# Unidad 6.1:
# Introducción a TypeScript en proyectos de React

- TypeScript es un superconjunto de JavaScript que añade tipado estático.
- React es una librería de JavaScript para construir interfaces de usuario.
- Unir TypeScript con React permite mayor:
  - Seguridad
  - Productividad
  - Mantenimiento.

# ¿Por qué usar TypeScript en React?

- Reduce errores en tiempo de ejecución.
- Facilita el autocompletado en el editor.
- Mejora la comprensión del código en equipos grandes.
- Hace el código más robusto y confiable.

# Instalación de TypeScript

**Create React App (CRA)** fue el estándar durante años, pero:

*npx create-react-app mi-app --template typescript*

- Es lento al arrancar y al hacer builds.
- Usa Webpack con una configuración pesada y poco flexible.
- Está prácticamente descontinuado: ya no recibe mejoras relevantes.
- No aprovecha las optimizaciones modernas del ecosistema.

# Instalación de TypeScript

**Vite** se ha convertido en el estándar moderno porque:

*npm create vite@latest mi-app --template react-ts*

- Arranca en milisegundos gracias a **ES modules**.
- El hot reload es instantáneo, incluso en proyectos grandes.
- La build final usa Rollup, muy optimizada.
- Tiene una configuración más simple y extensible.
- La plantilla oficial de React + TypeScript es muy ligera y limpia.

# Archivos .tsx

En proyectos React + TypeScript, los componentes deben estar en archivos ***.tsx***.

- La extensión *.tsx* es necesaria porque mezcla JSX con TypeScript.
- Ejemplo: Componente en MyComponent.tsx

```
function MyComponent() {
  return <h1>Hola TypeScript</h1>;
}
export default MyComponent;
```

# Tipos básicos en TypeScript

- **number**: números enteros o decimales.
- **string**: cadenas de texto.
- **boolean**: verdadero o falso.
- **any**: permite cualquier tipo (no se recomienda).
- **null y undefined**: valores vacíos.

```
let edad: number = 18;
let nombre: string = 'Ana';
let estudiante: boolean = true;
```

# Props en React con TypeScript

Las ***props*** son parámetros que reciben los componentes.

- En TS, **las props se definen con una interfaz** o tipo.
- Esto garantiza que se usen correctamente los datos.

```
type Props = { nombre: string, edad: number };

function Saludo({ nombre, edad }: Props) {
  return <p>Hola {nombre}, tienes {edad} años.</p>;
}
```

# Props opcionales

Podemos indicar que una prop es opcional usando **'?'**.

- Esto permite no pasarla en ciertos casos.

```
type Props = { nombre: string, edad?: number };

function Usuario({ nombre, edad }: Props) {
  return <p>{nombre} - {edad ?? 'Edad no indicada'}</p>;
}
```

# State con useState y TypeScript

***useState*** necesita saber el tipo de dato que maneja.

- Puede inferirse automáticamente. Pero podemos **especificarlo explícitamente para mayor seguridad**.

```
const [contador, setContador] =
useState<number>(0);
setContador(contador + 1);
```

# Estado complejo

- El estado puede ser un objeto o lista.
- Debemos definir el tipo correcto para evitar errores.

```
type Tarea = { id: number; titulo: string; completada: boolean };

const [tareas, setTareas] = useState<Tarea[]>([]);
```

# Eventos en React con TypeScript

Los eventos se tipan fuertemente en TypeScript.

- Ejemplo: onChange para un input.

```
function Formulario() {
  const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
      console.log(e.target.value);
  };
  return <input onChange={handleChange} />;
}
```

# Children en componentes

React permite pasar contenido como children.

- En TS, se tipa con **ReactNode**.

```
type Props = { children: React.ReactNode };

function Layout({ children }: Props) {
  return <div>{children}</div>;
}
```

# Componentes con clases

- Aunque hoy en día se usan más hooks, aún existen componentes con clases.
- Se tipan con Props y State genéricos.

```
class Contador extends React.Component<{ inicio: number }, { valor: number }>
{
  state = { valor: this.props.inicio };
  render() {
    return <p>{this.state.valor}</p>;
  }
}
```

# useEffect con TypeScript

useEffect no necesita tipado extra si usamos variables ya tipadas.

- Muy útil para efectos secundarios.

```
useEffect(() => {
  document.title = `Clicks: ${contador}`;
}, [contador]);
```

# useReducer con objeto tipado

***useReducer*** es útil para manejar estados complejos.

- Debemos tipar acciones y estados.

```
type Estado = { contador: number };
type Accion = { type: 'incrementar' } | { type: 'decrementar' };

function reducer(state: Estado, action: Accion): Estado {
  switch(action.type){
    case 'incrementar': return { contador: state.contador + 1 };
    case 'decrementar': return { contador: state.contador - 1 };
  }
}
```

# Context API con TS

Context provee datos a múltiples componentes.

