

[КАК СТАТЬ АВТОРОМ](#)[Неделя тестировщиков](#)[Знаешь Java? Держи квест!](#)**728.55**

Рейтинг

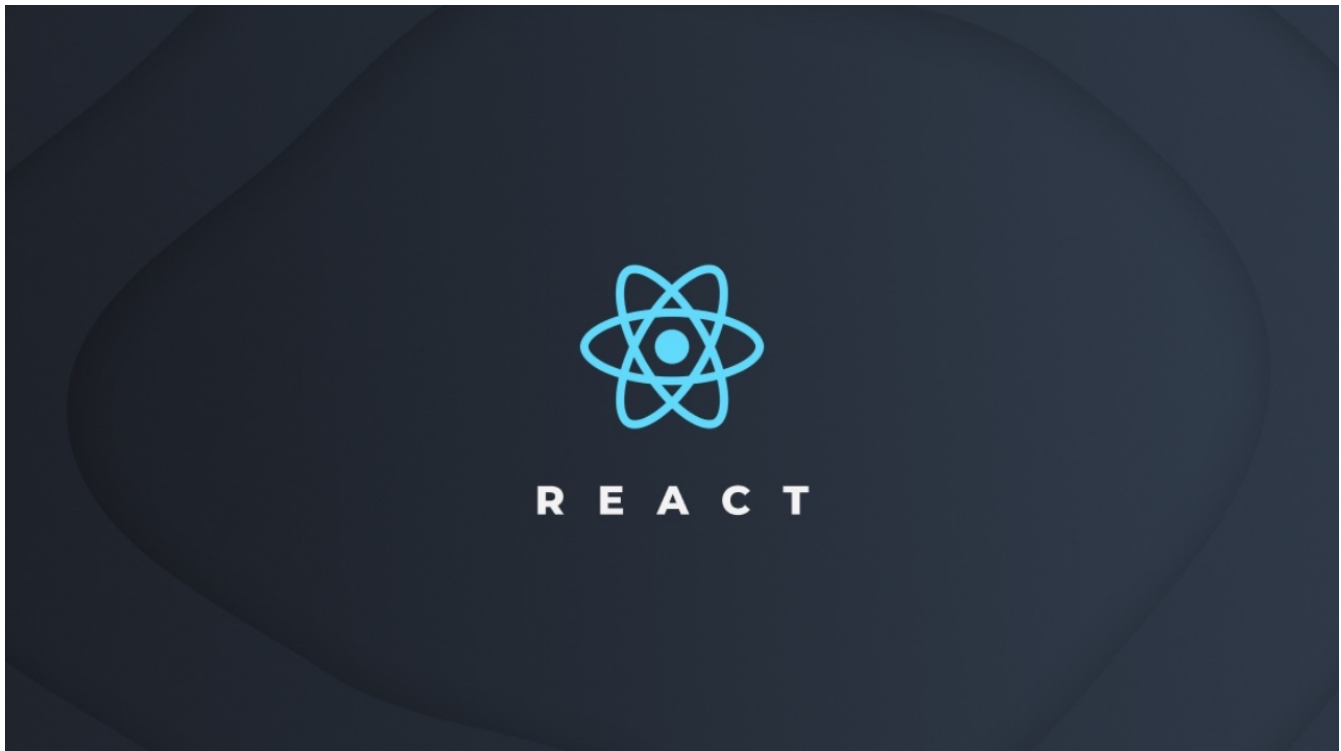
Timeweb Cloud

Облачная платформа для разработчиков и бизнеса

**aio350** 9 июня в 16:23

React: тестируем компоненты с помощью Jest и Testing Library

Блог компании Timeweb Cloud, JavaScript*, ReactJS*



Привет, друзья!

В данном tutorialе я покажу вам, как тестировать компоненты на [React](#) с помощью [Jest](#) и [Testing Library](#).

Список основных задач, которые мы решим на протяжении tutorialа:

1. Создание шаблона React-приложения с помощью [Vite](#).
2. Создание компонента для получения приветствия от сервера.

3. Установка и настройка Jest .
4. Установка и настройка Testing Library .
5. Тестирование компонента с помощью Testing Library :
 1. Используя стандартные возможности.
 2. С помощью кастомного рендера.
 3. С помощью кастомных запросов.
6. Тестирование компонента с помощью снимков Jest .

Репозиторий с кодом проекта.

Если вам это интересно, прошу под кат.

Создание шаблона

Обратите внимание: для работы с зависимостями я буду использовать [Yarn](#).

Vite — это продвинутый сборщик модулей (bundler) для JavaScript-приложений . Он более производительный и не менее кастомизируемый, чем [Webpack](#).

Vite CLI позволяет создавать готовые к разработке проекты, в том числе, с помощью некоторых шаблонов.

Создаем шаблон React-приложения :

```
# react-testing - название проекта
# --template react - используемый шаблон
yarn vite create react-testing --template react
```

Переходим в созданную директорию и устанавливаем зависимости:

```
cd react-testing
yarn
```

Убедиться в работоспособности приложения можно, выполнив команду `yarn dev` .

