Introducción a React

Sistemas y Tecnologías Web (CC3062) - 2026

Introducción a React

Semestre 01, 2026

Problemática

DOM manual

Imaginemos una tabla de productos con filtro y búsqueda

- El usuario escribe → actualizar el texto del input
- La tabla se filtra → re-renderizar las filas
- Se marca "solo en stock" → volver a filtrar

Con DOM puro: actualizaciones manuales, sincronización difícil, código frágil

El problema en código

```
const input = document.getElementById('busqueda');
const filas = document.querySelectorAll('#tabla tr');

input.addEventListener('input', () => {
  const valor = input.value.toLowerCase();

  filas.forEach(fila => {
```

```
    const visible = fila.textContent.toLowerCase().includes(valor);
    fila.style.display = visible ? '' : 'none';
  });
});
```

Con más filtros el código crece, se duplica y se rompe fácilmente

Cada cambio en el DOM hay que pedirlo manualmente

Componentes reactivos

- Describe cómo debe verse la UI según el estado
- React se encarga de actualizar el DOM automáticamente
- Un solo lugar para manejar cada dato

React hace que el estado de la UI sea predecible.

Definición

- Biblioteca JavaScript para construir interfaces de usuario
- Creada por Meta (Facebook) en 2013
- Basada en componentes reutilizables
- Reactiva: la UI se actualiza cuando cambian los datos
- No es un framework completo — solo la capa de vista

Programación Orientada a Componentes

Componentes

Un componente es una pieza reutilizable de UI

- Tiene su propia lógica

- Tiene su propia apariencia
- Se puede combinar con otros componentes

Pensémoslo como una función: recibe datos → devuelve HTML

Comparación con POO

| Concepto POO | Concepto React | |---|---| | Clase | Componente | | Instancia | Elemento renderizado | | Atributos | Props | | Estado interno | State | | Método | Función dentro del componente |

Árbol de componentes

Toda aplicación React es un árbol

```
App
├── Header
│   └── NavBar
├── ProductList
│   ├── SearchBar
│   └── ProductTable
│       ├── ProductRow
│       └── ProductRow
└── Footer
```

- Los componentes padres pasan datos a los hijos
- Los hijos notifican a los padres mediante funciones

Un componente básico

```
function Saludo() {
  return <h1>Hola, estudiante</h1>;
}
```

- Es una función JavaScript normal
- Devuelve JSX (HTML dentro de JS)
- El nombre siempre empieza con mayúscula

JSX

JSX = JavaScript + XML

```
const nombre = "Ana";

function Saludo() {
  return (
    <div>
      <h1>Hola, {nombre}</h1>
      <p>Bienvenida al curso</p>
    </div>
  );
}
```

- Las expresiones JS van entre `{ }`
- Se compila a `React.createElement(...)` internamente
- Obliga a tener un solo elemento raíz

Props

Definición

Propiedades que el padre pasa al hijo

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

```
// Uso:
<Saludo nombre="María" />
<Saludo nombre="Carlos" />
```

- Son de solo lectura — el hijo no las modifica
- Permiten que el mismo componente tenga distintos datos

Flujo de datos: una sola dirección

```
App  →  ProductList  →  ProductRow
     props               props
```

- Los datos fluyen hacia abajo (parent → child)
- Un hijo nunca modifica las props del padre
- Esto hace el flujo predecible y fácil de depurar

Props con distintos tipos

```
function Tarjeta({ titulo, precio, disponible }) {
  return (
    <div>
      <h2>{titulo}</h2>
      <p>Q{precio}</p>
      {disponible && <span>En stock</span>}
    </div>
  );
}

// Uso:
<Tarjeta
  titulo="Laptop"
  precio={3500}
  disponible={true}
/>
```

Error común: modificar props

```
// ❌ MAL — nunca hagas esto
function Componente({ nombre }) {
  nombre = "otro valor"; // Error: props son de solo lectura
  return <p>{nombre}</p>;
}

// ✅ BIEN — usa state si necesitas modificar datos
function Componente({ nombreInicial }) {
  const [nombre, setNombre] = useState(nombreInicial);
  return <p>{nombre}</p>;
}
```

Eventos

Manejo de eventos

Los eventos en React son similares a los del DOM, con dos diferencias:

- Los nombres usan camelCase: `onClick`, `onChange`, `onSubmit`
- Se pasa una función, no un string

```
// HTML puro
<button onclick="manejarClic()">Clic</button>

// React
<button onClick={manejarClic}>Clic</button>
```

Definir un manejador de evento

```
function Boton() {
  function handleClick() {
```

```
    alert('¡Botón presionado!');
  }

  return (
    <button onClick={handleClick}>
      Presionarme
    </button>
  );
}
```

- El manejador se define dentro del componente
- Se pasa como referencia — `handleClick`, no `handleClick()`