- Debemos tipar el valor y el Provider.

```
type Usuario = { nombre: string };

const UsuarioContext = React.createContext<Usuario | null>(null);

function App(){
  return <UsuarioContext.Provider value={{ nombre: 'Ana' }}>
    <Perfil />
  </UsuarioContext.Provider>
}
```

# Uso de Context con useContext

- useContext facilita el consumo de contextos.

- Siempre mantener tipos claros.

```
function Perfil(){
  const usuario = useContext(UsuarioContext);
  return <p>{usuario?.nombre}</p>;
}
```

# Ejercicio 1

Crea un componente ListaTareas con las siguientes características:

- Recibe una lista de tareas como prop.

- Cada tarea tiene id, título y completada.

- Muestra las tareas en un <ul>.

# Ejercicio 1 - Solución

- Definimos el tipo y mapeamos las tareas.

```
type Tarea = { id: number; titulo: string; completada: boolean };

type Props = { tareas: Tarea[] };

function ListaTareas({ tareas }: Props) {
  return <ul>{tareas.map(t => <li key={t.id}>{t.titulo}</li>)}</ul>;
}
```

# Ejercicio 2

Crea un contador con botones + y -.

- Usa useState con tipo number.

- Muestra el valor actualizado.

# Ejercicio 2 - Solución

- Definimos el estado tipado y manejadores de eventos.

```
function Contador(){
  const [valor, setValor] = useState<number>(0);
  return (
    <div>
      <button onClick={() => setValor(valor - 1)}>-</button>
      <span>{valor}</span>
      <button onClick={() => setValor(valor + 1)}>+</button>
    </div>
  );
}
```

# Tipos de utilidades en TS

- ***Partial<T>***: hace opcionales todas las propiedades.
- ***Pick<T,K>***: selecciona algunas propiedades.
- ***Omit<T,K>***: excluye propiedades.

```
type Usuario = { id: number; nombre: string; edad: number };
type UsuarioOpcional = Partial<Usuario>;
type UsuarioSinEdad = Omit<Usuario, 'edad'>;
```

# Default Props con TS

Podemos simular valores por defecto asignando directamente.

- TS asegura compatibilidad de tipos.

```
type Props = { mensaje?: string };

function Info({ mensaje = 'Sin info' }: Props) {
  return <p>{mensaje}</p>;
}
```

# Tipos para listas y mapas

- Listas: **Array<T>** o **T[]**.

```
const numeros: number[] = [1,2,3];
```

- Objetos clave-valor: **Record<K,V>**.

```
const edades: Record<string, number> = { Ana: 18, Luis: 20 };
```

# Tuplas en TypeScript

- Una tupla es un array con número fijo de elementos y tipos concretos para cada posición.
- Ejemplo: coordenadas con (x, y)

```
let coordenada: [number, number] = [10, 20];
```

# Tipos para funciones

- Podemos tipar parámetros y retorno.
- Ayuda a evitar valores incorrectos.

```
function suma(a: number, b:number): number {
  return a + b;
}
```

# Tipos genéricos con React

Los genéricos permiten crear componentes reutilizables.

- Muy útiles en listas y formularios.

```
function Lista<T>({ items }: { items: T[] }){
  return <ul>{items.map((i, idx) => <li key={idx}>{String(i)}</li>)}</ul>;
}
```

# Errores comunes

- Olvidar tipar estados complejos.
- Usar 'any' en exceso.
- No tipar eventos correctamente.
- No declarar interfaces para props.

# Beneficios a largo plazo

- Menos errores en producción.
- Mayor claridad en el código.
- Mejor colaboración en equipo.
- Facilidad para refactorizar proyectos grandes.

# Conclusiones

- TypeScript mejora React haciéndolo más seguro y predecible.
- Aprender tipado desde el inicio evita malas prácticas.
- La inversión de tiempo inicial se compensa con menos errores futuros.

# Referencias

- Documentación oficial TypeScript: https://www.typescriptlang.org/
- Documentación React: https://react.dev/
- Documentación TS + React: https://react-typescript-cheatsheet.netlify.app/

# Unidad 6.2:
# interfaces, type vs interface

Nos enfocaremos en:

- los tipos básicos
- interfaces
- la diferencia entre type e interface.

Entenderemos cómo aplicar tipado en componentes de React paso a paso.

# Introducción a Interfaces

Una interfaz describe la forma que debe tener un objeto.

- Se usa para tipar props, estados o datos en React.
- Facilita la reutilización de tipos en varias partes del código.

```
interface Persona {
  nombre: string;
  edad: number;
}
const user: Persona = { nombre: 'Ana', edad: 18 };
```

# Ventajas de las Interfaces

- Permiten definir contratos claros de datos.
- Hacen que el código sea más legible.
- Son reutilizables y pueden extenderse.

