https://jestjs.io

JEST

Introducción

- Jest es un marco de pruebas (test runner) de JavaScript que se centra en la simplicidad.
- Jest es un marco de prueba diseñado para garantizar la corrección de cualquier base de código JavaScript. Permite escribir pruebas con una API accesible, familiar y rica en funciones que brinda resultados rápidamente.
- Jest está bien documentado, requiere poca configuración y puede ampliarse para adaptarse a las necesidades.
- Jest utiliza un cargador personalizado para las importaciones en sus pruebas, lo que facilita la simulación de cualquier objeto fuera del alcance de la prueba. Se puede usar importaciones simuladas con la rica API de funciones simuladas para espiar llamadas a funciones con sintaxis de prueba legible.
- Jest puede recopilar información de cobertura de código de proyectos completos, incluidos archivos no probados.

Instalación, configuración y ejecución

- Para su instalación mediante npm:
  - npm install --save-dev jest @types/jest
- Por defecto, se utiliza la extensión .test.js para los ficheros de pruebas.
- La configuración se puede definir en el archivo package.json del proyecto, a través de un archivo jest.config.js o jest.config.ts y a través de la opción --config <path/to/file.js|ts|cjs|mjs|json>.

```
"scripts": {
  :
  "test": "react-scripts test --collectCoverage"
},
"jest": {
  "displayName": "Curso de React"
},
```

- Se puede automatizar su ejecución con webpack, Babel, Parcel, …
- Se puede ejecutar directamente desde Jest CLI (npm i jest --global):
  - jest my-test --notify --config=config.json

Suites

- Una “suite” es un nombre que describe al género o sección qué se va a pasar por un conjunto de pruebas unitarias, además es una herramienta que es el núcleo que se necesita para poder tener un orden en el momento de crear las pruebas.
- Las “suites” se crean con la función **describe**, que es una función global y con la cual se inicia toda prueba unitaria, además consta con dos parámetros:

```
describe("Una suite es sólo una función", function() {
//…
});
```

- El primer parámetro es una cadena de caracteres donde se define el nombre descriptivo de la prueba unitaria.
- El segundo parámetro es una función con el código que ejecutará la prueba de código.
- Se pueden anidar “suites” para estructurar conjuntos complejos y facilitar la legibilidad y la búsqueda, basta con crear un describe dentro de otro.

Especificaciones

- Una especificación contiene una o más expectativas (algo que se espera) que ponen a prueba el estado del código. Una expectativa de Jest es una afirmación que debe ser verdadera pero puede ser falsa.
- Una especificación con todas las expectativas verdaderas es una especificación que pasa la prueba, pero con una o más falsas es una especificación que falla.
- Las especificaciones se definen dentro de una Suite llamando a la función global del **test** (o el alias **it**) que, al igual que describe, recibe una cadena y una función, opcionalmente puede recibir un timeout. La cadena es el título de la especificación y la función es la especificación o prueba.
  - test("y así es una especificación", function() {
  - //…
  - });
- **describe** y **test** son funciones: pueden contener cualquier código ejecutable necesario para implementar la prueba y las reglas de alcance de JavaScript se aplican, por lo que las variables declaradas en un describe están disponibles para cualquier bloque test dentro de la suite.

Expectativas

- Las expectativas se construyen con la función expect que obtiene un valor real de una expresión y lo comparan mediante una función Matcher con un el valor esperado (constante).

  expect(valor actual).*matchers*(valor esperado);
- Los matchers (comparadores) son funciones que implementan comparaciones booleanas entre el valor actual y el esperado, ellos son los responsables de informar a Jest si la expectativa se cumple o es falsa.
- Cualquier matcher puede evaluarse como una afirmación negativa mediante el encadenamiento a la llamada expect de un not antes de llamar al matcher.

  expect(valor actual)**.not**.matchers(valor esperado);
- También existe la posibilidad de escribir matchers personalizados para cuando el dominio de un proyecto consiste en afirmaciones específicas que no se incluyen en los ya definidos.

