

Universidad Tecnológica Nacional

Facultad Regional Rosario



Especialidad: Ing. en Sistemas de Información.

Asignatura: Desarrollo de software

Comisión: 302

Fecha: 04/10/24

TRABAJO PRÁCTICO: Testing Library

Alumno	Legajo
Mondino, Juan Cruz	51922
Giampietro, Gustavo	50671
Dequelli, Gabriel	44073

Índice

Glosario	2
Introducción	3
Objetivo	3
Tema y alcance	3
Proyecto de pequeña escala	4
Definir una propuesta	4
Propósito	4
¿Cómo lo hace?	4
Ventajas de Testing Library	5
¿Cómo se usa?	5
Implementación	6
Conclusión	9
Establecer características de evaluación	9
Investigar las distintas tecnologías	10
Conclusión	11

Glosario

Vite: es una herramienta de desarrollo rápida y moderna para proyectos web que facilita la construcción y el desarrollo de aplicaciones frontend. Fue creada por Evan You, el autor de Vue.js, pero es compatible con múltiples frameworks de JavaScript, como React, Svelte, y Vue, entre otros.

Principales características de Vite:

1. Arranque rápido: Vite usa el navegador para realizar el "hot module replacement" (HMR), lo que permite que los cambios en el código se vean inmediatamente, sin necesidad de recargar toda la página. Esto mejora significativamente la velocidad de desarrollo.
2. Compilación eficiente: Durante el desarrollo, Vite solo convierte (transpila) los módulos cuando se necesitan, en lugar de empaquetar todo el proyecto desde el principio, como hacen otras herramientas tradicionales como Webpack.
3. Soporte para módulos ES: Vite aprovecha las capacidades nativas de los módulos de ECMAScript (ESM) en los navegadores modernos, lo que permite una entrega rápida y eficiente del código.
4. Optimización de producción: Aunque es muy rápido en el entorno de desarrollo, Vite también utiliza herramientas como Rollup para optimizar y empaquetar el proyecto para producción.

Render: es una función que simula el montaje de un componente dentro de un entorno de pruebas. Esto permite realizar pruebas sobre el componente como si estuviera siendo visualizado por el usuario real, aunque el renderizado ocurre en un entorno de simulación sin necesidad de un navegador real.

Herramienta de aserción (o framework de aserción): es una biblioteca o conjunto de funciones que se utilizan en el contexto de pruebas automatizadas para verificar si un resultado esperado coincide con el resultado real producido por el código. Las aserciones son declaraciones que evalúan una condición lógica y determinan si una prueba pasa o falla en función de si la condición es verdadera o falsa. Las herramientas de aserción son esenciales porque permiten a los desarrolladores automatizar la validación del comportamiento esperado del código. Esto facilita la identificación de errores, mejora la confianza en los cambios de código y ayuda a garantizar la calidad del software.

Datos mock: son datos falsos o simulados que se crean para probar el comportamiento de una aplicación o sistema sin depender de fuentes de datos reales, como bases de datos o APIs externas. Estos datos imitan el formato y la estructura de los datos reales, permitiendo

verificar cómo la aplicación manejaría las respuestas y errores de un servicio sin necesidad de conectarse a él.

Document Object Model (DOM): es una interfaz de programación que representa la estructura de un documento HTML o XML como una jerarquía de nodos y objetos, permitiendo a los lenguajes de programación interactuar con él. Cuando un navegador carga una página web, toma el archivo HTML y lo convierte en el DOM, que es una representación en memoria de esa página. Este modelo estructurado permite a los lenguajes como JavaScript manipular el contenido, la estructura y el estilo de la página de manera dinámica.

Introducción

Este trabajo se realiza con el fin de comparar diferentes Tests Suites o Frameworks, para tener la capacidad de elegir el que mejor se ajuste a las necesidades de cada aplicación.