# Extender Interfaces

Podemos crear una interfaz a partir de otra ya definida. Esto añade propiedades adicionales.

```
interface Persona {
  nombre: string;
}
interface Estudiante extends Persona {
  matricula: string;
}
const alumno: Estudiante = { nombre: 'Luis', matricula:
'1234A' };
```

# Interfaces en funciones

Además de objetos, podemos tipar funciones con interfaces.

- Así definimos la firma que debe cumplir una función.

```
interface Suma {
  (a: number, b: number): number;
}
const sumar: Suma = (x, y) => x + y;
```

# Interfaces en React

En React se usan para tipar las props.

- Esto asegura que el componente reciba las propiedades correctas.

```
interface Props {
  nombre: string;
}
function Saludo({ nombre }: Props) {
  return <h1>Hola {nombre}</h1>;
}
```

# ¿Qué es 'type' en TypeScript?

- Un type permite definir un alias de un tipo.
- Puede ser para objetos, funciones, uniones, etc.
- Es muy flexible y útil en React.

```
type Punto = { x: number; y: number };
let p1: Punto = { x: 3, y: 8 };
```

# Uniones de tipos con type

- Un type puede unir varios tipos posibles.
- Esto significa que una variable puede aceptar diferentes formas.

```
type ID = string | number;
let userId: ID;
userId = '123';
userId = 456;
```

# Diferencia entre type e interface

- interface se usa más para describir objetos y contratos reutilizables.
- type es más general y puede describir cualquier tipo, como uniones o literales.
- Ambos pueden parecer similares, pero interface es más extensible.

# Extensibilidad: interface vs type

- Las interfaces permiten herencia (extends).

```
interface Animal { nombre: string }
interface Perro extends Animal { raza: string }
```

- Los type usan combinaciones con '&'.

```
type Animal2 = { nombre: string }
type Perro2 = Animal2 & { raza: string }
```

Ambos logran resultados parecidos con distinta sintaxis.

# Ejercicio Interfaces

- Define una interfaz Producto con nombre, precio y disponible (boolean).
- Crea una variable usando esa interfaz.

# Solución Interfaces

- Definimos la interfaz y creamos un objeto siguiendo su contrato.

```
interface Producto {
  nombre: string;
  precio: number;
  disponible: boolean;
}

let telefono: Producto = {
  nombre: 'iPhone',
  precio: 999,
  disponible: true
};
```

# Ejercicio Interfaces en React

- Crea un componente que reciba como props un objeto tipo Persona.
- Debe mostrar el nombre y la edad.

# Solución Interfaces en React

- Usamos una interfaz y tipamos las props del componente.

```
interface Persona {
  nombre: string;
  edad: number;
}

function Perfil({ nombre, edad }: Persona) {
  return <p>{nombre} - {edad}</p>;
}
```

# Ejercicio Type vs Interface

- Define una interface Animal con nombre y especie.
- Define un type Mascota como unión Animal | null.
- Crea una variable usando ese tipo.

# Solución Type vs Interface

- Creamos la interface y luego un type que la usa en unión.

```
interface Animal {
  nombre: string;
  especie: string;
}

type Mascota = Animal | null;

let miMascota: Mascota = { nombre: 'Firulais', especie:
'Perro' };
```

# Resumen

- Los tipos básicos permiten un inicio seguro en TS.
- Interfaces describen contratos de objetos y props.
- Type es más flexible y se usa también para uniones.
- La diferencia principal: interface es extensible, type es más versátil.

# Conclusiones

- TypeScript ayuda a programar con menos errores y más claridad.
- Distinguir cuándo usar type o interface es clave en proyectos grandes.
- Ambos conceptos son fundamentales en React con TypeScript.

# Referencias


- Documentación Oficial de TypeScript: https://www.typescriptlang.org/
- React con TypeScript Cheatsheets: https://react-typescript-cheatsheet.netlify.app/
- Documentación Oficial de React: https://react.dev/

# Unidad 6.3:
# Tipado de Props y Estado

- En React los componentes pueden recibir datos (props) y manejar datos internos (estado).

- En TypeScript debemos definir tipos para ambos, asegurando que se usan correctamente.

- Aprenderemos paso a paso cómo tipar props y estado con claridad.

# ¿Qué son las Props?

- Las props son datos que recibe un componente desde el exterior.
- Son similares a parámetros de una función.
- En TypeScript, debemos especificar qué tipo de datos esperamos recibir.

# Ejemplo de Props en JavaScript

- En JavaScript no existe tipado explícito.
- Esto puede provocar errores en ejecución, sin avisarnos antes.

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

<Saludo nombre='Ana' />
```

# Ejemplo de Props en TypeScript

- Definimos un tipo o interfaz para las props.
- Esto obliga a pasar los datos correctos al componente.

```
interface Props {
  nombre: string;
}

function Saludo({ nombre }: Props) {
  return <h1>Hola {nombre}</h1>;
}
```

# Ventajas de tipar Props

- Aumenta la seguridad del código.
- El editor de texto muestra sugerencias automáticas.
- Se detectan errores antes de ejecutar el programa.
- Facilita el trabajo en equipo.

# Props con múltiples propiedades

- Podemos definir varias propiedades dentro de la interfaz.
- Ejemplo: un mensaje con nombre, edad y activo.

