Оглавление

Глава 1

Тестирование и Отладка

За счет разделения React приложений на компоненты, их очень удобно тестировать. Есть множество инструментов для создания тестов с React, и в этой главе мы разберем самые популярные из них, чтобы понять, какую выгоду мы можем из них извлечь.

Jest - тестовый фреймворк, который поддерживается силами *Кристофером Пожером(Christopher Pojer)* из Facebook и членами сообщества; но ничто не мешает вам решить использовать **Mocha**. Мы посмотрим на оба способа создания лучшего тестового окружения.

Также вы узнаете о разнице между **Поверхностной отрисовкой (Shallow rendering)** и полной отрисовкой DOM с помощью **TestUtils** и **Enzyme**, как работает **Snapshot** тестирование и как собирать информацию о покрытии кода тестами.

После разбора самих инструментов мы перейдем к примеру покрытия тестами компонента из репозитория Redux и посмотрим на распространенные подходы, которые могут быть применены в сложных тестовых сценариях.

После разбора этой главы, вы сможете создать с нуля свое собственное тестовое окружение и написать тесты для компонентов вашего приложения.

В этой главе мы разберем следующие вопросы:

- Почему важно покрывать приложение тесты, и как это ускоряет разработку
- Как настроить окружение с помощью Jest и начать писать тесты с TestUtils

- Как создать такое же тестовое окружение с Mocha
- Что такое Enzyme, и почему он рекомендуется для создания тестов React компонентов
- Как создать тесты для компонента из реального приложения
- Снимки (snapshots) Jest и процент покрытия с помощью библиотеки Istanbul
- Основные способы тестирования компонентов высшего порядка и сложных страниц с множеством дочерних компонентов
- Инструменты разработчика в React и подходы к обработке ошибок

1.1 Польза от тестирования

Тестирование web UI никогда не было простой задачей. Какие бы тесты мы не рассматривали, от модульных (unit) до сквозных (end-to-end), интерфейс всегда зависит от браузеров, взаимодействия с пользователем и множества других параметров, которые затрудняют создание оптимальной стратегии тестирования.

Если вы когда-либо писали сквозные тесты, то должны знать, как трудно получить стабильно воспроизводимые результаты из-за различных факторов влияющих на выполнение тестов (например, нестабильность сети). Помимо этого, пользовательские интерфейсы часто обновляются, чтобы улучшить конверсию или просто добавить новые функции.

Если тесты становится сложно писать и поддерживать, у разработчиков пропадает мотивация в их создании. С другой стороны тесты очень важны, так как увеличивают доверие к коду, что увеличивает скорость и качество разработки. Если какая-то часть кода покрыта хорошими тестами, то разработчик, даже если вносит изменения, может быть уверен, что этот код работает корректно и готов к поставке.

Часто разработчики могут быть сосредоточены над созданием новых возможностей для приложения, и в этот момент им может быть трудно понять, не ломают ли они уже существующий код. Наличие тестов может уберечь от регрессии приложения, так как их падение очень наглядно говорит о поломке в коде. Таким образом тесты добавляют уверенности в

работоспособности кода и уменьшают время, необходимое для его релиза.

Также тесты помогают улучшить качество кода в целом. Даже если обнаруживается какая-либо ошибка в приложении, ничто не мешает, не только исправить эту ошибку, но и создать специальный тест, который воспроизводит эту ошибку. Такой прием позволит в будущем быстрее обнаруживать появление ранее встречаемых ошибок.

К счастью, React упрощает написание тестов для UI. Тестирование отдельных компонентов (или деревьев компонентов) не так трудоемко, по сравнению с полным сквозным тестирование, особенно если компоненты обладают своей строго ограниченной областью ответственности.

А если компоненты не обладают внутренним состоянием, то они могут тестироваться как обычные функции.

Еще одна великолепная возможность, которую принесли современные инструменты, это возможность запускать тесты при помощи Node и консоли. Потребность в запуске браузера для проверки тестов значительно замедляет процесс разработки и ухудшает воспроизводимость тестов; что собственно и исправляется запуском тестов в консоли.

Когда компоненты тестируются только в консоли, могут всплыть неожиданные вещи при из запуске в браузере, но в общем и целом это происходит крайне редко.

Когда мы тестируем React компоненты, мы хотим быть уверены, что они работают корректно для различных комбинаций параметров, которые им можно передать.

Также мы можем захотеть покрыть тестами различные состояния компонента, если таковые есть. Если состояние компонента меняется по нажатию на кнопку или какому-либо другому событию, то мы можем покрыть тестами обработчики событий, чтобы всегда быть уверенными, что они работают так, как должны работать.