Matchers

- .toEqual(y); verifica si ambos valores son iguales ==.
- .toBe(y); verifica si ambos objetos son idénticos ===.
- .toMatch(pattern); verifica si el valor pertenece al patrón establecido.
- .toMatchObject(object); verifica que un objeto coincide en propiedad y valor con un subconjunto de otro objeto.
- .toBeDefined(); verifica si el valor está definido.
- .toBeUndefined(); verifica si el valor es indefinido.
- .toBeNull(); verifica si el valor es nulo.
- .toBeNaN(); verifica si el valor es NaN.
- .toBeCloseTo (n, d); verifica la precisión matemática (número de decimales).
- .toContain(y); verifica si el valor actual contiene el esperado.

Matchers

- .toBeTruthy(); verifica si el valor es verdadero.
- .toBeFalsy(); verifica si el valor es falso.
- .toBeLessThan(y); verifica si el valor actual es menor que el esperado.
- .toBeLessThanOrEqual (y); verifica si el valor actual es menor o igual que el esperado.
- .toBeGreaterThan(y); verifica si el valor actual es mayor que el esperado.
- .toBeGreaterThanOrEqual (y); verifica si el valor actual es mayor o igual que el esperado.
- .toBeInstanceOf(Class); verifica que es instancia de cierta clase.
- .toThrow(); verifica si una función lanza una excepción.
- .toThrow(error?); verifica si una función lanza una excepción específica.

Código asincrónico

- Es común en JavaScript ejecutar código de forma asincrónica. En estos casos, Jest debe saber cuando ha terminado el código que se está probando, antes de que puede pasar a otra test.
- Jest tiene varias formas de manejar esto.
  - Devoluciones de llamada (Callbacks)
  - Promesas
  - Async / Await

Devoluciones de llamada (Callbacks)

- El patrón asincrónico más común son las devoluciones de llamada: pasar como argumento la función que se desea ejecutar cuando termine el proceso asíncrono.
- El problema es que la prueba se completará tan pronto como se complete la llamada asíncrona, antes de llamar a la devolución de llamada.
- Hay una forma alternativa de test que soluciona este problema: definir como parámetro (por convenio denominada done) con la función culla llamada Jest esperará para dar como finalizada la prueba.

```
test('the data is peanut butter', done => {
  function callback(data) {
    try {
      expect(data).toBe('peanut butter');
      done();
    } catch (error) { done(error); }
  }
  fetchData(callback);
});
```

- Si done() nunca se llama, la prueba fallará (con un error de tiempo de espera), que es lo que se espera que suceda. Si la declaración expect falla, arroja un error y done() no se llama por lo que hay que invocarla en un bloque try.

Promesas

- Si el código usa promesas, existe una forma más sencilla de manejar las pruebas asincrónicas.
- Si la función test devuelve una promesa, Jest esperará a que esa promesa se resuelva. Si se rechaza la promesa, la prueba fallará automáticamente.

```
test('the fetch fails with an error', () => {
  expect.assertions(1); // verificar que invoca la aserción
  return fetchData().catch(e => expect(e).toMatch('error'));
});
```

- También se puede utilizar los marcadores .resolves y .rejects en la declaración de "expect" y Jest esperará a que esa promesa resuelva o rechace.

```
test('the data is peanut butter', () => {
  return expect(fetchData()).resolves.toBe('peanut butter');
});
```


Async / Await

- Como sintaxis alternativa a las promesas se puede usar async/await en las pruebas. Para escribir una prueba asíncrona, hay que usar la palabra clave async delante de la función pasada a test.

```
test('the data is peanut butter', async () => {
  const data = await fetchData();
  expect(data).toBe('peanut butter');
});
test('the fetch fails with an error', async () => {
  expect.assertions(1);
  try {
    await fetchData();
  } catch (e) {
    expect(e).toMatch('error');
  }
});
```

Montaje y desmontaje

- Para montar el escenario de pruebas suele ser necesario definir e inicializar un conjunto de variables. Para evitar la duplicidad de código y mantener las variables inicializadas en un solo lugar además de mantener la modularidad, Jest suministra las funciones globales, a nivel de fichero, o local, dentro de una suite:
  - beforeAll(fn) se ejecuta solo una vez antes empezar a ejecutar las especificaciones del "describe".
  - beforeEach(fn) se ejecuta antes de cada especificación dentro del "describe".
  - afterEach(fn) se ejecuta después de cada especificación dentro del "describe".
  - afterAll(fn) se ejecuta solo una vez después de ejecutar todas las especificaciones del "describe".

```
describe("operaciones aritméticas", function(){
    var cal;
    beforeEach(function(){ calc = new Calculadora(); });
    it("adicion", function(){ expect(calc.suma(4)).toEqual(4); });
    it("multiplicacion", function(){ expect(calc.multiplica(7)).toEqual(0); });
    // …
```

- Otra manera de compartir las variables entre una beforeEach, test y afterEach es a través de la palabra clave this. Cada expectativa beforeEach/test/afterEach tiene el mismo objeto vacío this que se restablece de nuevo a vacío para de la siguiente expectativa beforeEach/test/afterEach.

