Clase 8 - React Recap

IIC2182 - Interfaces y Experiencia de Usuario



PONTIFICIA
UNIVERSIDAD
CATÓLICA
DE CHILE

Primer Semestre 2026

Agenda

Fundamentos

- Componentes funcionales
- Expresiones JSX y `{}`
- Props, callbacks y TypeScript
- Renderizado condicional

Hooks y Patrones

- useState y estado inmutable
- useEffect y anti-patrones
- Listas, .map() y keys
- useContext para estado global

Browser APIs

- localStorage
- sessionStorage
- Cookies

Componentes Funcionales

La base de React moderno

¿Qué es un componente?

Definición

"Un componente es una pieza reutilizable de la interfaz que encapsula su propia estructura, estilo y comportamiento."

Componente Funcional

```
function Greeting({ name }: { name: string }) {
  return <h1 className="text-xl font-bold">
    Hello, {name}!
    </h1>
}
```

Reutilización

```
function App() {
  return (
    <div>
      <Greeting name="Ana" />
      <Greeting name="Pedro" />
    </div>
  )
}
```

En React moderno usamos solo componentes funcionales — los class components son legacy. Son más simples, fáciles de entender y funcionan mejor con hooks.

Expresiones en JSX: { }

Dentro de JSX, las llaves { } permiten insertar cualquier expresión JavaScript

</> Variables y expresiones

```
const name = "Ana";
const age = 25;

return (
  <div>
    <h1>Hello, {name}</h1>
    <p>Age: {age}</p>
    <p>Next year: {age + 1}</p>
    <p>Uppercase: {name.toUpperCase()}</p>
  </div>
);
```

Condicionales y listas

```
return (
  <div>
    {/* Ternario */}
    {isAdmin ? <AdminPanel /> : <Login />}

    {/* && (solo si true) */}
    {error && <p>{error}</p>}

    {/* .map() para listas */}
    {items.map(item => (
      <li key={item.id}>{item.name}</li>
    ))}
  </div>
);
```

Renderizado Condicional

En JSX no puedes usar `if` directamente — usa expresiones

Ternario (if/else)

```
function Status({ isOnline }: { isOnline: boolean })
  return (
    <span>{isOnline ? "Online" : "Offline"}</span>
  );
}
```

Cuando necesitas mostrar algo en ambos casos

&& (solo if)

```
function Alert({ count }: { count: number }) {
  return (
    <div>
      {count > 0 && <span>You have {count} items</sp
    </div>
  );
}
```

Cuando solo muestras algo si la condición es true

Early return (para casos complejos)

```
function Dashboard({ user }: { user: User | null }) {
  if (!user) return <p>Please log in</p>;
  return <h1>Welcome, {user.name}</h1>;
}
```

Ternarios anidados: el problema

Encadenar ternarios dentro de JSX se vuelve ilegible rápidamente

❌ Ternarios encadenados — difícil de leer

```
function Badge({ role }: { role: string }) {
  return (
    <span>
      {role === "admin"
        ? "Admin"
        : role === "editor"
          ? "Editor"
          : role === "viewer"
            ? "Viewer"
            : "Unknown"}
    </span>
  );
}
```

✅ Extraer a una función — claro y testeable

```
function getRoleLabel(role: string): string {
  const labels: Record<string, string> = {
    admin: "Admin",
    editor: "Editor",
    viewer: "Viewer",
  };
  return labels[role] ?? "Unknown";
}

function Badge({ role }: { role: string }) {
  return <span>{getRoleLabel(role)}</span>;
}
```

Si tienes más de un nivel de ternario, extrae la lógica a una función. Es más legible, testeable y fácil de mantener.

Props: datos del padre al hijo

Las props son el mecanismo para pasar datos de un componente padre a un componente hijo

Recibir props

```tsx
interface PersonProps {
  name: string
  age: number
}

// Opción 1: objeto props
function Person(props: PersonProps) {
  return <p>{props.name}, {props.age}</p>
}

// Opción 2: destructuring (preferido)
function Person({ name, age }: PersonProps) {
  return <p>{name}, {age}</p>
}
```

Pasar props

```tsx
// Explícitas
<Person name="Ana" age={25} />

// Con spread operator
const data: PersonProps = { name: "Ana", age: 25 }
<Person {...data} />
```

Las props son read-only — un componente nunca debe modificar sus propias props.