После покрытия тестами основного функционала компонента, мы можем покрыть проверить **Пограничные случае (Edge cases)**. К пограничным случаям мы можем отнести ситуации, когда все параметры компонента приняли значение *null*, или когда произошла какая-то ошибка. После того, как все тесты написаны, мы можем быть уверены в достаточной степени, что компонент ведет себя в соответствии с нашими ожиданиями.

Очень важно тестировать компоненты по отдельности, но это не гарантирует, что их совокупность также будет работать корректно. Как

мы увидим далее, с React мы можем отрисовывать дерево элементов и тестировать взаимодействие компонентов внутри этого дерева.

Есть различные подходы в написании тестов, но один из самых популярных - **Разработка через тестирование (Test Driven Development, TDD)**. Использование TDD подразумевает написание тестов перед написанием основного кода, за счет чего мы получаем возможность оценки корректности будущего кода до начала его разраотки.

Следование этому подходу помогает писать код лучше, так как мы задумываемся о его дизайне еще до того, как начнем писать сам код, что обычно ведет к повышению качества.

1.2 Тестирование JavaScript с Jest

Лучший способ научиться тестировать React компоненты... это протестировать React компоненты. Поэтому в этой части мы попробуем написать небольшие компоненты и покрыть их тестами.

Документация React говорит о том, что в Facebook для тестирования используется Jest. Но в общем случае ничто не запрещает вам использовать любой другой тестовый фреймворк.

А в следующей части вы научитесь тестировать компоненты при помощи Mocha.

Чтобы посмотреть, как работает Jest, мы создадим с нуля проект, установим все необходимые зависимости, и создадим компонент, который покроем тестами. Это будет весело!

Начнем с того, что создадим проект в пустой директории:

```
npm init
```

После того, как *package.json* будет создан, мы можем начать устанавливать зависимости. Первой из них будет сам Jest:

```
npm install --save-dev jest
```

Для того, чтобы сказать npm, что мы хотим использовать команду *jest* для запуска тестов, мы должны добавить соответствующую команду в файл *package.json*:

```
"scripts": {
  "test": "jest"
},
```

Для того, чтобы иметь возможность использовать все возможности ES2015 и JSX, мы должны установить Babel с соответствующими плагинами:

```
npm install --save-dev babel-jest babel-preset-es2015
    babel-preset-react
```

Как вы уже можете догадаться, для конфигурации Babel нам понадобится файл *.babelrc*, в котором мы укажем, какие пресеты мы хотим использовать в нашем проекте:

```
{
  "presets": ["es2015", "react"]
}
```

Само собой нам потребуются React и ReactDom, чтобы иметь возможность создавать и запускать React компоненты:

```
npm install --save react react-dom
```

Настройка проекта закончена, мы можем запускать Jest для тестирования ES2015 и JSX кода, а также создавать и отрисовывать компоненты, но есть еще одна вещь, которую необходимо сделать.

Как мы уже сказали, мы хотим запускать тесты в консоли с Node. Но в этом случае мы не можем использовать ReactDOM, так как он требует DOM браузера.

Команда Facebook создала специальный инструмент, который называется *TestUtils*. Этот инструмент позволяет без проблема тестировать React компоненты в любом тестовом фреймворке.

Давайте для начала его установим и посмотрим, какие возможности он предоставляет:

```
npm i --save-dev react-addons-test-utils
```

Теперь у нас есть все необходимое для тестирования компонентов. TestUtils позволяет выполнять поверхностную (shallow) отрисовку компонентов или отрисовывать компоненты в специальный DOM, отделенный от браузера. Также эта библиотека позволяет получать ссылки на компоненты, отрисованные в DOM, для проверки их состояния в целях тестирования.

Также с TestUtils возможно симулировать события браузера для проверки работоспособности обработчиков событий.

Давайте начнем с создания компонента, который в дальнейшем будет покрывать тестами.

Мы создадим компонент *Button*, который будет получать из параметров текст и отрисовывать кнопку с этим текстом. Также в нем будет обработчик событий для этой кнопки. Для начала мы создадим только скелет этого компонента, а затем создадим к нему тесты, чтобы следовать подходу TDD.

Нам будет необходимо создать компонент класс, так как TestUtils на данный момент не умеет работать с функциональными компонентами.

Создадим файл *button.js* и импортируем в нем React:

```
import React from 'react'
```

Теперь мы можем определить сам компонент:

```
class Button extends React.Component
```

В компоненте на данный момент будет только метод render, который будет возвращать пустой *div*:

```
render() {
  return <div />
}
```

И в конце добавим экспорт этого компонента:

```
export default Button
```

Компонент подготовлен к покрытию тестами, теперь мы можем создать файл *button.spec.js* и приступить к написанию тестов.

