernos.

**Nota:** A diferencia de create-react-app, Vite es más ligero y ofrece un mejor rendimiento durante el desarrollo. Además, es muy flexible y permite configuraciones avanzadas si es necesario.

**05 - React**

En este capítulo, aprenderás a **integrar React** en tu proyecto de Astro para añadir interactividad y componentes dinámicos. Veremos cómo instalar las dependencias de React, cómo “hidratar” componentes en Astro y ejemplos prácticos de uso.

**1. ¿Por Qué Integrar React en Astro?**

Astro genera contenido estático para optimizar el rendimiento, pero a veces necesitas **interactividad** (formularios dinámicos, contadores, etc.). Aquí entra React, que puede convivir con Astro sin convertir todo el sitio en una Single-Page App. Podrás:

* **Hidratar** solo los componentes que requieran comportamiento en el navegador.
* Mantener el resto de la página como HTML estático.

**2. Instalar y Configurar React**

1. **Instalar React y React DOM**  
   Desde la carpeta raíz de tu proyecto Astro:

npm install react react-dom

(O yarn add react react-dom si usas Yarn.)

1. **Integración con Astro**  
   Si creaste tu proyecto con la plantilla básica de Astro (o sin React preconfigurado), no necesitas un plugin extra: Astro sabe manejar archivos .jsx o .tsx.
   * Asegúrate de que tu astro.config.\* no excluya .jsx/.tsx de los lugares a buscar.
2. **Estructura de Archivos**  
   Una forma común es:
3. src/
4. components/
5. BotonInteractivo.jsx
6. pages/
7. index.astro
   * BotonInteractivo.jsx será un componente React.
   * index.astro (o cualquier .astro) podrá importar y usar BotonInteractivo.

**3. Añadir un Componente React**

**Ejemplo: Contador Interactivo**

1. **Crea el Componente en React**  
   src/components/Contador.jsx:
2. import { useState } from 'react';
3. export default function Contador() {
4. const [cuenta, setCuenta] = useState(0);
5. return (
6. <div className="p-4 border rounded bg-gray-50">
7. <p className="font-bold">Cuenta: {cuenta}</p>
8. <button
9. className="px-2 py-1 bg-blue-500 text-white rounded"
10. onClick={() => setCuenta(cuenta + 1)}
11. >
12. Incrementar
13. </button>
14. </div>
15. );

}

1. **Importar y Hidratar en Astro**  
   src/pages/index.astro:
2. ---
3. import Contador from '../components/Contador.jsx';
4. ---
5. <html>
6. <head>
7. <title>Mi Sitio con Astro + React</title>
8. </head>
9. <body class="p-8">
10. <h1 class="text-2xl font-bold mb-4">Página Principal</h1>
11. <!-- Para que React funcione en el cliente, agregamos el directive 'client:load' -->
12. <Contador client:load />
13. </body>

</html>

**Explicación**:

* **import Contador from …**: Cargamos el componente React.
* **<Contador client:load />**: La directiva client:load le dice a Astro que cargue e hidrate el componente en el **lado del cliente** apenas se cargue la página. Esto convierte el HTML estático en un componente interactivo de React.

**Otras Directivas de Hidratación**

* **client:idle**: Hidrata el componente cuando la página está inactiva (ideal para performance).
* **client:visible**: Hidrata cuando el elemento entra en el viewport.
* **client:media="(min-width: 768px)"**: Hidrata según una media query.
* **client:only="react"**: Exclusivo para componentes que no se renderizan como HTML estático, sino que se cargarán completamente en el navegador.

**4. Ejemplo Práctico: Botón de Cambio de Tema**

Supón que quieres un botón que cambie de “modo claro” a “modo oscuro” en tu sitio:

1. **Componente React**: src/components/TemaToggler.jsx
2. import { useState, useEffect } from 'react';
3. export default function TemaToggler() {
4. const [modoOscuro, setModoOscuro] = useState(false);
5. useEffect(() => {
6. if (modoOscuro) {
7. document.documentElement.classList.add('dark');
8. } else {
9. document.documentElement.classList.remove('dark');
10. }
11. }, [modoOscuro]);
12. return (
13. <button
14. onClick={() => setModoOscuro(!modoOscuro)}
15. className="px-3 py-1 bg-gray-700 text-white rounded"
16. >
17. {modoOscuro ? 'Modo Claro' : 'Modo Oscuro'}
18. </button>
19. );

}

**Explicación**:

* + useState mantiene un boolean para saber si estamos en modo oscuro.
  + useEffect aplica/remueve la clase dark en <html> cada vez que cambie modoOscuro.

1. **Astro File**: src/pages/index.astro
2. ---
3. import TemaToggler from '../components/TemaToggler.jsx';
4. ---
5. <html>
6. <head>
7. <meta charset="UTF-8" />
8. <title>Tema con React</title>
9. <!-- Ejemplo de uso de Tailwind con modo oscuro. -->
10. <script>
11. // Si quieres, detecta preferencia del usuario al cargar la página
12. </script>
13. </head>
14. <body class="p-4 dark:bg-gray-900 dark:text-white">
15. <h1 class="text-2xl font-bold mb-4">Modo Oscuro/Claro</h1>
16. <TemaToggler client:load />
17. </body>

</html>

* + **client:load**: Hidrata en cuanto la página cargue.
  + **Tailwind**: Asumiendo que tienes configurada la opción darkMode: 'class' en tailwind.config.js. Así, cuando .dark está en <html>, se activan estilos oscuros.

Resultado: El botón alterna el modo oscuro del sitio.

**5. Compilación y Build**

Astro compilará tu sitio estático y los componentes React que hayas marcado con client:\* se empaquetarán y cargarán en el cliente. Para ver esto:

1. **Modo Desarrollo**:

npm run dev

Abre <http://localhost:3000>. Al hacer clic en tu componente React, verás la interactividad.

1. **Build para Producción**:
2. npm run build

npm run preview

* + En la carpeta dist/, Astro generará HTML estático.
  + Incluye tus scripts empacados de React para la parte interactiva.

**6. Más Ejemplos Prácticos**

* **Formulario de Contacto**:  
  Crea un componente React con validaciones en tiempo real.
* import { useState } from 'react';
* export default function FormularioContacto() {
* const [nombre, setNombre] = useState('');
* const [mensaje, setMensaje] = useState('');
* const manejarSubmit = (e) => {
* e.preventDefault();
* alert(`Enviando mensaje de ${nombre}`);
* };
* return (
* <form onSubmit={manejarSubmit}>
* <input
* type="text"
* placeholder="Tu Nombre"
* value={nombre}
* onChange={(e) => setNombre(e.target.value)}
* />
* <textarea
* placeholder="Escribe tu mensaje"
* value={mensaje}
* onChange={(e) => setMensaje(e.target.value)}
* ></textarea>
* <button type="submit">Enviar</button>
* </form>
* );

}

Luego en .astro:

---

import FormularioContacto from '../components/FormularioContacto.jsx';

---

<FormularioContacto client:load />

* **Carousel**:  
  Un carrusel con React que maneje estados de la imagen actual, flechas de siguiente/anterior, etc.

**7. Buenas Prácticas**

1. **Usar Directivas de Hidratación con Cautela**
   * Solo aplica client:load o client:idle a los componentes que necesiten interactividad real.
   * El resto puede ser HTML estático, para mantener tu sitio rápido.
2. **Mantén Pequeños tus Componentes de React**
   * Astro te permite separar la interactividad en pequeñas partes, en vez de un gran bundle global. Así optimizas la carga.
3. **Combina con MD/MDX si Quieres**
   * Si deseas contenido en Markdown con secciones interactivas de React, podrías instalar la integración [MDX](https://docs.astro.build/en/guides/integrations-guide/mdx/).
4. **Refactoriza Lógica**
   * Para componentes React muy grandes, considera extraerlos a su propia carpeta components/ y reusarlos en múltiples .astro o .mdx.

**8. Conclusión**

Agregar **React** a Astro te brinda lo mejor de ambos mundos:

* **Velocidad** y *build* estático de Astro.
* **Interactividad** en componentes específicos con React.

Pasos Clave:

1. **Instalar** react y react-dom.
2. **Crear** componentes React en .jsx o .tsx.
3. **Importar** en tu archivo .astro con la directiva client:\*.
4. **Disfrutar** de un sitio rápido con solo las partes interactivas necesarias.