Приводим структуру проекта к следующему виду:

```
- node_modules
- src
  - App.jsx
  - main.jsx
- .gitignore
- index.html
- package.json
- vite.config.js
- yarn.lock
```

```
{ task: 'setup project', status: 'done' }
```

Создание компонента

Для обращения к API мы будем использовать [Axios](#):

```
yarn add axios
```

Создаем в директории `src` файл `FetchGreeting.jsx` следующего содержания:

```
import { useState } from 'react'
import axios from 'axios'

// пропом компонента является адрес конечной точки
// для получения приветствия от сервера
const FetchGreeting = ({ url }) => {
  // состояние приветствия
  const [greeting, setGreeting] = useState('')
  // состояние ошибки
  const [error, setError] = useState(null)
  // состояние нажатия кнопки
  const [btnClicked, setBtnClicked] = useState(false)

  // метод для получения приветствия от сервера
  const fetchGreeting = (url) =>
    axios
      .get(url)
      // если запрос выполнен успешно
```

```
.then((res) => {
  const { data } = res
  const { greeting } = data
  setGreeting(greeting)
  setBtnClicked(true)
})
// если возникла ошибка
.catch((e) => {
  setError(e)
})

// текст кнопки
const btnText = btnClicked ? 'Готово' : 'Получить приветствие'

return (
  <div>
    <button onClick={() => fetchGreeting(url)} disabled={btnClicked}>
      {btnText}
    </button>
    {/* если запрос выполнен успешно */}
    {greeting && <h1>{greeting}</h1>}
    {/* если возникла ошибка */}
    {error && <p role='alert'>Не удалось получить приветствие</p>}
  </div>
)
}

export default FetchGreeting
```

```
{ task: 'create component', status: 'done' }
```

Установка и настройка Jest

Устанавливаем Jest :

```
yarn add jest
```

По умолчанию средой для тестирования является [Node.js](#), поэтому нам потребуется еще один пакет:

```
yarn add jest-environment-jsdom
```

Создаем в корне проекта файл `jest.config.js` (настройки Jest) следующего содержания:

```
module.exports = {  
  // среда тестирования - браузер  
  testEnvironment: 'jest-environment-jsdom',  
}
```

Для транспиляции кода перед запуском тестов Jest использует [Babel](#). Поскольку мы будем работать с **JSX** нам потребуется два "пресета":

```
yarn add @babel/preset-env @babel/preset-react
```

Создаем в корне проекта файл `babel.config.js` (настройки Babel) следующего содержания:

```
module.exports = {  
  presets: [  
    '@babel/preset-env',  
    ['@babel/preset-react', { runtime: 'automatic' }]  
  ]  
}
```

Настройка `runtime: 'automatic'` добавляет React в глобальную область видимости, что позволяет не импортировать его явно в каждом файле.

Дефолтной директорией с тестами для Jest является `__tests__`. Создаем эту директорию в корне проекта.

Создаем в директории `__tests__` файл `fetch-greeting.test.jsx` следующего содержания:

```
test.todo('получение приветствия')
```

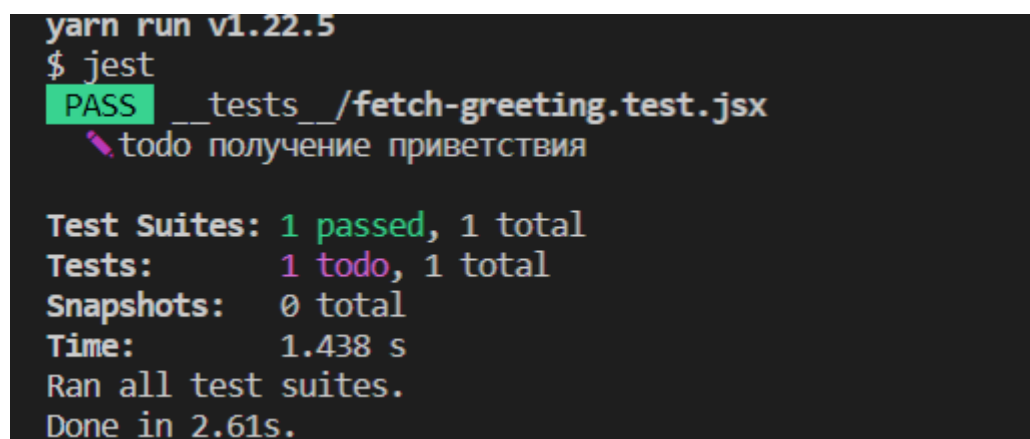
Объекты `describe`, `test`, `expect` и другие импортируются в пространство модуля `Jest`. Почитать об этом можно [здесь](#) и [здесь](#).

`test.todo(name: string)` — это своего рода заглушка для теста, который мы собираемся писать.

Добавляем в раздел `scripts` файла `package.json` команду для запуска тестов:

```
"test": "jest"
```

Выполняем эту команду с помощью `yarn test`:

A terminal window with a dark background. The first line shows the command 'yarn run v1.22.5' followed by '\$ jest'. The next line shows 'PASS' in a green box, followed by '___tests___/fetch-greeting.test.jsx'. Below that is a comment '/* todo получение приветствия */'. Then, a summary of test results is shown: 'Test Suites: 1 passed, 1 total', 'Tests: 1 todo, 1 total', 'Snapshots: 0 total', 'Time: 1.438 s', 'Ran all test suites.', and 'Done in 2.61s.'

```
yarn run v1.22.5
$ jest
PASS ___tests___/fetch-greeting.test.jsx
/* todo получение приветствия */

Test Suites: 1 passed, 1 total
Tests: 1 todo, 1 total
Snapshots: 0 total
Time: 1.438 s
Ran all test suites.
Done in 2.61s.
```

Получаем в терминале нашу "тудушку" и сообщение об успешном выполнении "теста".

