Основные шаги при написании тестов с React Testing Library

**1. Рендер компонента**

Первый шаг — это отрисовать компонент, который ты хочешь протестировать. Для этого используется функция render() из React Testing Library.

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

import MyComponent from "./MyComponent";

test("renders MyComponent", () => {

  render(<MyComponent />); // Отрисовываем компонент

  // Проверяем, что компонент отобразил определённый текст

  expect(screen.getByText(/hello, world!/i)).toBeInTheDocument();

});

💡 screen используется для поиска элементов, которые отобразил render(). Например:

screen.getByText(/текст/) — найти элемент с определённым текстом.

screen.getByRole("button") — найти элемент с определённой ролью (например, кнопку, заголовок).

screen.getByPlaceholderText("Введите текст") — найти поле ввода по подсказке (placeholder).

**2. Проверка взаимодействия**

Ты можешь тестировать пользовательские действия, например, нажатие кнопок или ввод текста в поля. Для этого используется функция fireEvent.

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

import MyComponent from "./MyComponent";

test("button click changes text", () => {

  render(<MyComponent />);

    // Находим кнопку по тексту

  const button = screen.getByText("Кликни меня");

    // Проверяем, что начальный текст "Привет"

  expect(screen.getByText("Привет")).toBeInTheDocument();

    // Эмулируем клик по кнопке

  fireEvent.click(button);

  // Убедимся, что текст изменился

  expect(screen.getByText("Пока")).toBeInTheDocument();

});

💡 Такой тест проверяет, как компонент реагирует на клики, ввод текста, выбор из списков и т. д.

**3. Работа с мокающими функциями**

Моки (mock) — это поддельные функции или данные, которые заменяют настоящие. Например, можно замокать API или хук.

Пример: мок функции

test("calls a function on button click", () => {

  const onClickMock = jest.fn(); // Создаём мок-функцию

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

  fireEvent.click(screen.getByText("Click me")); // Эмулируем клик

  // Проверяем, что мок-функция вызвалась 1 раз

  expect(onClickMock).toHaveBeenCalledTimes(1);

});

jest.mock("react-router-dom", () => ({

  ...jest.requireActual("react-router-dom"),

  useParams: jest.fn(),

}));

Пример: мок хуков из react-router-dom

Если твой компонент использует хуки типа useParams или useSearchParams, их можно замокать.

jest.mock("react-router-dom", () => ({

  ...jest.requireActual("react-router-dom"),

  useParams: jest.fn(),

}));

import { useParams } from "react-router-dom";

test("renders data based on useParams", () => {

  useParams.mockReturnValue({ id: "123" }); // Мокируем возвращаемое значение useParams

  render(<MyComponent />);

  // Проверяем, что компонент отображает ID 123 из useParams

  expect(screen.getByText("ID: 123")).toBeInTheDocument();

});

**Рабочие примеры**

**Пример 1: Проверяем текст на странице**

Простой тест, который проверяет отрисовку статических элементов.

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

import MyPage from "./MyPage";

test("renders page header", () => {

  render(<MyPage />);

  // Проверяем, что заголовок "Моя страница" есть на экране

  expect(screen.getByText(/моя страница/i)).toBeInTheDocument();

});

**Пример 2: Тестируем взаимодействие с кнопкой**

Добавим интерактивность — будем тестировать, как компонент обновляет текст при клике.

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

import Counter from "./Counter";

test("increments counter on button click", () => {

  render(<Counter />);

  // Проверяем, что начальное значение счётчика — 0

  expect(screen.getByText("Счётчик: 0")).toBeInTheDocument();

  // Эмулируем клик по кнопке

  const button = screen.getByRole("button", { name: "Увеличить" });

  fireEvent.click(button);

  // Проверяем, что значение счётчика стало 1

  expect(screen.getByText("Счётчик: 1")).toBeInTheDocument();

});

**Пример 3: Мокаем API**

Предположим, компонент загружает данные с сервера. Обычно в тестах мы мокаем этот запрос, чтобы компонент не делал настоящий вызов.