**02 - Pages**

En este segundo capítulo, profundizaremos en el sistema de **páginas** y **componentes** de Astro, centrándonos en los archivos .astro y cómo se genera contenido estático. Al finalizar, veremos cómo compilar el proyecto, obtener su salida estática y las ventajas de este enfoque.

**1. Estructura de Páginas en Astro**

Astro utiliza un sistema de **routes** basado en archivos, lo que significa que cada archivo .astro dentro de src/pages/ se convierte automáticamente en una ruta. Por ejemplo:

src/

pages/

index.astro -> Ruta: /

about.astro -> Ruta: /about

blog/

index.astro -> Ruta: /blog

post1.astro -> Ruta: /blog/post1

* **index.astro** en la raíz de pages corresponde a la página principal (/).
* Cualquier archivo .astro dentro de una subcarpeta (por ejemplo, blog/post1.astro) se corresponde con esa ruta anidada (/blog/post1).

**Ejemplo básico: index.astro**

---

const title = "Bienvenido a Astro";

---

<!DOCTYPE html>

<html lang="es">

<head>

<meta charset="UTF-8" />

<title>{title}</title>

</head>

<body>

<h1>{title}</h1>

<p>Esta es la página de inicio generada con Astro.</p>

</body>

</html>

1. **Sección Frontmatter (---)**: Aquí puedes incluir lógica en JavaScript o TypeScript, como variables (title) o datos traídos de un fetch.
2. **HTML + sintaxis de interpolación** ({title}): Astro te permite usar expresiones de JavaScript dentro de llaves.
3. **Salida en HTML**: Tras la compilación, este archivo generará una página estática con todo el HTML resultante.

**2. Componentes de Astro**

Además de las páginas, Astro te permite crear **componentes** en archivos .astro dentro de la carpeta src/components/. Esto sirve para reutilizar partes comunes (por ejemplo, un header o footer).

**Ejemplo: Header.astro**

---

const siteTitle = "Mi Sitio en Astro";

---

<header style="background: #ddd; padding: 1rem;">

<h2>{siteTitle}</h2>

</header>

Para usar este componente en una página, simplemente lo importas y lo incluyes en el HTML de otro archivo .astro:

---

import Header from '../components/Header.astro';

---

<html>

<body>

<Header />

<main>

<h1>Contenido principal</h1>

</main>

</body>

</html>

**Ventajas de los componentes .astro:**

* **Reutilizables**: Puedes tener un header, footer, sidebar, etc.
* **Livianos**: Se compilan a HTML estático, sin JavaScript adicional en el cliente (a menos que tú lo requieras).

**3. Compilar y Comprobar la Salida Estática**

Para ver el resultado estático final que genera Astro, debes ejecutar el proceso de compilación (*build*):

1. **Ejecutar el build**  
   Desde la terminal, en la carpeta del proyecto, usa:

npm run build

Esto creará una carpeta dist/ (por defecto) con todos los archivos HTML, CSS y demás recursos listos para producción.

1. **Vista previa local**  
   Después de compilar, puedes hacer:

npm run preview

Esto inicia un servidor local que sirve el contenido de dist/. Ingresa a la URL que te indique (por defecto, http://localhost:4173 u otro puerto) para ver cómo lucen tus páginas generadas.

1. **Revisar la carpeta dist**  
   Verás que cada archivo .astro se ha convertido en un archivo HTML:
2. dist/
3. ├─ index.html (de index.astro)
4. ├─ about/index.html (si había about.astro)
5. └─ assets/...

También se incluyen estilos, imágenes y otros recursos en subcarpetas correspondientes.

**4. Ventajas del Enfoque Estático**

1. **Rendimiento Mejorado**: Las páginas se sirven como HTML puro, sin necesidad de JavaScript adicional para mostrar contenido estático. Esto se traduce en un *Time to First Byte (TTFB)* más rápido.
2. **Coste de Hospedaje Reducido**: Como es una web estática, puedes desplegarla en servicios de hosting estático (Netlify, GitHub Pages, Vercel, etc.) sin complejas configuraciones de servidor.
3. **Menor Complejidad**: No necesitas un backend para servir contenido dinámico básico, y Astro se encarga de la estructura final de los archivos.
4. **Mejor SEO**: Los motores de búsqueda pueden rastrear el contenido HTML directamente, sin esperar a la ejecución de JavaScript.

**03 - Pages 2**

En este archivo profundizaremos en la creación de páginas más avanzadas, como un **blog** con rutas dinámicas. Veremos cómo Astro permite manejar slugs (identificadores de página) de forma sencilla, cómo aprovechar funcionalidades como Astro.params, el uso de archivos Markdown para manejar contenido y, muy importante, cómo usar **getStaticPaths** para generar múltiples rutas estáticas automáticamente. Además, veremos brevemente cómo organizar un pequeño blog y permitir que cada entrada tenga su propia ruta, incluyendo el uso de **rutas comodín** con [...slug].astro.

**1. Rutas Dinámicas con Slugs**

Una de las grandes ventajas de Astro es que puedes crear rutas dinámicas de forma declarativa. Para ello, se emplean archivos con el patrón [slug].astro dentro de src/pages/. Por ejemplo:

src/

pages/

blog/

[slug].astro

index.astro

* **blog/[slug].astro** definirá una ruta de la forma /blog/lo-que-sea.
* slug se convierte en un parámetro que podrás leer dentro del archivo .astro usando Astro.params.

**Ejemplo Básico de [slug].astro**

---

const { slug } = Astro.params;

// Podrías usar esta variable para buscar contenido, por ejemplo, de un CMS o de un archivo Markdown.

// const postData = await obtenerPostPorSlug(slug);

const postData = {

titulo: `Título para el slug: ${slug}`,

contenido: `Contenido simulado para el slug "${slug}".`,

};

---

<html>

<head>

<title>{postData.titulo}</title>

</head>

<body>

<h1>{postData.titulo}</h1>

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

</body>

</html>

**Explicación:**

* Astro.params.slug te devuelve la parte de la URL que aparece después de /blog/.
* En este ejemplo, simulamos que obtenerPostPorSlug(slug) retorna la información del post.
* Puedes crear tantas rutas dinámicas como quieras, por ejemplo: /blog/mi-post, /blog/otro-post, etc.

**Rutas Comodín: [...slug].astro**

Cuando quieras capturar **uno o varios segmentos** en la ruta, puedes usar un **archivo comodín** [...slug].astro. Por ejemplo:

src/

pages/

blog/

[...slug].astro

* **blog/[...slug].astro** definirá rutas como:
  + /blog/mi-post → Astro.params.slug = ["mi-post"]
  + /blog/2023/mayo/nuevo-post → Astro.params.slug = ["2023","mayo","nuevo-post"]
* Esto te permite tener múltiples niveles de anidación sin crear varias subcarpetas.

**Ejemplo básico de [...slug].astro:**

---

const { slug } = Astro.params; // slug será un array, ej. ["2023","mayo","mi-post"]

let titulo = "Página Dinámica con Slug";

let contenido = `Ruta capturada: ${slug.join("/")}`;

// Podrías parsear slug[0] como el año, slug[1] como el mes, etc.

---

<html>

<head>

<title>{titulo}</title>

</head>

<body>

<h1>{titulo}</h1>

<p>{contenido}</p>

</body>

</html>

**2. Organización de un Blog**

Para un blog básico, puedes combinar **páginas estáticas** y **rutas dinámicas**:

1. **Página de Listado de Posts** (blog/index.astro):  
   Muestra enlaces a cada post dinámico. Este archivo se renderiza en /blog/.
2. **Páginas de Posts** (blog/[slug].astro o blog/[...slug].astro):  
   Contenido individual de cada post, basado en el slug.

**Ejemplo de blog/index.astro**

---

const posts = [

{ slug: 'post-1', titulo: 'Primer Post' },

{ slug: 'post-2', titulo: 'Segundo Post' },

];

---

<html>

<body>

<h1>Blog</h1>

<ul>

{posts.map(({ slug, titulo }) => (

<li>

<a href={`/blog/${slug}`}>{titulo}</a>

</li>

))}

</ul>

</body>

</html>

**Explicación:**

* Declaramos un array de posts con slug y titulo.
* Generamos un <ul> que enlaza a cada post mediante <a href={/blog/${slug}}>.