```
interface Props {
  nombre: string;
  edad: number;
  activo: boolean;
}

function Usuario({ nombre, edad, activo }: Props) {
  return <p>{nombre} - {edad} años ({activo ? 'Activo' : 'Inactivo'})</p>;
}
```

# Props opcionales

- A veces un componente no necesita recibir todas las props.
- Con '?' podemos indicar propiedades opcionales.
- Esto evita errores si no se pasa esa prop.

```
interface Props {
  nombre: string;
  edad?: number;
}

function Usuario({ nombre, edad }: Props) {
  return <p>{nombre} - {edad ?? 'Edad no indicada'}</p>;
}
```

# Props con valores por defecto

- Podemos asignar valores por defecto a una prop usando destructuring.
- Así, si no se pasa valor, se usa el valor predeterminado.

```
interface Props {
  mensaje?: string;
}

function Info({ mensaje = 'Sin información' }: Props) {
  return <p>{mensaje}</p>;
}
```

# Props con children

- React permite pasar contenido dentro de un componente usando children.
- En TS se define el tipo con React.ReactNode.

```
interface Props {
  children: React.ReactNode;
}

function Contenedor({ children }: Props) {
  return <div>{children}</div>;
}
```

# Ejercicio 1: Props

Crea un componente Tarjeta que reciba como props:

- título (string)
- contenido (string)

Debe mostrar esta información en una tarjeta simple.

# Ejercicio 1 - Solución

- Definimos una interfaz y aplicamos en el componente.

```
interface Props {
  titulo: string;
  contenido: string;
}

function Tarjeta({ titulo, contenido }: Props) {
  return (
    <div>
      <h2>{titulo}</h2>
      <p>{contenido}</p>
    </div>
  );
}
```

# ¿Qué es el estado (state)?

- El estado es información interna de un componente que puede cambiar.
- Se maneja usando el Hook useState en componentes funcionales.
- En TS debemos definir el tipo del estado.

# Ejemplo de useState en TS

- Si inicializamos con un valor, TS infiere el tipo.
- Podemos también indicar el tipo explícitamente.

```
const [contador, setContador] = useState<number>(0);
setContador(contador + 1);
```

# Estado con strings

- El estado también puede almacenar texto.
- Ejemplo: guardar un nombre escrito en un formulario.

```
const [nombre, setNombre] = useState<string>('');
```

# Estado con objetos

- El estado puede ser un objeto con varias propiedades.
- Debemos definir un tipo o interfaz para ese objeto.

```
interface Usuario {
  nombre: string;
  edad: number;
}

const [usuario, setUsuario] = useState<Usuario>({ nombre: 'Ana', edad:
18 });
```

# Estado con listas

- Muy común en React manejar listas (ej: tareas, productos).
- Se usa un tipo array en useState.

```
interface Tarea {
  id: number;
  titulo: string;
  completada: boolean;
}

const [tareas, setTareas] = useState<Tarea[]>([]);
```

# Actualizar estado en listas

- Para agregar elementos a un estado tipo lista, se crea una nueva lista.
- Nunca se modifica directamente el array original.

```
setTareas([...tareas, { id: 1, titulo: 'Aprender TS', completada:
false }]);
```

# Ejercicio 2: Estado

Crea un componente Contador con estado interno.

- Debe mostrar el número y dos botones (+ y -).
- Tipar el estado correctamente con useState.

# Ejercicio 2 - Solución

- Definimos el estado tipado (number) y usamos botones para actualizar.

```
function Contador(){
  const [valor, setValor] = useState<number>(0);
  return (
    <div>
      <button onClick={() => setValor(valor - 1)}>-</button>
      <span>{valor}</span>
      <button onClick={() => setValor(valor + 1)}>+</button>
    </div>
  );
}
```

# Combinando Props y Estado

- Un componente puede recibir props y además manejar su propio estado.
- Ejemplo: Botón que recibe un mensaje como prop, pero cuenta clics en el estado.

```
interface Props {
  texto: string;
}

function Boton({ texto }: Props) {
  const [clics, setClics] = useState<number>(0);

  return <button onClick={() => setClics(clics + 1)}>{texto} ({clics})</button>;
}
```

# Ejercicio 3: Props + Estado

- Crea un componente ListaTareas que reciba como prop inicial una lista de tareas.
- Debe manejar un estado propio con esa lista y permitir añadir más tareas.

# Ejercicio 3 - Solución

- Definimos una interfaz para Tarea y usamos props + estado.

