## 🔰 React Testing with Jest — Phase 1: Foundation (Absolute Basics)

### ✅ Objective

To build a solid understanding of the **fundamentals of testing in React** using **Jest** and **React Testing Library (RTL)**. This phase focuses on the absolute basics, enabling you to test React components confidently.

### 📘 1. What is Testing and Why Do We Test?

#### ✅ Definition:

Testing is the process of writing code to automatically verify that your application behaves as expected.

#### 🎯 Why Do We Test?

* To **ensure correctness** of components and logic.
* To **catch bugs early** in development.
* To **prevent regressions** when refactoring code.
* To help write **better, more modular code**.
* To **automate verification** instead of manual testing.

#### ✅ Analogy:

Think of testing like a safety net for a trapeze artist — it prevents catastrophic falls (bugs) when you make changes.

### 📘 2. Types of Testing

| **Type** | **Scope** | **Examples** | **Tooling** |
| --- | --- | --- | --- |
| **Unit** | Tests a single function/component | Button, utils, pure functions | Jest |
| **Integration** | Tests multiple units together | Form with input + submit | Jest + React Testing Library |
| **E2E** | Tests user flow end-to-end | Login flow, cart to checkout | Cypress, Playwright |

### 📘 3. Setting Up React Testing Library and Jest

#### ✅ Install Dependencies

For most modern React apps (especially if using Vite, CRA, or Next.js), testing support is built-in or easy to add.

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

#### ✅ Configure Jest (if not auto-configured)

In package.json:

"scripts": {

"test": "jest"

}

#### ✅ Optional: Create jest.config.js

module.exports = {

testEnvironment: 'jsdom',

setupFilesAfterEnv: ['@testing-library/jest-dom/extend-expect'],

};

### 📘 4. Your First Test Case: Button Click

#### 🔹 Component (Counter.jsx):

import { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0);

return (

<div>

<p data-testid="count-display">Count: {count}</p>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

}

export default Counter;

#### 🔹 Test File (Counter.test.js):

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

import Counter from './Counter';

test('increments count on click', () => {

render(<Counter />);

const button = screen.getByText('Increment');

fireEvent.click(button);

expect(screen.getByTestId('count-display')).toHaveTextContent('Count: 1');

});

#### ✅ Breakdown:

* render(<Component />) renders the component in a virtual DOM.
* screen.getByText() finds the button.
* fireEvent.click() simulates a user click.
* expect(...).toHaveTextContent() asserts the result.

### 📘 5. Understanding render(), screen, and fireEvent

| **Method** | **Purpose** |
| --- | --- |
| render() | Renders a React component in virtual DOM |
| screen | Global helper to access queries (getBy, etc.) |
| fireEvent() | Simulates DOM events like click, change |

#### 🔹 Example:

render(<MyForm />);

const input = screen.getByPlaceholderText('Enter name');

fireEvent.change(input, { target: { value: 'Likan' } });

expect(input.value).toBe('Likan');

### 📘 6. Role of describe, test, it, expect

#### ✅ Syntax:

describe('Group of related tests', () => {

test('Individual test case', () => {

expect(true).toBe(true);

});

it('Another test case (alias of test)', () => {

expect(1 + 1).toBe(2);

});

});

| **Function** | **Purpose** |
| --- | --- |
| describe | Groups related tests |
| test | Defines a test case |
| it | Alias for test, can be used interchangeably |
| expect | Makes an assertion about some value |

#### 🔹 Chained Matchers with expect()

expect(button).toBeInTheDocument();

expect(input).toHaveAttribute('placeholder', 'Enter name');

### ✅ Summary of Phase 1

* ✔️ Learned **why** we test.
* ✔️ Understood **types** of testing.
* ✔️ Set up **Jest** and **React Testing Library**.
* ✔️ Wrote **first test case** (click button).
* ✔️ Explored **screen**, **render**, **fireEvent**.
* ✔️ Practiced test grouping and assertions.

## ⚙️ React Testing with Jest — Phase 2: Core Concepts (Practice & Patterns)

### ✅ Objective

Learn how to test user interactions, DOM elements, and write clean, maintainable tests using fireEvent and core patterns.