**3. Uso de getStaticPaths para Generar Múltiples Páginas**

Cuando desees compilar múltiples rutas dinámicas a HTML estático sin crearlas manualmente, Astro ofrece el **getStaticPaths**. Esto permite definir qué slugs se van a generar durante la etapa de construcción (*build*).

**Ejemplo: Generar varios posts con getStaticPaths**

Supongamos que queremos compilar estáticamente tres posts: post-1, post-2 y post-3. Creamos blog/[slug].astro con la siguiente estructura:

---

export async function getStaticPaths() {

// Retorna un array con los slugs que queremos generar

return [

{ params: { slug: 'post-1' } },

{ params: { slug: 'post-2' } },

{ params: { slug: 'post-3' } },

];

}

const { slug } = Astro.params;

// En un caso real, aquí podrías hacer un fetch de una API o leer un archivo Markdown:

const postData = {

titulo: `Título para slug: ${slug}`,

contenido: `Contenido para ${slug}`,

};

---

<html>

<head>

<title>{postData.titulo}</title>

</head>

<body>

<h1>{postData.titulo}</h1>

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

</body>

</html>

**¿Qué ocurre aquí?**

* **getStaticPaths**: Se ejecuta en la etapa de build y retorna un array de objetos. Cada objeto contiene un params con las variables (en este caso, { slug: 'post-1' }, etc.).
* Astro crea estáticamente las rutas /blog/post-1, /blog/post-2 y /blog/post-3.
* Durante la compilación, cada params.slug se inyecta en Astro.params.slug, permitiendo que tu archivo .astro renderice el contenido apropiado.

**¿Y si los slugs vienen de un archivo Markdown o un CMS?**

export async function getStaticPaths() {

// Imagina que tenemos un array de posts [{ slug: 'post-1' }, { slug: 'post-2' }, ...]

const posts = await fetchPostsFromCMS();

return posts.map((post) => ({

params: { slug: post.slug },

}));

}

* Aquí cada ruta se generará según la lista que obtengas de un CMS, de archivos .md, etc. Esto hace que el sitio se compile con todas las páginas necesarias.

**4. Uso de Markdown y Astro para Contenido**

Además de manejar datos simulados o un CMS, Astro te permite usar archivos Markdown (.md) y Markdown con frontmatter (.mdx). Cada archivo se puede **importar** y **renderizar** directamente en tu página Astro, haciendo muy fácil la creación de blogs basados en contenido estático.

**Ejemplo:**

---

import PostMD from '../../content/posts/mi-post.md';

---

<html>

<body>

<PostMD />

</body>

</html>

* Aquí, mi-post.md podría tener un frontmatter con título, fecha, etc.
* Astro lo convertirá a HTML en tiempo de compilación, generando una salida estática.

Para un flujo más avanzado, Astro provee un sistema de *collections* (usando content/config.\*).

**5. Compilar y Ver la Salida**

Como en el capítulo anterior, al ejecutar:

npm run build

Astro generará una carpeta dist/ que incluirá:

* Una carpeta blog/ con:
  + index.html (listado del blog).
  + [slug]/index.html para cada slug solicitado en getStaticPaths.
* Además de los recursos estáticos y otros archivos HTML.

Con npm run preview, podrás comprobar cómo se han creado estáticamente las páginas /blog/post-1, /blog/post-2, etc., sin requerir un servidor dinámico.

**6. Ventajas de las Rutas Dinámicas en Astro**

1. **Flexibilidad**: Permite manejar muchos posts o productos sin crear manualmente cada archivo .astro.
2. **Rendimiento Estático**: En modo SSG (Static Site Generation), Astro crea un HTML estático por cada ruta especificada, ofreciendo cargas rápidas y excelente SEO.
3. **SEO Mejorado**: Cada página dinámica se convierte en un archivo HTML con URL propia, en lugar de depender de un sistema SPA que cargue dinámicamente el contenido desde el navegador.
4. **Integración de Contenido**: Fácil de combinar con archivos Markdown, bases de datos o APIs externas sin necesidad de un backend complejo.
5. **Rutas Comodín**: Con [...slug].astro, puedes capturar múltiples segmentos en una sola página, ideal para URLs de estructura variable.
6. **Automatización con getStaticPaths**: Astro genera automáticamente las rutas que definas, simplificando el mantenimiento y escalado de tu sitio.

**7. Conclusión**

Has aprendido cómo:

* Crear rutas dinámicas con [slug].astro y [...slug].astro.
* Utilizar getStaticPaths para generar múltiples rutas de forma automática, lo cual es ideal para blogs o sitios con numerosos posts.
* Combinar contenido de un CMS, archivos Markdown o datos simulados para crear páginas en Astro.
* Compilar todo a un sitio estático y disfrutar de un rendimiento y SEO sobresalientes.

**Blog con Astro y Tailwind CSS**

En este tutorial aprenderás a combinar **Astro** con **Tailwind CSS** para crear un blog sencillo. Nos basaremos en la [guía oficial de Tailwind para Astro](https://docs.astro.build/en/guides/integrations-guide/tailwind/), añadiendo pasos para configurar un blog estático y aplicar estilos con Tailwind.

**1. Crear un Nuevo Proyecto Astro**

1. **Instala o actualiza Node.js**  
   Asegúrate de tener Node.js 16+ instalado. Puedes verificar con:

node -v

1. **Crea el proyecto con npm create astro@latest**  
   En la carpeta donde quieras tu proyecto, ejecuta:

npm create astro@latest

* + El asistente te pedirá nombre de proyecto y tipo de plantilla.
  + Elige “**Basic**” para comenzar con lo mínimo.

1. **Instala dependencias**  
   Entra al directorio generado y ejecuta:
2. cd mi-blog-astro

npm install

1. **Verifica la instalación**  
   Arranca el servidor de desarrollo:

npm run dev

**2. Agregar la Integración de Tailwind**

Astro ofrece una integración oficial para Tailwind que facilita la configuración. Sigue estos pasos basados en la [guía oficial](https://docs.astro.build/en/guides/integrations-guide/tailwind/):

1. **Instala el paquete de integración de Tailwind**  
   Desde el directorio del proyecto, ejecuta:
2. npx astro add tailwind

npm install @astrojs/tailwind tailwindcss

Esto incluye tanto la integración (@astrojs/tailwind) como la dependencia tailwindcss en tu proyecto.

1. **Configura la integración en tu astro.config.\***  
   Abre o crea el archivo astro.config.mjs (o .js) y añade:
2. import { defineConfig } from 'astro/config';
3. import tailwind from '@astrojs/tailwind';
4. export default defineConfig({
5. integrations: [
6. tailwind({
7. // Configuración opcional (ej. aplicar la transform de tailwind)
8. // Example:
9. // config: { applyBaseStyles: false },
10. }),
11. ],

});

1. **Crear archivos de configuración de Tailwind**  
   En la terminal, genera el archivo tailwind.config.js (opcionalmente, si deseas personalizarlo):

npx tailwindcss init # SOLO si no tienes creado el archivo

Esto creará tailwind.config.js donde podrás especificar tus rutas, colores, plugins, etc.

1. **Asegúrate de que Astro procese el CSS**  
   Si la integración está activa, Astro inyecta automáticamente Tailwind en tu build. Por defecto, la guía de Astro te sugiere crear o editar un archivo src/styles/global.css donde se importen las directivas de Tailwind:
2. /\* src/styles/global.css \*/
3. @tailwind base;
4. @tailwind components;

@tailwind utilities;

Y luego, este archivo será inyectado por la integración sin que necesites importarlo manualmente en cada página.

1. **Inicia el servidor**

npm run dev

Astro y Tailwind CSS ya están funcionando juntos. Para verificarlo, ve a tus archivos .astro y aplica clases de Tailwind (por ejemplo, class="text-red-500").

**3. Estructura del Blog**

Crearemos un blog con:

* **Página de listado** de posts (src/pages/blog/index.astro).
* **Rutas dinámicas** para cada post (src/pages/blog/[slug].astro).

La idea es simular que tenemos varios posts, ya sea en un array o en un sistema de archivos.