Callbacks como props

Los event handlers en React reciben funciones callback, no invocaciones de funciones

✅ Correcto: pasar referencia

```
<button onClick={() => alert('Hello!')}>
  Greet
</button>

const handleClick = () => alert('Hello!');
<button onClick={handleClick}>
  Greet
</button>
```

❌ Incorrecto: invocación directa

```
// ⚠️ Se ejecuta en cada render!
<button onClick={alert('Hello!')}>
  Greet
</button>

<button onClick={handleClick()}>
  Greet
</button>
```

↑ Callback del hijo al padre

Un padre puede pasar un callback como prop para que el hijo le notifique cambios

```
function Parent() {
  const handleNameChange = (n: string) => console.log(n);
  return <Child onChange={handleNameChange} />
}
```

Ciclo de Vida del Componente

Trigger → Render → Commit

¿Cuándo se re-renderiza un componente?

El ciclo de actualización de React tiene 3 fases

Trigger → Render → Commit

Trigger

1. Render inicial (mount)
2. Cambio de state
3. Re-render del padre

Render

React ejecuta la función del componente y calcula el Virtual DOM (barato)

Commit

Compara Virtual DOM con Real DOM y aplica solo los cambios mínimos (costoso, se batchea)

useState

Estado reactivo en componentes funcionales

¿Qué es el Estado?

Definición

"El estado es un dato que puede cambiar con el tiempo y que, al cambiar, provoca que el componente se vuelva a renderizar con el nuevo valor."

Props vs Estado

Props — datos que vienen del padre (read-only)

State — datos internos del componente (mutables)

Ejemplos de estado

El valor de un input de texto

Si un modal está abierto o cerrado

La lista de items de un carrito

Los datos cargados de una API

Sin estado, un componente siempre renderiza lo mismo. El estado es lo que hace a React reactivo — cambio de estado = nuevo render.

useState: lo básico

El hook `useState` retorna un par: el valor actual y una función para actualizarlo

```
// TypeScript infiere el tipo del valor inicial
const [count, setCount] = useState(0);          // number
const [name, setName] = useState("Ana");  // string
const [name, setName] = useState<string | undefined>();          // string
const [items, setItems] = useState<string[]>([]); // necesita genérico
```

Sobrescribir valor

```
setCount(42);  // count → 42
```

Reemplaza el estado con un valor nuevo

+ Actualizar desde anterior

```
setCount(prev ⇒ prev + 1);
```

Usa el valor previo — nunca falla

`setState` es asíncrono — el valor no cambia inmediatamente después de llamarlo, sino en el siguiente render.

¿Por qué setState es asíncrono?

El nuevo valor no está disponible en la misma ejecución — recién se refleja en el siguiente render

❌ El log muestra el valor anterior

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

  const handleClick = () => {
    setCount(count + 1);
    console.log(count); // ← sigue siendo 0!
  };
}
```

✅ Usar updater function para encadenar

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

  const handleClick = () => {
    setCount(prev => prev + 1); // 0 → 1
    setCount(prev => prev + 1); // 1 → 2
    // count aún es 0 aquí
  };
}
```

React agrupa (batches) las actualizaciones de estado por rendimiento. Si necesitas encadenar cambios, usa la updater function `(prev => ...)`.

Estado inmutable: arrays y objetos

React detecta cambios por referencia, no por contenido. Nunca mutes el estado directamente.

