## 🔰 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().

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

**Waitfor ???**

**Jest function more cases ??**

**Test custom hooks**

**Usage of act**

**Cleabing up mocks after each**

**Matchers ..all matchers …**

**Good to know -   
  
mock service worker   
snapshiot testing**