Jest ищет тесты во всех файлах, которые оканчиваются на *.spec* и *.test*, а также во всех файлах директории *__tests__*; но вы можете изменить это поведение в настройках Jest, если этого требует ваш проект.

В начале файла *button.spec.js* мы импортируем все необходимые зависимости:

```
import React from 'react'
import TestUtils from 'react-addons-test-utils'
import Button from './button'
```

Нам нужен React, чтобы писать JSX код, TestUtils, на который мы посмотрим немного дальше, и только что созданный компонент *Button*.

Для начала создадим простейший тест, чтобы убедиться, что система тестирования в принципе функционирует:

```
test('works', () => {
  expect(true).toBe(true)
})
```

Функция *test* принимает два параметра: описание теста и функцию с реализацией самого теста. Внутри мы используем функцию *expect* для передачи Jest объекта, относительно которого мы хотим выполнить предсказание. Функция *expect* возвращает объект с методами, которые позволяют конкретизировать предсказание. Например, функция *toBe* проверяет, что переданный объект в точности соответствует заданному.

Теперь мы можем выполнить в терминале команду:

```
npm test
```

Вы должны увидеть следующий результат:

```
 PASS  ./button.spec.js
     works (3ms)
 Test Suites: 1 passed, 1 total
 Tests: 1 passed, 1 total
 Snapshots: 0 total
 Time: 1.48s
 Ran all test suites.
```

Если вы увидели в выводе в консоль слово PASS, вы готовы для создания реальных тестов.

Как мы уже сказали, с помощью тестов мы хотим убедиться, что компонент корректно обрабатывает полученные параметры, а обработчики событий выполняют свою работу.

Существует два основных способа тестировать React компонента:

- Поверхностная отрисовка
- Монтирование компонентов в специальный DOM

Начнем с первого из них, так как он проще для понимания. При поверхностной отрисовке, как можно догадаться из названия, отрисовывается не все дерево компонентов, а только его верхняя часть *высотой 1*, относительно которой мы можем выполнять различные проверки.

Отрисовка одного уровня дерева означает, что мы будет тестировать исходный компонент независимо от дочерних сколь сложны они бы не были. Таким образом, отрисовка дочерних компонентов не будет проводиться в принципе, поэтому они никак не смогут повлиять на результаты теста.

Первый тест, который мы можем сделать, это проверить, что переданный компоненту текст отрисовывается внутри кнопки.

Для начала создадим сам тест:

```
test('renders with text', () => {
```

Создадим переменную с заданным текстом, которую мы будем передавать в параметры проверяемого компонента:

```
const text = 'text'
```

И теперь можно выполнить поверхностную отрисовку компонента, для чего достаточно следующих трех строк:

```
const renderer = TestUtils.createRenderer()
renderer.render(<Button text={text} />)
const button = renderer.getRenderOutput()
```

Сначала мы создаем *renderer*, с помощью которого отрисовываем компонент *Button*, и в последней строчке получаем результат отрисовки.

Результат отрисовки будет выглядеть примерно следующим образом:

```
{
  '$$typeof': Symbol(react.element),
  type: 'button',
  key: null,
  ref: null,
  props: { onClick: undefined, children: 'text' },
  _owner: null,
  _store: {}
}
```

Если долго и пристально вглядываться, то можно увидеть в этом объекте React элемента. Его параметр *props* отвечает за переданные элементу параметры, в том числе за дочерние (атрибут *children*) элементы.

Теперь мы знаем, как выглядит результат отрисовки, а значит может легко проверить, что отрисовалась именно кнопка, а дочерним элементом является значение переменной *text*:

```
expect(button.type).toBe('button')
expect(button.props.children).toBe(text)
```

И не забудем в конце закрыть все скобки:

```
})
```

Теперь, если вы запустите в консоли команду:

```
npm test
```

Вы должны увидеть что-то следующего вида:

```
FAIL  ./button.spec.js
    renders with text
    expect(received).toBe(expected)
    Expected value to be (using ===):
      "button"
    Received:
      "div"
```

Тест упал, чего мы вообще говоря и ожидали, так как запустили тест для еще не реализованного компонента в соответствии с TDD подходом. Теперь мы можем вернуться к компоненту и поправить метод *render* так, чтобы компонент проходил данный тест:

```
render() {
  return (
    <button>
      {this.props.text}
    </button>
  )
}
```

Теперь при запуске тестов нас должна встретить зеленая галочка:

```
PASS  ./button.spec.js
    renders with text (9ms)
    Test Suites: 1 passed, 1 total
    Tests:
    Snapshots:
    Time:
    Ran all test suites.
```

Поздравляю! Ваш первый тест для компонента, написанный в соответствии с TDD, выполнился успешно.