❌ Mutación directa — React NO re-renderiza

```
const [items, setItems] = useState<string[]>([]);
const [user, setUser] = useState({ name: "James", age: 30

// ❌ Arrays
items.push("New Item");
setItems(items);

// ❌ Objects
user.name = "Ana";
setUser(user);
```

✅ Copia nueva — React detecta el cambio

```
// ✅ Arrays (con updater function)
setItems([ ...items, "New Item"]);
setItems(prev => [ ...prev, "New Item"]);
setItems(items.filter(i => i.id =/= id));
setItems(prev => prev.filter(i => i.id =/= id));

// ✅ Objects (con updater function)
setUser(prev => ({  ...prev, name: "Ana" }));
setUser({  ...user, name: "Ana" });
```

Siempre crear un nuevo array/objeto con spread (...) para que React pueda detectar el cambio y re-renderizar.

Estado derivado: no siempre necesitas useState

Si un valor se puede calcular a partir de props o de otro estado, no lo pongas en un useState

❌ Anti-patrón: estado redundante

```
function Person() {
  const [name, setName] = useState("James");
  const [initial, setInitial] = useState("J");

  useEffect(() => {
    setInitial(name[0]);
  }, [name]);
}
```

Doble render innecesario y código más complejo

✅ Preferido: variable derivada

```
function Person() {
  const [name, setName] = useState("James");
  const initial = name[0]; // Se calcula cada render
}
```

Simple, directo y sin renders extra

Si un valor se puede calcular de otro estado o props, simplemente usa una `const` .

useMemo: optimizar cálculos costosos

Cuando el cálculo derivado es costoso, `useMemo` lo memoriza hasta que cambien sus dependencias

Sintaxis

```
const memoizedValue = useMemo(() => {
  return expensiveCalculation(data);
}, [data]); // Solo recalcula cuando 'data' cambia
```

Cuándo usarlo

Filtrar/ordenar listas grandes

Cálculos matemáticos pesados

Transformaciones de datos complejas

Cuándo NO usarlo

Operaciones simples (acceso a propiedades)

Si el cálculo toma menos de 1ms

Si las dependencias cambian en cada render

useEffect

Sincronizar con sistemas externos — y sus anti-patrones

useEffect: para qué sirve

Definición

"useEffect se usa para ejecutar código después del commit, específicamente para sincronizar con sistemas externos (APIs, timers, suscripciones)."

```
useEffect(callback, dependencies);
```

Solo al montar

```
useEffect(() => {
  // Solo una vez
}, []);
```

Al cambiar deps

```
useEffect(() => {
  // Cuando name cambie
}, [name]);
```

Cada render

```
useEffect(() => {
  // ⚠️ Cuidado
}); // Sin array
```

Caso de uso legítimo: fetch de datos en un useEffect con [] al montar.

useEffect: el anti-patrón

`useEffect` es para sincronizar con sistemas externos, NO para derivar estado

❌ Anti-patrón: estado derivado con useEffect

```
function Person() {
  const [name, setName] = useState("James");
  const [initial, setInitial] = useState("J");

  useEffect(() => {
    setInitial(name[0]); // ❌ Redundante
  }, [name]);
}
```

✅ Correcto: variable calculada

```
function Person() {
  const [name, setName] = useState("James");
  const initial = name[0]; // ✅ Directo
}
```

⚠ ¿Por qué es un problema?

1. Código más difícil de leer y mantener
2. Causa 2 renders en vez de 1 (doble commit al DOM)
3. La documentación oficial de React lo desaconseja explícitamente

Fetch con useEffect: el patrón correcto

El caso de uso principal de useEffect: cargar datos de una API al montar

```
interface Recipe {
  id: string
  title: string
}

function RecipeList() {
  const [recipes, setRecipes] = useState<Recipe[]>([]);

  useEffect(() => {
    fetch("https://api.example.com/recipes")
      .then(res => res.json())
      .then((data: Recipe[]) => setRecipes(data));
  }, []);  // ← Array vacío = solo al montar

  return (
    <div>
      {recipes.map(r => <RecipeCard key={r.id} {...r} />)}
    </div>
  );
}
```

Usa `setRecipes(data)` (sobreescribir) en vez de `setRecipes(prev => [...prev, ...data])` para evitar duplicados con hot reload.

