**Моки в Jest и React** — это важный инструмент, который упрощает тестирование кода. Они позволяют имитировать поведение функций,

модулей или компонентов, isolating зависимости для выполнения тестов. Разберём их максимально подробно, шаг за шагом.

**Что такое моки?**

Моки — это подделки, которые заменяют реальные зависимости вашего приложения, чтобы они возвращали ожидаемые значения или имитировали поведение. Скажем, у вас есть компонент, который вызывает API. При тестировании вы бы не хотели отправлять настоящие запросы — вместо этого API заменяется моком, который возвращает предопределённые данные.

**Почему моки важны?**

**Изоляция тестов:**

   Моки позволяют тестировать компонент или функцию в изоляции от других зависимостей.

   Например, вы можете протестировать компонент, который получает данные, без необходимости запускать сервер или подключаться к сети.

**Контроль поведения:**

   Вы можете заранее определить, что вернёт ваша мок-функция, что упрощает проверку различных сценариев (успешные ответы, ошибки и т. д.).

**Быстрота тестов:**

   Моки ускоряют выполнение тестов, так как реальный сетевой запрос или сложная логика не выполняется.

**Моки в Jest — Основы**

Jest предоставляет встроенные возможности для создания моков. Вот ключевые методы:

**jest.fn():**

  Создаёт простую мок-функцию.

**jest.mock():**

  Подменяет модуль целиком или его отдельные методы.

**jest.spyOn():**

  Наблюдает за реальными методами и позволяет частично их подменять.

**Пример использования jest.fn()**

jest.fn() создаёт простую поддельную функцию, которая может использоваться вместо настоящей:

// Функция, которую мы хотим протестировать:

const fetchData = (callback) => {

  // Например, представим, что вызывается некое API:

  const data = "Real Data";

  callback(data);

};

test("тестирование функции с использованием мока", () => {

  const mockCallback = jest.fn(); // Создаем мок

  fetchData(mockCallback);

  // Проверяем, вызвалась ли мок-функция

  expect(mockCallback).toHaveBeenCalled();

  // Проверяем параметры вызова

  expect(mockCallback).toHaveBeenCalledWith("Real Data");

});

**Объяснение:**

Мок заменяет реальный callback.

Мы проверяем не только факт вызова мока, но и то, с какими аргументами он был вызван (toHaveBeenCalledWith("Real Data")).

**Примеры с jest.fn()**

jest.fn() используется для создания мок-функций, которые можно настроить для тестирования вызовов, аргументов и возвращаемых значений.

**Пример 1: Тестирование callback-функции**

Допустим, у нас есть функция, которая принимает callback:

// Функция, которую будем тестировать

const processData = (data, callback) => {

  callback(data + 1);

};

Протестируем её с помощью jest.fn():

test("Тестирование processData с мок-функцией", () => {

  const mockCallback = jest.fn(); // Создаем мок-функцию

  processData(2, mockCallback);

  // Проверяем, что мок-функция вызвана

  expect(mockCallback).toHaveBeenCalled();

  // Проверяем, что она вызвана с правильным аргументом

  expect(mockCallback).toHaveBeenCalledWith(3);

  // Проверяем, сколько раз она была вызвана

  expect(mockCallback).toHaveBeenCalledTimes(1);

});

**Пример 2: Настройка возвращаемого значения**

С помощью jest.fn() можно заранее задать возвращаемые значения мок-функции:

test("Мок-функция с возвращаемым значением", () => {

  const mockFn = jest.fn().mockReturnValue(42);

  const result = mockFn();

  // Проверяем, что функция возвращает заданное значение

  expect(result).toBe(42);

  // Проверяем вызов функции

  expect(mockFn).toHaveBeenCalled();

});

**Пример 3: Проверка аргументов вызовов**

Если функция вызывается несколько раз с разными аргументами, мы можем это протестировать:

test("Мок-функция с несколькими вызовами", () => {

  const mockFn = jest.fn();

  mockFn(1);

  mockFn(2);

  // Проверяем, что мок-функция вызвана дважды

  expect(mockFn).toHaveBeenCalledTimes(2);

  // Проверяем аргументы первого и второго вызова

  expect(mockFn.mock.calls0).toBe(1); // Первый вызов

  expect(mockFn.mock.calls0).toBe(2); // Второй вызов

});

**Примеры с jest.mock()**

jest.mock() используется для полной подмены модулей, что позволяет изолировать тесты от настоящих зависимостей.