Кажется, что можно приступить к тестированию компонента. Почти, есть один нюанс.

Дело в том, что `Jest` спроектирован для работы с `Node.js` и не поддерживает `ESM` из коробки. Более того, поддержка `ESM` является экспериментальной и в будущем может претерпеть некоторые изменения. Почитать об этом можно [здесь](#).

Для того, чтобы все работало, как ожидается, нужно сделать 2 вещи.

Можно определить в `package.json` тип кода как модуль (`"type": "module"`), но это ломает `Vite`. Можно изменить расширение файла с тестом на `.mjs`, но мне такой вариант не нравится. А можно сообщить `Jest` расширения файлов, которые следует обрабатывать как `ESM`:

```
// jest.config.js
module.exports = {
  testEnvironment: 'jest-environment-jsdom',
  // !
  extensionsToTreatAsEsm: ['.jsx'],
}
```

Также необходимо каким-то образом передать Jest флаг `--experimental-vm-modules`. Существует несколько способов это сделать, но наиболее подходящим с точки зрения обеспечения совместимости с разными ОС является следующий:

1. Устанавливаем `cross-env` с помощью `yarn add cross-env`.
2. Редактируем команду для запуска тестов:

```
"test": "cross-env NODE_OPTIONS=--experimental-vm-modules jest"
```

```
{ task: 'setup jest', status: 'done' }
```

Установка и настройка Testing Library

Устанавливаем обертку Testing Library для React :

```
yarn add @testing-library/react
```

Для обеспечения интеграции с Jest нам также потребуется следующий пакет:

```
yarn add @testing-library/jest-dom
```

Для тестирования отправки запроса на сервер и получения от него приветствия необходим фиктивный (mock) сервер. Одним из самых простых решений для этого является `msw`:

```
yarn add msw
```

```
{ task: 'setup testing library', status: 'done' }
```

Тестирование компонента с помощью Testing Library

Стандартные возможности

Реализуем тестирование компонента с помощью стандартных возможностей, предоставляемых Testing Library .

Начнем с импорта зависимостей:

```
// msw
import { rest } from 'msw'
import { setupServer } from 'msw/node'
// см. ниже
import { render, fireEvent, waitFor, screen } from '@testing-library/react'
// см. ниже
import '@testing-library/jest-dom'
// компонент
import FetchGreeting from '../src/FetchGreeting'
```

Создаем фиктивный сервер:

```
const server = setupServer(
  rest.get('/greeting', (req, res, ctx) =>
    res(ctx.json({ greeting: 'Привет!' })))
)
```

В ответ на GET HTTP-запрос сервер будет возвращать объект с ключом `greeting` и значением `Привет!` .

Определяем глобальные хуки:

```
// запускаем сервер перед выполнением тестов
beforeAll(() => server.listen())
// сбрасываем обработчики к дефолтной реализации после каждого теста
afterEach(() => server.resetHandlers())
```



```
// останавливаем сервер после всех тестов
afterAll(() => server.close())
```

Мы напишем 2 теста:

- для получения приветствия и его рендеринга;
- для обработки ошибки сервера.

Поскольку тестируется один и тот же функционал, имеет смысл сгруппировать тесты с помощью `describe`:

```
describe('получение приветствия', () => {
  // todo
})
```

Начнем с теста для получения приветствия и его рендеринга:

```
test('-> успешное получение и отображение приветствия', async function () {
  // рендерим компонент
  // https://testing-library.com/docs/react-testing-library/api/#render
  render(<FetchGreeting url='/greeting' />)

  // имитируем нажатие кнопки для отправки запроса
  // https://testing-library.com/docs/dom-testing-library/api-events#fireEvent
  //
  // screen привязывает (bind) запросы к document.body
  // https://testing-library.com/docs/queries/about/#screen
  fireEvent.click(screen.getByText('Получить приветствие'))

  // ждем рендеринга заголовка
  // https://testing-library.com/docs/dom-testing-library/api-async/#waitFor
  await waitFor(() => screen.getByRole('heading'))

  // текстом заголовка должно быть `Привет!`
  expect(screen.getByRole('heading')).toHaveTextContent('Привет!')
  // текстом кнопки должно быть `Готово`
  expect(screen.getByRole('button')).toHaveTextContent('Готово')
  // кнопка должна быть заблокированной
  expect(screen.getByRole('button')).toBeDisabled()
})
```

```
} )
```

О запросах типа `getByRole` можно почитать [здесь](#), а список всех стандартных запросов можно найти [здесь](#).

О кастомных сопоставлениях (matchers), которыми `@testing-library/jest-dom` расширяет объект `expect` из `Jest` можно почитать [здесь](#).

