Testing y Linting

Sistemas y Tecnologías Web (CC3062) - 2026

Testing y Linting en React

Semestre 01, 2026

¿Por qué hacer pruebas?

Un componente funciona hoy.

Un colega modifica una función que usa ese componente.

Dos semanas después, algo falla en producción.

Nadie recuerda que ese componente dependía de esa función.

Las pruebas automáticas resuelven esto

Las pruebas verifican que el código funciona **después de cada cambio**.

Si algo se rompe, el test falla antes de llegar a producción.

No hay que recordar qué probar — las pruebas lo hacen por el equipo.

Tres razones concretas

Confianza: se puede refactorizar sin miedo — si los tests pasan, el comportamiento no cambió.

Regresiones: detectan cuándo un cambio nuevo rompe algo viejo.

Documentación viva: un test bien escrito muestra qué hace el componente y cómo usarlo.

Tipos de pruebas

| Tipo | ¿Qué prueba? | Velocidad | Confianza | | --- | --- | --- | --- | | **Unitaria** | Una función o componente aislado | Muy rápida | Baja cobertura del sistema | | **Integración** | Varios componentes juntos | Rápida | Alta para flujos comunes | | **E2E** | El flujo completo en el navegador | Lenta | Muy alta |

En React, la mayoría de las pruebas son de **integración** — probar componentes con sus interacciones.

Vitest

Vitest es el framework de pruebas del ecosistema Vite.

Compatible con la API de Jest — misma sintaxis, pero más rápido.

```
npm install -D vitest @vitest/ui jsdom
```

Configurar en `vite.config.js`:

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';

export default defineConfig({
  plugins: [react()],
  test: {
    globals: true,
    environment: 'jsdom',
```

```
    setupFiles: './src/test/setup.js',
  },
});
```

Scripts en package.json

```
{
  "scripts": {
    "test":       "vitest",
    "test:ui":    "vitest --ui",
    "test:run":   "vitest run",
    "coverage":   "vitest run --coverage"
  }
}
```

- `vitest` — modo watch (re-ejecuta al guardar).
- `vitest run` — ejecuta una vez (para CI).
- `vitest --ui` — interfaz visual en el navegador.

Sintaxis básica

```
import { describe, it, expect } from 'vitest';
import { sumar, esPar } from './utils';

describe('utilidades matemáticas', () => {

  it('suma dos números', () => {
    expect(sumar(2, 3)).toBe(5);
  });

  it('reconoce números pares', () => {
    expect(esPar(4)).toBe(true);
    expect(esPar(7)).toBe(false);
  });
```

```
  it('lanza error si el argumento no es número', () => {
    expect(() => sumar('a', 1)).toThrow();
  });

});
```

Matchers comunes de expect

| Matcher | ¿Qué verifica? | | --- | --- | | `toBe(valor)` | Igualdad estricta (`===`) | | `toEqual(objeto)` | Igualdad profunda de objetos/arrays | | `toBeTruthy()` / `toBeFalsy()` | Valor verdadero o falso | | `toBeNull()` | Es `null` | | `toBeUndefined()` | Es `undefined` | | `toContain(elemento)` | Array o string contiene el elemento | | `toHaveLength(n)` | Longitud es `n` | | `toThrow()` | La función lanza un error | | `toBeGreaterThan(n)` | Es mayor que `n` |

Testing Library

Testing Library es una familia de librerías para probar componentes como lo haría un usuario.

No prueba implementación interna — prueba comportamiento observable.

```
npm install -D @testing-library/react @testing-library/user-event @testing-library/jest-d
```

Archivo de setup (`src/test/setup.js`):

```
import '@testing-library/jest-dom';
```

La filosofía de Testing Library

> "Cuanto más se parezcan tus pruebas a cómo el software se usa, más confianza dan."

- **No** buscar componentes por su nombre de clase o estructura interna.
- **Sí** buscar elementos por su texto, rol o label — como lo haría el usuario.
- **No** probar el estado interno de React.
- **Sí** probar lo que el usuario ve y puede hacer.

render y screen

```
import { render, screen } from '@testing-library/react';
import { Saludo } from './Saludo';

it('muestra el nombre del usuario', () => {
  render(<Saludo nombre="Ana" />);