### 📘 1. Testing Input Fields and Form Submission

#### 🔹 Component: LoginForm.jsx

import React, { useState } from 'react';

function LoginForm() {

const [username, setUsername] = useState('');

const [submitted, setSubmitted] = useState(false);

const handleSubmit = (e) => {

e.preventDefault();

setSubmitted(true);

};

return (

<form onSubmit={handleSubmit}>

<input

type="text"

placeholder="Enter username"

value={username}

onChange={(e) => setUsername(e.target.value)}

/>

<button type="submit">Login</button>

{submitted && <p>Submitted as: {username}</p>}

</form>

);

}

export default LoginForm;

#### 🔹 Test: LoginForm.test.js

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

import LoginForm from './LoginForm';

test('should update input and show submitted message', () => {

render(<LoginForm />);

const input = screen.getByPlaceholderText('Enter username');

const button = screen.getByRole('button', { name: /login/i });

fireEvent.change(input, { target: { value: 'Likan' } });

fireEvent.click(button);

expect(screen.getByText('Submitted as: Likan')).toBeInTheDocument();

});

#### ✅ Explanation:

* fireEvent.change() updates the input field.
* fireEvent.click() triggers the form submission.
* We verify with expect(...).toBeInTheDocument().

### 📘 2. Using beforeEach and afterEach

#### 🔹 Purpose:

* beforeEach: Runs **before each test**.
* afterEach: Cleans up or resets things **after each test**.

beforeEach(() => {

render(<LoginForm />);

});

afterEach(() => {

cleanup(); // optional, RTL does it automatically

});

test('input updates correctly', () => {

const input = screen.getByPlaceholderText('Enter username');

fireEvent.change(input, { target: { value: 'Likan' } });

expect(input.value).toBe('Likan');

});

### 📘 3. Mocking Functions (jest.fn())

#### 🔹 Scenario: Mock a login function to see if it was called.

const mockLogin = jest.fn();

function DummyForm() {

return <button onClick={mockLogin}>Mock Login</button>;

}

test('calls mockLogin when button is clicked', () => {

render(<DummyForm />);

fireEvent.click(screen.getByText('Mock Login'));

expect(mockLogin).toHaveBeenCalledTimes(1);

});

#### ✅ Explanation:

* jest.fn() creates a mock function.
* We assert how many times it was called using .toHaveBeenCalledTimes().

### 📘 4. Querying Elements

#### 🔹 Common Queries:

| **Query** | **Description** |
| --- | --- |
| getByRole | Finds by semantic role (button, etc.) |
| getByText | Finds by visible text |
| getByPlaceholderText | Finds by input placeholder |

screen.getByRole('button', { name: /login/i });

screen.getByText('Mock Login');

screen.getByPlaceholderText('Enter username');

#### 🔹 queryBy vs findBy vs getBy

| **Method** | **Behavior** |
| --- | --- |
| getBy | Fails immediately if not found |
| queryBy | Returns null if not found |
| findBy | Async: waits for element |

expect(screen.queryByText('Nonexistent')).toBeNull();

const element = await screen.findByText('Async Loaded');

### 📘 5. Assertions with expect()

expect(screen.getByText('Submitted as: Likan')).toBeInTheDocument();

expect(input).toHaveValue('Likan');

expect(button).toHaveAttribute('type', 'submit');

#### ✅ Explanation:

* expect() checks if a value or element meets a condition.
* Common matchers: .toBe, .toEqual, .toHaveTextContent, .toHaveAttribute, .toBeInTheDocument

### ✅ Summary of Phase 2

* ✔️ Tested **input** and **form submission** using fireEvent
* ✔️ Used **setup/teardown** hooks (beforeEach, afterEach)
* ✔️ Practiced **jest.fn()** for mocking
* ✔️ Learned all major **element queries**
* ✔️ Used **assertions** to validate outcomes

You’re now ready for **Phase 3: Testing Logic & Props**, where we dive into mocking APIs, handling props, and dynamic rendering.

**A bit more on jest function – explore**

## 🚦 Phase 3: Testing Logic & Props

### 🎯 Goal:

To learn how to test logic inside components, handle props, perform API mocking, and handle conditional rendering.

### 📘 1. Testing Props and Dynamic Rendering

#### ✅ Component: Greeting.jsx

function Greeting({ name }) {

return <h1>Hello, {name}!</h1>;

}

export default Greeting;

#### ✅ Test: Greeting.test.js

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

import Greeting from './Greeting';

test('renders dynamic name from props', () => {

render(<Greeting name="Likan" />);

const heading = screen.getByText('Hello, Likan!');

expect(heading).toBeInTheDocument();

});

#### 🧠 Explanation:

* Props are passed into the component.
* We assert the output based on prop value.
* Verifies dynamic rendering is working.

### 📘 2. Conditional Rendering (Show/Hide Logic)

#### ✅ Component: Welcome.jsx

function Welcome({ isLoggedIn }) {

return (

<div>

{isLoggedIn ? <p>Welcome back!</p> : <p>Please log in.</p>}

</div>

);

}

export default Welcome;

#### ✅ Test: Welcome.test.js

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

import Welcome from './Welcome';

test('renders login message when not logged in', () => {

render(<Welcome isLoggedIn={false} />);

expect(screen.getByText('Please log in.')).toBeInTheDocument();

});

test('renders welcome message when logged in', () => {

render(<Welcome isLoggedIn={true} />);

expect(screen.getByText('Welcome back!')).toBeInTheDocument();

});

#### 🧠 Explanation:

* Test covers both branches of conditional logic.
* We pass different props and assert the UI output accordingly.