**3.1 Crear la Página de Listado**

Crea src/pages/blog/index.astro:

---

const posts = [

{ slug: 'hola-astro', title: 'Hola Astro', date: '2023-07-15' },

{ slug: 'otro-post', title: 'Mi Otro Post', date: '2023-07-16' },

];

---

<html>

<head>

<title>Blog - Mi Astro + Tailwind</title>

</head>

<body class="bg-gray-100 text-gray-800">

<main class="max-w-screen-md mx-auto p-4">

<h1 class="text-3xl font-bold mb-8">Blog</h1>

<ul class="space-y-4">

{posts.map(({ slug, title, date }) => (

<li class="bg-white shadow p-4 rounded">

<h2 class="text-xl font-semibold">

<a href={`/blog/${slug}`} class="text-blue-600 hover:text-blue-800">

{title}

</a>

</h2>

<p class="text-gray-600 text-sm">{date}</p>

</li>

))}

</ul>

</main>

</body>

</html>

* **Clases de Tailwind**:
  + bg-gray-100, text-gray-800, text-3xl, etc.
  + Observa cómo añaden estilos sin escribir CSS manual.
* **Listamos** los posts (simulados en un array) y creamos enlaces a cada slug.

**3.2 Crear Rutas Dinámicas**

Para mostrar el contenido de cada post, crea src/pages/blog/[slug].astro:

---

const { slug } = Astro.params;

// Simulación de datos: en producción podrías fetch de un CMS o un archivo .md

const postData = {

hola-astro: {

title: 'Hola Astro',

date: '2023-07-15',

content: 'Este es mi primer post en Astro con Tailwind.',

},

'otro-post': {

title: 'Mi Otro Post',

date: '2023-07-16',

content: 'Contenido genial de mi segundo post.',

},

};

const post = postData[slug] ?? { title: 'No Encontrado', date: '', content: '' };

---

<html>

<head>

<title>{post.title}</title>

</head>

<body class="bg-gray-100 text-gray-800">

<main class="max-w-screen-md mx-auto p-4">

<article class="bg-white p-4 rounded shadow">

<h1 class="text-3xl font-bold mb-2">{post.title}</h1>

<p class="text-gray-600 mb-4">{post.date}</p>

<div class="prose">

{post.content}

</div>

</article>

<div class="mt-8">

<a href="/blog" class="text-blue-600 hover:text-blue-800">« Volver al Blog</a>

</div>

</main>

</body>

</html>

* **Astro.params.slug** obtiene la parte de la URL que viene después de /blog/.
* **Clases de Tailwind** añaden padding, fondos, tipografías, etc.

**4. Estilizar con Tailwind**

En los ejemplos anteriores, ya se usan clases de Tailwind (bg-gray-100, text-blue-600, etc.). Para personalizar más, edita tu tailwind.config.js:

/\*\* @type {import('tailwindcss').Config} \*/

export default {

content: [

// Ajusta rutas a tu proyecto Astro

"./src/\*\*/\*.astro",

"./src/\*\*/\*.js",

"./src/\*\*/\*.jsx",

"./src/\*\*/\*.ts",

"./src/\*\*/\*.tsx",

],

theme: {

extend: {

// Puedes añadir colores, tipografías personalizadas, etc.

},

},

plugins: [

// Si quieres plugins como @tailwindcss/typography

],

};

Si deseas un mejor estilo para el contenido del post (texto, títulos, etc.), instala **@tailwindcss/typography**:

npm install @tailwindcss/typography

Agrega en tailwind.config.js:

plugins: [

require('@tailwindcss/typography'),

],

Ahora en tus archivos .astro, asigna la clase prose para generar un aspecto agradable:

<div class="prose">

{post.content}

</div>

**5. Ejecución y Build Final**

1. **Ejecutar en modo dev**:

npm run dev

Visita <http://localhost:3000/blog> para ver la lista de posts y acceder a cada slug.

1. **Build para Producción**:
2. npm run build

npm run preview

* + Se generará la carpeta dist/ con un sitio estático de alto rendimiento.
  + preview es solo para probar localmente; al desplegar a un hosting (Netlify, Vercel, etc.) sube la carpeta dist/.

**Conclusión**

* **Astro** te ofrece la generación estática y rutas dinámicas, facilitando que tu sitio sea rápido y fácil de desplegar.
* **Tailwind** te da un sistema de utilidades CSS que evita escribir CSS manual y te permite prototipar y dar estilo rápidamente.
* **Sigue explorando**: Usa Content Collections, archivos Markdown, o integraciones con React/Vue para enriquecer más el blog.

**View Transitions**

Astro admite **view transitions opcionales por página** con solo unas pocas líneas de código. Las view transitions actualizan el contenido de tu página sin recargar la navegación completa normal del navegador, lo que proporciona animaciones fluidas entre páginas.

Astro ofrece un componente de enrutamiento <ViewTransitions /> que se puede agregar dentro del <head> de una sola página para controlar las transiciones de página mientras navegas hacia otra página. Proporciona un enrutador ligero del lado del cliente que [intercepta la navegación](https://docs.astro.build/es/guides/view-transitions/#proceso-de-navegaci%C3%B3n-del-lado-del-cliente) y te permite personalizar la transición entre páginas.

Agrega este componente a un componente .astro reutilizable, como un encabezado común o un diseño, para lograr [transiciones animadas de página en todo tu sitio (modo SPA)](https://docs.astro.build/es/guides/view-transitions/#view-transitions-completas-en-todo-el-sitio-modo-spa).

El soporte de las view transitions en Astro está impulsado por la nueva API del navegador [View Transitions](https://developer.chrome.com/docs/web-platform/view-transitions/) y también incluye:

* Algunas [opciones de animación integradas](https://docs.astro.build/es/guides/view-transitions/#directivas-de-animaci%C3%B3n-integradas), como fade, slide y none.
* Soporte para animaciones de navegación hacia adelante y hacia atrás.
* La capacidad de [personalizar completamente todos los aspectos de la animación de transición](https://docs.astro.build/es/guides/view-transitions/#personalizando-animaciones) y crear tus propias animaciones.
* La opción de [impedir la navegación del lado del cliente para enlaces que no sean de página](https://docs.astro.build/es/guides/view-transitions/#previniendo-la-navegaci%C3%B3n-del-lado-del-cliente).
* [Control sobre el comportamiento de respaldo](https://docs.astro.build/es/guides/view-transitions/#control-de-respaldo) para navegadores que aún no admiten las API de View Transitions.
* Soporte automático para [prefers-reduced-motion](https://docs.astro.build/es/guides/view-transitions/#prefers-reduced-motion).

Nota

Por defecto, cada página utilizará la navegación normal del navegador, ocupando toda la página. Debes optar por ver las view transitions y puedes usarlas por página o en todo el sitio.

**Agregando las View Transitions a una página**

Opta por utilizar view transitions en páginas individuales importando y añadiendo el componente de enrutamiento <ViewTransitions /> dentro del <head> en cada página deseada.

src/pages/index.astro

---

import { ViewTransitions } from 'astro:transitions';

---

<html lang="es">

<head>

<title>Mi página de inicio</title>

<ViewTransitions />

</head>

<body>

<h1>¡Bienvenido a mi sitio web!</h1>

</body>

</html>

**View transitions completas en todo el sitio (modo SPA)**

Importa y agrega el componente <ViewTransitions /> a tu componente <head> común o de diseño compartido. Astro creará animaciones de página predeterminadas basadas en las similitudes entre la página antigua y la nueva, y también proporcionará un comportamiento de respaldo para los navegadores que no lo admitan.

El ejemplo a continuación muestra cómo agregar las animaciones de navegación de página predeterminadas de Astro en todo el sitio, incluida la opción de control de respaldo predeterminado para navegadores que no las admiten, importando y agregando este componente a un componente <CommonHead /> de Astro:

src/components/CommonHead.astro

---

import { ViewTransitions } from 'astro:transitions';

---

<link rel="icon" type="image/svg+xml" href="/favicon.svg" />

<meta name="generator" content={Astro.generator} />

<!-- Etiquetas Meta Primarias -->

<title>{title}</title>

<meta name="title" content={title} />

<meta name="description" content={description} />

<ViewTransitions />

¡No es necesario realizar ninguna otra configuración para habilitar la navegación predeterminada del lado del cliente en Astro!