A este trabajo se lo denomina PoC (Proof of Concept), este es un estudio de viabilidad que se lleva adelante previo al desarrollo o implementación de una nueva idea, proyecto o producto. En particular en el área de desarrollo de software esto puede aplicar a un nuevo producto de software o también a la aplicación de nuevas técnica, metodología o tecnologías de desarrollo de software. Una PoC sirve a todos los involucrados y stakeholders para evaluar la conveniencia práctica de realizar un proyecto o implementar nuevas técnicas, metodologías o tecnologías; más allá de las conveniencias teóricas.

La prueba de concepto en una fase temprana permite:

- Identificar los posibles riesgos y obstáculos.
- Determinar la viabilidad del proyecto.
- Proporcionar pruebas evidentes de funcionalidad a las partes interesadas antes de invertir más tiempo, esfuerzo y dinero.

Objetivo

Llevar adelante una investigación y Prueba de Concepto (PoC) de implementación de tecnologías de desarrollo de software, de forma intergrupal e implementarlo en un desarrollo o demostración técnica para evaluar su conveniencia.

Tema y alcance

Los grupos deberán elegir de la lista propuesta o proponer una tecnología para investigar.

Las tecnologías propuestas se agrupan según el tema y los grupos de un mismo tema deberán trabajar de manera conjunta para llevar adelante la PoC.

Etapas:

1. Definir una propuesta.
2. Establecer características de evaluación.
3. Investigar las distintas tecnologías.
4. Implementar un proyecto de pequeña escala para comparar entre las tecnologías que investiga cada grupo.
5. Redactar un informe y conclusión conjunta de la PoC.
6. Realizar una presentación conjunta frente al curso en base a la implementación y al informe.

Proyecto de pequeña escala

El proyecto de pequeña escala en el que se implementará el testeo es una aplicación web de prueba, generada por React con el comando:

```
npx create-react-app my-test-app
```

Por más que sea simple, tiene las funcionalidades más comunes (caja de texto, botones, lista dinámica) de una página web.

Definir una propuesta

Nosotros elegimos Testing Library y compararemos esta tecnología con: Mock Service Worker, Mocha, Vitest y Jest. El objetivo de esta comparación es, como se aclaró anteriormente, conocer el funcionamiento de estas distintas tecnologías de testeo para tener la capacidad de decidir cuál es la mejor en cada caso.

Testing Library es una colección de bibliotecas enfocadas en facilitar la escritura de pruebas unitarias y de integración para aplicaciones web. Se utiliza principalmente con frameworks de JavaScript, como React, Angular, y Vue, pero tiene variantes para diferentes entornos y plataformas.

Propósito

El objetivo principal de Testing Library es ayudar a los desarrolladores a realizar pruebas de componentes o aplicaciones basadas en la interacción del usuario, asegurando que las pruebas sean lo más cercanas posible al uso real de la interfaz. En lugar de centrarse en la implementación interna de los componentes, fomenta una filosofía de prueba orientada al comportamiento y la accesibilidad.

¿Cómo lo hace?

Esta librería está centrada en la interacción con la interfaz de usuario (UI) de manera similar a como lo haría un usuario real. Esto se logra mediante la selección de elementos basados en atributos semánticos y accesibles, como `role`, `label`, `text`, entre otros. Por lo tanto, Testing Library enfatiza el uso de consultas que reflejan cómo un usuario interactuaría con la aplicación.

Ejemplo de funciones comunes:

1. **Renderizado del componente:** Se utiliza ``render`` para montar un componente en un entorno de prueba.
2. **Búsqueda de elementos:** Provee métodos, como, por ejemplo, ``getByText``, ``getByRole`` o ``getByLabelText``, para seleccionar los elementos con los que se desea interactuar.
3. **Interacción del usuario:** Para simular eventos del usuario, se utiliza ``userEvent``, que permite realizar acciones como clics, escritura de texto, etc.
4. **Afirmaciones:** Una vez realizadas las interacciones, se verifica el estado final de la UI mediante afirmaciones (``expect``) con ``Jest`` u otra herramienta de aserción.