### 📘 3. Testing API Calls (Mocking fetch/axios)

#### ✅ Component: User.jsx

import { useEffect, useState } from 'react';

function User() {

const [name, setName] = useState('');

useEffect(() => {

fetch('/api/user')

.then(res => res.json())

.then(data => setName(data.name));

}, []);

return <div>{name ? `Hello, ${name}` : 'Loading...'}</div>;

}

export default User;

#### ✅ Test: User.test.js

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

import User from './User';

beforeEach(() => {

global.fetch = jest.fn(() =>

Promise.resolve({

json: () => Promise.resolve({ name: 'Likan' })

})

);

});

afterEach(() => {

jest.resetAllMocks();

});

test('renders fetched user data', async () => {

render(<User />);

const greeting = await screen.findByText('Hello, Likan');

expect(greeting).toBeInTheDocument();

});

#### 🧠 Explanation:

* jest.fn() mocks fetch.
* We return a resolved promise with the expected API data.
* findByText is used for async UI updates.

### 📘 4. Mocking Child Components

#### ✅ Component: Parent.jsx

import Child from './Child';

function Parent() {

return (

<div>

<h1>Parent</h1>

<Child />

</div>

);

}

export default Parent;

#### ✅ Test: Parent.test.js

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

import Parent from './Parent';

// This mocks the entire Child component

jest.mock('./Child', () => () => <div>Mocked Child</div>);

test('renders parent and mocked child', () => {

render(<Parent />); // Render Parent component (Child is now mocked)

// Check that the Parent text appears in DOM

expect(screen.getByText('Parent')).toBeInTheDocument();

// Check that the mocked version of Child also appears

expect(screen.getByText('Mocked Child')).toBeInTheDocument();

});

#### 🧠 Explanation (Line by Line):

* jest.mock('./Child', ...) tells Jest to replace the real Child with a fake one.
* The fake Child component is just a div with text "Mocked Child".
* This helps us test the Parent component in isolation (without needing real Child behavior).
* render(<Parent />) runs the test by rendering the Parent component.
* screen.getByText() is used to check if the Parent heading and mocked Child are rendered.

**Child taking props –**

**import Child from './Child';**

**function Parent() {**

**return (**

**<div>**

**<h1>Parent</h1>**

**<Child message="Hello from Parent" />**

**</div>**

**);**

**}**

**export default Parent;  
  
// Parent.test.js**

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

**import Parent from './Parent';**

**// Mock data object**

**const mockData = {**

**message: 'Mock data message',**

**};**

**// Mock the Child component with destructured mockData (ignoring actual passed props)**

**jest.mock('./Child', () => ({ message }) => {**

**return <div>Mocked Child: {message}</div>;**

**});**

**test('renders parent and mocked child with mock data', () => {**

**render(<Parent />);**

**expect(screen.getByText('Parent')).toBeInTheDocument();**

**// The mocked child renders with mockData.message regardless of what Parent passes**

**expect(screen.getByText('Mocked Child: Mock data message')).toBeInTheDocument();**

**});**

**What this does:**

**You create a mockData object inside your test.**

**The mock function’s parameter destructures from mockData by default.**

**So even if Parent passes a different message prop, the mock will always use mockData.message.**

**This completely ignores the actual props passed from Parent.**

# What is jest.mock()?

* jest.mock() is a **Jest function** used to **mock (replace) a module dependency** during testing.
* It tells Jest: “Hey, whenever this module is imported in the code being tested, replace it with something else (a mock) I provide or auto-generate.”
* This lets you isolate your tests by **replacing complex dependencies** (like components, API calls, or utilities) with simple mocks.

# Basic syntax

js

CopyEdit

jest.mock(moduleName, factoryFunction?, options?);

* **moduleName (string)** — The path or name of the module you want to mock, e.g. './Child'.
* **factoryFunction (function, optional)** — A function returning the mock implementation of the module.
* **options (object, optional)** — Additional options (rarely used).

# How does it work?

* When you import the module anywhere in your test (or code under test), **Jest replaces that import with your mock.**
* If you **don’t provide** a factory function, Jest automatically mocks the module by replacing all exports with Jest mock functions (jest.fn()).
* If you **do provide** a factory function, Jest uses **the return value of that function as the module**.