  // Buscar por texto visible
  expect(screen.getByText('Hola, Ana')).toBeInTheDocument();
});
```

- `render(...)` monta el componente en un DOM virtual.
- `screen` da acceso a los elementos renderizados.
- `toBeInTheDocument()` viene de `@testing-library/jest-dom`.

Queries de screen

| Query | Cuándo usarla | | --- | --- | | `getByText('texto')` | Elemento con texto exacto visible | | `getByRole('button', { name: 'Enviar' })` | Elemento por rol ARIA y nombre | | `getByLabelText('Email')` | Input asociado a un label | | `getByPlaceholderText('Buscar...')` | Input por placeholder | | `getByTestId('mi-id')` | Último recurso — `data-testid` | | `queryByText(...)` | Como get pero

devuelve `null` si no existe | | `findByText(...)` | Como get pero espera a que aparezca (async) |

Prioridad recomendada: `getByRole` > `getByLabelText` > `getByText` > `getByTestId`

userEvent — simular interacciones

```
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import { Contador } from './Contador';

it('incrementa el conteo al hacer clic', async () => {
  const user = userEvent.setup();

  render(<Contador />);

  const boton = screen.getByRole('button', { name: 'Incrementar' });
  expect(screen.getByText('Conteo: 0')).toBeInTheDocument();

  await user.click(boton);
  expect(screen.getByText('Conteo: 1')).toBeInTheDocument();

  await user.click(boton);
  expect(screen.getByText('Conteo: 2')).toBeInTheDocument();
});
```

`userEvent.setup()` crea una instancia que simula eventos reales del navegador.

Las interacciones son `async` porque simulan el comportamiento real.

Pruebas de componentes

Componente: Saludo

```jsx
// Saludo.jsx
export function Saludo({ nombre, activo }) {
  return (
    <div>
      <h1>Hola, {nombre}</h1>
      {activo ? <span>En línea</span> : <span>Desconectado</span>}
    </div>
  );
}
```

```jsx
// Saludo.test.jsx
import { render, screen } from '@testing-library/react';
import { Saludo } from './Saludo';

describe('Saludo', () => {

  it('muestra el nombre', () => {
    render(<Saludo nombre="Carlos" activo={true} />);
    expect(screen.getByText('Hola, Carlos')).toBeInTheDocument();
  });

  it('muestra "En línea" cuando activo es true', () => {
    render(<Saludo nombre="Ana" activo={true} />);
    expect(screen.getByText('En línea')).toBeInTheDocument();
  });

  it('muestra "Desconectado" cuando activo es false', () => {
    render(<Saludo nombre="Ana" activo={false} />);
    expect(screen.getByText('Desconectado')).toBeInTheDocument();
  });

});
```

Componente: Formulario controlado

```
// Buscador.jsx
export function Buscador({ onBuscar }) {
  const [texto, setTexto] = useState('');
  return (
    <form onSubmit={e => { e.preventDefault(); onBuscar(texto); }}>
      <label>
        Buscar
        <input value={texto} onChange={e => setTexto(e.target.value)} />
      </label>
      <button type="submit">Buscar</button>
    </form>
  );
}
```

```
// Buscador.test.jsx
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import { Buscador } from './Buscador';

it('llama a onBuscar con el texto ingresado', async () => {
  const user = userEvent.setup();
  const mockBuscar = vi.fn(); // función espía

  render(<Buscador onBuscar={mockBuscar} />);

  await user.type(screen.getByLabelText('Buscar'), 'React testing');
  await user.click(screen.getByRole('button', { name: 'Buscar' }));

  expect(mockBuscar).toHaveBeenCalledWith('React testing');
  expect(mockBuscar).toHaveBeenCalledTimes(1);
});
```

Componente: Lista condicional

```
// ListaProductos.jsx
export function ListaProductos({ productos }) {
```

```
  if (productos.length === 0) {
    return <p>No hay productos disponibles</p>;
  }
  return (
    <ul>
      {productos.map(p => <li key={p.id}>{p.nombre}</li>)}
    </ul>
  );
}
```

