Testing y pruebas

React con Jest, (E2E) con Cypress

Julian F. Latorre

September 10, 2024

Contenido

¿Qué es el testing y por qué es importante?

- Evaluación sistemática del software
- Identifica diferencias entre comportamiento real y esperado
- Crucial en React por su naturaleza dinámica y componentizada
- Beneficios:
  - Verifica renderizado correcto de componentes
  - Asegura funcionamiento de la lógica de la aplicación
  - Previene regresiones
  - Mejora la calidad del software
  - Facilita la refactorización

Tipos de pruebas en el desarrollo frontend

1. Pruebas Unitarias
2. Pruebas de Integración
3. Pruebas de Snapshot
4. Pruebas End-to-End (E2E)

Introducción a Jest

- Desarrollado por Facebook
- Diseñado para React, pero versátil
- Características clave:
  - Configuración Zero
  - Pruebas en paralelo
  - Cobertura de código integrada
  - Mocking robusto
  - Snapshots

```
npm install --save-dev jest @testing-library/react
    @testing-library/jest-dom

// jest.config.js
module.exports = {
  testEnvironment: 'jsdom',
  setupFilesAfterEnv: ['<rootDir>/jest.setup.js'],
  moduleNameMapper: {
    '\\.(css|less|sass|scss)$': 'identity-obj-proxy',
  },
};

// jest.setup.js
import '@testing-library/jest-dom';

// package.json
{
  "scripts": {
    "test": "jest"
```

Escribiendo pruebas unitarias con Jest

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

describe('MiComponente', () => {
  test('renderiza correctamente', () => {
    render(<MiComponente />);
    expect(screen.getByText('Texto esperado')).toBeInTheDocument();
  });

  test('maneja clics correctamente', () => {
    const mockFn = jest.fn();
    render(<MiComponente onClick={mockFn} />);
    const button = screen.getByRole('button');
    button.click();
    expect(mockFn).toHaveBeenCalledTimes(1);
  });
```

Mocking de módulos y funciones

```
jest.mock('./miModulo', () => ({
  funcionImportante: jest.fn(() => 'resultado simulado')
}));

test('usa el módulo simulado', () => {
  const { funcionImportante } = require('./miModulo');
  expect(funcionImportante()).toBe('resultado simulado');
});

global.fetch = jest.fn(() =>
  Promise.resolve({
    json: () => Promise.resolve({ data: 'mocked data' }),
  })
);
```

Pruebas asíncronas

```
// Usando async/await
test('datos cargados correctamente', async () => {
  const data = await fetchData();
  expect(data).toBeDefined();
});

// Usando callbacks
test('callback llamado', done => {
  function callback(data) {
    try {
      expect(data).toBe('hola');
      done();
    } catch (error) {
      done(error);
    }
```

Pruebas de snapshot

```
import renderer from 'react-test-renderer';

test('componente renderiza correctamente', () => {
  const tree = renderer
    .create(<MiComponente mensaje="Hola" />)
    .toJSON();
  expect(tree).toMatchSnapshot();
});
```

Introducción a React Testing Library

- Solución ligera para probar componentes React
- Enfoque en comportamiento, no implementación
- Principios clave:
  - Prueba el comportamiento, no la implementación
  - Encuentra elementos por accesibilidad roles, etiquetas, y texto
  - Evita probar detalles de implementación internos

Queries en React Testing Library

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

test('muestra el mensaje correcto', () => {
  render(<MiComponente />);

  // Por texto
  expect(screen.getByText('Bienvenido')).toBeInTheDocument();

  // Por rol
  const button = screen.getByRole('button', { name: 'Enviar' });
  expect(button).toBeInTheDocument();

  // Por label
  const input = screen.getByLabelText('Nombre de usuario');
  expect(input).toBeInTheDocument();

  // Por placeholder
```

Simulando interacciones de usuario

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

test('maneja la entrada de usuario', () => {
  render(<Formulario />);

  // Usando fireEvent
  const input = screen.getByLabelText('Email');
  fireEvent.change(input, { target: { value: 'test@example.com' } })

  // Usando userEvent (más cercano a la interacción real del usuari
  const submitButton = screen.getByRole('button', { name: 'Enviar' }
  userEvent.click(submitButton);

  expect(screen.getByText('Formulario enviado')).toBeInTheDocument()
});
```

Introducción a Cypress

- Herramienta para pruebas E2E
- Características:
  - Recarga automática en tiempo real
  - Depuración sencilla
  - Esperas automáticas y reintentos
  - Control sobre el estado de la aplicación y tráfico de red

Configuración de Cypress en un proyecto React

```
npm install --save-dev cypress

// package.json
{
  "scripts": {
    "cypress:open": "cypress open",
    "cypress:run": "cypress run"
  }
}

npx cypress open
```

Escribiendo pruebas E2E con Cypress

```
describe('Mi aplicación React', () => {
  beforeEach(() => {
    cy.visit('http://localhost:3000')
  })

  it('muestra el título correcto', () => {
    cy.get('h1').should('contain', 'Bienvenido a mi aplicació
  })

  it('puede navegar al perfil de usuario', () => {
    cy.get('nav').contains('Perfil').click()
    cy.url().should('include', '/perfil')
    cy.get('h2').should('contain', 'Perfil de Usuario')
  })
```

Buenas prácticas en pruebas E2E con Cypress

- Usa selectores de datos específicos para pruebas
- Organiza las pruebas en archivos lógicos
- Utiliza `cy.intercept()` para simular respuestas de API
- Aprovecha los comandos personalizados de Cypress
- Ejecuta las pruebas E2E en un entorno de CI/CD

Conclusiones

- El testing es crucial para el desarrollo de aplicaciones React robustas
- Jest proporciona un framework poderoso para pruebas unitarias y de integración
- React Testing Library facilita pruebas centradas en el comportamiento
- Cypress ofrece una solución completa para pruebas E2E
- Una estrategia de testing completa combina todos estos enfoques