Utiliza [directivas de transición](https://docs.astro.build/es/guides/view-transitions/#directivas-de-transici%C3%B3n) o [anula la navegación predeterminada del lado del cliente](https://docs.astro.build/es/guides/view-transitions/#previniendo-la-navegaci%C3%B3n-del-lado-del-cliente) en elementos individuales para un control más preciso.

**Directivas de transición**

Astro asignará automáticamente a los elementos encontrados tanto en la página antigua como en la nueva un view-transition-name compartido y único. Esta pareja de elementos coincidentes es inferida por el tipo de elemento y su ubicación en el DOM.

Utiliza las directivas opcionales transition:\* en los elementos de la página dentro de tus componentes .astro para un control más fino sobre la animación de transición de la página durante la navegación.

* transition:name: Te permite anular la coincidencia de elementos predeterminada de Astro para la animación del contenido antiguo/nuevo y [especificar un nombre de transición](https://docs.astro.build/es/guides/view-transitions/#nombrando-una-transici%C3%B3n) para asociar un par de elementos DOM.
* transition:animate: Te permite modificar la animación predeterminada de Astro mientras reemplaza el elemento antiguo por el nuevo especificando un tipo de animación. Utiliza las [directivas de animación integradas](https://docs.astro.build/es/guides/view-transitions/#directivas-de-animaci%C3%B3n-integradas) de Astro o [crea animaciones de transición personalizadas](https://docs.astro.build/es/guides/view-transitions/#personalizando-animaciones).
* transition: persist: Te permite anular el reemplazo predeterminado de los elementos antiguos por los nuevos de Astro y, en su lugar, [persistir los componentes y elementos HTML](https://docs.astro.build/es/guides/view-transitions/#manteniendo-el-estado) al navegar a otra página.

**Nombrando una transición**

En algunos casos, es posible que desees o necesites identificar los elementos de transición de vista correspondientes tú mismo. Puedes especificar un nombre para un par de elementos utilizando la directiva transition:name.

src/pages/old-page.astro

<aside transition:name="hero">

src/pages/new-page.astro

<aside transition:name="hero">

Ten en cuenta que el valor proporcionado transition:name solo se puede utilizar una vez en cada página. Establece esto manualmente solo cuando Astro no pueda inferir un nombre adecuado por sí mismo, o para un control más fino sobre los elementos coincidentes.

**Manteniendo el estado**

**Agregado en:** astro@2.10.0

Puedes persistir componentes y elementos HTML (en lugar de reemplazarlos) a través de las navegaciones de página utilizando la directiva transition:persist.

Por ejemplo, el siguiente <video> continuará reproduciéndose a medida que navegas a otra página que contiene el mismo elemento de video. Esto funciona tanto para la navegación hacia adelante como hacia atrás.

src/components/Video.astro

<video controls="" autoplay="" transition:persist>

<source src="https://ia804502.us.archive.org/33/items/GoldenGa1939\_3/GoldenGa1939\_3\_512kb.mp4" type="video/mp4">

</video>

También puedes colocar la directiva en una [isla de Astro](https://docs.astro.build/es/concepts/islands/) (un componente de framework UI con una directiva [client:](https://docs.astro.build/es/reference/directives-reference/#directivas-del-cliente)). Si ese componente existe en la siguiente página, la isla de la página antigua **con su estado actual** continuará mostrándose, en lugar de reemplazarla con la isla de la nueva página.

En el siguiente ejemplo, el estado interno del componente del recuento no se restablecerá cuando se navegue hacia adelante y hacia atrás por páginas que contengan el componente <Counter /> con el atributo transition:persist.

components/Header.astro

<Counter client:load transition:persist initialCount={5} />

También puedes [identificar manualmente los elementos correspondientes](https://docs.astro.build/es/guides/view-transitions/#nombrando-una-transici%C3%B3n) si la isla/elemento está en un componente diferente entre las dos páginas.

src/pages/old-page.astro

<Video controls="" autoplay="" transition:name="media-player" transition:persist />

src/pages/new-page.astro

<MyVideo controls="" autoplay="" transition:name="media-player" transition:persist />

Una alternativa práctica, transition:persist puede tomar un nombre de transición como valor.

src/pages/index.astro

<video controls="" autoplay="" transition:persist="media-player">

**transition:persist-props**

**Agregado en:** astro@4.5.0

Esto te permite controlar si los props de una isla deben persistir o no tras la navegación.

Por defecto, cuando se añade transition:persist a una isla, el estado se mantiene durante la navegación, pero el componente se vuelve a renderizar con nuevos props. Esto es útil, por ejemplo, cuando un componente recibe props específicos de la página, como el title de la página actual.

Puede anular este comportamiento con transition:persist-props. Al añadir esta directiva se conservarán los accesorios existentes de una isla (no se volverá a renderizar con nuevos valores) además de mantener su estado existente.

**Directivas de animación integradas**

Astro viene con algunas animaciones integradas para anular la transición fade predeterminada. Agrega la directiva transition:animate a elementos individuales para personalizar el comportamiento de transiciones específicas.

* fade (por defecto): Una animación de fundido cruzado opinionada. El contenido antiguo se desvanece y el nuevo contenido se desvanece al aparecer.
* initial: Optar por no usar la animación de fundido cruzado opinionada de Astro y en su lugar utilizar el estilo predeterminado del navegador.
* slide: Una animación donde el contenido antiguo se desliza hacia la izquierda y el nuevo contenido se desliza desde la derecha. En la navegación hacia atrás, las animaciones son opuestas.
* none: Desactiva las animaciones predeterminadas del navegador. Úsalo en el elemento <html> de una página para desactivar el fade predeterminado para cada elemento en la página.

Combina directivas para tener un control total sobre la animación de tu página. Establece un valor predeterminado para la página en el elemento <html>, y anúlalo en los elementos individuales que desees.

El siguiente ejemplo produce una animación de deslizamiento para el contenido del cuerpo, al mismo tiempo que desactiva la animación de fundido predeterminada del navegador para el resto de la página:

---

import CommonHead from '../components/CommonHead.astro';

---

<html transition:animate="none">

<head>

<CommonHead />

</head>

<body>

<header>

...

</header>

<!-- Anula la configuración predeterminada de tu página en un solo elemento -->

<main transition:animate="slide">

...

</main>

</body>

</html>

**Personalizando Animaciones**

Puedes personalizar todos los aspectos de una transición utilizando las propiedades de animación de CSS.

Para personalizar una animación incorporada, primero importa la animación desde astro:transitions, y luego proporciona opciones de personalización.

El ejemplo a continuación personaliza la duración de la animación incorporada fade:

---

import { fade } from 'astro:transitions';

---

<header transition:animate={fade({ duration: '0.4s' })}>

También puedes definir tus propias animaciones para usar con transition:animate, definiendo tanto el comportamiento hacia adelante como hacia atrás, así como las páginas nuevas y antiguas, de acuerdo con los siguientes tipos:

export interface TransitionAnimation {

name: string; // El nombre del keyframe

delay?: number | string;

duration?: number | string;

easing?: string;

fillMode?: string;

direction?: string;

}

export interface TransitionAnimationPair {

old: TransitionAnimation | TransitionAnimation[];

new: TransitionAnimation | TransitionAnimation[];

}

export interface TransitionDirectionalAnimations {

forwards: TransitionAnimationPair;

backwards: TransitionAnimationPair;

}

El siguiente ejemplo muestra todas las propiedades necesarias para definir una animación personalizada de fade:

---

const anim = {

old: {

name: 'fadeIn',

duration: '0.2s',

easing: 'linear',

fillMode: 'forwards',

},

new: {

name: 'fadeOut',

duration: '0.3s',

easing: 'linear',

fillMode: 'backwards',

}

};

const myFade = {

forwards: anim,

backwards: anim,

};

---

<header transition:animate={myFade}> ... </header>

**Controlar el enrutador**

El enrutador <ViewTransitions /> maneja la navegación escuchando:

* Los clics en elementos <a>.
* Eventos de navegación hacia atrás y hacia adelante.

