Зачем имитировать (мокать) данные

В этом материале вы узнаете, как имитировать (мокать) функционал из внешних библиотек в процессе разработки unit-тестов для React-приложений.

Имитировать (мокать) функции или объекты — это необходимый навык, который пригодится во время проверки модулей кода.

С его помощью в тестах можно добиться:

Предсказуемости. Благодаря искусственно созданным объектам можно добиться однозначного поведения тестов.

Изолированности. Моки позволяют избавиться от внешних зависимостей и работать с тестом конкретного модуля, не влияя на результат.

Гибкости. Можно проверять тестовые сценарии, подставляя легко настраиваемый объект, который покрывает различные варианты работы модуля.

Как выполнить имитацию написанных модулей, понятно: можно передать моки через аргументы в компоненты или функции. Но в случае с самостоятельным функционалом, на который нельзя повлиять через аргументы, всё не так просто. Например, при использовании внешних библиотек для запросов, когда нужно замокать поведение сервера и отдать данные, или библиотеки react-router-dom, которая управляет URL браузера.

Имитация простейших функций

Для начала разберём моки для компонентов. Чтобы имитировать поведение функции, которую вы используете, можно воспользоваться командой:

jest.fn()

Она возвращает функцию-заглушку с дополнительными опциями для тестов.

Пример: есть простой компонент Button, на который можно нажать, чтобы вызвать обработчик onClick.

import { ButtonHTMLAttributes, FC } from ' react'

export const Button: FC<ButtonHTMLAttributes<HTMLButtonElement>> = ({

onClick,

})=> {

return ‹button onClick={onClick}>Click me!</button>

}

Нам нужно написать тест, который будет проверять, что onClick вызывается. Тест будет выглядеть следующим образом:

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

import { Button } from './test'

describe( 'Test Button', () = {

afterEach(() => {

jest.clearAllMocks()

})

it( 'Check onClick prop', () → f

const onClick = jest.fn()

render (<Button onClick={onClick) />)

fireEvent.click(screen.getByRole( 'button'))

expect(onClick).toHaveBeenCalled()

})

})

В примере мы передаём функцию-заглушку в пропсы компоненту Button, затем имитируем нажатие на эту кнопку и ждём вызов заглушки.

Имитация функционала из внешних библиотек

В примере выше всё понятно: делаем функцию-заглушку, передаём её в компонент и проверяем, что она была вызвана при нажатии на кнопку. Теперь давайте рассмотрим, что делать, если передать что-либо извне нельзя.

Например, если используется внешний модуль или сторонняя библиотека, алгоритм из предыдущего примера не сработает. Но выход есть — это специальная конструкция в jest, которая позволяет имитировать целые модули. Разберём её на примере.

Допустим, у нас есть простое навигационное меню с тремя элементами-ссылками:

Main ведёт на ‘/’.

Settings — на ‘/settings’.

Dashboard — на ‘/Dashboard.

Если значение ссылки совпадает с текущим значением URL, ей назначается дополнительный CSS-класс, чтобы подчеркнуть ссылку.

Этот класс можно добавить элементу с помощью хука из react-router-dom useLocation.

import { Link, useLocation } from "react-router-dom'

import './styles.css'

export const NavMenu = () => {

const { pathname } = useLocation()

return (

‹div className="container">

<Link to="/" className={pathname == '/' ? 'active' : ''}>

Main

</Link>

<Link

to="/settings"

className=(pathname === '/settings' ? 'active' : ''}

>

Settings

</Link>

<Link

to="/dashboard"

className={pathname === '/dashboard' ? 'active' : ''}

>

Dashboard

</Link>

</div>

)}

Нужно проверить: если pathname совпадает со значением ссылки, назначаем ей соответствующий класс.

Таким образом, задача нашего тестирования будет звучать следующим образом:

мы хотим получить значение pathname и проверить, что соответствующая ссылка содержит класс active.

Попробуем это реализовать:

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

import { NavMenu } from ' /NavMenu'

describe('Test Nav Menu', () => {

afterEach(() => {

jest.clearAllMocks()

})

it('Check active settings link', () = {

/\*

Какой-либо функционал, который позволит получить '/settings' в pathname

из хука useLocation.

\*/

render (<NavMenu />)

const settingsLink = screen.getByText('Settings')

expect(settingsLink.classList.contains('active')).toBeTruthy()

})

В тесте находим ссылку с текстом Settings и проверяем, что она содержит класс active. Теперь нужно вернуть '/settings' в pathname, чтобы тест заработал. Сделаем это, добавив перед всеми тестами следующую конструкцию:

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

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

}))

Из всего пакета react-router-dom мы делаем заглушку, чтобы добавить опции для других заглушек на конкретный хук. Jest.requireActual('react-router-dom') оставляет реализацию функций, чтобы их поведение не нарушалось.

Затем необходимо сделать заглушку из хука useLocation, она вернёт нужный pathname.

jest.spyOn(Router, 'useLocation').mockReturnValue({

pathname: '/settings',

hash: '',

key: '',

search: '',

state: '',

})

В описании хука обязательно должны быть заданы остальные свойства. Но можно оставить их значения пустыми, так как они нам не нужны.

Запустим тест и проверим его работоспособность.

Появилась ошибка. В ней нет ничего страшного, она возникла из-за того, что для работы react-router-dom нужно обернуть всё приложение в BrowserRouter.

import { BrowserRouter } from 'react-router-dom'

import './app.css'

import { NavMenu } from './NavMenu'

export function App() {

return (

<BrowserRouter>

<NavMenu />

</BrowserRouter>

)

}

Выше показан App.tsx-файл приложения.

Чтобы обернуть тестируемый компонент в BrowserRouter, у render-функции есть второй аргумент, который принимает объект. Добавим поле Wrapper.

render(<NavMenu />, { wrapper: MemoryRouter })

Итоговый код теста для этого компонента будет выглядеть следующим образом:

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

import { NavMenu } from './NavMenu'

import Router, { MemoryRouter } from 'react-router-dom'

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

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

}))

describe('Test Nav Menu', () => {

afterEach(() => {

jest.clearAllMocks()

})

it('Check active settings link', () => {

jest.spyOn(Router, 'useLocation').mockReturnValue({

pathname: '/settings',

hash: '',

key: '',

search: '',

state: '',

})

render(<NavMenu />, { wrapper: MemoryRouter })

const settingsLink = screen.getByText('Settings')

expect(settingsLink.classList.contains('active')).toBeTruthy()

})

})

    expect(settingsLink.classList.contains('active')).toBeTruthy()

  })

})

Выводы

В этом материале вы научились делать заглушки для целых библиотек, что позволяет реализовывать более сложные тесты при работе с приложениями. Помимо разобранных методов, есть множество способов мокать поведение сторонних библиотек. Изучайте другие источники информации, чтобы понять, как их использовать.

Чтобы ещё больше разобраться в теме, рекомендуем следующие материалы:

React Testing Library

React Router

React Testing Library с примерами тестов React Router