**Пример 1: Мокирование модуля API**

Допустим, у нас есть модуль api.js:

// api.js

export const fetchData = () => {

  return Promise.resolve({ data: "Real Data" });

};

Теперь мы хотим протестировать компонент, использующий этот модуль:

// Component.js

import { fetchData } from "./api";

export const Component = async () => {

  const response = await fetchData();

  return response.data;

};

Создаём мок для модуля:

jest.mock("./api", () => ({

  fetchData: jest.fn(), // Мокируем функцию fetchData

}));

import { fetchData } from "./api";

import { Component } from "./Component";

test("Мокирование модуля API", async () => {

  fetchData.mockResolvedValue({ data: "Mocked Data" }); // Настраиваем возвращаемое значение

  const result = await Component();

  // Проверяем, что функция возвращает замокированные данные

  expect(result).toBe("Mocked Data");

  // Проверяем вызов функции fetchData

  expect(fetchData).toHaveBeenCalled();

});

**Пример 2: Мокирование нескольких функций в модуле**

Предположим, модуль содержит несколько функций:

// utils.js

export const add = (a, b) => a + b;

export const subtract = (a, b) => a - b;

Мокируем весь модуль:

jest.mock("./utils", () => ({

  add: jest.fn(),

  subtract: jest.fn(),

}));

import { add, subtract } from "./utils";

test("Мокирование нескольких функций", () => {

  add.mockReturnValue(10);

  subtract.mockReturnValue(5);

  expect(add(2, 3)).toBe(10);

  expect(subtract(10, 5)).toBe(5);

  // Проверяем вызовы

  expect(add).toHaveBeenCalledWith(2, 3);

  expect(subtract).toHaveBeenCalledWith(10, 5);

});

**Пример 3: Мокирование зависимостей у React-компонента**

Допустим, компонент вызывает API:

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

import { fetchData } from "./api";

export const MyComponent = () => {

  const [data, setData] = useState("");

  useEffect(() => {

    fetchData().then((res) => setData(res.data));

  }, []);

  return {data};

};

Тестируем с мокированным API:

jest.mock("./api", () => ({

  fetchData: jest.fn(),

}));

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

import { MyComponent } from "./MyComponent";

import { fetchData } from "./api";

test("Мокирование API-запроса в React компоненте", async () => {

  fetchData.mockResolvedValue({ data: "Hello, World!" });

  render();

  // Проверяем, что компонент сначала показывает пустое состояние

  expect(screen.queryByText("Hello, World!")).toBeNull();

  // Ждём, пока компонент обновится

  await waitFor(() => {

    expect(screen.getByText("Hello, World!")).toBeInTheDocument();

  });

});

**Пример использования jest.mock() (Замена модулей)**

Часто приходится подменять целые зависимости/модули.

Представим, что у нас есть модуль api.js, который содержит вызов API:

// api.js

export const getUser = () => fetch("<https://api.example.com/user>").then((res) => res.json());

Мы хотим протестировать компонент, который использует getUser:

// MyComponent.js

import { getUser } from "./api";

export const MyComponent = async () => {

  const user = await getUser();

  return Hello, ${user.name};

};

Мокируем модуль в тесте:

// MyComponent.test.js

import { getUser } from "./api";

import { MyComponent } from "./MyComponent";

// Подменяем весь модуль

jest.mock("./api", () => ({

  getUser: jest.fn(),

}));

test("тестирование компонента с мокнутым API", async () => {

  // Настраиваем мок-ответ функции

  getUser.mockResolvedValue({ name: "John" });

  const result = await MyComponent();

  expect(result).toBe("Hello, John");

  expect(getUser).toHaveBeenCalled(); // Проверяем, что функция вызвана

});

**Объяснение:**

jest.mock() заменяет модуль ./api.

Внутри теста мы используем mockResolvedValue, чтобы указать, что возвращает функция.

Компонент вызывается, но вместо реального вызова API возвращаются наши данные.

**Примеры с jest.spyOn()**

jest.spyOn() позволяет наблюдать за вызовами реальной функции, но при этом оставлять её поведение (или частично мимикрировать).