Перед тем, как приступить к реализации теста для обработки ошибки сервера установим еще один пакет:

```
yarn add @testing-library/user-event
```

Данный пакет рекомендуется использовать для имитации пользовательских событий (типа нажатия кнопки) вместо `fireEvent`. Почитать об этом можно [здесь](#).

```
// `it` - синоним `test`
it('-> обработка ошибки сервера', async () => {
  // после этого сервер в ответ на запрос
  // будет возвращать ошибку со статус-кодом `500`
  server.use(rest.get('/greeting', (req, res, ctx) => res(ctx.status(500)))

  // рендерим компонент
  render(<FetchGreeting url='greeting' />)

  // имитируем нажатие кнопки
  // рекомендуемый подход
  // https://testing-library.com/docs/user-event/setup
  const user = userEvent.setup()
  // если не указать `await`, тогда `Testing Library`
  // не успеет обернуть обновление состояния компонента
  // в `act` и мы получим предупреждение в терминале
  await user.click(screen.getByText('Получить приветствие'))

  // ждем рендеринга сообщения об ошибке
  await waitFor(() => screen.getByRole('alert'))

  // текстом сообщения об ошибке должно быть `Не удалось получить приветствие`
  expect(screen.getByRole('alert')).toHaveTextContent(
    'Не удалось получить приветствие'
  )
})
```

```
)  
// кнопка не должна быть заблокированной  
expect(screen.getByRole('button')).not.toBeDisabled()  
})
```

Запускаем тесты с помощью команды `yarn test` :

```
PASS __tests__/fetch-greeting.test.jsx  
  получение приветствия  
    ✓ -> успешное получение и отображение приветствия (179 ms)  
    ✓ -> обработка ошибки сервера (125 ms)  
  
Test Suites: 1 passed, 1 total  
Tests:       2 passed, 2 total  
Snapshots:   0 total  
Time:        2.724 s  
Ran all test suites.  
Done in 4.24s.
```

```
{ task: 'default testing', status: 'done' }
```

Кастомный рендер

Предположим, что мы хотим распределить состояние приветствия между несколькими компонентами, например, с помощью провайдера.

Создаем в директории `src` файл `GreetingProvider.jsx` следующего содержания:

```
import { createContext, useContext, useReducer } from 'react'  
  
// начальное состояние  
const initialState = {  
  error: null,  
  greeting: null  
}  
  
// константы  
const SUCCESS = 'SUCCESS'  
const ERROR = 'ERROR'  
  
// редуктор  
function greetingReducer(state, action) {
```

```
switch (action.type) {
  case SUCCESS:
    return {
      error: null,
      greeting: action.payload
    }
  case ERROR:
    return {
      error: action.payload,
      greeting: null
    }
  default:
    return state
}
}

// создатель операций
const createGreetingActions = (dispatch) => ({
  setSuccess(success) {
    dispatch({
      type: SUCCESS,
      payload: success
    })
  },
  setError(error) {
    dispatch({
      type: ERROR,
      payload: error
    })
  }
})

// контекст
const GreetingContext = createContext()

// провайдер
export const GreetingProvider = ({ children }) => {
  const [state, dispatch] = useReducer(greetingReducer, initialState)

  const actions = createGreetingActions(dispatch)

  return (
    <GreetingContext.Provider value={{ state, actions }}>
      {children}
    </GreetingContext.Provider>
  )
}
```

```
    )  
  }  
  
  // кастомный хук  
  export const useGreetingContext = () => useContext(GreetingContext)
```

Оборачиваем компонент `FetchGreeting` провайдером в файле `App.jsx` :

```
import { GreetingProvider } from './GreetingProvider'  
import FetchGreeting from './FetchGreeting'  
  
function App() {  
  return (  
    <div className='App'>  
      <GreetingProvider>  
        <FetchGreeting url='/greeting' />  
      </GreetingProvider>  
    </div>  
  )  
}  
  
export default App
```

Редактируем `FetchGreeting.jsx` :

```
import { useState } from 'react'  
import axios from 'axios'  
import { useGreetingContext } from './GreetingProvider'  
  
const FetchGreeting = ({ url }) => {  
  // извлекаем состояние и операции из контекста  
  const { state, actions } = useGreetingContext()  
  const [btnClicked, setBtnClicked] = useState(false)  
  
  const fetchGreeting = (url) =>  
    axios  
      .get(url)  
      .then((res) => {  
        const { data } = res  
        const { greeting } = data  
        // !  
      })  
}
```

```
        actions.setSuccess(greeting)
        setBtnClicked(true)
    })
    .catch((e) => {
        // !
        actions.setError(e)
    })

    const btnText = btnClicked ? 'Готово' : 'Получить приветствие'

    return (
      <div>
        <button onClick={() => fetchGreeting(url)} disabled={btnClicked}>
          {btnText}
        </button>
        {/* ! */}
        {state.greeting && <h1 data-cy='heading'>{state.greeting}</h1>}
        {state.error && <p role='alert'>Не удалось получить приветствие</p>}
      </div>
    )
  }

  export default FetchGreeting
```

Для того, чтобы не оборачивать явно каждый тестируемый компонент в провайдеры, из которых он потребляет тот или иной контекст (состояние, тема, локализация и т.д.), предназначен **кастомный рендер**.