Listas y Keys

Renderizar colecciones de forma eficiente

Mapping: de datos a componentes

Usa `.map()` para transformar un array de datos en un array de componentes JSX

El patrón map

```
function RecipeList() {
  const [recipes, setRecipes] = useState<Recipe[]>([
  // ... fetch en useEffect

  return (
    <div>
      {recipes.map(r => (
        <RecipeCard key={r.id} {...r} />
      ))}
    </div>
  );
}
```

¿Por qué map?

Declarativo — describes qué mostrar, no cómo

Dinámico — funciona con cualquier cantidad de datos

Limpio — una línea por transformación

La prop `key`: por qué importa

React usa `key` para identificar qué elementos cambiaron, se agregaron o se eliminaron

❌ Sin key o usando índice

```
// ❌ Sin key — warning en consola
{items.map(item => <Item {...item} />)}

// ⚠️ Índice como key — problemas al reordenar
{items.map((item, i) => (
  <Item key={i} {...item} />
))}
```

✅ Key única y estable

```
// ✅ Usar un ID único del dato
{items.map(item => (
  <Item key={item.id} {...item} />
))}
```

Única entre hermanos

La key debe ser única entre los siblings del mismo map

No usar índice

Si el orden puede cambiar, el índice causa bugs de estado

Estable

Usar IDs del servidor o generados al crear el dato

¿Por qué no usar índice como key?

Cuando el orden cambia (eliminar, reordenar), el índice ya no identifica al mismo elemento

❌ key={index} — estado se mezcla al eliminar

```
// items = ["A", "B", "C"]  → keys: 0, 1, 2
// Eliminas "A" → ["B", "C"] → keys: 0, 1
// React cree que key=0 sigue siendo "A"
// → el input de "A" se queda pegado en "B"!

{items.map((item, i) => (
  <li key={i}>
    {item} <input placeholder={item} />
  </li>
))}
```

✅ key={id} — cada elemento mantiene su estado

```
// items = [{id: 1, text: "A"}, ...]
// Eliminas id=1 → React sabe que id=2 y id=3 no cambiaro
// → inputs mantienen su valor correcto

{items.map(item => (
  <li key={item.id}>
    {item.text} <input placeholder={item.text} />
  </li>
))}
```

Con key={index}, React reutiliza el DOM del índice — si el dato cambió de posición, el estado interno (inputs, checkboxes) se queda en el lugar equivocado.

useContext

Estado compartido sin prop drilling

El problema: prop drilling

Cuando muchos componentes necesitan el mismo dato, pasarlo por props a cada nivel es tedioso

```
App (theme: "dark")
  └─ Layout (theme: "dark")        ← Solo pasa la prop
      └─ Sidebar (theme: "dark")   ← Solo pasa la prop
          └─ Button (theme: "dark") ← El único que la usa
```

Prop Drilling

Cada nivel intermedio recibe y pasa la prop

Difícil de mantener si hay muchos niveles

Los componentes intermedios no usan el dato

useContext

El dato se provee una vez desde arriba

Cualquier hijo lo consume directamente

Sin pasar por intermediarios

useContext: paso a paso

3 pasos: crear contexto, proveer valor, consumir

1. Crear el contexto

```
type Theme = "light" | "dark";
const ThemeContext = createContext<Theme>("light");
```

2. Proveer un valor (en el padre)

```
function App() {
  const [theme, setTheme] = useState<Theme>("dark");
  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      <Layout />
    </ThemeContext.Provider>
  );
}
```

3. Consumir (en cualquier hijo)

```
function Button() {
  const { theme, setTheme } = useContext(ThemeContext);
  return <button className={theme}>Toggle</button>;
}
```

useContext: cuándo usarlo

Buenos casos de uso

- Tema (dark/light mode)
- Datos del usuario autenticado
- Idioma / internacionalización
- Carrito de compras

Cuándo NO usarlo

- × Estado que solo usa 1 componente
- × Props que bajan solo 1–2 niveles
- × Datos que cambian muy frecuentemente

`useContext` re-renderiza todos los consumidores cuando el valor cambia. Para estado complejo y frecuente, considera librerías como Zustand o Redux.