### 🧪 6. Debugging Tests

#### ✅ Tools: debug(), logTestingPlaygroundURL()

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

import MyComponent from './MyComponent';

test('debugging UI output', () => {

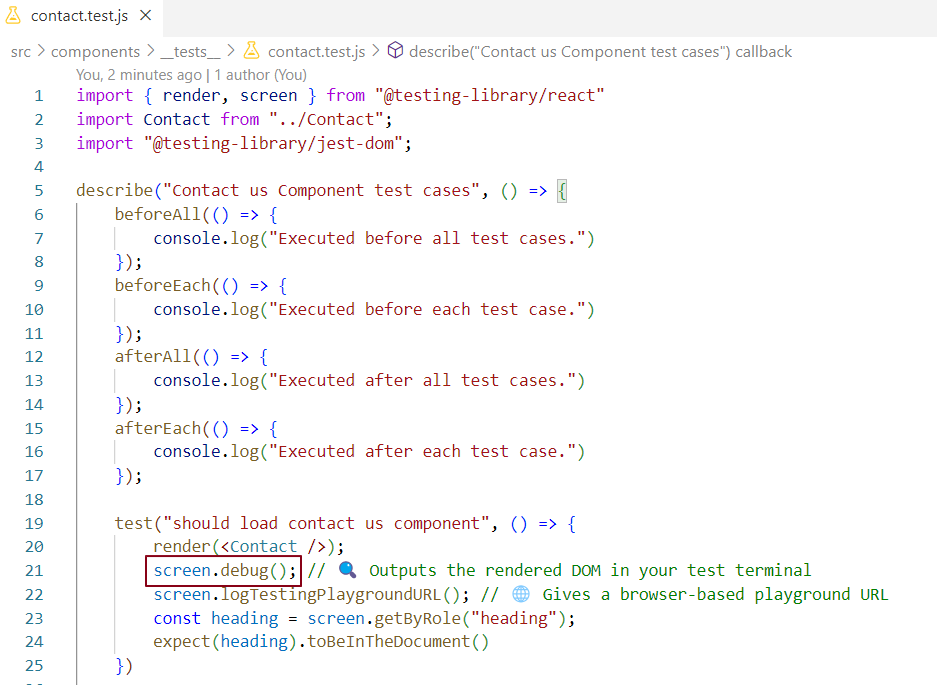
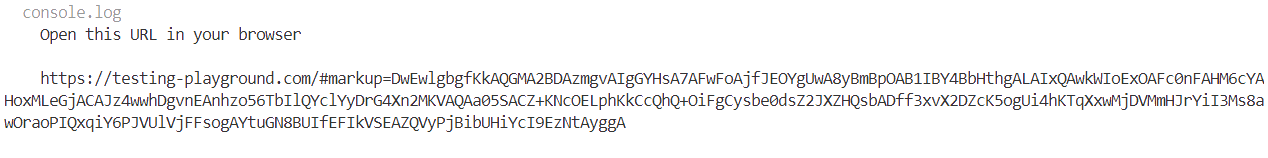
render(<MyComponent />);

screen.debug(); // Prints the DOM in the test output

screen.logTestingPlaygroundURL(); // Logs a playground link to inspect element queries

});

#### 🧠 Explanation:

* screen.debug() outputs the current DOM structure in the terminal.
* 
* 
* logTestingPlaygroundURL() gives a URL to open in your browser for query suggestions.
* 

#### ✅ Flaky Test Handling:

* Flaky tests are tests that fail sometimes due to async behavior or timing issues.
* Solutions:
  + Use findBy queries for async content.
  + Add await waitFor() for delayed UI updates.
  + Avoid relying on setTimeout inside tests.
  + Always clean up mocks using afterEach().

# 🔍 jest.fn() — Complete Guide

jest.fn() is used to create a **mock function** in Jest. It's a powerful tool for testing how functions are **called, what arguments they receive, what they return**, and how many times they're called.

# What is jest.fn()?

jest.fn() creates a **mock function** — a fake function used in tests that:

* **Records how it was called** (number of calls, arguments)
* **Tracks what it returns**
* Can be customized to **return specific values** or **simulate different behaviors**
* Can **replace real functions** to isolate tests and avoid side effects

# Why use jest.fn()?

* **Spy on functions** to see how and when they’re called
* **Control function behavior** without relying on real implementation
* **Mock external dependencies or callbacks** to isolate units of code
* **Test event handlers, props, custom hooks, utilities, etc.**

# Common Use Cases of jest.fn()

### 1. ****Spy on function calls****

Check if a function was called, how many times, and with what arguments.

js

CopyEdit

const mockFn = jest.fn();

mockFn('apple', 42);

mockFn('banana');

expect(mockFn).toHaveBeenCalled(); // Was it called at least once?