Создаем в корне проекта директорию `testing`. В этой директории создаем файл `test-utils.jsx` следующего содержания:

```
import { render } from '@testing-library/react'
import { GreetingProvider } from '../src/GreetingProvider'

// все провайдеры приложения
const AllProviders = ({ children }) => (
  <GreetingProvider>{children}</GreetingProvider>
)

// кастомный рендер
const customRender = (ui, options) =>
  render(ui, {
    // обертка для компонента
```

```
    wrapper: AllProviders,  
    ...options  
  })  
  
  // повторно экспортируем `Testing Library`  
  export * from '@testing-library/react'  
  // перезаписываем метод `render`  
  export { customRender as render }
```

Для того, чтобы иметь возможность импортировать кастомный рендер просто из `test-utils` необходимо сделать 2 вещи:

1. Сообщить Jest названия директорий с модулями:

```
// jest.config.js  
module.exports = {  
  testEnvironment: 'jest-environment-jsdom',  
  extensionsToTreatAsEsm: ['.jsx'],  
  // !  
  moduleDirectories: ['node_modules', 'testing']  
}
```

1. Добавить синоним пути в файле `jsconfig.json` (создаем этот файл в корне проекта):

```
{  
  "compilerOptions": {  
    "baseUrl": "src",  
    "paths": {  
      "test-utils": [  
        "./testing/test-utils"  
      ]  
    }  
  }  
}
```

Для TypeScript-проекта синонимы путей (и другие настройки) определяются в

файле `tsconfig.json`.

Редактируем файл `fetch-greeting.test.jsx`:

```
// импортируем стандартные утилиты `Testing Library` и кастомный рендер
import { render, fireEvent, waitFor, screen } from 'test-utils'
```

Запускаем тест с помощью `yarn test` и убеждаемся в том, что тесты по-прежнему выполняются успешно.

```
{ task: 'testing with custom render', status: 'done' }
```

Кастомные запросы

Что если нам оказалось недостаточно стандартных запросов, предоставляемых `Testing Library`? Что если мы, например, хотим получать ссылку на DOM-элемент с помощью атрибута `data-cy`? Для этого предназначены **кастомные запросы**.

Создаем в директории `testing` файл `custom-queries.js` следующего содержания:

```
import { queryHelpers, buildQueries } from '@testing-library/react'

const queryAllByDataCy = (...args) =>
  queryHelpers.queryAllByAttribute('data-cy', ...args)

const getMultipleError = (c, dataCyValue) =>
  `Обнаружено несколько элементов с атрибутом data-cy: ${dataCyValue}`

const getMissingError = (c, dataCyValue) =>
  `Не обнаружен элемент с атрибутом data-cy: ${dataCyValue}`

// генерируем кастомные запросы
const [
  queryByDataCy,
  getAllByDataCy,
  getByDataCy,
  findAllByDataCy,
  findByDataCy
```



```
] = buildQueries(queryAllByDataCy, getMultipleError, getMissingError)

// и экспортируем их
export {
  queryByDataCy,
  queryAllByDataCy,
  getByDataCy,
  getAllByDataCy,
  findByDataCy,
  findAllByDataCy
}
```

Далее кастомные запросы можно внедрить в кастомный рендер:

```
// test-utils.js
import { render, queries } from '@testing-library/react'
import * as customQueries from './custom-queries'

const customRender = (ui, options) =>
  render(ui, {
    wrapper: AllProviders,
    // !
    queries: { ...queries, ...customQueries },
    ...options
  })
```

Определяем атрибут `data-cy` у заголовка в компоненте `FetchGreeting` :

```
{state.greeting && <h1 data-cy='heading'>{state.greeting}</h1>}
```

И получаем ссылку на этот элемент в тесте с помощью кастомного запроса:

```
const { getByDataCy } = render(<FetchGreeting url='/greeting' />)

expect(getByDataCy('heading')).toHaveTextContent('Привет!')
```

Запускаем тест с помощью `yarn test` :

И получаем ошибку.

```
function getQueriesForElement(element, queries = defaultQueries, initialValue) {
  return Object.keys(queries).reduce((helpers, key) => {
    // получаем запрос по ключу
    const fn = queries[key];
    // и передаем в запрос элемент в качестве аргумента
    // здесь возникает ошибка
    // `fn.bind` не является функцией`
    helpers[key] = fn.bind(null, element);
    return helpers;
  }, initialValue);
}
```

18 of 31

```
// test-utils.jsx  
console.log(customQueries)
```

Запускаем тест:

```
console.log  
[Module: null prototype] {  
  __esModule: true,  
  default: {  
    findAllByDataCy: [Function (anonymous)],  
    findByDataCy: [Function (anonymous)],  
    getAllByDataCy: [Function (anonymous)],  
    getByDataCy: [Function (anonymous)],  
    queryAllByDataCy: [Function: queryAllByDataCy],  
    queryByDataCy: [Function (anonymous)]  
  },  
  findAllByDataCy: [Function (anonymous)],  
  findByDataCy: [Function (anonymous)],  
  getAllByDataCy: [Function (anonymous)],  
  getByDataCy: [Function (anonymous)],  
  queryAllByDataCy: [Function: queryAllByDataCy],  
  queryByDataCy: [Function (anonymous)]  
}  
  
at log (testing/test-utils.jsx:6:9)
```

Видим ключ `__esModule` со значением `true`. Свойство `__esModule` функцией не является, поэтому при попытке вызова `bind` на нем выбрасывается исключение. Но откуда оно взялось в нашем модуле?

Коротко о главном:

- `test-utils.jsx` является модулем для тестирования;
- Jest автоматически создает "моковые" версии таких модулей — объекты заменяются, API сохраняется;
- перед созданием мока код модуля транпилируется с помощью `Babel` ;
- Jest запускается в режиме поддержки ESM , поэтому `Babel` добавляет свойство `__esModule` в каждый мок.

Одним из самых простых способов решения данной проблемы является запрос оригинального модуля (без создания его моковой версии) с помощью метода `requireActual` объекта `jest`.