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

import MyComponent from "./MyComponent";

// Мокаем глобальный fetch

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

  Promise.resolve({

    json: () => Promise.resolve({ data: "Hello World" }),

  })

);

test("fetches and displays data", async () => {

  render(<MyComponent />);

  // Проверяем, что данные отобразились

  expect(await screen.findByText("Hello World")).toBeInTheDocument();

});

💡 Здесь findByText используется, чтобы дождаться асинхронного обновления UI.

**Пример 4: Компонент с хранилищем состояния (context)**

Если компонент использует контекст, то тест должен обернуть его в Provider.

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

import { MyContext } from "./MyContext";

import MyComponent from "./MyComponent";

test("renders greeting from context", () => {

  render(

    <MyContext.Provider value={{ greeting: "Привет, мир!" }}>

      <MyComponent />

    </MyContext.Provider>

  );

  // Проверяем, что компонент показывает данные из контекста

  expect(screen.getByText("Привет, мир!")).toBeInTheDocument();

});

Советы:

Используй screen вместо container: screen помогает писать читабельные тесты ("ищи текст на экране").

Асинхронные действия: Для ожидания рендеринга при асинхронных действиях используй findBy вместо getBy.

Мокаем правильно:

   Функции: Используй jest.fn().

   Модули: Используй jest.mock().

   API: Замени fetch или любую API функцию на поддельную с jest.fn().

**Моки (Mocks)** — это подделки функций, данных, зависимостей или модулей, которые используются при тестировании для замены их реальных версий. Они позволяют изолировать тестируемый компонент или функции, делая тестирование более предсказуемым, быстрым и независимым.

**Для чего нужны моки**

**Изоляция компонентов:**

   Компонент может зависеть от внешнего API, хуков или других объектов. Чтобы не использовать реальные данные, мы замещаем их моками.

   Например, избегаем реального вызова API.

**Ускорение тестов:**

   Вместо тяжелых операций (например, запросов на сервер или работы с большим количеством данных) моки используют простой заранее заданный результат.

**Проверка побочных эффектов:**

   Бывают ситуации, когда важно не то, что делает функция, а то, что она была вызвана (например, setState или useSearchParams).

**Имитирование поведения:**

   Можно точно контролировать, что возвращают замоканные зависимости (например, хук возвращает конкретные параметры).

**Когда использовать моки**

Внешние API-запросы.\*\*

Контекст (React Context).\*\*

React-хуки (например, useParams, useSearchParams, useNavigate).\*\*

Взаимодействие с Redux или другими хранилищами состояния.\*\*

Сторонние модули, библиотеки или функции.\*\*

**Как создавать моки**

1. Мокаем функции с jest.fn()

jest.fn() создает поддельную функцию, которая может перехватывать вызовы, параметры и возвращать значения.

// Подделываем функцию обратного вызова

const mockFunction = jest.fn();

// Вызываем поддельную функцию с аргументами

mockFunction(1, "Hello");

// Проверяем, что функция вызвана 1 раз

expect(mockFunction).toHaveBeenCalledTimes(1);

// Проверяем, что функция вызвана с правильными аргументами

expect(mockFunction).toHaveBeenCalledWith(1, "Hello");

// Можно настроить, что функция возвращает

mockFunction.mockReturnValue("Мок-результат");

expect(mockFunction()).toBe("Мок-результат");

**2. Мокаем модули с jest.mock()**

jest.mock() позволяет подменять весь модуль.