```
// ListaProductos.test.jsx
describe('ListaProductos', () => {

  it('muestra mensaje cuando la lista está vacía', () => {
    render(<ListaProductos productos={[]} />);
    expect(screen.getByText('No hay productos disponibles')).toBeInTheDocument();
  });

  it('renderiza cada producto', () => {
    const productos = [
      { id: 1, nombre: 'Laptop' },
      { id: 2, nombre: 'Mouse' },
    ];
    render(<ListaProductos productos={productos} />);
    expect(screen.getByText('Laptop')).toBeInTheDocument();
    expect(screen.getByText('Mouse')).toBeInTheDocument();
  });

});
```

Mocking con vi

vi.fn() — función espía

```
const mockFn = vi.fn();

// Llamar la función
mockFn('argumento');

// Verificar que fue llamada
expect(mockFn).toHaveBeenCalled();
expect(mockFn).toHaveBeenCalledWith('argumento');
expect(mockFn).toHaveBeenCalledTimes(1);

// Hacer que retorne un valor
const mockSuma = vi.fn().mockReturnValue(42);
expect(mockSuma()).toBe(42);
```

vi.mock() — mockear módulos

```
// Mockear fetch para no hacer llamadas reales en los tests
vi.mock('./api', () => ({
  obtenerProductos: vi.fn().mockResolvedValue([
    { id: 1, nombre: 'Laptop', precio: 5000 },
    { id: 2, nombre: 'Mouse', precio: 150 },
  ]),
}));

import { render, screen } from '@testing-library/react';
import { CatalogoProductos } from './CatalogoProductos';

it('muestra los productos de la API', async () => {
  render(<CatalogoProductos />);

  // Esperar a que aparezca el contenido async
  expect(await screen.findByText('Laptop')).toBeInTheDocument();
  expect(screen.getByText('Mouse')).toBeInTheDocument();
});
```

Pruebas de componentes async

```
// Componente que carga datos
function CatalogoProductos() {
  const [productos, setProductos] = useState([]);
  const [cargando, setCargando] = useState(true);

  useEffect(() => {
    obtenerProductos()
      .then(setProductos)
      .finally(() => setCargando(false));
  }, []);

  if (cargando) return <p>Cargando...</p>;
  return <ul>{productos.map(p => <li key={p.id}>{p.nombre}</li>)}</ul>;
}

// Test con findBy (espera a que aparezca el elemento)
it('muestra los productos después de cargar', async () => {
  render(<CatalogoProductos />);

  // El componente empieza en estado de carga
  expect(screen.getByText('Cargando...')).toBeInTheDocument();

  // findBy espera (async) hasta que aparezca el elemento
  expect(await screen.findByText('Laptop')).toBeInTheDocument();
  expect(screen.queryByText('Cargando...')).not.toBeInTheDocument();
});
```

Linting con ESLint

El linter analiza el código **sin ejecutarlo** y reporta problemas de estilo, posibles bugs y malas prácticas.

No reemplaza las pruebas — complementa.

```
npm install -D eslint @eslint/js eslint-plugin-react eslint-plugin-react-hooks
```

Configuración ESLint (`eslint.config.js`)

```
import js from '@eslint/js';
import reactPlugin from 'eslint-plugin-react';
import reactHooksPlugin from 'eslint-plugin-react-hooks';

export default [
  js.configs.recommended,
  {
    plugins: {
      react: reactPlugin,
      'react-hooks': reactHooksPlugin,
    },
    rules: {
      // React
      'react/prop-types': 'warn',
      'react/jsx-no-duplicate-props': 'error',
      'react/self-closing-comp': 'warn',

      // React Hooks — las reglas de los hooks
      'react-hooks/rules-of-hooks': 'error',
      'react-hooks/exhaustive-deps': 'warn',

      // JavaScript general
      'no-unused-vars': 'warn',
      'no-console': 'warn',
      'eqeqeq': 'error', // usar === en lugar de ==
    },
  },
];
```

Qué detecta ESLint

```
// ⚠ react-hooks/exhaustive-deps — dependencia faltante
useEffect(() => {
  fetchDatos(userId); // userId se usa pero no está en el array
}, []);
```