Для того, чтобы иметь возможность использовать этот объект в ESM, его следует импортировать из `@jest/globals`:

```
yarn add @jest/globals
```

```
import { jest } from '@jest/globals'
// import * as customQueries from './custom-queries'
const customQueries = jest.requireActual('./custom-queries')
```

Запускаем тест. Теперь все работает, как ожидается.

```
{ task: 'testing with custom queries', status: 'done' }
```

Тестирование компонента с помощью снимков Jest

Конечно, можно исследовать каждый DOM-элемент компонента по отдельности с помощью сопоставлений типа `toHaveTextContent`, но, согласитесь, что это не очень удобно. Легко можно пропустить какой-нибудь элемент или атрибут.

Для исследования текущего состояния всего UI за один раз предназначены **снимки** (snapshots).

На самом деле, в нашем распоряжении уже имеется все необходимое для тестирования компонента с помощью снимков. Одним из значений, возвращаемых методом `render` является `container`, который можно передать в метод `expect` и вызвать метод `toMatchSnapshot`:

```
describe('получение приветствия', () => {
  test('-> успешное получение и отображение приветствия', async function () {
    // получаем контейнер
    const { container, getByDataCy } = render(<FetchGreeting url='/greeting
    // тестируем текущее состояние `UI` с помощью снимка
    expect(container).toMatchSnapshot()
```

```
fireEvent.click(screen.getByText('Получить приветствие'))

await waitFor(() => screen.getByRole('heading'))
// состояние `UI` изменилось, поэтому нужен еще один снимок
expect(container).toMatchSnapshot()

expect(getByDataCy('heading')).toHaveTextContent('Привет!')
expect(screen.getByRole('button')).toHaveTextContent('Готово')
expect(screen.getByRole('button')).toBeDisabled()
})

it('-> обработка ошибки сервера', async () => {
  server.use(rest.get('/greeting', (req, res, ctx) => res(ctx.status(500)

  // получаем контейнер
  const { container } = render(<FetchGreeting url='greeting' />)
  // снимок 1
  expect(container).toMatchSnapshot()

  const user = userEvent.setup()
  await user.click(screen.getByText('Получить приветствие'))

  await waitFor(() => screen.getByRole('alert'))
  // снимок 2
  expect(container).toMatchSnapshot()

  expect(screen.getByRole('alert')).toHaveTextContent(
    'Не удалось получить приветствие'
  )
  expect(screen.getByRole('button')).not.toBeDisabled()
})
})
```

Запускаем тест:

```
yarn run v1.22.5
$ cross-env NODE_OPTIONS=--experimental-vm-modules jest
(node:6080) ExperimentalWarning: VM Modules is an experimental feature. This feature could change at any time
(Use `node --trace-warnings ...` to show where the warning was created)
PASS __tests__/fetch-greeting.test.jsx
  получение приветствия
    ✓ -> успешное получение и отображение приветствия (181 ms)
    ✓ -> обработка ошибки сервера (88 ms)

  > 4 snapshots written.
Snapshot Summary
  > 4 snapshots written from 1 test suite.

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   4 written, 4 total
Time:        3.105 s, estimated 4 s
Ran all test suites.
Done in 4.53s.
```

При первом выполнении теста, в котором используются снимки, Jest генерирует снимки и складывает их в директорию `__snapshots__` в директории с тестом. В нашем случае запуск теста привел к генерации файла `fetch-greeting.test.jsx.snap` следующего содержания:

```
exports[`получение приветствия -> обработка ошибки сервера 1`] = `
<div>
  <div>
    <button>
      Получить приветствие
    </button>
  </div>
</div>
`;

exports[`получение приветствия -> обработка ошибки сервера 2`] = `
<div>
  <div>
    <button>
      Получить приветствие
    </button>
  <p
    role="alert"
  >
    <div>
      <div>
        <button>
          Получить приветствие
        </button>
      </div>
    </div>
  </p>
</div>
`;
```

```
    >
    Не удалось получить приветствие
  </p>
</div>
</div>
`;

exports[`получение приветствия -> успешное получение и отображение приветс
<div>
  <div>
    <button>
      Получить приветствие
    </button>
  </div>
</div>
`;

exports[`получение приветствия -> успешное получение и отображение приветс
<div>
  <div>
    <button
      disabled=""
    >
      Готово
    </button>
    <h1
      data-cy="heading"
    >
      Привет!
    </h1>
  </div>
</div>
`;
```

Как видим, снимок правильно отражает все изменения состояния UI компонента.

Снова запускаем тест:

```
yarn run v1.22.5
$ cross-env NODE_OPTIONS=--experimental-vm-modules jest
(node:2644) ExperimentalWarning: VM Modules is an experimental feature. This feature could change at any time
(Use `node --trace-warnings ...` to show where the warning was created)
PASS __tests__/fetch-greeting.test.jsx
  получение приветствия
    ✓ -> успешное получение и отображение приветствия (185 ms)
    ✓ -> обработка ошибки сервера (70 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   4 passed, 4 total
Time:        2.655 s, estimated 3 s
Ran all test suites.
Done in 4.17s.
```

```
{ task: 'snapshot testing', status: 'done' }
```

Парочка полезных советов:

- для обновления снимка следует передать флаг `--updateSnapshot` или просто `-u` при вызове Jest: `yarn test -u`;
- для указания тестов для выполнения или снимков для обновления при вызове Jest можно передать флаг `--testPathPattern` со значением директории с тестами (в виде строки или регулярного выражения): `yarn test -u --testPathPattern=components/fetchGreeting`.