expect(mockFn).toHaveBeenCalledTimes(2); // Called exactly twice

expect(mockFn).toHaveBeenCalledWith('apple', 42); // Called with specific args

expect(mockFn).toHaveBeenNthCalledWith(2, 'banana'); // 2nd call with 'banana'

**Example Component — Calls onClick multiple times**

jsx

CopyEdit

import React from 'react';

function MultiCallButton({ onClick }) {

return (

<div>

<button onClick={() => onClick('apple', 42)}>Call 1</button>

<button onClick={() => onClick('banana')}>Call 2</button>

</div>

);

}

export default MultiCallButton;

**Jest test verifying multiple calls and arguments**

js

CopyEdit

import React from 'react';

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

import MultiCallButton from './MultiCallButton';

test('calls onClick multiple times with correct arguments', () => {

const onClickMock = jest.fn();

render(<MultiCallButton onClick={onClickMock} />);

fireEvent.click(screen.getByText('Call 1'));

fireEvent.click(screen.getByText('Call 2'));

expect(onClickMock).toHaveBeenCalled(); // called at least once

expect(onClickMock).toHaveBeenCalledTimes(2); // called exactly twice

expect(onClickMock).toHaveBeenCalledWith('apple', 42); // called with first args at least once

expect(onClickMock).toHaveBeenNthCalledWith(2, 'banana'); // 2nd call was with 'banana'

});

**What this does:**

* Renders two buttons that call onClick with different arguments.
* Simulates clicking each button.
* Checks the mock was called twice.
* Verifies the arguments of both calls, including the order.

### 2. ****Control return values****

Simulate different return values without running real code.

js

CopyEdit

const mockFn = jest.fn();

// Always return 10

mockFn.mockReturnValue(10);

console.log(mockFn()); // 10

console.log(mockFn()); // 10

// Return different values on consecutive calls

mockFn.mockReturnValueOnce(1).mockReturnValueOnce(2);

console.log(mockFn()); // 1

console.log(mockFn()); // 2

console.log(mockFn()); // 10 (default return)

ument();

// Also check how many times the mock was called

expect(mockFn).toHaveBeenCalledTimes(3);

});

How this works:

mockReturnValueOnce(1) makes the first call return 1.

mockReturnValueOnce(2) makes the second call return 2.

mockReturnValue(10) makes all other calls return 10 (default).

The test clicks the button 3 times and checks the displayed value each time.

### 3. ****Simulate custom behavior****

Make the mock function execute your own logic.

js

CopyEdit

const mockFn = jest.fn((x) => x \* 2);

console.log(mockFn(3)); // 6

console.log(mockFn(5)); // 10

### ✅ 1. React Component

jsx

CopyEdit

import React, { useState } from 'react';

function DoubleButton({ onDouble }) {

const [result, setResult] = useState(null);

const number = 4; // hardcoded number

const handleClick = () => {

const doubled = onDouble(number); // mock will handle logic

setResult(doubled);

};

return (

<div>

<button onClick={handleClick}>Double it</button>

<p>Result: {result !== null ? result : 'Click to see result'}</p>

</div>

);

}

export default DoubleButton;

### ✅ 2. Test: Simulate Custom Logic

js

CopyEdit

import React from 'react';

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

import DoubleButton from './DoubleButton';

test('calls mock function and executes custom logic', () => {

// Mock with custom logic: double the input

const mockFn = jest.fn(x => x \* 2);

render(<DoubleButton onDouble={mockFn} />);

fireEvent.click(screen.getByText('Double it'));

// Was called with 4

expect(mockFn).toHaveBeenCalledWith(4);

// Result should be 8 (4 \* 2)

expect(screen.getByText('Result: 8')).toBeInTheDocument();

});

### ✅ Summary

| **Feature** | **Outcome** |
| --- | --- |
| jest.fn(x => x \* 2) | Custom behavior (doubles input) |
| mockFn(3) | Returns 6 |
| Component behavior | Doubles a hardcoded value (4) |
| Test | Confirms mock logic & output |

### 4. ****Replace real functions or modules****

Use jest.fn() to mock callbacks, props, or dependencies during testing.

// Example: testing a component with an onClick handler

const onClickMock = jest.fn();

render(<Button onClick={onClickMock}>Click me</Button>);

fireEvent.click(screen.getByText('Click me'));

expect(onClickMock).toHaveBeenCalledTimes(1);

### ======== Button Component with Hardcoded Label

jsx

CopyEdit

import React from 'react';

function Button({ onClick }) {

return (

<button onClick={onClick}>

Click me

</button>

);

}

export default Button;

### Jest Test for This Component

js

CopyEdit

import React from 'react';

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

import Button from './Button';

test('calls onClick when button is clicked', () => {

const onClickMock = jest.fn();

render(<Button onClick={onClickMock} />);

fireEvent.click(screen.getByText('Click me'));

expect(onClickMock).toHaveBeenCalledTimes(1);

});

So the label "Click me" is fixed inside the component and not passed as a prop. The only prop expected is onClick.