Control de ejecución

- El marcador **.skip** permite desactivar las suites y las especificaciones, estas se omiten cuando se ejecutan las pruebas y aparecerán como skipped entre los resultados de la prueba.
- De igual forma, las especificaciones pendientes de implementar se pueden marcar con **.todo** (solo texto, sin función) y también aparecerán como skipped.
- El marcador **.only** permite limitar las pruebas que se ejecutan a determinadas suites y especificaciones, el resto se omiten cuando se ejecutan las pruebas y aparecerán como skipped entre los resultados de la prueba. Si una suite .only no tiene ninguna especificación .only se ejecutan todas sus especificaciones, pero si tiene alguna solo ellas se ejecutaran. Una especificación .only se ejecuta siempre aunque su suite este desactivada.

```
describe.skip('Pruebas de expect', () => {
    test.todo('Pendiente...');
    test.only('null', () => {
```

Pruebas parametrizadas

- Las pruebas parametrizadas permiten ejecutar una prueba, suite o especificación, varias veces con diferentes argumentos. Se declaran con el marcador .each asociado a una matriz que proporcionará los argumentos para cada invocación, una fila por cada juego de argumentos, a utilizar en los parámetros de la función de prueba. Se pueden generar títulos de prueba únicos mediante la inyección posicional de parámetros con formato printf:

```
describe.each([
  [1, 1, 2],  [1, 2, 3],  [2, 1, 3],
])('.add(%i, %i)', (a, b, expected) => {
  test(`returns ${expected}`, () => {
    expect(a + b).toBe(expected);
  });
  test(`returned value not be greater than ${expected}`, () => {
    expect(a + b).not.toBeGreaterThan(expected);
  });
  test(`returned value not be less than ${expected}`, () => {
    expect(a + b).not.toBeLessThan(expected);
  });
});
```


Mock

- Jest tiene funciones dobles de prueba llamados mock, también se conocen como "espías", porque permiten espiar el comportamiento de una función que es llamada indirectamente por algún otro código, en lugar de solo probar la salida.
- Se puede crear una función simulada con jest.fn(). Si no se proporciona una implementación, la función simulada devolverá undefined cuando se invoca.

```
const mockCallback = jest.fn(x => 42 + x);
```

- Las funciones mock también se pueden usar para inyectar valores de retorno simulados durante una prueba:

```
const myMock = jest.fn();
myMock.mockReturnValueOnce(false).mockReturnValueOnce(true);
const result = [11, 12, 15].filter(num => myMock(num));
expect(result).toHaveLength(2); // [12, 15]
expect(result[0]).toBe(12);
```

Valores de retorno

- .mockReturnValue(value) Acepta un valor que se devolverá siempre que se llame a la función simulada.
- .mockReturnValueOnce(value) Acepta un valor que se devolverá para una llamada a la función simulada. Se pueden encadenar para que las sucesivas llamadas a la función simulada devuelvan valores diferentes.

```
const myMockFn = jest.fn()
  .mockReturnValue('default')
  .mockReturnValueOnce('first call')
  .mockReturnValueOnce('second call');
```

- Si solo cuenta con .mockReturnValueOnce, el último fijara el valor a devolver en las siguientes llamadas.
- .mockResolvedValue(value),.mockResolvedValueOnce(value), .mockRejectedValue(value) y .mockRejectedValueOnce(value) son versiones con azúcar sintáctica para devolver promesas.

```
test.only('async test', async () => {
   const asyncMock = jest.fn().mockResolvedValue(43);
   let rslt = await asyncMock(); // 43
   expect(rslt).toBe(43);
});
```