Для того, чтобы иметь возможность импортировать статику (изображения, шрифты, аудио, видео и т.д.) в тестируемых компонентах, необходимо реализовать [кастомный трансформер](#) для Jest.

Создаем в директории `testing` файл `file-transformer.js` следующего содержания:

```
// формат `CommonJS` в данном случае является обязательным
const path = require('path')

module.exports = {
```



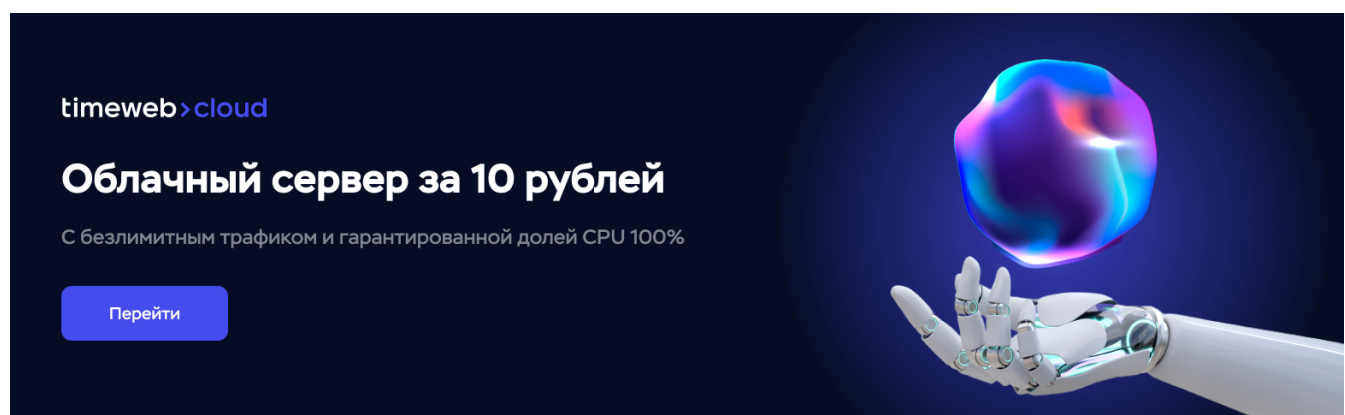
```
process: (sourceText, sourcePath, options) => ({
  code: `module.exports = ${JSON.stringify(path.basename(sourcePath))}`
})
}
```

И настраиваем трансформацию в файле `jest.config.js` :

```
module.exports = {
  testEnvironment: 'jest-environment-jsdom',
  extensionsToTreatAsEsm: ['.jsx'],
  moduleDirectories: ['node_modules', 'testing'],
  // !
  transform: {
    // дефолтное значение, в случае кастомизации должно быть указано явно
    '^.+\\.jsx?$': 'babel-jest',
    // трансформация файлов
    '^.+\\.jsx?$': 'babel-jest',
    '^(?!\\.)(\\.?(png|jpeg|jpg|gif|eot|otf|webp|svg|ttf|woff|woff2|mp4|webm|wav|mp3|
    <rootDir>/testing/file-transformer.js'
  }
}
```

Пожалуй, это все, что я хотел рассказать о тестировании React-компонентов с помощью Jest и Testing Library. Надеюсь, вы нашли для себя что-то интересное и не зря потратили время.

Благодарю за внимание и happy coding!



timeweb > cloud

Облачный сервер за 10 рублей

С безлимитным трафиком и гарантированной долей CPU 100%

[Перейти](#)

Теги: javascript, react.js, reactjs, react, testing, test, testing library, testing library react, jest, тестирование, тест

Хабы: [Блог компании Timeweb Cloud](#), [JavaScript](#), [ReactJS](#) +5 38 2 +2

Редакторский дайджест



Присылаем лучшие статьи раз в месяц

**Timeweb Cloud**

Облачная платформа для разработчиков и бизнеса

**120**

Карма

51.6

Рейтинг

Igor Agarov @aio350

JavaScript Developer

Комментарии 2

**j8kin**

09.06.2022 в 23:04

Мы сверху на jest прикрутили jest-cucumber-fusion и пишем высокоуровневые огуречные тесты.

Жаль нет возможности запускать по сценариям.

Но в остальном с точки зрения тестирования бизнес требований и бизнес логики - очень нравится.

 0[Ответить](#)**vss414**

16.06.2022 в 13:49

В статье есть некоторые конструкции, которые считаются ошибочными по мнению автора React Testing Library. Настоятельно рекомендую к прочтению его статью, где он их подробно разбирает - <https://kentcdodds.com/blog/common-mistakes-with-react-testing-library>.

 0[Ответить](#)

Только полноправные пользователи могут оставлять комментарии. [Войдите](#), пожалуйста.