Ventajas de Testing Library

1. **Pruebas centradas en el usuario:** Las pruebas están orientadas a cómo el usuario interactúa con la aplicación, lo que mejora la fiabilidad y el valor de las pruebas.
2. **Accesibilidad:** Fomenta el uso de atributos accesibles.
3. **Mantenimiento más sencillo:** Al evitar pruebas basadas en la implementación interna de los componentes, las pruebas suelen ser más resistentes a cambios en el código.
4. **Desacople de la implementación:** No se preocupa por los detalles específicos de cómo se implementa un componente, sino por cómo este es visible y usable desde el punto de vista del usuario.

¿Cómo se usa?

Testing Library se utiliza principalmente dentro de un entorno de pruebas, en combinación con herramientas como Jest (para correr pruebas en JavaScript/TypeScript) o Karma (usado en Angular). Un ejemplo de uso en Angular sería:

JavaScript

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

import { MyComponent } from './my-component';

test('debería mostrar el texto', async () => {

  await render(MyComponent, { componentProperties: { text: 'Hola Mundo' } });

  const element = screen.getByText('Hola Mundo');

  expect(element).toBeInTheDocument();

});
```

En este ejemplo, ``render`` monta el componente, luego ``screen.getByText`` selecciona el texto visible, y finalmente ``expect`` verifica que el texto está en el DOM.

Implementación

Una vez que se tiene la aplicación que se quiere testear, se ejecuta el comando:

```
npm install @testing-library/react
```

Con el cual se instalan los paquetes necesarios para que uno pueda utilizar los métodos o funciones que permitan testear la aplicación y cada elemento que conforman la UI, como por ejemplo:

En el caso del testeo de la App, primeramente se limpia el DOM para que un testeo previo no influya en los que estamos por realizar. Luego se implementan tests tanto como para verificar el funcionamiento de algún elemento, como para corroborar que aparezcan otros tantos.

```
1  import { render, fireEvent, screen, cleanup } from '@testing-library/react';
2  import App from './App';
3
4  // Limpia el DOM después de cada test
5  afterEach(() => {
6    cleanup();
7  });
8
9  //1. Verificar si el texto "Bienvenido a mi App" está presente
10 test('renders Bienvenido a mi App link', () => {
11   render(<App />);
12   const linkElement = screen.getByText(/Bienvenido a mi App/i);
13   expect(linkElement).toBeInTheDocument();
14 });
15
16 //2. Verificar si el Logo de React está presente
17 test('renders React logo', () => {
18   render(<App />);
19   const logo = screen.getByAltText(/logo/i);
20   expect(logo).toBeInTheDocument();
21 });
22
23 //3. Verificar que un enlace abre una nueva ventana
24 test('renders a link to React documentation with target _blank', () => {
25   render(<App />);
26   const linkElement = screen.getByText(/Aprende React/i);
27   expect(linkElement).toHaveAttribute('target', '_blank');
28 });
29
30 //4. Verificar que el enlace tiene el href correcto
31 test('renders a link with the correct href', () => {
32   render(<App />);
33   const linkElement = screen.getByText(/Aprende React/i);
34   expect(linkElement).toHaveAttribute('href', 'https://reactjs.org');
35 });
```

```

37 //5. Verificar que el componente se renderiza correctamente
38 test('renders without crashing', () => {
39   const { container } = render(<App />);
40   expect(container).toBeInTheDocument();
41 });
42
43 //6. Pruebas de Atributos y Clases de CSS
44 test('renders the logo with the correct class', () => {
45   render(<App />);
46   const logo = screen.getByAltText('logo');
47   expect(logo).toHaveClass('App-logo');
48 });
49
50 //7. Verificar que el campo de texto se actualiza cuando el usuario escribe
51 test('updates the text field when the user types', () => {
52   render(<App />);
53
54   const input = screen.getByLabelText(/ingresa un texto/i);
55
56   // Simular que el usuario escribe en el campo de texto
57   fireEvent.change(input, { target: { value: 'Hola Mundo' } });
58
59   // Verificar que el valor del input se ha actualizado
60   expect(input.value).toBe('Hola Mundo');
61 });