Resumen: hooks y cuándo usarlos

useState Estado local reactivo — el más usado

useEffect Sincronizar con sistemas externos (fetch, timers) — no para estado derivado

useMemo Memorizar cálculos costosos — solo cuando hay un problema de performance real

useContext Estado global compartido — evita prop drilling

Regla de oro: empieza simple. Usa variables derivadas antes de useMemo. Usa props antes de useContext. Evita useEffect si puedes calcular directo.

Almacenamiento en el Browser

localStorage, sessionStorage y Cookies

¿Qué problema resolvemos?

El estado de React vive en memoria — al recargar la página, se pierde todo

⚠ Sin persistencia

El usuario elige tema dark → recarga → vuelve a light

El carrito tiene 5 items → navega a otra página → vacío

Completa paso 3 de un formulario → refresh → vuelve al paso 1

Inicia sesión → cierra pestaña → tiene que loguearse de nuevo

✓ Con almacenamiento del browser

localStorage — preferencias que persisten (tema, idioma)

sessionStorage — datos temporales por pestaña (formularios multi-paso)

Cookies — autenticación que se envía al servidor automáticamente

💡 React maneja el estado en memoria. Para que sobreviva al refresh, necesitamos guardarlo en el browser con estas APIs.

localStorage

Almacena datos en el navegador de forma persistente — sobrevive al cierre del browser

API básica

```
// Guardar
localStorage.setItem("theme", "dark");

// Leer
const theme = localStorage.getItem("theme"); // "dar

// Eliminar
localStorage.removeItem("theme");

// Limpiar todo
localStorage.clear();
```

Guardar objetos

```
// localStorage solo guarda strings
const user = { name: "Ana", age: 25 };

// Guardar: serializar a JSON
localStorage.setItem("user", JSON.stringify(user));

// Leer: parsear JSON
const saved = localStorage.getItem("user");
const parsed = saved ? JSON.parse(saved) : null;
```

Límite de ~5MB por origen. Se comparte entre todas las pestañas del mismo origen. Nunca guardes datos sensibles (tokens, contraseñas) — cualquier script puede leerlo.

sessionStorage

Misma API que localStorage, pero los datos se borran al cerrar la pestaña

API (idéntica a localStorage)

```
// Mismo API, diferente scope
sessionStorage.setItem("step", "3");
const step = sessionStorage.getItem("step");
sessionStorage.removeItem("step");
```

Diferencias clave

Scope — aislado por pestaña (cada tab tiene su propia copia)

Duración — se borra al cerrar la pestaña

Uso típico — formularios multi-paso, estado temporal

Ideal para datos que solo tienen sentido durante la sesión activa — como el progreso en un wizard o filtros temporales.

Cookies

El mecanismo más antiguo — se envían automáticamente al servidor en cada request HTTP

API (más compleja)

```
// Crear (expira en 7 días)
document.cookie = "theme=dark; max-age=604800; path=/

// Leer (retorna un string con todas las cookies)
const cookies = document.cookie; // "theme=dark; lang

// Parsear una cookie específica
function getCookie(name: string): string | null {
  const match = document.cookie
    .split("; ")
    .find(row => row.startsWith(`${name}=`));
  return match ? match.split("=")[1] : null;
}
```

Características

Se envían al server — en cada request HTTP (headers)

Compartidas — entre todas las pestañas del mismo origen

HttpOnly — el servidor puede hacerlas inaccesibles desde JS

Secure — solo se envían por HTTPS

SameSite — protección contra CSRF

Límite — ~4KB por cookie

Comparación: ¿cuál usar?

	localStorage	sessionStorage	Cookies
Persistencia	Permanente	Hasta cerrar tab	Configurable
Tamaño	~5 MB	~5 MB	~4 KB
Se envía al server	No	No	Sí
Acceso JS	Siempre	Siempre	Depende (HttpOnly)

localStorage

Preferencias de usuario, tema, idioma

sessionStorage

Datos temporales, formularios multi-paso

Cookies

Autenticación, tokens de sesión

¿Preguntas?

Clase 8 - React Recap



PONTIFICIA
UNIVERSIDAD
CATÓLICA
DE CHILE

IIC2182 - Primer Semestre 2026