```
interface Tarea {
  id: number;
  titulo: string;
  completada: boolean;
}

interface Props {
  inicial: Tarea[];
}

function ListaTareas({ inicial }: Props) {
  const [tareas, setTareas] = useState<Tarea[]>(inicial);

  const agregarTarea = () => {
    const nueva = { id: Date.now(), titulo: 'Nueva tarea', completada:
false };
    setTareas([...tareas, nueva]);
  };

  return (
    <div>
      <ul>
        {tareas.map(t => <li key={t.id}>{t.titulo}</li>)}
      </ul>
      <button onClick={agregarTarea}>Agregar</button>
    </div>
  );
}
```

# Eventos y Estado

- El estado se actualiza normalmente con eventos.
- En TS, los eventos tienen un tipo especial que debemos definir.
- Ejemplo: onChange en un input.

```
function Formulario(){
  const [texto, setTexto] = useState<string>('');

  const handleChange = (e: React.ChangeEvent<HTMLInputElement>) =>
{
    setTexto(e.target.value);
  };

  return <input value={texto} onChange={handleChange} />;
}
```

# Errores comunes en Props y Estado

- No definir los tipos → provoca errores en ejecución.
- Usar 'any' excesivamente → se pierde seguridad de TS.
- Modificar directamente un estado de objeto o array → React no detecta cambios.
- Mezclar props y estado para lo mismo → usar uno u otro.

# Buenas prácticas

- Siempre definir tipos claros para props y estado.
- Evitar el uso de 'any'.
- Usar interfaces para datos con varias propiedades.
- Separar responsabilidades: props vienen de fuera, estado es interno.

# Resumen

- Props: datos externos que llegan al componente, se tipan con interfaces o types.
- Estado: datos internos que cambian, se tipan en useState.
- Es clave dominar ambos en React + TypeScript.
- Esto hace que las aplicaciones sean más fiables y mantenibles.

# Conclusiones

- El tipado de props y estado previene errores comunes.
- TypeScript obliga a pensar mejor la estructura de los datos.
- Es una base sólida para proyectos React de cualquier tamaño.

# Referencias


- Documentación Oficial de TypeScript: https://www.typescriptlang.org/
- Documentación Oficial de React: https://react.dev/
- React + TS Cheatsheet: https://react-typescript-cheatsheet.netlify.app/

# Unidad 6.4:
# Tipado de Hooks y funciones asíncronas

- React incluye Hooks para manejar estado y ciclos de vida en componentes funcionales.
- En TypeScript, es importante tipar correctamente los Hooks para prevenir errores.
- Además, muchas aplicaciones modernas usan funciones asíncronas con fetch o axios.
- Aprenderemos a tiparlas junto con React Hooks.

# ¿Qué son los Hooks?

- Los Hooks son funciones especiales que permiten usar características de React en componentes funcionales.
- Evitan la necesidad de trabajar con clases.
- Los más usados: useState, useEffect, useContext, useReducer.
- En TypeScript debemos indicar sus tipos para mayor seguridad.

# useState en detalle

- *useState* gestiona estado dentro de un componente.
- En *TypeScript*, se puede inferir el tipo por el valor inicial.
- Pero también podemos especificar el tipo de forma explícita.

```
const [contador, setContador] = useState<number>(0);
setContador(contador + 1);
```

# Tipado de useState con objetos

- Cuando el estado es un objeto complejo, conviene definir un tipo o interfaz.
- Esto garantiza que el estado siempre tenga la forma definida.

```
interface Usuario {
  nombre: string;
  edad: number;
}

const [usuario, setUsuario] = useState<Usuario>({ nombre: 'Ana', edad: 18 });
```

# Ejercicio 1: useState Tipado

- Crea un componente que guarde en el estado tu color favorito.
- Debe mostrar el color y tener un input para modificarlo.

# Ejercicio 1 - Solución

- Definimos el estado como string y lo actualizamos con un input.

```
function ColorFavorito() {
  const [color, setColor] = useState<string>('Rojo');

  const handleChange = (e: React.ChangeEvent<HTMLInputElement>) =>
{
    setColor(e.target.value);
  };

  return (
    <div>
      <p>Mi color favorito es: {color}</p>
      <input value={color} onChange={handleChange} />
    </div>
  );
}
```

# useEffect en TypeScript

- *useEffect* permite ejecutar efectos secundarios (ej: API calls, timers).
- En TypeScript normalmente no requiere anotaciones extra.
- Usa variables ya tipadas dentro del efecto.

```
useEffect(() => {
  document.title = `Clics: ${contador}`;
}, [contador]);
```

# useContext en TypeScript

- *useContext* permite acceder a datos globales definidos con Context API.
- El contexto debe crearse con un tipo claro para evitar valores undefined.

```
interface Usuario {
  nombre: string;
}

const UsuarioContext = React.createContext<Usuario | null>(null);

function Perfil() {
  const usuario = useContext(UsuarioContext);
  return <p>{usuario?.nombre}</p>;
}
```

# useReducer en TypeScript

- *useReducer* es útil para estados complejos (ej: reducers como en Redux).
- Necesitamos definir claramente el tipo del estado y el tipo de las acciones.