```
// ⚠ react-hooks/rules-of-hooks — hook dentro de condicional
if (condicion) {
  const [valor, setValor] = useState(0); // rompe la regla de hooks
}

// ⚠ no-unused-vars — variable declarada pero no usada
const resultado = calcular(); // resultado nunca se usa

// ⚠ eqeqeq — comparación débil
if (valor == null) { ... } // debería ser ===
```

Script de linting

```
{
  "scripts": {
    "lint":       "eslint src/",
    "lint:fix":   "eslint src/ --fix"
  }
}
```

`--fix` corrige automáticamente los problemas que puede resolver (espacios, punto y coma, comillas).

Prettier — formateo automático

ESLint verifica calidad. Prettier formatea el estilo visual del código.

```
npm install -D prettier eslint-config-prettier
```

Archivo `.prettierrc`:

```
{
  "semi": true,
  "singleQuote": true,
```

```
  "tabWidth": 2,
  "trailingComma": "es5",
  "printWidth": 100
}
```

Con `eslint-config-prettier` se desactivan las reglas de ESLint que entran en conflicto con Prettier.

Automatización con Husky y lint-staged

El problema: los scripts de `npm run lint` y `npm run test` se ejecutan solo cuando alguien recuerda hacerlo.

La solución: ejecutarlos automáticamente **antes de cada commit** con Git hooks.

Instalación

```
npm install -D husky lint-staged
npx husky init
```

Esto crea el directorio `.husky/` con un hook `pre-commit`.

Configurar lint-staged

`lint-staged` ejecuta los comandos **solo sobre los archivos modificados** en el commit.

En `package.json`:

```
{
  "lint-staged": {
    "src/**/*.{js,jsx}": [
      "eslint --fix",
```

```
      "prettier --write"
    ],
    "src/**/*.{js,jsx,json,css}": [
      "prettier --write"
    ]
  }
}
```

Hook pre-commit

Archivo `.husky/pre-commit`:

```
#!/bin/sh
npx lint-staged
```

Ahora, cada vez que alguien intenta hacer `git commit`:

1. lint-staged ejecuta ESLint sobre los archivos modificados.
2. Si hay errores, el commit se bloquea.
3. Si todo pasa, el commit se realiza.

Código con errores de lint nunca llega al repositorio.

Agregar tests al pre-commit (opcional)

```
#!/bin/sh
npx lint-staged
npm run test:run
```

Si algún test falla, el commit se bloquea.

Útil en proyectos críticos. En proyectos grandes puede ser lento — evaluarlo.

Flujo completo con automatización

```
git add .
git commit -m "feat: agregar formulario de contacto"
    ↓
.husky/pre-commit ejecuta:
    ├── lint-staged → eslint --fix sobre archivos cambiados
    ├── lint-staged → prettier --write sobre archivos cambiados
    └── vitest run (si está configurado)
    ↓
¿Errores? → commit bloqueado, ver el mensaje de error
¿Todo OK? → commit realizado 
```

Estructura recomendada de archivos

```
src/
├── components/
│   ├── Buscador.jsx
│   ├── Buscador.test.jsx    ← test junto al componente
│   ├── ListaProductos.jsx
│   └── ListaProductos.test.jsx
├── hooks/
│   ├── useFetch.js
│   └── useFetch.test.js
├── utils/
│   ├── formatear.js
│   └── formatear.test.js
└── test/
    └── setup.js             ← configuración global de Testing Library
```

El test junto al archivo que prueba — fácil de encontrar, fácil de mantener.

Principios clave

- Las pruebas verifican comportamiento, no implementación interna
- Usar `getByRole` y `getByLabelText` — las queries más cercanas al usuario

- `userEvent` para simular interacciones reales
- `vi.fn()` para espiar funciones; `vi.mock()` para aislar módulos externos
- `findBy` (async) para elementos que aparecen después de una operación asíncrona
- ESLint detecta bugs y malas prácticas antes de ejecutar el código
- Prettier formatea automáticamente — elimina debates de estilo en el equipo
- Husky + lint-staged garantizan que el código en el repositorio siempre pasa el linter
- Un test que nunca falla no prueba nada — el test debe fallar cuando el comportamiento es incorrecto