Eventos comunes

| Evento | Cuándo se dispara | |---|---| | `onClick` | Clic en cualquier elemento | | `onChange` | Cambio en un input, select o checkbox | | `onSubmit` | Envío de un formulario | | `onKeyDown` | Tecla presionada | | `onMouseEnter` | El cursor entra al elemento |

El objeto de evento

```
function Input() {
  return (
    <input
      onChange={(e) => {
        console.log(e.target.value); // texto ingresado
        console.log(e.target.type);  // "text", "checkbox"...
      }}
    />
  );
}
```

- `e` (o `event`) es el evento del navegador

- `e.target` es el elemento que disparó el evento
- En formularios: `e.preventDefault()` evita el reload de la página

State

Definición

Datos que cambian con el tiempo

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

function Contador() {
  const [conteo, setConteo] = useState(0);

  return (
    <div>
      <p>Conteo: {conteo}</p>
      <button onClick={() => setConteo(conteo + 1)}>
        Incrementar
      </button>
    </div>
  );
}
```

- `useState` devuelve `[valor, función para cambiar el valor]`
- Cuando el state cambia, React re-renderiza el componente

Props vs State

| | Props | State | |---|---|---| | ¿Quién lo define? | El componente padre | El propio componente | | ¿Se puede cambiar? | No (solo lectura) | Sí, con la función setter | | ¿Para qué sirve? | Recibir datos | Guardar datos que cambian | | Ejemplo | `nombre="Ana"` | `const [activo, setActivo]` |

Cuándo usar state

Hagámonos estas preguntas:

1. ¿El dato cambia con el tiempo o la interacción del usuario?
2. ¿No puede calcularse desde props u otro state?
3. ¿El padre no lo controla?

Si la respuesta es sí a todas → es state

State y eventos juntos

```
function InputControlado() {
  const [texto, setTexto] = useState('');

  return (
    <div>
      <input
        value={texto}
        onChange={(e) => setTexto(e.target.value)}
      />
      <p>Escribiste: {texto}</p>
    </div>
  );
}
```

- El input es controlado — su valor viene del state
- `onChange` actualiza el state → React re-renderiza → el input muestra el valor nuevo
- React siempre sabe qué hay en el input

Tipos de componentes

Funcionales vs Clase

Existen dos formas de escribir componentes en React

```
// Funcional — estándar actual
function Saludo({ nombre }) {
  return <h1>Hola, {nombre}</h1>;
}
```

```
// Clase — forma antigua (pre-2019)
class Saludo extends React.Component {
  render() {
    return <h1>Hola, {this.props.nombre}</h1>;
  }
}
```

- Los funcionales son más simples y son el estándar hoy
- Los de clase siguen existiendo en código legacy — es importante reconocerlos

State en componentes de clase

```
class Contador extends React.Component {
  constructor(props) {
    super(props);
    this.state = { conteo: 0 };
  }

  render() {
    return (
      <div>
        <p>Conteo: {this.state.conteo}</p>
        <button onClick={() => this.setState({ conteo: this.state.conteo + 1 })}>
          Incrementar
        </button>
      </div>
    );
```

```
  }
}
```

- Se usa `this.state` para leer y `this.setState()` para actualizar
- Es más verboso — por eso se prefieren los funcionales con `useState`

Presentacionales vs Contenedor

Una separación útil para organizar el código:

| Tipo | Responsabilidad | |---|---| | Presentacional | Solo renderiza UI — recibe props, devuelve JSX | | Contenedor | Maneja lógica y estado — pasa datos a los presentacionales |

No es una regla estricta, pero ayuda a mantener los componentes pequeños y enfocados

Componente presentacional

```
// Solo se encarga de mostrar — no sabe de dónde vienen los datos
function TarjetaUsuario({ nombre, email, avatarUrl }) {
  return (
    <div className="tarjeta">
      <img src={avatarUrl} alt={nombre} />
      <h2>{nombre}</h2>
      <p>{email}</p>
    </div>
  );
}
```

- No tiene state propio
- Fácil de reutilizar y testear
- Solo depende de sus props

Componente contenedor

```
// Se encarga de obtener y manejar los datos
function PerfilUsuario({ userId }) {
  const [usuario, setUsuario] = useState(null);

  // lógica para cargar el usuario...

  if (!usuario) return <p>Cargando...</p>;

  return <TarjetaUsuario
    nombre={usuario.nombre}
    email={usuario.email}
    avatarUrl={usuario.avatar}
  />;
}
```

- Maneja state y lógica
- Delega la presentación a `TarjetaUsuario`
- Si cambia el diseño, solo se toca el presentacional

Principios clave

- Divide la UI en componentes pequeños y reutilizables
- Los datos fluyen en una sola dirección
- No almacenes state que puede calcularse
- El state vive en el ancestor común de quienes lo necesitan