**Пример 1: Отслеживание вызовов метода объекта**

const mathUtils = {

  multiply: (a, b) => a \* b,

};

test("Шпионаж за методом объекта", () => {

  const spy = jest.spyOn(mathUtils, "multiply");

  const result = mathUtils.multiply(2, 3);

  // Проверяем вызывалась ли функция

  expect(spy).toHaveBeenCalled();

  // Проверяем аргументы

  expect(spy).toHaveBeenCalledWith(2, 3);

  // Проверяем результат

  expect(result).toBe(6);

  spy.mockRestore(); // Восстанавливаем оригинальное поведение

});

**Пример 2: Подмена результата функции**

const mathUtils = {

  divide: (a, b) => a / b,

};

test("Шпионаж с подменой возвращаемого значения", () => {

  const spy = jest.spyOn(mathUtils, "divide").mockReturnValue(10);

  const result = mathUtils.divide(4, 2);

  // Проверяем, что вернули подменённое значение

  expect(result).toBe(10);

  spy.mockRestore(); // Возвращаем оригинальную функцию

});

**Пример использования jest.spyOn()**

Иногда нужно не полностью заменять функцию, а просто "подсмотреть" её вызов или вернуть определённый результат:

const api = {

  fetchData: () => "Real Data",

};

test("шпионаж за вызовом метода", () => {

  // Подключаемся к существующей функции

  const spy = jest.spyOn(api, "fetchData").mockReturnValue("Mocked Data");

  const data = api.fetchData();

  expect(data).toBe("Mocked Data"); // Проверяем подставленный результат

  expect(spy).toHaveBeenCalled(); // Проверяем вызов функции

  spy.mockRestore(); // Восстанавливаем оригинальную функцию

  expect(api.fetchData()).toBe("Real Data");

});

**Объяснение:**

jest.spyOn() отслеживает вызовы реальной функции, но также позволяет контролировать её поведение через mockReturnValue.

**Моки в контексте React**

Когда вы работаете с React-компонентами, моки часто используются для:

Замены методов API.

Тестирования взаимодействий с пропсами или контекстом.

Имитирования дочерних компонентов.

Мокирование API-запросов с React Testing Library

Пример компонента, в котором используется API:

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

import { getUser } from "./api";

export const UserComponent = () => {

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

  useEffect(() => {

    getUser().then((data) => setUser(data));

  }, []);

  if (!user) {

    return <div>Loading...</div>;

  }

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

};

**Тестируем компонент:**

import React from "react";

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

import { UserComponent } from "./UserComponent";

import { getUser } from "./api";

// Мокируем API

jest.mock("./api", () => ({

  getUser: jest.fn(),

}));

test("рендер компонента с мокированным API", async () => {

  getUser.mockResolvedValue({ name: "Jane" }); // Настраиваем мок

  render(<UserComponent />);

  expect(screen.getByText(/Loading.../i)).toBeInTheDocument(); // Спиннер указан

  // Ждем, пока компонент обновится

  await waitFor(() => {

    expect(screen.getByText("Hello, Jane")).toBeInTheDocument();

  });

});

**Объяснение:**

jest.mock() подменяет зависимость getUser.

Мы эмулируем разные сценарии, такие как успешный ответ (используя mockResolvedValue).

С помощью waitFor проверяем, что компонент корректно обновляется по приведённым данным.

**Полезные методы Jest для моков**

mockReturnValue(value) Заставляет мок-функцию вернуть определённое значение.

mockResolvedValue(data) Возвращает промис с данным значением (для работы с async/await).

mockImplementation(fn) Полностью заменяет выполнение функции на пользовательский алгоритм.

jest.spyOn(object, method) Отслеживает вызовы реальной функции объекта, с возможностью подмены.

**Ключевые моменты**

Моки упрощают тестирование сложных приложений, изолируя зависимости.

Используйте:

   jest.fn() для отдельных функций.

   jest.mock() для целых модулей.

   jest.spyOn() для шпионажа или частичной подмены реальных функций.

В React чаще всего моки применяются для подмены API или имитации поведения дочерних компонентов.

Вот несколько подробных примеров применения jest.fn(), jest.mock() и jest.spyOn(). Мы рассмотрим реальные кейсы для каждого из этих методов, чтобы вы могли понять их различия и лучшие случаи применения.

**Резюме**

Используйте jest.fn(), если вы тестируете отдельные функции или callback'и.

Используйте jest.mock(), если необходимо замокировать внешний модуль или целую зависимость.

Используйте jest.spyOn(), если вы хотите лишь следить за вызовами функции, сохраняя её поведение.