Las siguientes opciones te permiten controlar aún más cuándo ocurre la navegación dentro del enrutador:

* data-astro-reload: un atributo de la etiqueta <a> para [forzar una navegación de página completa](https://docs.astro.build/es/guides/view-transitions/#previniendo-la-navegaci%C3%B3n-del-lado-del-cliente).
* data-astro-history="auto | push | replace": un atributo de la etiqueta <a> para [controlar el historial del navegador](https://docs.astro.build/es/guides/view-transitions/#reemplazar-entradas-en-el-historial-del-navegador).
* navigate(href, options): un método disponible para cualquier script o componente cliente para [desencadenar la navegación](https://docs.astro.build/es/guides/view-transitions/#desencadenar-la-navegaci%C3%B3n).

**Previniendo la navegación del lado del cliente**

Existen casos en los que no se puede navegar a través del enrutamiento del lado del cliente ya que ambas páginas involucradas deben utilizar el enrutador <ViewTransitions /> para evitar una recarga de página completa. También puede que no desees la navegación del lado del cliente en cada cambio de navegación y prefieras una navegación de página tradicional en rutas selectas en su lugar.

Puedes optar por no utilizar la navegación del lado del cliente de manera selectiva para cada enlace añadiendo el atributo data-astro-reload a cualquier etiqueta <a> o <form>. Este atributo anulará cualquier componente <ViewTransitions /> existente y, en su lugar, provocará una recarga del navegador durante la navegación.

El siguiente ejemplo muestra cómo evitar la navegación del lado del cliente al navegar a un artículo desde la página de inicio únicamente. Esto aún te permite tener animaciones en elementos compartidos, como una imagen destacada, al navegar a la misma página desde una página de listado de artículos:

src/pages/index.astro

<a href="/articles/emperor-penguins" data-astro-reload>

src/pages/articles.astro

<a href="/articles/emperor-penguins">

Los enlaces con el atributo data-astro-reload serán ignorados por el enrutador y se producirá una navegación de página completa.

**Desencadenar la navegación**

También puedes desencadenar la navegación del lado del cliente a través de eventos que normalmente no son escuchados por el enrutador <ViewTransitions /> utilizando navigate. Esta función del módulo astro:transitions/client se puede utilizar en scripts y en componentes del framework que se hidratan con una [directiva de cliente](https://docs.astro.build/es/reference/directives-reference/#directivas-del-cliente).

El siguiente ejemplo muestra un componente de Astro que lleva al visitante a otra página que selecciona desde un menú:

src/components/Form.astro

<script>

import { navigate } from 'astro:transitions/client';

// Navega a la opción seleccionada automáticamente.

document.querySelector('select').onchange = (ev) => {

let href = ev.target.value;

navigate(href);

};

</script>

<select>

<option value="/play">Jugar</option>

<option value="/blog">Blog</option>

<option value="/about">Acerca de</option>

<option value="/contact">Contacto</option>

</select>

src/pages/index.astro

---

import Form from "../components/Form.astro";

import { ViewTransitions } from "astro:transitions";

---

<html>

<head>

<ViewTransitions />

</head>

<body>

<Form />

</body>

</html>

El siguiente ejemplo implementa lo mismo usando navigate() en un componente <Form /> de React:

src/components/Form.jsx

import { navigate } from "astro:transitions/client";

export default function Form() {

return (

<select onChange={(e) => navigate(e.target.value)}>

<option value="/play">Jugar</option>

<option value="/blog">Blog</option>

<option value="/about">Acerca de</option>

<option value="/contact">Contacto</option>

</select>

);

}

El componente <Form /> puede ser renderizado en una página de Astro que utiliza el enrutador <ViewTransitions />, con una directiva del cliente:

src/pages/index.astro

---

import Form from "../components/Form.jsx";

import { ViewTransitions } from "astro:transitions";

---

<html>

<head>

<ViewTransitions />

</head>

<body>

<Form client:load />

</body>

</html>

El método navigate toma los siguientes argumentos:

* href (obligatorio) - La nueva página a la que se va a navegar.
* options - Un objeto opcional con las siguientes propiedades:
  + history: 'push' | 'replace' | 'auto'
    - 'push': el enrutador utilizará history.pushState para crear una nueva entrada en el historial del navegador.
    - 'replace': el enrutador utilizará history.replaceState para actualizar la URL sin agregar una nueva entrada en la navegación.
    - 'auto' (por defecto): el enrutador intentará utilizar history.pushState, pero si la URL no es una que se pueda transicionar, la URL actual permanecerá sin cambios en el historial del navegador.
  + formData: Un objeto FormData para solicitudes POST.

Para navegar hacia atrás y hacia delante por el historial del navegador, puedes combinar navigate() con las funciones integradas history.back(), history.forward() e history.go() del navegador. Si navigate() es llamado durante la renderización del lado del servidor de tu componente, no tendrá efecto alguno.

**Reemplazar entradas en el historial del navegador**

Normalmente, cada vez que navegas, se crea una nueva entrada en el historial del navegador. Esto permite la navegación entre páginas utilizando los botones atrás y adelante del navegador.

El enrutador <ViewTransitions /> te permite sobrescribir las entradas del historial al agregar el atributo data-astro-history a cualquier etiqueta <a> individual.

El atributo data-astro-history puede establecerse en los mismos tres valores que la [opción history de la función navigate()](https://docs.astro.build/es/guides/view-transitions/#desencadenar-la-navegaci%C3%B3n):

data-astro-history: 'push' | 'replace' | 'auto'

* 'push': el enrutador utilizará history.pushState para crear una nueva entrada en el historial del navegador.
* 'replace': el enrutador utilizará history.replaceState para actualizar la URL sin agregar una nueva entrada en la navegación.
* 'auto' (por defecto): el enrutador intentará utilizar history.pushState, pero si la URL no se puede transicionar, la URL actual se mantendrá sin cambios en el historial del navegador.

El siguiente ejemplo navega a la página /main, pero no agrega una nueva entrada al historial de navegación. En cambio, reutiliza la entrada actual en el historial (/confirmation) y la sobrescribe.

src/pages/confirmation.astro

<a href="/main" data-astro-history="replace">

Esto tiene el efecto de que si retrocedes desde la página /main, el navegador no mostrará la página /confirmation, sino la página anterior a ella.

**Transiciones con formularios**

**Agregado en:** astro@4.0.0

El enrutador <ViewTransitions /> activará transiciones en la página desde elementos <form>, admitiendo tanto peticiones GET como POST.

Por defecto, Astro envía los datos de tu formulario como multipart/form-data cuando el atributo method está configurado como POST. Si deseas que coincida con el comportamiento predeterminado de los navegadores web, utiliza el atributo enctype para enviar tus datos codificados como application/x-www-form-urlencoded:

src/components/Form.astro

<form action="/contact" method="POST" enctype="application/x-www-form-urlencoded">

<!-- -->

</form>

Puedes optar por excluir las transiciones del enrutador en formularios individuales mediante el atributo data-astro-reload:

src/components/Form.astro

<form action="/contact" data-astro-reload>

<!-- -->

</form>

**Control de respaldo**

El enrutador <ViewTransitions /> funciona mejor en navegadores que admiten las View Transitions (p.ej. navegadores basados en Chromium), pero también incluye soporte predeterminado de respaldo para otros navegadores. Incluso si el navegador no admite la API de View Transitions, Astro seguirá proporcionando navegación en el navegador utilizando una de las opciones de respaldo para obtener una experiencia comparable.

Puedes anular el soporte de respaldo predeterminado de Astro agregando una propiedad fallback en el componente <ViewTransitions /> y estableciéndolo en swap o none:

* animate (predeterminado, recomendado) - Astro simulará view transitions utilizando atributos personalizados antes de actualizar el contenido de la página.
* swap - Astro no intentará animar la página. En su lugar, la página antigua será reemplazada inmediatamente por la nueva.
* none - Astro no realizará ninguna transición animada de página. En su lugar, obtendrás navegación de página completa en navegadores que no admitan esta función.

---

import { ViewTransitions } from 'astro:transitions';

---

<title>Mi sitio</title>

<ViewTransitions fallback="swap" />

Limitaciones conocidas

La animación del navegador initial no es simulada por Astro. Por lo tanto, cualquier elemento que utilice esta animación no será animado.