Пример: Мок библиотеки

// Подделываем модуль fetch

jest.mock("node-fetch", () => jest.fn());

import fetch from "node-fetch";

test("тестируем fetch", async () => {

  // Указываем, что fetch будет возвращать "мок-ответ"

  fetch.mockResolvedValue({

    json: () => Promise.resolve({ message: "Привет, мир!" }),

  });

  const response = await fetch();

  const data = await response.json();

  // Проверяем, что fetch вернул нужные данные

  expect(data.message).toBe("Привет, мир!");

});

Пример: Мок кастомного модуля

import \* as api from "./api";

// Замокаем весь модуль

jest.mock("./api");

test("замена поведения функции в модуле", () => {

  api.getData.mockResolvedValue("мок-данные");

  const result = api.getData();

  expect(result).resolves.toBe("мок-данные");

});

**3. Мокаем хуки React**

Многие хуки из React и сторонних библиотек (например, react-router-dom) нужно мокаем, чтобы управлять их поведением в тестах.

Пример 1: Мокаем useParams из react-router-dom

Хук useParams используется для получения параметров из URL (<http://site.com/users/:id>). Замокаем его, чтобы вернуть нужный параметр.

jest.mock("react-router-dom", () => ({

  ...jest.requireActual("react-router-dom"), // Сохраняем оригинальную функциональность

  useParams: jest.fn(), // Замокаем useParams

}));

import { useParams } from "react-router-dom";

test("useParams возвращает userId", () => {

  useParams.mockReturnValue({ id: "123" }); // Указываем, что будет возвращено хук

  render(<MyComponent />);

  // Проверяем, что компонент корректно отображает userId

  expect(screen.getByText("ID пользователя: 123")).toBeInTheDocument();

});

**Пример 2: Мокаем useSearchParams**

Хук useSearchParams используется для работы с параметрами строки запроса (?search=name). Замокаем его.

jest.mock("react-router-dom", () => ({

  ...jest.requireActual("react-router-dom"),

  useSearchParams: jest.fn(),

}));

import { useSearchParams } from "react-router-dom";

test("useSearchParams устанавливает поисковые параметры", () => {

  const setSearchParamMock = jest.fn(); // Мокируем метод для изменения параметров

  useSearchParams.mockReturnValue([{ get: jest.fn() }, setSearchParamMock]);

  render(<MyComponent />);

  // Выполняем пользовательское действие (например, ввод текста в поле)

  fireEvent.change(screen.getByLabelText("Поиск:"), { target: { value: "test" } });

  // Проверяем, что setSearchParams вызван с аргументами

  expect(setSearchParamMock).toHaveBeenCalledWith({ search: "test" });

});

**Пример 3: Мокаем useNavigate**

Хук useNavigate используется для программной навигации.

jest.mock("react-router-dom", () => ({

  ...jest.requireActual("react-router-dom"),

  useNavigate: jest.fn(),

}));

import { useNavigate } from "react-router-dom";

test("useNavigate вызывается при клике", () => {

  const navigateMock = jest.fn(); // Мокируем navigate

  useNavigate.mockReturnValue(navigateMock);

  render(<MyComponent />);

  fireEvent.click(screen.getByText("Перейти")); // Выполняем клик

  // Проверяем, что useNavigate вызван с правильным URL

  expect(navigateMock).toHaveBeenCalledWith("/some-path");

});

**4. Мокаем асинхронные функции**

Иногда ваш компонент вызывает асинхронные функции, например, запросы к API. Такие вызовы можно замокаем.

// Мокаем fetch

global.fetch = jest.fn();

test("загрузка данных с API", async () => {

  fetch.mockResolvedValue({

    json: jest.fn().mockResolvedValue({ message: "Hello World" }),

  });

  render(<MyComponent />);

  // Ждем, пока данные отобразятся

  expect(await screen.findByText("Hello World")).toBeInTheDocument();

  // Проверяем, что fetch вызван

  expect(fetch).toHaveBeenCalledTimes(1);

});

**Синтаксис использования моков**

Создание простого мока:

      const mockFunction = jest.fn();

   mockFunction.mockReturnValue("value");

Создание мока модуля:

      jest.mock("some-library", () => ({

     someFunction: jest.fn().mockReturnValue("data"),

   }));

Мокаем хуки из сторонних библиотек:

      jest.mock("react-router-dom", () => ({

     useNavigate: jest.fn(),

     useParams: jest.fn(),

   }));