### 5. ****Mock asynchronous functions****

Mock async behaviors using mockResolvedValue or mockRejectedValue.

### ✅ Example Async Component

jsx

CopyEdit

import React, { useState } from 'react';

function FetchMessage({ fetchData }) {

const [message, setMessage] = useState('');

const [error, setError] = useState('');

const handleClick = async () => {

try {

const res = await fetchData();

setMessage(res.data);

} catch (err) {

setError(err.message);

}

};

return (

<div>

<button onClick={handleClick}>Fetch</button>

{message && <p>Message: {message}</p>}

{error && <p>Error: {error}</p>}

</div>

);

}

export default FetchMessage;

### ✅ Test: Mock Resolved & Rejected Async Behavior

js

CopyEdit

import React from 'react';

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

import FetchMessage from './FetchMessage';

test('displays message when fetch resolves', async () => {

const mockFetch = jest.fn().mockResolvedValue({ data: 'Hello World' });

render(<FetchMessage fetchData={mockFetch} />);

fireEvent.click(screen.getByText('Fetch'));

const message = await screen.findByText('Message: Hello World');

expect(message).toBeInTheDocument();

expect(mockFetch).toHaveBeenCalledTimes(1);

});

test('displays error when fetch rejects', async () => {

const mockFetch = jest.fn().mockRejectedValue(new Error('Failed'));

render(<FetchMessage fetchData={mockFetch} />);

fireEvent.click(screen.getByText('Fetch'));

const error = await screen.findByText('Error: Failed');

expect(error).toBeInTheDocument();

expect(mockFetch).toHaveBeenCalledTimes(1);

});

### ✅ Summary

| **Behavior** | **Method** | **Example** |
| --- | --- | --- |
| Resolve mock | mockResolvedValue | mockResolvedValue({ data: 'Hi' }) |
| Reject mock | mockRejectedValue | mockRejectedValue(new Error('Oops')) |
| Test resolve | await findByText(...) | Verifies resolved output is displayed |
| Test reject | await findByText(...) | Verifies error message is displayed |

### 6. ****Test custom hooks or utilities****

Isolate dependencies of hooks/utilities by mocking their input functions.

## ✅ Scenario: Custom Hook + Mocked Dependency

### 🔧 Step 1: Create a Custom Hook (useUser.js)

jsx

CopyEdit

import { useState, useEffect } from 'react';

function useUser(fetchUser) {

const [user, setUser] = useState(null);

useEffect(() => {

const result = fetchUser(); // This is mocked in the test

setUser(result);

}, [fetchUser]);

return user;

}

export default useUser;

## 🧪 Step 2: Test the Hook (with Mocked Dependency)

If you're using @testing-library/react:

### ✅ Option 1: Hook used inside a component (no extra lib needed)

jsx

CopyEdit

// UserComponent.jsx

import React from 'react';

import useUser from './useUser';

function UserComponent({ fetchUser }) {

const user = useUser(fetchUser);

if (!user) return <div>Loading...</div>;

return <div>Hello, {user.name}</div>;

}

export default UserComponent;

js

CopyEdit

// UserComponent.test.js

import React from 'react';

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

import UserComponent from './UserComponent';

test('calls fetchUser and shows user name', () => {

const mockFetchUser = jest.fn().mockReturnValue({ id: 1, name: 'Alice' });

render(<UserComponent fetchUser={mockFetchUser} />);

expect(mockFetchUser).toHaveBeenCalled();

expect(screen.getByText('Hello, Alice')).toBeInTheDocument();

});

### ✅ Option 2: Isolate the hook itself (advanced)

If you want to **test just the hook** directly, you can use [@testing-library/react-hooks](https://react-hooks-testing-library.com/), which requires installing:

Serach online >>>>

**where to use beforeEach, afterEach, beforeAll, and afterAll** in your test files — with **real, practical examples** using React components and utilities.

## ✅ **What These Do**

| **Hook** | **Runs when?** |
| --- | --- |
| beforeAll | Once **before all** tests in file |
| beforeEach | Before **each** test |
| afterEach | After **each** test |
| afterAll | Once **after all** tests in file |

### ✅ What's included:

1. fetchUser.js – Actual mockable fetch logic (can be real or fake API).
2. UserProfile.js – Component that uses fetchUser with async/await.
3. UserProfile.test.js – Test using jest.mock() + full lifecycle hooks.