**Proceso de navegación del lado del cliente**

Cuando se utiliza el enrutador <ViewTransitions />, se siguen los siguientes pasos para llevar a cabo la navegación del lado del cliente en Astro:

1. Un visitante de tu sitio desencadena la navegación mediante cualquiera de las siguientes acciones:
   * Haciendo clic en una etiqueta <a> que enlaza internamente a otra página de tu sitio.
   * Haciendo clic en el botón de retroceso.
   * Haciendo clic en el botón de avance.
2. El enrutador comienza a buscar la siguiente página.
3. El enrutador agrega el atributo data-astro-transition al elemento HTML con un valor de 'forward' o 'back' según corresponda.
4. El enrutador llama a document.startViewTransition. Esto desencadena el propio [proceso de transición de vista](https://developer.mozilla.org/en-US/docs/Web/API/View_Transitions_API#the_view_transition_process) del navegador. Es importante destacar que el navegador captura una captura de pantalla del estado actual de la página.
5. Dentro del callback de startViewTransition, el enrutador realiza un **intercambio**, que consta de la siguiente secuencia de eventos:
   * El contenido de <head> se intercambia, manteniendo algunos elementos:
     + Los nodos DOM de las hojas de estilo se mantienen si existen en la nueva página, para evitar el parpadeo de contenido no estilizado (FOUC).
     + Los scripts se mantienen si existen en la nueva página.
     + Cualquier otro elemento de <head> con transition:persist se mantiene si hay un elemento correspondiente en la nueva página.
   * <body> se reemplaza completamente con el cuerpo de la nueva página.
   * Los elementos marcados con transition:persist se trasladan al nuevo DOM si existen en la nueva página.
   * La posición de desplazamiento se restaura si es necesario.
   * Se desencadena el evento astro:after-swap en el document. Esto marca el final del proceso de **intercambio**.
6. El enrutador espera a que se carguen las nuevas hojas de estilo antes de resolver la transición.
7. El enrutador ejecuta cualquier nuevo script agregado a la página.
8. Se dispara el evento astro:page-load. Esto marca el final del proceso de navegación.

**Comportamiento del script con view transitions**

Al añadir view transitions a un proyecto de Astro existente, es posible que algunos de tus scripts ya no se vuelvan a ejecutar tras la navegación por la página, como ocurría con las actualizaciones de página completa del navegador. Utiliza la siguiente información para asegurarte de que tus scripts se ejecutan como se espera.

**Orden del Script**

Al navegar entre páginas con el componente <ViewTransitions />, los scripts se ejecutan en orden secuencial para ajustarse al comportamiento del navegador.

**Reejecución del script**

Los [Bundled module scripts](https://docs.astro.build/es/guides/client-side-scripts/#procesamiento-de-scripts), que son los scripts por defecto en Astro, sólo se ejecutan una vez. Después de la ejecución inicial serán ignorados, incluso si el script existe en la nueva página después de una transición.

A diferencia de los scripts de módulos empaquetados, los [scripts inline](https://docs.astro.build/es/guides/client-side-scripts/#optar-por-no-procesar) pueden volver a ejecutarse durante la visita de un usuario a un sitio si existen en una página que se visita varias veces. Los scripts inline también pueden volver a ejecutarse cuando un visitante navega a una página sin el script y luego vuelve a una con el script.

**data-astro-rerun**

**Agregado en:** astro@4.5.0

Para forzar a los scripts inline a reejecutarse después de cada transición, añade la propiedad data-astro-rerun. Al añadir cualquier atributo a un script también se añade implícitamente is:inline, por lo que esto sólo está disponible para scripts que no están empaquetados y procesados por Astro.

<script is:inline data-astro-rerun>...</script>

Para garantizar que un script se ejecuta cada vez que se carga una página durante la navegación del lado del cliente, debe ser ejecutado por un [evento del ciclo de vida](https://docs.astro.build/es/guides/view-transitions/#eventos-del-ciclo-de-vida). Por ejemplo, los escuchadores de eventos para DOMContentLoaded pueden sustituirse por el evento de ciclo de vida [astro:page-load](https://docs.astro.build/es/guides/view-transitions/#astropage-load).

Si tienes código que establece un estado global en un script inline, este estado necesitará tener en cuenta que el script podría ejecutarse más de una vez. Comprueba el estado global en tu etiqueta <script>, y ejecuta condicionalmente tu código cuando sea posible. Esto funciona porque window se mantiene.

<script is:inline>

if (!window.SomeGlobal) {

window.SomeGlobal = {} // ....

}

</script>

**Eventos del ciclo de vida**

El enrutador <ViewTransition /> proporciona varios eventos en el document durante la navegación. Estos eventos proporcionan hooks en el ciclo de vida de la navegación, permitiéndote realizar acciones como mostrar indicadores de que la nueva página está cargado, sobrescribe el comportamiento predeterminado y restablece el estado mientras se completa la navegación.

El proceso de navegación implica una fase de **preparación**, cuando el nuevo contenido es cargado; una fase de **intercambio de DOM**, donde el contenido de la página antigua se sustituye por el de la nueva; y una fase de **finalización** donde los scripts son ejecutados, la carga se informa como completada y se realiza un trabajo de limpieza.

Los eventos del ciclo de vida de la API de View Transitions de Astro, en orden, son:

* [astro:before-preparation](https://docs.astro.build/es/guides/view-transitions/#astrobefore-preparation)
* [astro:after-preparation](https://docs.astro.build/es/guides/view-transitions/#astroafter-preparation)
* [astro:before-swap](https://docs.astro.build/es/guides/view-transitions/#astrobefore-swap)
* [astro:after-swap](https://docs.astro.build/es/guides/view-transitions/#astroafter-swap)
* [astro:page-load](https://docs.astro.build/es/guides/view-transitions/#astropage-load)

Consejo

El evento before- te permite influir y modificar acciones que están a punto de producirse y los after- son notificaciones de que se ha completado una fase.

Mientras algunas acciones pueden ser disparadas durante cualquier evento, algunas tareas solo pueden realizarse durante un evento especifico para mejores resultados, como mostrar un spinner de carga antes de la preparación o sobrescribir pares de animaciones antes de intercambiar el contenido.

**astro:before-preparation**

**Agregado en:** astro@3.6.0

Un evento que es disparado al inicio de la fase de preparación, después de que la navegación ha comenzado (p. ej. después de que el usuario haya clicado un enlace), pero antes de que el contenido es cargado.

Este evento es usado:

* Para hacer algo antes de que la carga haya comenzado, como mostrar un spinner de carga.
* Para modificar la carga, como cargar el contenido que has definido en una plantilla en lugar de una URL externa.
* Para cambiar la dirección de la navegación (que suele ser adelante o atrás) para realizar una animación personalizada.

Aquí hay un ejemplo usando el evento astro:before-preparation para cargar un spinner antes de que se cargue el contenido y pararlo inmediatamente depués la carga. Ten en cuenta que el uso del callback del loader de esta manera permite la ejecución asíncrona del código.

<script is:inline>

document.addEventListener('astro:before-preparation', ev => {

const originalLoader = ev.loader;

ev.loader = async function() {

const { startSpinner } = await import('./spinner.js');

const stop = startSpinner();

await originalLoader();

stop();

};

});

</script>

**astro:after-preparation**

**Agregado en:** astro@3.6.0

Un evento que es diparado al final de la fase de preparación, después de que el contenido de la nueva página haya sido cargado y analizado en un documento. Este evento ocurre antes de la fase de view transitions.

Este ejemplo usa el evento astro:before-preparation para comenzar un indicador de carga y el evento astro:after-preparation para pararlo:

<script is:inline>

document.addEventListener('astro:before-preparation', () => {

document.querySelector('#loading').classList.add('show');

});

document.addEventListener('astro:after-preparation', () => {

document.querySelector('#loading').classList.remove('show');

});

</script>

Esta es una versión más sencilla de un spinner de carga que el ejemplo mostrado anteriormente: si todo el código de los listener’s puede ejecutarse de forma síncrona, no es necesario conectarse al callback del loader.

**astro:before-swap**

**Agregado en:** astro@3.6.0

Un evento que es disparado antes de que el nuevo documento (que se completa durante la fase de preparación) reemplace el documento actual. Este evento ocurre dentro de una view transition, donde el usuario sigue viendo una instantánea de la página anterior.