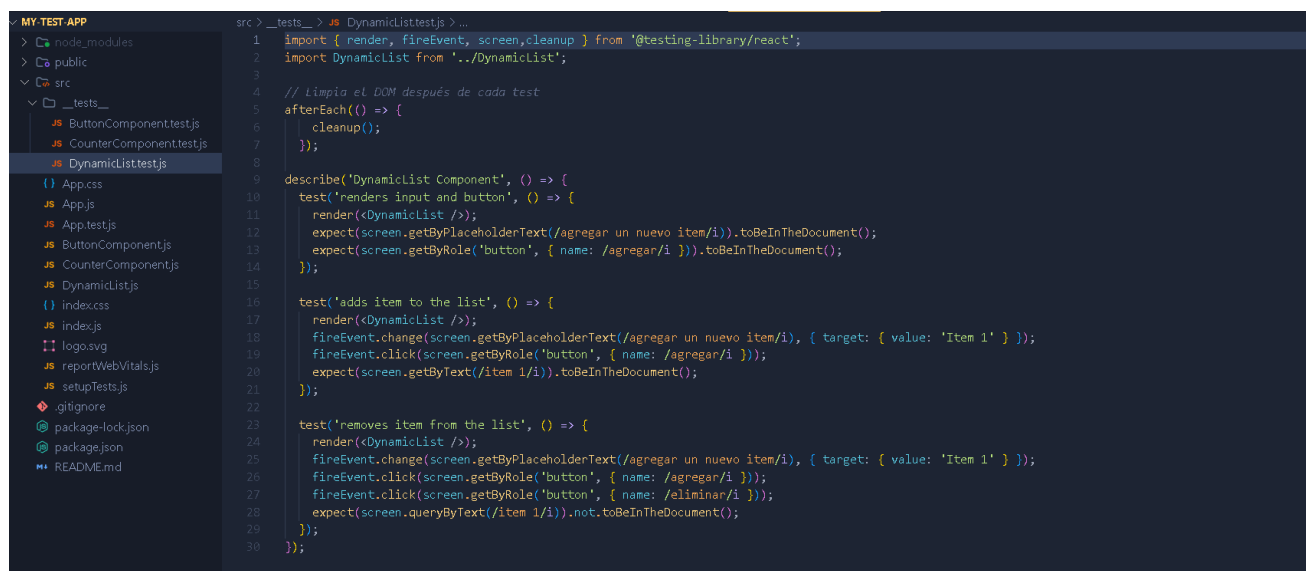
```

```

63 //8. Verificar que el texto enviado se muestra después de hacer clic en "Enviar".
64 test('displays the text sent when "Send" is clicked', () => {
65   render(<App />);
66
67   const input = screen.getByLabelText(/ingresa un texto/i);
68   const submitButton = screen.getByText(/enviar/i);
69
70   // Simular que el usuario escribe y hace clic en enviar
71   fireEvent.change(input, { target: { value: 'Prueba de formulario' } });
72   fireEvent.click(submitButton);
73
74   // Verificar que el texto enviado aparece en la interfaz
75   expect(screen.getByText(/texto enviado: prueba de formulario/i)).toBeInTheDocument();
76 });
77
78
79 //9. Verifica que al enviar el formulario con el campo vacío no ocurra ningún error.