## 1️⃣ fetchUser.js – The async logic

js

CopyEdit

// fetchUser.js

export async function fetchUser() {

// In real life, this might be:

// const response = await fetch('/api/user');

// return await response.json();

return new Promise((resolve) => {

setTimeout(() => {

resolve({ name: 'Likan' });

}, 100); // simulate network delay

});

}

## 2️⃣ UserProfile.js – The React component

jsx

CopyEdit

// UserProfile.js

import React, { useEffect, useState } from 'react';

import { fetchUser } from './fetchUser';

function UserProfile() {

const [user, setUser] = useState(null);

useEffect(() => {

let isMounted = true;

const getUser = async () => {

try {

const res = await fetchUser();

if (isMounted) setUser(res);

} catch (err) {

if (isMounted) setUser({ name: 'Error fetching user' });

}

};

getUser();

return () => {

isMounted = false;

};

}, []);

return <div>{user ? `Welcome, ${user.name}` : 'Loading user...'}</div>;

}

export default UserProfile;

## 3️⃣ UserProfile.test.js – Full test with mock + lifecycle

js

CopyEdit

// UserProfile.test.js

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

import UserProfile from './UserProfile';

import \* as api from './fetchUser'; // import \* for mocking

jest.mock('./fetchUser'); // mock the module

beforeAll(() => {

console.log('✅ Global setup before all tests');

});

beforeEach(() => {

api.fetchUser.mockResolvedValue({ name: 'Likan' });

render(<UserProfile />);

console.log('🔁 Fresh render before each test');

});

afterEach(() => {

cleanup();

jest.clearAllMocks();

console.log('🧹 Cleaned up after each test');

});

afterAll(() => {

console.log('🚫 All tests done, clean global stuff');

});

test('shows loading initially', () => {

expect(screen.getByText('Loading user...')).toBeInTheDocument();

});

test('displays fetched user name', async () => {

const text = await screen.findByText('Welcome, Likan');

expect(text).toBeInTheDocument();

expect(api.fetchUser).toHaveBeenCalledTimes(1);

});

### 📦 Folder Structure

bash

CopyEdit

/src

├── fetchUser.js

├── UserProfile.js

└── UserProfile.test.js

# What is waitFor?

* **waitFor is a helper function in React Testing Library** that lets your test **wait for something to happen asynchronously** before continuing.
* It keeps **re-running a function you give it until that function stops throwing an error or a timeout happens**.

# Why do we need waitFor?

* React apps often update the UI **after some delay** — like after data fetching, animations, timers, or user events.
* When testing, sometimes you want to **wait for those updates before making assertions**.
* But JavaScript tests run super fast — they might check for something **before it appears on screen**, causing tests to fail.
* waitFor solves this by **retrying your assertion repeatedly** until it passes or times out.

# How does waitFor work?

* You pass a function (usually with an assertion) to waitFor.
* waitFor runs it.
* If it throws (meaning the thing you want isn't ready), waitFor waits a bit and tries again.
* It keeps trying until either:
  + Your function succeeds (no error thrown), OR
  + The timeout is reached (test fails).

# Example:

js

CopyEdit

await waitFor(() => {

// This assertion might fail at first if 'Hello' is not yet on the screen

expect(screen.getByText('Hello')).toBeInTheDocument();

});

* The test will keep trying the above until "Hello" appears or the timeout happens.

waitFor = a tool to "wait and retry" until your UI or condition is ready, preventing flaky tests caused by asynchronous UI updates.

### Can you replace waitFor with findBy?

* **Yes, you often can!** Because findBy **already waits** for the element to appear and resolves with it.
* findBy is basically a **built-in wait + get** for a single element.

### When to prefer findBy over waitFor?

* When you just want to **get one element that appears asynchronously**, use findBy — it’s simpler and cleaner.
* Example:

js

CopyEdit

const helloElement = await screen.findByText('Hello');

expect(helloElement).toBeInTheDocument();

## Using waitFor with getBy (for complex assertion)

jsx

CopyEdit

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

import React, { useEffect, useState } from 'react';

function ExampleComponent() {

const [loading, setLoading] = useState(true);

useEffect(() => {

setTimeout(() => setLoading(false), 500); // Simulate async load

}, []);

return (

<div>

{loading ? <p>Loading...</p> : <p>Hello</p>}

</div>

);

}

// Test

test('waitFor example: waits for Loading... to disappear and Hello to appear', async () => {

render(<ExampleComponent />);

// Loading... is initially present

expect(screen.getByText('Loading...')).toBeInTheDocument();