```
type Estado = { contador: number };
type Accion = { type: 'incrementar' } | { type: 'decrementar' };

function reducer(state: Estado, action: Accion): Estado {
  switch(action.type){
    case 'incrementar': return { contador: state.contador + 1 };
    case 'decrementar': return { contador: state.contador - 1 };
    default: return state;
  }
}

const [state, dispatch] = useReducer(reducer, { contador: 0 });
```

# Ejercicio 2: useReducer

- Crea un componente con un contador manejado por useReducer.
- Debe tener un botón + y - para actualizar.

# Ejercicio 2 - Solución

- Definimos tipos de acciones y el reducer con tipado explícito.

```
type Estado = { valor: number };
type Accion = { type: 'inc' } | { type: 'dec' };

function reducer(state: Estado, action: Accion): Estado {
  switch(action.type){
    case 'inc': return { valor: state.valor + 1 };
    case 'dec': return { valor: state.valor - 1 };
    default: return state;
  }
}

function Contador() {
  const [state, dispatch] = useReducer(reducer, { valor: 0 });
  return (
    <div>
      <button onClick={() => dispatch({ type: 'dec'
})}>-</button>
      {state.valor}
      <button onClick={() => dispatch({ type: 'inc'
})}>+</button>
    </div>
  );
```

# Custom Hooks en TypeScript

- Podemos crear nuestros propios Hooks para reutilizar lógica.
- En TS es fundamental definir qué retorna y cuál es el tipo de entrada.
- Ejemplo: un Hook que maneja un contador.

```
function useContador(inicial: number){
  const [valor, setValor] =
useState<number>(inicial);
  const incrementar = () => setValor(v => v + 1);
  return { valor, incrementar };
}
```

# Ejemplo de Custom Hook

- Crearemos un Hook que detecta si la ventana es pequeña (mobile).
- Debemos tipar el estado y los listeners de eventos.

```
function useEsMobile() {
  const [esMobile, setEsMobile] = useState<boolean>(window.innerWidth <
768);

  useEffect(() => {
    const handleResize = () => setEsMobile(window.innerWidth < 768);
    window.addEventListener('resize', handleResize);
    return () => window.removeEventListener('resize', handleResize);
  }, []);

  return esMobile;
}
```

# ¿Qué son funciones asíncronas?

- Una función asíncrona devuelve siempre una Promesa.
- Permite trabajar con await para esperar resultados.
- En TS debemos definir qué tipo devuelve la promesa.

```
async function sumarAsync(a: number, b: number): Promise<number>
{
  return a + b;
}
```

# Promises en TypeScript

- Una Promise<T> significa que la promesa devolverá un valor de tipo T.
- Ejemplo sencillo retornando un string.

```
function obtenerNombre(): Promise<string> {
  return new Promise(res => {
    setTimeout(() => res('Ana'), 1000);
  });
}
```

# fetch con TypeScript

- Cuando consumimos una API con fetch, debemos tipar la respuesta esperada.
- Esto evita errores de propiedades inexistentes.

```
interface Usuario {
  id: number;
  nombre: string;
}

async function obtenerUsuarios(): Promise<Usuario[]> {
  const res = await fetch('https://api.example.com/usuarios');
  return res.json();
}
```

# Ejercicio 3: API Fetch

- Crea un Hook useUsuarios que obtenga una lista de usuarios desde una API.
- Debe manejar estado de loading, error y datos.

# Ejercicio 3 - Solución

- Usamos useState, useEffect y tipado de la respuesta.

```
interface Usuario {
  id: number;
  nombre: string;
}

function useUsuarios() {
  const [usuarios, setUsuarios] = useState<Usuario[]>([]);
  const [cargando, setCargando] = useState<boolean>(true);
  const [error, setError] = useState<string>('');

  useEffect(() => {
    fetch('https://api.example.com/usuarios')
      .then(res => res.json())
      .then(data => setUsuarios(data))
      .catch(() => setError('Error al cargar'))
      .finally(() => setCargando(false));
  }, []);

  return { usuarios, cargando, error };
}
```

# axios con TypeScript

- axios es una librería popular para consumir APIs.
- La versión con TypeScript permite tipar directamente la respuesta.

```
import axios from 'axios';

interface Usuario { id: number; nombre: string; }

async function getUsuarios(): Promise<Usuario[]> {
  const res = await axios.get<Usuario[]>('https://api.example.com/usuarios');
  return res.data;
}
```

# Errores comunes en funciones async

- No tipar correctamente la promesa devuelta.
- Usar any para la respuesta de la API.
- No capturar errores con try/catch o catch().

# Ejercicio 4: axios + Hook

- Crea un Hook usePosts que con axios traiga publicaciones.
- Debe devolver posts, cargando y error.

# Ejercicio 4 - Solución

- Definimos tipos para la respuesta y estados.