80 test('submitting empty form does not crash', () => {
81   render(<App />);
82   fireEvent.submit(screen.getByRole('button', { name: /enviar/i }));
83   expect(screen.getByText(/ingresa un texto/i)).toBeInTheDocument();
84 });
85
86 //10. Verifica que el botón esté deshabilitado si no hay texto en el input.
87 test('button is disabled when input is empty', () => {
88   render(<App />);
89   const button = screen.getByRole('button', { name: /enviar/i });
90   expect(button).toBeDisabled();
91 });

```


A su vez, si se quiere realizar tests sobre los componentes de la UI, generalmente, se crean unos tests aparte (en otros archivos) para cada componente en específico. Es una forma de trabajo más ordenada e intuitiva.

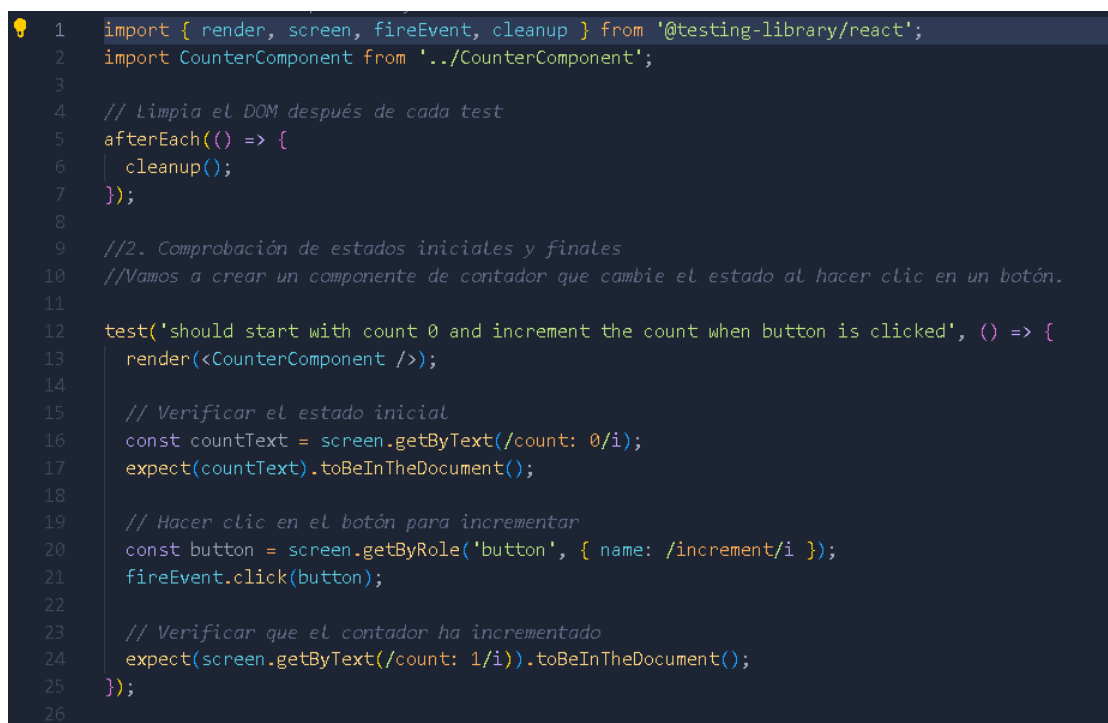