Seguimiento de llamadas

- Todas las funciones simuladas tienen la propiedad especial .mock, que es donde se guardan los datos sobre cómo se ha llamado a la función y qué devolvió. La propiedad .mock también rastrea el valor de this para cada llamada.
  - .calls: Una matriz que contiene arrays con los argumentos de llamada pasados en cada invocación a la función simulada.
  - .results: Una matriz que contiene los resultados de cada llamada: un objeto con las propiedades type (return, throw, incomplete) y value.
  - .instances: Una matriz que contiene todas las instancias que se han creado utilizando la función simulada con new (como función constructora).
- Las propiedades de .mock son muy útiles en las pruebas para afirmar cómo se llaman, instancian o devuelven estas funciones.

```
expect(someMockFunction.mock.calls.length).toBe(1);
expect(someMockFunction.mock.calls[1][0]).toBe('second exec, 'first arg');
expect(someMockFunction.mock.results[0].value).toBe('return value');
expect(someMockFunction.mock.instances.length).toBe(2);
expect(someMockFunction.mock.instances[0].name).toEqual('test');
```

Seguimiento de llamadas

- Hay comparadores (matchers) especiales para interactuar con los espías.
  .toHaveBeenCalled() pasará si el espía fue llamado.
  .toHaveBeenCalledTimes(n) pasará si el espía se llama el número de veces especificado.
  .toHaveBeenCalledWith(…) pasará si la lista de argumentos coincide con alguna de las llamadas grabadas a la espía.
  .toHaveBeenNthCalledWith(nthCall, arg1, arg2, ...) pasará si n llamada se invoco con determinados argumentos.
  .toHaveReturned() pasará si el espía devolvió con éxito (es decir, no arrojó un error) al menos una vez.
  .toHaveReturnedTimes(number) para asegurarse de que el espía devuelve correctamente (es decir, no arroje un error) un número exacto de veces.
  .toHaveReturnedWith(value) pasará si el espía devolvió un valor específico.
  .toHaveLastReturnedWith(esperado): pasará si el espía devolvió el valor en la última llamada.
  .toHaveNthReturnedWith(nthCall, value) para probar el valor específico que devolvió el espía para la enésima llamada.

Cambiar comportamiento

- Un mock puede interceptar cualquier función y hacer un seguimiento a las llamadas y todos los argumentos.

```
let c = new Calc();
const spy = jest.spyOn(c, 'sum');
spy.mockReturnValue(4);
expect(c.sum(1,2)).toBe(4);
expect(spy).toBeCalled();
```

- Se pueden interceptar módulos completos:

```
import axios from 'axios';
import UsersRest from './users';
jest.mock('axios');
test('should fetch users', () => {
  const users = [{name: 'Bob'}];
  axios.get.mockResolvedValue({data: users});
  return UsersRest.getAll().then(data => expect(data).toEqual(users));
});
```

Otras opciones

- Opcionalmente, se puede proporcionar un nombre para las funciones simuladas, que se mostrarán en lugar de "jest.fn ()" en la salida de error de la prueba.

```
const spy = jest.spyOn(c, 'sum').mockName('Mock sum');
```

- jest.isMockFunction(fn) determina si la función dada es una función simulada.
- jest.clearAllMocks() borra las propiedades mock.calls y mock.instances de todos los simulacros.
- jest.resetAllMocks() restablece el estado de todos los simulacros.
- jest.restoreAllMocks() restaura todas las imitaciones a su valor original.
- mockFn.mockClear(), mockFn.mockReset() y mockFn.mockRestore() lo realiza a nivel de funciones simuladas individuales.

Cobertura de código

- Con Jest podemos ejecutar pruebas unitarias y crear informes de cobertura de código. Los informes de cobertura de código nos permiten ver que parte del código ha sido o no probada adecuadamente por nuestras pruebas unitarias.
- Para generar el informe de cobertura:

```
"scripts": {
  :
  "test": "react-scripts test --collectCoverage"
},
```

- Una vez que las pruebas se completen, aparecerá una nueva carpeta /coverage en el proyecto. Si se abre el archivo index.html en el navegador se debería ver un informe con el código fuente y los valores de cobertura del código.
- Usando los porcentajes de cobertura del código, podemos establecer la cantidad de código (instrucciones, líneas, caminos, funciones) que debe ser probado. Depende de cada organización determinar la cantidad de código que deben cubrir las pruebas unitarias.