```
interface Post { id: number; titulo: string }

function usePosts() {
  const [posts, setPosts] = useState<Post[]>([]);
  const [loading, setLoading] = useState<boolean>(true);
  const [error, setError] = useState<string>('');

  useEffect(() => {
    axios.get<Post[]>('https://api.example.com/posts')
      .then(res => setPosts(res.data))
      .catch(() => setError('Error al cargar'))
      .finally(() => setLoading(false));
  }, []);

  return { posts, loading, error };
}
```

# Resumen Hooks + Async

- useState y useReducer deben siempre usar tipos claros.
- Custom Hooks permiten reutilizar lógica con tipos definidos.
- Funciones async siempre devuelven Promesas tipadas.
- Al consumir APIs, definir interfaces para las respuestas.

# Conclusiones

- Tipar Hooks y funciones async evita errores comunes en React.
- El uso de interfaces para datos de API es obligatorio en proyectos serios.
- Con TypeScript se logra mayor seguridad y escalabilidad.
- Son temas clave para construir aplicaciones robustas.

# Referencias


- Documentación oficial TypeScript: https://www.typescriptlang.org/
- React Docs: https://react.dev/
- React + TypeScript Cheatsheet: https://react-typescript-cheatsheet.netlify.app/
- Axios con TS: https://axios-http.com/docs/intro

# Unidad 6.5:
# Uso de librerías en React con TypeScript

- En proyectos React utilizamos librerías externas para ampliar funcionalidades.
- TypeScript nos ayuda a evitar errores al usar estas librerías.
- Un ejemplo muy común es React Router, utilizado para navegación entre pantallas.

# ¿Qué es React Router?

- Es la librería más popular para manejar rutas en aplicaciones React.
- Permite navegar entre páginas sin recargar toda la aplicación.
- En TypeScript debemos tipar props y parámetros de las rutas.

# Instalación de React Router

- Para instalar React Router con soporte en TS usamos npm o yarn:

```
npm install react-router-dom
npm install --save-dev @types/react-router-dom
```

# Ejemplo de Router básico

- Se utiliza BrowserRouter para envolver nuestra app.
- Dentro definimos Routes con cada Route.
- Cada Route asocia una URL con un componente.

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

function App(){
  return (
    <BrowserRouter>
      <Routes>
        <Route path='/' element={<Home />} />
        <Route path='/contacto' element={<Contacto />} />
      </Routes>
    </BrowserRouter>
  );
}
```

# Tipado de Componentes en Rutas

- Cada componente en una ruta debe estar correctamente tipado con props si las usa.
- Si no recibe props, no es necesario definir tipos adicionales.

```
function Home(){
  return <h1>Página de inicio</h1>;
}

function Contacto(){
  return <h1>Contacto</h1>;
}
```

# Parámetros de Ruta

- Podemos definir rutas dinámicas, ej: /usuarios/:id
- Estos parámetros deben tiparse correctamente en TypeScript.

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

function PerfilUsuario(){
  const { id } = useParams<{ id: string }>();
  return <p>Usuario con ID: {id}</p>;
}
```

# Ejercicio 1: Rutas con parámetros

- Crea una ruta /producto/:id que muestre el ID del producto.
- Tipa correctamente el parámetro en TypeScript.

# Ejercicio 1 - Solución

- Definimos la ruta y usamos useParams con tipado explícito.

```
function Producto(){
  const { id } = useParams<{ id: string }>();
  return <h2>Producto número {id}</h2>;
}

<Routes>
  <Route path='/producto/:id' element={<Producto />} />
</Routes>
```

# useNavigate en TypeScript

- *useNavigate* permite redirigir al usuario a otra ruta.
- En TS se infiere su tipo por defecto, pero podemos tipar argumentos.

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

function Boton(){
  const navigate = useNavigate();
  return <button onClick={() => navigate('/contacto')}>Ir a
contacto</button>;
}
```

# useLocation en TypeScript

- *useLocation* devuelve información de la ubicación actual.
- El resultado puede incluir pathname, search, hash y state.

```
import { useLocation } from 'react-router-dom';

function InfoRuta(){
  const location = useLocation();
  return <pre>{JSON.stringify(location, null, 2)}</pre>;
}
```

# Paso de estado entre rutas

- Podemos pasar estado al navegar con navigate('/ruta', { state: datos }).
- Ese estado debe tiparse correctamente al hacer uso de useLocation.

```
const navigate = useNavigate();
navigate('/perfil', { state: { nombre: 'Ana' } });

// En Perfil.tsx
const location = useLocation();
const state = location.state as { nombre: string };
<p>Bienvenida, {state.nombre}</p>
```

# Rutas anidadas

- React Router permite definir rutas hijas dentro de otras.
- El componente Outlet sirve para representar dónde se renderiza la ruta hija.

```
function Dashboard(){
  return (
    <div>
      <h1>Panel</h1>
      <Outlet />
    </div>
  );
}

<Routes>
  <Route path='/panel' element={<Dashboard />}>
    <Route path='perfil' element={<Perfil />} />
  </Route>
</Routes>
```

# Ejercicio 2: Rutas anidadas

- Crea una ruta /blog que tenga rutas hijas para /blog/articulos y /blog/autores.
- Usa Outlet para renderizar la vista correspondiente.