ПОХОЖИЕ ПУБЛИКАЦИИ

3 февраля в 12:20

React: WebRTC Media Call

◆ +6

👁 5.7K

🔖 54

💬 2 +2

27 января в 12:19

React: Code Editor

◆ +9

👁 7.9K

🔖 46

💬 3 +3

21 января в 12:14

JavaScript: захват медиапотока из DOM элементов

◆ +7

👁 9K

🔖 74

💬 6 +6

ЛУЧШИЕ ПУБЛИКАЦИИ ЗА СУТКИ

вчера в 11:16

Bash отладчик с поддержкой произвольных точек останова

◆ +44

👁 4.1K

🔖 74

💬 5 +5

вчера в 14:40

Новые нули дзета-функции

◆ +31

👁 4K

🔖 14

💬 10 +10

вчера в 21:20

Личный опыт выгорания

◆ +26

👁 8K

🔖 31

💬 23 +23

вчера в 17:00

DIY квантовые вычисления: как я начал собирать квантовые схемы

◆ +21

👁 8.2K

🔖 41

💬 22 +22

вчера в 21:00

Антиматерия и бариогенезис. Три причины, почему нет антивещества, но есть мы

◆ +15

👁 2.8K

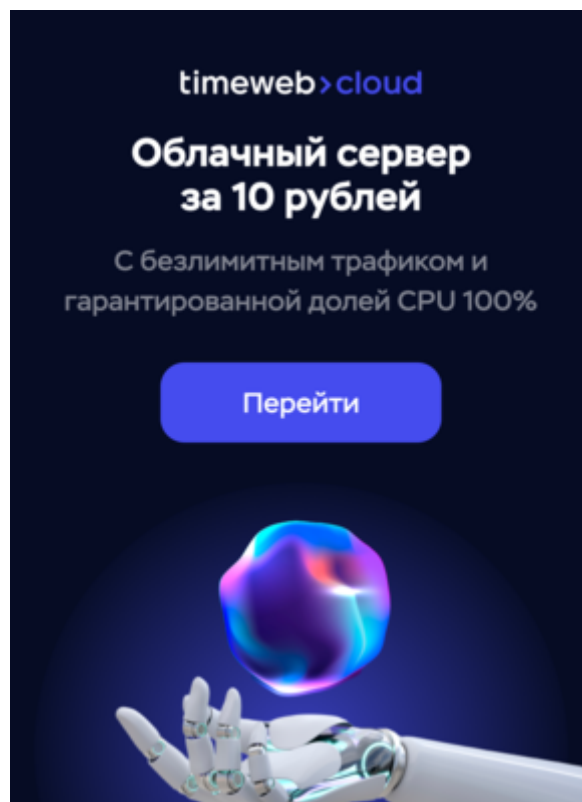
🔖 20

💬 1 +1

ИНФОРМАЦИЯ

Дата основания	25 мая 2006
Местоположение	Россия
Сайт	cloud.timeweb.com
Численность	201–500 человек
Дата регистрации	11 августа 2011

ВИДЖЕТ



timeweb > cloud

**Облачный сервер
за 10 рублей**

С безлимитным трафиком и
гарантированной долей CPU 100%

Перейти

ВИДЖЕТ





ССЫЛКИ

Серверы под любые задачи
cloud.timeweb.com

Облачные базы данных (DBaaS)
cloud.timeweb.com

Объектное S3-хранилище
cloud.timeweb.com

Партнерская программа
timeweb.com

Timeweb News — актуальные новости и скидки
t.me

«Релиз в пятницу» — подкаст от команды Timeweb Cloud
www.youtube.com

Craftum — конструктор сайтов
craftum.com

НОВОСТИ

Развертывание приложений Python с помощью Gunicorn
30 июня

Копирование файлов по SSH

29 июня

Обзор ZeroTier One: работа с программно-конфигурируемой сетью и создание VPN

27 июня

Как установить Docker на Ubuntu

27 июня

MySQL: для чего нужна, как устроена, основные преимущества и недостатки

24 июня

RDP-протокол: что это такое, для чего используется и как работает

24 июня

Установка NextCloud на Ubuntu 20.04

24 июня

Проекты и новая база знаний

23 июня


Установка NextCloud на Ubuntu 20.04

23 июня

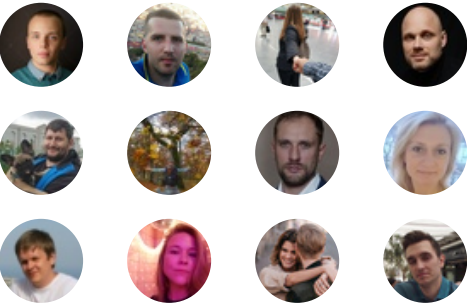
Как настроить Laravel, Nginx и MySQL с помощью Docker Compose

22 июня

ВКОНТАКТЕ

 **Timeweb: всё про хостинг, IT ...**

54,856 followers



☐ Follow on VK

ПРИЛОЖЕНИЯ



Приложение для VDS Evo

Управляйте VDS Evo со своего мобильного в удобное для вас время.

Информация о работе ваших VDS и консультация со службой поддержки теперь доступны на мобильном устройстве.

[Android](#) [iOS](#)



Приложение для хостинга

Управляйте виртуальным хостингом прямо со смартфона. Все данные о хостинге и возможность связаться со Службой поддержки внутри одного приложения для Android и iOS.

[Android](#) [iOS](#)

Ваш аккаунт

Войти

Регистрация

Разделы

Публикации

Новости

Хабы

Компании

Авторы

Песочница

Информация

Устройство сайта

Для авторов

Для компаний

Документы

Соглашение

Конфиденциальность

Услуги

Корпоративный блог

Медийная реклама

Нативные проекты

Образовательные

программы

Мегапроекты



Настройка языка

Техническая поддержка

Вернуться на старую версию

© 2006–2022, Habr

1.6K

2 +2