```

src > _tests_ > DynamicListtest.js ...
1 import { render, fireEvent, screen, cleanup } from '@testing-library/react';
2 import DynamicList from '../DynamicList';
3
4 // Limpia el DOM después de cada test
5 afterEach(() => {
6   cleanup();
7 });
8
9 describe('DynamicList Component', () => {
10   test('renders input and button', () => {
11     render(<DynamicList />);
12     expect(screen.getByPlaceholderText(/agregar un nuevo item/i)).toBeInTheDocument();
13     expect(screen.getByRole('button', { name: /agregar/i })).toBeInTheDocument();
14   });
15
16   test('adds item to the list', () => {
17     render(<DynamicList />);
18     fireEvent.change(screen.getByPlaceholderText(/agregar un nuevo item/i), { target: { value: 'Item 1' } });
19     fireEvent.click(screen.getByRole('button', { name: /agregar/i }));
20     expect(screen.getByText(/item 1/i)).toBeInTheDocument();
21   });
22
23   test('removes item from the list', () => {
24     render(<DynamicList />);
25     fireEvent.change(screen.getByPlaceholderText(/agregar un nuevo item/i), { target: { value: 'Item 1' } });
26     fireEvent.click(screen.getByRole('button', { name: /agregar/i }));
27     fireEvent.click(screen.getByRole('button', { name: /eliminar/i }));
28     expect(screen.queryByText(/item 1/i)).not.toBeInTheDocument();
29   });
30 });
  
```

En la imagen de arriba se pueden observar los tests que apuntan a corroborar el funcionamiento del componente DynamicList, por ello es que, como se ve al principio, se importa el archivo del componente correspondiente.

De igual manera sucede con los tests del CounterComponent y del ButtonComponent, que dentro de la carpeta “_tests_” se crearon, también, los archivos .js con la finalidad de testear los componentes restantes:



```

1 import { render, screen, fireEvent, cleanup } from '@testing-library/react';
2 import CounterComponent from '../CounterComponent';
3
4 // Limpia el DOM después de cada test
5 afterEach(() => {
6   cleanup();
7 });
8
9 //2. Comprobación de estados iniciales y finales
10 //Vamos a crear un componente de contador que cambie el estado al hacer clic en un botón.
11
12 test('should start with count 0 and increment the count when button is clicked', () => {
13   render(<CounterComponent />);
14
15   // Verificar el estado inicial
16   const countText = screen.getByText(/count: 0/i);
17   expect(countText).toBeInTheDocument();
18
19   // Hacer clic en el botón para incrementar
20   const button = screen.getByRole('button', { name: /increment/i });
21   fireEvent.click(button);
22
23   // Verificar que el contador ha incrementado
24   expect(screen.getByText(/count: 1/i)).toBeInTheDocument();
25 });
26
  
```

```
1 import { render, fireEvent, screen, cleanup } from '@testing-library/react';
2 import ButtonComponent from '../ButtonComponent';
3
4 // Limpia el DOM después de cada test
5 afterEach(() => {
6   cleanup();
7 });
8
9 test('button changes text when clicked', () => {
10   render(<ButtonComponent />);
11   const button = screen.getByText('Click me');
12   fireEvent.click(button);
13   expect(button).toHaveTextContent('You clicked me!');
14 });
```

También es necesario, antes de realizar los tests, ejecutar el comando:

```
npm install @testing-library/jest-dom
```

Ya que, como se mencionará más adelante, las Testing Librarys utilizan Jest para realizar las aserciones, pero si solo se ejecuta el primer comando, el de instalación de las Testing Librarys, no se tendrá acceso a las funcionalidades adicionales de aserciones extendidas que Jest-DOM proporciona, como: `.toBeInTheDocument()`, `.toHaveTextContent()`, `.toBeVisible()` u otras aserciones más específicas para el DOM. Estas aserciones son útiles para hacer las pruebas más legibles y detalladas. Sin ellas, estarías limitado a las aserciones básicas que ofrece Jest, como `.toBe()` o `.toEqual()`, lo que podría hacer que tus pruebas sean menos expresivas y claras.

Conclusión

Testing Library es una herramienta poderosa y flexible para garantizar que los componentes funcionen correctamente desde la perspectiva del usuario final, lo que resulta en una experiencia de usuario de alta calidad y accesible.

Establecer características de evaluación

Las pruebas que realizamos cubren una amplia gama de escenarios a la hora de la utilización de la aplicación:

- ~ Verificación de texto.
- ~ Verificación de la presencia del logo.
- ~ Verificación de enlaces con `target = "_blank"`.
- ~ Verificación del atributo `href` correcto en un enlace.
- ~ Verificación de que el componente se renderiza sin errores.
- ~ Verificación de clases CSS aplicadas.
- ~ Simulación de escritura en un campo de texto.

- ~ Verificación de que el texto enviado se muestra después de enviar el formulario.
- ~ Verificación de que el envío del formulario vacío no genera errores.
- ~ Verificación de que el botón está deshabilitado cuando el campo de texto está vacío.

Investigar las distintas tecnologías

Testing library, como fue desarrollado anteriormente, son un conjunto de herramientas que permiten crear pruebas para analizar el frontend, más específicamente la interacción del usuario con el UI y prueba de componentes en aplicaciones web. Es utilizado en frameworks como React, Angular y Vue, buscando que los componentes de la UI funcionen correctamente para cada interacción del usuario. Al utilizarlo se busca que las funcionalidades de cada componente se mantengan simples, porque estas series de tests no funcionan correctamente para pruebas más complejas, pero en vez de verlo como una desventaja, mantener las funcionalidades simples, permite que el código tenga mayor flexibilidad y accesibilidad a la hora de realizar cambios.

En cambio, Jest es un framework de pruebas completo para Javascript, que incluye un motor de pruebas, un corredor de pruebas y herramientas de aserción. Es usado tanto para pruebas unitarias como de integración. Es utilizado para testear tanto el frontend como el backend, y puede corroborar funciones simples o complejas. La Testing Library misma usa Jest como herramienta de aserción, aunque Jest en sí cubre pruebas a un nivel más amplio, no está limitado a la UI.

Vitest es un framework de pruebas basado en Vite. Como Vite es extremadamente rápido y modular, Vitest aprovecha estas características para ejecutar pruebas de manera más eficiente, especialmente en proyectos frontend. Al estar diseñado específicamente para el ecosistema de Vite, Vitest integra de forma nativa el soporte de este para características como HMR, lo que acelera tanto el desarrollo como las pruebas. Este framework es similar a Jest, pero tiene la ventaja de estar optimizado para el ecosistema Vite, esto significa que tampoco está limitado al frontend únicamente, también son pruebas para el backend; además, está más orientado a la velocidad en el entorno de desarrollo y es particularmente útil para proyectos que utilizan Vite como herramienta de construcción.

Mocha es un framework de pruebas flexible para JavaScript que no tiene muchas dependencias integradas. Es usado para pruebas tanto del frontend como del backend. Es usado en aplicaciones de Node.js y en proyectos JavaScript generales, permitiendo flexibilidad en el entorno y elección de herramientas complementarias como bibliotecas de aserción. Mocha es un framework de pruebas que te permite estructurar y ejecutar pruebas, pero no incluye herramientas de aserción (para verificar los resultados) ni de simulación (mocks) por defecto. Para agregar estas capacidades, necesitas usar bibliotecas adicionales, como Chai para las aserciones y Sinon.js para los mocks. Por otro lado, Testing Library se centra en pruebas de interfaces de usuario y puede ser usada con Mocha para validar componentes o interacciones. Mocha simplemente ejecuta las pruebas, mientras que Testing Library gestiona las interacciones del usuario, y para verificar los resultados necesitarías integrar alguna herramienta de aserción como Chai.

Mock Service Worker (MSW) es utilizado para simular APIs en pruebas y desarrollo, permitiendo interceptar solicitudes de red y responder con datos mock, sin necesidad de acceder a servicios reales. En resumen, es usado para probar cómo las aplicaciones manejan las respuestas del servidor. A diferencia de la Testing Library, MSW se centra en la simulación de redes y backend.

Conclusión

Cada una de estas tecnologías cumple un rol importante en el ecosistema de pruebas, no se puede decir que una tecnología es mejor que las demás sino que cada una tiene un enfoque y propósito específico, complementándose entre sí, y la elección entre ellas depende del tipo de pruebas y del entorno de desarrollo de cada proyecto.