# Ejercicio 2 - Solución

- Definimos una ruta padre Blog con sus hijos.

```
function Blog(){
  return (
    <div>
      <h1>Blog</h1>
      <Outlet />
    </div>
  );
}

<Routes>
  <Route path='/blog' element={<Blog />}>
    <Route path='articulos' element={<p>Lista de
artículos</p>} />
    <Route path='autores' element={<p>Lista de autores</p>} />
  </Route>
</Routes>
```

# Rutas protegidas en TS

- Podemos crear un componente que verifique si el usuario está autenticado antes de mostrar una ruta.
- Esto es común en aplicaciones con login.

```
interface Props {
  children: JSX.Element;
}

function RutaProtegida({ children }: Props){
  const autenticado = false;
  return autenticado ? children : <Navigate to='/login' />;
}

<Route path='/panel' element={<RutaProtegida><Dashboard /></RutaProtegida>} />
```

# Ejercicio 3: Ruta protegida

- Crea una ruta /perfil que sólo se muestre si autenticado = true.
- Sino, redirige a /login usando Navigate.

# Ejercicio 3 - Solución

- Encapsulamos la lógica de autenticación en RutaProtegida.

```
function RutaProtegida({ children }: { children: JSX.Element }){
  const autenticado = false;
  return autenticado ? children : <Navigate to='/login' />;
}

<Route path='/perfil' element={<RutaProtegida><Perfil /></RutaProtegida>} />
```

# Tipos comunes en React Router

- ***Location***: describe la ubicación actual.
- ***NavigateFunction***: función que devuelve useNavigate.
- ***Params***: parámetros de ruta tipados con genéricos.
- ***Outlet***: sirve para renderizar rutas hijas.

# Ejemplo completo con Router

- Mostramos un proyecto con navegación básica, props tipadas y parámetros.
- Incluye rutas protegidas y públicas.

```
function App(){
  return (
    <BrowserRouter>
      <Routes>
        <Route path='/' element={<Home />} />
        <Route path='/login' element={<Login />} />
        <Route path='/perfil/:id' element={<Perfil />} />
        <Route path='/panel' element={<RutaProtegida><Dashboard /></RutaProtegida>} />
      </Routes>
    </BrowserRouter>
  );
}
```

# Errores comunes

- No tipar correctamente los parámetros de ruta (ej: usar any).
- No usar aserciones en location.state y depender de any.
- Olvidar proporcionar tipos al contexto de autenticación.

# Buenas prácticas en TS + Router

- Siempre tipar los parámetros de useParams.
- Definir interfaces explícitas para datos pasados en navigate().
- Encapsular rutas protegidas en un componente que reciba children.
- Mantener las rutas organizadas en un archivo de configuración.

# Ejercicio 4 Final

- Crea una mini-app con navegación:
- - Ruta '/' → Página de inicio.
- - Ruta '/productos' → Lista de productos.
- - Ruta '/productos/:id' → Detalle de producto.
- Tipa parámetros y datos de productos en TS.

# Ejercicio 4 - Solución

- Definimos interfaces y rutas correctamente tipadas.

```
interface Producto { id: number; nombre: string }

const productos: Producto[] = [
  { id: 1, nombre: 'Libro' },
  { id: 2, nombre: 'Mochila' }
];

function ListaProductos(){
  return (
    <ul>
      {productos.map(p => <li key={p.id}><Link
to={`/productos/${p.id}`}>{p.nombre}</Link></li>)}
    </ul>
  );
}

function DetalleProducto(){
  const { id } = useParams<{ id: string }>();
  return <h2>Detalles del producto #{id}</h2>;
}

<Routes>
  <Route path='/' element={<h1>Inicio</h1>} />
  <Route path='/productos' element={<ListaProductos />} />
  <Route path='/productos/:id' element={<DetalleProducto />} />
</Routes>
```

# Resumen

- React Router gestiona la navegación en aplicaciones React.
- Con TypeScript tipamos parámetros, estado y props en rutas.
- Hooks como useNavigate, useParams y useLocation se benefician del tipado.
- Esto mejora seguridad, reduce errores y hace el código más legible.

# Conclusiones

- Unir React Router con TypeScript es esencial para aplicaciones grandes.
- El tipado correcto evita errores de rutas mal definidas.
- Las rutas protegidas y parámetros tipados hacen la app más robusta.
- Es una habilidad clave para desarrollo profesional en React.

# Referencias


- Documentación oficial React Router: https://reactrouter.com/
- Documentación oficial TypeScript: https://www.typescriptlang.org/
- React + TS Cheatsheet: https://react-typescript-cheatsheet.netlify.app/