Cobertura de código

- Para establecer un mínimo de 80% de cobertura de código cuando las pruebas unitarias se ejecuten en el proyecto:

```
"jest": {
  "displayName": "Curso de React",
  "coverageThreshold": {
    "global": { "branches": 80, "functions": 80, "lines": 80, "statements": 80 }
  }
```

- Si se especifican rutas al lado global, los datos de cobertura para las rutas coincidentes se restarán de la cobertura general y los umbrales se aplicarán de forma independiente.

```
"coverageThreshold": {
    "global": { "branches": 50,"functions": 50,"lines": 50,"statements": 50 },
    "./src/components/": { "branches": 40,"statements": 40 },
    "./src/reducers/**/*.js": { "statements": 90 },
    "./src/api/critical.js": { "branches": 100,"functions": 100,"statements": 100 }
  }
```


https://testing-library.com/

REACT TESTING LIBRARY

DOM Testing Library

- La biblioteca esencial DOM Testing Library es una solución liviana para probar páginas web consultando e interactuando con nodos DOM (ya sea simulado con JSDOM/ Jest o en el navegador). Las principales utilidades que proporciona implican consultar el DOM en busca de nodos de una manera similar a la forma en que el usuario encuentra elementos en la página. De esta manera, la biblioteca ayuda a garantizar que las pruebas brinden la confianza de que la aplicación funcionará cuando un usuario real la use.
- La biblioteca esencial se ha empaquetado para proporcionar APIs ergonómicas para varios marcos, incluidos React, Angular y Vue. También hay un complemento para usar consultas de biblioteca de pruebas para pruebas de un extremo a otro en Cypress y una implementación para React Native.
- React Testing Library se basa en la DOM Testing Library para trabajar con componentes de React.

React Testing Library

- Para cuando se desea escribir pruebas sostenibles para los componentes de React.
- Como parte de este objetivo, se desea que las pruebas eviten incluir detalles de implementación de sus componentes y, en cambio, se centren en hacer que sus pruebas le brinden la confianza para la que están diseñadas. Como parte de esto, desea que su base de pruebas se pueda mantener a largo plazo para que la refactorización de sus componentes (cambios en la implementación pero no en la funcionalidad) no interrumpan las pruebas y lo ralenticen a usted y a su equipo.
- El React Testing Library es una solución muy ligera para probar componentes React. Proporciona funciones de utilidad en la parte superior de react-dom y react-dom/test-utils, de una manera que alienta mejores prácticas de prueba. Su principio rector principal es: Cuanto más se asemejen sus pruebas a la forma en que se utiliza su software, más confianza le pueden dar.

React Testing Library

- Entonces, en lugar de tratar con instancias de componentes React renderizados, sus pruebas funcionarán con nodos DOM reales. Las utilidades que proporciona esta biblioteca facilitan la consulta del DOM de la misma manera que lo haría el usuario.
- Encontrar elementos de formulario por su texto de etiqueta (como lo haría un usuario), encontrar enlaces y botones de su texto (como lo haría un usuario).
- También expone una forma recomendada de encontrar elementos mediante el data-testid como una "trampilla de escape" para elementos donde el contenido del texto y la etiqueta no tienen sentido o no son prácticos.
- Esta biblioteca fomenta que sus aplicaciones sean más accesibles y le permite acercar sus pruebas al uso de sus componentes de la forma en que lo hará un usuario, lo que permite que sus pruebas le den más confianza de que funcionará cuando un usuario real la use.

Proceso

- El método render procesa un elemento React en el DOM y genera el documento o página.
- Las consultas son los métodos que Testing Library brinda para encontrar elementos en la página.
- Después de seleccionar un elemento, se puede usar la API de eventos para activar eventos y simular las interacciones del usuario con la página, o usar Jest y jest-dom para hacer afirmaciones sobre el elemento.
- A medida que los elementos aparecen y desaparecen en respuesta a las acciones, las API asíncronas como waitFor o las consultas findBy se pueden utilizar para esperar los cambios en el DOM. Para buscar solo elementos secundarios de un elemento específico, se puede usar within. Si es necesario, también hay algunas opciones que se puede configurar, como el tiempo de espera para reintentos y el atributo testID predeterminado.