// waitFor to wait for Loading... to disappear AND Hello to appear

await waitFor(() => {

expect(screen.queryByText('Loading...')).not.toBeInTheDocument();

expect(screen.getByText('Hello')).toBeInTheDocument();

});

});

## Using findBy (only for waiting for one element to appear)

jsx

CopyEdit

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

import React, { useEffect, useState } from 'react';

function ExampleComponent() {

const [loading, setLoading] = useState(true);

useEffect(() => {

setTimeout(() => setLoading(false), 500); // Simulate async load

}, []);

return (

<div>

{loading ? <p>Loading...</p> : <p>Hello</p>}

</div>

);

}

// Test

test('findBy example: waits for Hello to appear', async () => {

render(<ExampleComponent />);

// findBy waits for Hello to appear

const helloElement = await screen.findByText('Hello');

expect(helloElement).toBeInTheDocument();

// You can also check Loading... has disappeared after that

expect(screen.queryByText('Loading...')).not.toBeInTheDocument();

});

### Key points:

* **waitFor with getBy** lets you combine multiple assertions and conditions (both disappearance and appearance) in one place.
* **findBy** only waits for the **appearance** of a single element.

==================================================================

**Usage of act-**

# What is act?

act is a **testing helper function from React** that ensures **all updates related to React components (like rendering, state changes, effects)** are finished and applied before you run your test assertions.

# Why do we need act?

React batches and applies state updates asynchronously. If you make changes that update the component’s state or cause effects, and then immediately run your test assertions **without waiting for React to finish updating the DOM**, your tests might:

* Fail or behave unpredictably
* Show warnings like **"An update to your component was not wrapped in act(...)"**

act tells React:

“Hey, wait until all updates are done, then let me continue with my test assertions.”

# Simple Example (Without act) — Causes Warning

jsx

CopyEdit

import React, { useState } from 'react';

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

function Counter() {

const [count, setCount] = useState(0);

return (

<>

<div>Count: {count}</div>

<button onClick={() => setCount(count + 1)}>Increment</button>

</>

);

}

test('increments counter', () => {

render(<Counter />);

fireEvent.click(screen.getByText('Increment'));

// React warning: Update not wrapped in act

expect(screen.getByText('Count: 1')).toBeInTheDocument();

});

You may get this warning:

vbnet

CopyEdit

Warning: An update to Counter inside a test was not wrapped in act(...).

# Simple Example (With act) — No Warning

jsx

CopyEdit

import React, { useState } from 'react';

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

function Counter() {

const [count, setCount] = useState(0);

return (

<>

<div>Count: {count}</div>

<button onClick={() => setCount(count + 1)}>Increment</button>

</>

);

}

test('increments counter', () => {

render(<Counter />);

// Wrap the click event in act so React knows to process updates before assertion

act(() => {

fireEvent.click(screen.getByText('Increment'));

});

expect(screen.getByText('Count: 1')).toBeInTheDocument();

});

No warning here, because act ensures React processes the state update before the assertion.

# Complete Example with Async State Update & act

Imagine you have a component that fetches data asynchronously and updates state:

jsx

CopyEdit

import React, { useEffect, useState } from 'react';

function AsyncComponent() {

const [message, setMessage] = useState('Loading...');

useEffect(() => {

setTimeout(() => {

setMessage('Data loaded');

}, 500);

}, []);

return <div>{message}</div>;

}

### Test without act (May fail or warn)

jsx

CopyEdit

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

test('shows loaded message', async () => {

render(<AsyncComponent />);

// Assertion immediately after render — React may still update state after 500ms

expect(screen.getByText('Data loaded')).toBeInTheDocument(); // FAILS

});

### Test using act and waiting for 5ms

jsx

CopyEdit

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

import AsyncComponent from './AsyncComponent'; // adjust path accordingly

test('shows loaded message after 5ms delay', async () => {

render(<AsyncComponent />);

// Wait for React to process the setTimeout update after 5ms

await act(async () => {

await new Promise(resolve => setTimeout(resolve, 10)); // wait a bit longer than 5ms to be safe

});

expect(screen.getByText('Data loaded')).toBeInTheDocument(); // PASSES

});

Act summary –

| **Aspect** | **Explanation** |
| --- | --- |
| **What** | React testing helper to wait for updates |
| **Why** | To avoid state update warnings and flaky tests |
| **When** | When code causes React state/DOM updates |
| **How** | Wrap updates (sync or async) inside act |
| **With RTL helpers** | Usually done automatically (e.g. fireEvent) |
| **Without RTL helpers** | You need to call act manually |

**Good to know -   
  
1.mock service worker   
2.snapshiot testing**

**3.Isolate testing of hooks ()Test custom hooks (we need additional packages I think )**

**4. Matchers ….all matchers …**