Este evento puede ser usado para realizar cambios antes de que el intercambio ocurra. La propiedad newDocument en el evento representa el documento entrante. Aquí hay un ejempl para asegurar que la preferencia del modo claro u oscuro del navegador en localStorage se traslada a la nueva página:

<script is:inline>

function setDarkMode(document) {

let theme = localStorage.darkMode ? 'dark' : 'light';

document.documentElement.dataset.theme = theme;

}

setDarkMode(document);

document.addEventListener('astro:before-swap', ev => {

// Pasa el documento entrante para establecer el tema en él

setDarkMode(ev.newDocument);

});

</script>

El evento astro:before-swap también puede ser usado para cambiar la **implementación** del intercambio. La implementación del intercambio por defecto compara el contenido principal, mueve elementos **persistentes** del antiguo documento al newDocument, y luego reemplaza todo el body con el body del nuevo documento.

En este punto del ciclo de vida, podrías optar por definir tu propia implementación de intercambio, por ejemplo, para comparar todo el contenido del documento existente (como hacen algunos otros enrutadores):

<script is:inline>

document.addEventListener('astro:before-swap', ev => {

ev.swap = () => {

diff(document, ev.newDocument);

};

});

</script>

**astro:after-swap**

Un evento que se dispara inmediatamente después de que la nueva página reemplace la página antigua. Puedes escuchar este evento en el document y desencadenar acciones que ocurrirán antes de que se rendericen los elementos DOM de la nueva página y se ejecuten los scripts.

Este evento, cuando se escucha en la **página saliente**, es útil para pasar y restaurar cualquier estado en el DOM que necesite transferirse a la nueva página.

Este es el último punto en el ciclo de vida donde sigue siendo seguro para, por ejemplo, añadir un nombre de clase para el modo oscuro (<html class="dark-mode">), aunque podrías hacerlo en un evento anterior.

El evento astro:after-swap ocurre inmediatamente después de que el historial del navegador haya sido actualizado y la posición del scroll haya sido establecida. Por lo tanto, un uso de este evento es sobreescribir la restauración predeterminada para la navegación en el historial. El siguiente ejemplo restablece la posición del scroll horizontal y vertical en la esquina superior izquierda de la página para cada navegación.

document.addEventListener('astro:after-swap',

() => window.scrollTo({ left: 0, top: 0, behavior: 'instant' }))

**astro:page-load**

Un evento que se dispara al final de la navegación de la página, después de que la nueva página es visible para el usuario y se han cargado los estilos y scripts bloqueantes. Puedes escuchar este evento en el document.

El componente <ViewTransitions /> dispara este evento se dispara tanto en la navegación inicial de la página para una página pre-renderizada como en cualquier navegación posterior, ya sea hacia delante o hacia atrás.

Puedes usar este evento para ejecutar código en cada navegación de página, o sólo una vez:

<script>

document.addEventListener('astro:page-load', () => {

// Esto solo se ejecuta una vez.

setupStuff();

}, { once: true });

</script>

**Accesibilidad**

Habilitar la navegación del lado del cliente y animar las view transitions presentan desafíos de accesibilidad, y Astro tiene como objetivo hacer que los sitios que optan por las View Transitions sean accesibles por defecto en la medida de lo posible.

**Anuncio de ruta**

**Agregado en:** astro@3.2.0

El componente <ViewTransitions /> incluye un anunciador de ruta para la navegación de páginas durante la navegación del lado del cliente. No se necesita configuración ni acción para habilitar esto.

Las tecnologías de asistencia permiten que los visitantes sepan que la página ha cambiado anunciando el nuevo título de la página después de la navegación. Cuando se utiliza la navegación del lado del servidor con recargas tradicionales de la página completa en el navegador, esto ocurre automáticamente después de que la nueva página se carga. En la navegación del lado del cliente, el componente <ViewTransitions /> realiza esta acción.

Para agregar un anuncio de ruta a la navegación del lado del cliente, el componente agrega un elemento a la nueva página con el atributo aria-live configurado en assertive. Esto indica a las AT (tecnologías de asistencia) que deben anunciar inmediatamente. El componente también verifica lo siguiente, en orden de prioridad, para determinar el texto del anuncio:

* El <title>, si existe.
* El primer <h1> que encuentre.
* La pathname de la página.

Recomendamos encarecidamente que siempre incluyas un elemento <title> en cada página por razones de accesibilidad.

**prefers-reduced-motion**

El componente <ViewTransitions /> de Astro incluye una media query de CSS que deshabilita *todas* las animaciones de transición de vista, incluida la animación de respaldo, siempre que se detecte la configuración [prefer-reduced-motion](https://developer.mozilla.org/en-US/docs/Web/CSS/@media/prefers-reduced-motion). En su lugar, el navegador simplemente intercambiará los elementos DOM sin una animación.

**Actualizar a la v3.0 desde la v2.x**

Las view transitions ya no están detrás de una bandera experimental en Astro v3.0.

Si **no** habías habilitado esta bandera experimental en Astro 2.x, esto no causará ningún cambio disruptivo en tu proyecto. La nueva API de View Transitions no afecta tu código existente.

Si anteriormente estabas utilizando view transitions experimentales, es posible que haya algunos cambios disruptivos cuando actualices tu proyecto Astro desde una versión anterior.

Por favor, sigue las instrucciones a continuación según corresponda para actualizar **un proyecto de Astro v2.x configurado con experimental.viewTransitions: true** a la v3.0.

**Actualizar desde experimental.viewTransitions**

Si anteriormente habías habilitado la bandera experimental para las view transitions, deberás actualizar tu proyecto para Astro v3.0, que ahora permite las view transitions de manera predeterminada.

**Eliminar la bandera experimental.viewTransitions**

Elimina la bandera experimental:

astro.config.mjs

import { defineConfig } from 'astro/config';

export default defineConfig({

experimental: {

viewTransitions: true

}

});

**Actualizar la fuente de importación**

El componente <ViewTransitions /> ha sido movido de astro:components a astro:transitions. Actualiza la fuente de importación en todas las ocurrencias de tu proyecto.

src/layouts/BaseLayout.astro

---

import { ViewTransitions } from " astro:transitions"

---

<html lang="en">

<head>

<title>Mi Página de Inicio</title>

<ViewTransitions />

</head>

<body>

<h1>¡Bienvenido a mi sitio web!</h1>

</body>

</html>

**Actualizar directivas transition:animate**

**Cambiado:** El valor morph de la directiva transition:animate ha sido renombrado a initial. Además, esta ya no es la animación por defecto. Si no se especifica ninguna directiva transition:animate, tus animaciones ahora serán por defecto fade.

1. Renombra cualquier animación morph a initial.

src/components/MyComponent.astro

<div transition:name="name" transition:animate=" initial" />

 Para mantener cualquier animación que previamente usara morph por defecto, añade explícitamente transition:animate="initial" a esas animaciones.

src/components/MyComponent.astro

<div transition:name="name" transition:animate="initial" />

 Puedes eliminar de manera segura cualquier animación configurada explícitamente como fade. Esto es ahora el comportamiento por defecto:

src/components/MyComponent.astro

<div transition:name="name" />

**Añadido:** Astro también admite un nuevo valor para transition:animate, llamado none. Este valor puede ser utilizado en el elemento <html> de una página para desactivar las transiciones animadas de página completa en toda la página. Esto solo anulará el **comportamiento de animación predeterminado** en elementos de la página que no tengan una directiva de animación. Aún puedes establecer animaciones en elementos individuales y estas animaciones específicas ocurrirán.

1. Ahora puedes desactivar todas las transiciones predeterminadas en una página individual, animando solamente los elementos que utilicen explícitamente una directiva transition:animate:

<html transition:animate="none">

<head></head>

<body>

<h1>¡Hola Mundo!</h1>

</body>

</html>

**Actualizar nombres de eventos**

El evento astro:load ha sido renombrado a astro:page-load. Renombra todas las instancias en tu proyecto.

src/components/MyComponent.astro

<script>

document.addEventListener(' astro:page-load', runSetupLogic);

</script>

El evento astro:beforeload ha sido renombrado como astro:after-swap. Renombra todas las instancias en tu proyecto.

src/components/MyComponent.astro

<script>

document.addEventListener(' astro:after-swap', setDarkMode);

</script>