Método render

- El método render procesa un elemento React en un contenedor al que se adjunta document.body. Como un segundo argumento se puede especificar opciones.

```
render(<Calculadora />)
```

- De forma predeterminada render creará un div, donde se renderizará el componente React, y lo agregará al document.body. Si se proporciona un elemento HTML a través de la opción container, no se agregará automáticamente a document.body (la opción baseElement como el elemento base para las consultas).

```
const table = document.createElement('table')
const { container } = render(<TableBody {...props} />, {
  container: document.body.appendChild(table),
})
```

- Con la opción wrapper se puede pasar un React Component para renderizarlo alrededor del elemento interno. Esto es más útil para crear funciones de renderizado personalizadas reutilizables para proveedores de datos comunes.

Método render

- El método render devuelve un objeto con las propiedades:
  - container: El nodo DOM (HTMLElement) que lo contiene el React Element renderizado (resultado de ReactDOM.render).
  - baseElement: El nodo DOM del contenedor donde se procesa el React Element.
  - ...queries: Las consultas automáticamente vinculadas al baseElement.
- El objeto devuelve los métodos:
  - debug: Muestra en consola el innerHtml del contenedor.
  - rerender: Fuerza un repintado con nuevos valores en las props para asegurarse de que las propiedades se actualicen correctamente.
  - unmount: Hará que se desmonte el componente renderizado para verificar lo que sucede cuando su componente se elimina de la página (fugas de memoria o proceso).
  - asFragment: Devuelve una instantánea del componente para poder compararla posteriormente con una nueva instantánea.
- El método cleanup desmonta árboles de React que se montaron con render.

Configuración/limpieza

- Para cada prueba, usualmente queremos renderizar nuestro árbol de React en un elemento del DOM que esté asociado a document. Esto es importante para poder recibir eventos del DOM. Cuando la prueba termina, queremos "limpiar" y desmontar el árbol de document.

```
import { unmountComponentAtNode } from "react-dom";
let container = null;
beforeEach(() => { // configurar un elemento del DOM como objetivo del renderizado
  container = document.createElement("div");
  document.body.appendChild(container);
});
 test('Creación', () => {
      let comp = render(<Contador init={10} />, {container});
        // …
      comp.unmount(); // abstracción personalizada de ReactDOM.unmountComponentAtNode
});
afterEach(() => { // limpieza al salir
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});
```


Método act()

- Cuando se escriben pruebas de interfaz de usuario, tareas como el renderizado, los eventos de usuario, o la obtención de datos pueden considerarse "unidades" de interacción con la interfaz de usuario. react-dom/test-utils proporciona una utilidad llamada act(), de Arrange-Act-Assert, que asegura que todas las actualizaciones relacionadas con estas "unidades" hayan sido procesadas y aplicadas al DOM antes de que se haga cualquier aserción:

```
act(() => {
  // renderizar componentes
});
// afirmaciones
```

- En la biblioteca React Testing Library las utilidades ya están envueltas con act().

Consultas

- Las consultas son los métodos que Testing Library le brinda para encontrar elementos en la página.
- Los diferentes tipos de consultas (prefijos de las funciones) indican si devolverá un elemento, o arrojará un error (get) o un null (find) si no se encuentra ningún elemento, o si devolverá una Promesa (query) que se rechazará si no se encuentra ningún elemento. Las consultas pueden devolver elementos individuales o múltiples elementos (All). queryAll devuelve una matriz vacía si ningún elemento coincide.
- La prueba debe parecerse a la forma en que los usuarios interactúan con su código (componente, página, etc.) tanto como sea posible. Los nombres de las funciones se completan indicado que busca por el atributo role (ByRole), el campo etiquetado con un texto (ByLabelText), el atributo placeholder (ByPlaceholderText), el contenido de texto (ByText), el valor actual de un elemento de formulario (ByDisplayValue), el atributo alt (ByAltText) y el atributo title (ByTitle).
- También expone una forma recomendada de encontrar elementos mediante el atributo data-testid (ByTestId) para otros casos.

Consultas

- El argumento principal de una consulta puede ser una cadena, una expresión regular o una función (predicado). Aceptan un objeto como argumento final que puede contener:
  - exact (solo aplicable a cadenas): Predeterminado a true; coincide con cadenas completas, distingue entre mayúsculas y minúsculas. Cuando es falso, coincide con subcadenas y no distingue entre mayúsculas y minúsculas.
  - normalizer: Una función opcional que anula el comportamiento de normalización. La normalización consiste en recortar los espacios en blanco al principio y al final del texto y contraer varios caracteres de espacios en blanco adyacentes en un solo espacio.
- Las funciones son accesibles globalmente (requieren que pase el container como primer argumento), como métodos del objeto respuesta de render o como métodos del objeto global screen para consultar todo document.body.
- Además de las consultas proporcionadas por la biblioteca de pruebas, puede utilizar el querySelector del API DOM para consultar elementos a través del elemento contenedor.

https://testing-playground.com/

Métodos asíncronos

- A veces es necesario probar que un elemento está presente y luego desaparece o viceversa. Otras veces es necesario esperar a que el servidor responda.
- Las utilidades de espera asíncrona permiten esperar a que se satisfaga una afirmación antes de continuar, reintentan hasta que la consulta pase o se agote el tiempo de espera. Los métodos asíncronos devuelven una Promesa, por lo que siempre se deben usar con await o .then(done) al llamarlos.
- Se puede utilizar waitFor cuando se necesite esperar un período de tiempo a que las expectativas pasen.

```
await waitFor(() => expect(mockAPI).toHaveBeenCalledTimes(1))
```

- Para esperar a que se eliminen los elementos del DOM, se puede usar waitForElementToBeRemoved.

```
waitForElementToBeRemoved(document.querySelector('div.out')).then(() =>
  // …
)
```

- Las consultas findBy son una combinación de los métodos getBy y waitFor.

Acciones de usuario

- La función fireEvent permite disparar eventos DOM para simular las acciones de usuario, mediante un elemento y un objeto que define el evento:

```
fireEvent(
  getByText(container, 'Submit'),
  new MouseEvent('click', { bubbles: true, cancelable: true, })
)
```

- También dispone de métodos de conveniencia para simplificar el disparar eventos DOM.

```
fireEvent.change(getByLabelText(/username/i), { target: { value: 'a' } })
fireEvent.keyDown(domNode, { key: 'Enter', code: 'Enter' })
fireEvent.click(screen.getByDisplayValue('+'))
```

- user-event es una biblioteca complementaria para Testing Library que proporciona una simulación más avanzada de las interacciones del navegador que el método integrado fireEvent.

Acciones de usuario

- user-event es una biblioteca complementaria para Testing Library que proporciona una simulación más avanzada de las interacciones del navegador que el método integrado fireEvent.
  - click(element, eventInit, options)
  - dblClick(element, eventInit, options)
  - type(element, text, [options])
  - upload(element, file, [{ clickInit, changeInit }])
  - clear(element)
  - selectOptions(element, values)
  - deselectOptions(element, values)
  - tab({shift, focusTrap})
  - hover(element)
  - unhover(element)
  - paste(element, text, eventInit, options)

Acciones de usuario

```
import userEvent from '@testing-library/user-event'

test('type', () => {
  render(<textarea />)
  userEvent.type(screen.getByRole('textbox'), 'Hello,{enter}World!')
  expect(screen.getByRole('textbox')).toHaveValue('Hello,\nWorld!')
  userEvent.clear(screen.getByRole('textbox'))
  expect(screen.getByRole('textbox')).toHaveAttribute('value', '')
})
test('selectOptions', () => {
  render(
    <select multiple>
      <option value="1">A</option>
      <option value="2">B</option>
      <option value="3">C</option>
    </select>
  )
  userEvent.selectOptions(screen.getByRole('listbox'), ['1', '3'])
  expect(screen.getByRole('option', { name: 'A' }).selected).toBe(true)
  expect(screen.getByRole('option', { name: 'B' }).selected).toBe(false)
  expect(screen.getByRole('option', { name: 'C' }).selected).toBe(true)
})
```

jest-dom

- jest-dom es una biblioteca complementaria para Testing Library que proporciona comparadores de elementos DOM personalizados para Jest.
  - toBeDisabled
  - toBeEnabled
  - toBeEmpty
  - toBeEmptyDOMElement
  - toBeInTheDocument
  - toBeInvalid
  - toBeRequired
  - toBeValid
  - toBeVisible
  - toContainElement
  - toContainHTML
  - toHaveAttribute
  - toHaveClass
  - toHaveFocus
  - toHaveFormValues
  - toHaveStyle
  - toHaveTextContent
  - toHaveValue
  - toHaveDisplayValue
  - toBeChecked
  - toBePartiallyChecked
  - toHaveDescription