Теперь давайте посмотрим, как проверить, что компонент получил обработчик событий *onClick*, и что этот обработчик вызывается при нажатии на кнопку.

Но перед тем, как мы начнем, нужно рассказать про две концепции: моки (mock) и откреплённый (detached) DOM.

Первая упрощает проверку работы функций внутри теста. В данном тесте мы хотим передать компоненту функцию через параметр *onClick* и проверить, что функция вызывается, когда происходит нажатие на кнопку.

Для того, чтобы сделать это, нам нужно создать специальную **мок (mock)** функцию (в других фреймворках такая функция может иметь

другое название, например *spy*). Такая функция работает как обыкновенная, но расширена дополнительными возможностями.Например, можно проверить, сколько раз и с какими параметрами была вызвана функция.

Для того, чтобы создать мок функцию при помощи Jest, мы можем использовать *jest.fn()*.

Вторую концепцию нам нужно разобрать из-за того, что мы не можем симулировать события DOM с помощью *TestUtils* при поверхностной отрисовке.

Это происходит из-за того, что для тестирования событий с *TestUtils*, нам нужны реальные компоненты, а не React элементы.

Поэтому, для тестирования событий браузера, нам необходимо отрисовать наш компонент в откреплённый DOM. Отрисовка компонента в полноценный DOM требует наличия браузера, но вместе с Jest идет специальный DOM, в который можно что-то отрисовать из консоли.

Отрисовка компонента в откреплённый DOM несколько отличается от поверхностной отрисовки, поэтому давайте посмотрим, как это будет выглядеть в коде.

Для начала создадим новый тест:

```
test('fires the onClick callback', () => {
```

Создадим мок функцию *onClick* при помощи Jest:

```
const onClick = jest.fn()
```

Теперь мы отрисуем компонент в DOM полностью:

```
const tree = TestUtils.renderIntoDocument(
  <Button onClick={onClick} />
)
```

Если мы распечатаем *tree* в консоль, то увидим не React элемент, а полноценный компонент.

Из-за этого мы уже не можем просто проверить, что вернула функции *renderIntoDocument*, но с помощью специального метода *TestUtils* мы можем получить элемент кнопки, которая нас интересует:

```
const button = TestUtils.findRenderedDOMComponentWithTag(
  tree,
  'button'
)
```

Как можно догадаться из названия функции, она ищет внутри дерева элемент с заданным тегом.

Теперь мы можем воспользоваться другим методом из *TestUtils* для симуляции события:

```
TestUtils.Simulate.click(button)
```

Объект *Simulate* предоставляет функции, которые имеют названия, аналогичные названиям событий, и принимают один параметр для цели события.

И в конце выполняем проверку того, что функция была вызвана:

```
expect(onClick).toBeCalled()
```

То есть мы просто проверяем, что *мок* функция была вызвана.

Если мы снова запустим тесты, то увидим сообщение об ошибке, что ожидаемо, так как мы еще не реализовали работу функции *onClick*:

```
FAIL  ./button.spec.js
  fires the onClick callback
  expect(jest.fn()).toBeCalled()
  Expected mock function to have been called.
```

Именно так мы и работает при TDD подходе. Теперь вернемся в файл *button.js* и реализуем обработчик событий:

```
render() {
  return (
    <button onClick={this.props.onClick}>
      {this.props.text}
    </button>
  )
}
```

Теперь тесты должны показывать зеленый свет:

```
PASS  ./button.spec.js
    renders with text (10ms)
    fires the onClick callback (17ms)
Test Suites: 1 passed, 1 total
Tests: 2 passed, 2 total
Snapshots: 0 total
Time: 1.401s, estimated 2s
Ran all test suites.
```

Теперь наш компонент полностью протестирован и реализован в соответствии с написанными тестами.

1.3 Гибкий тестовый фреймворк Mocha

В этой части мы проделаем то же самое, чтобы показать, что вы можете использовать с React любой тестовый фреймворк по вашему желанию. Также будет полезным увидеть разницу между интегрированным фреймворком Jest, который старается все автоматизировать для более плавного использования (прим.пер. smooth developer experience что бы это не значило), и Mocha, который не делает никаких предположений относительно того, какие инструменты вам нужны. С Mocha вы можете установить любые библиотеки, которые вам нужны для тестирования React компонентов.

Для начала создадим новый npm проект в пустой директории:

```
npm init
```

И добавим саму библиотеку *mocha*:

```
npm install --save-dev mocha
```

Так же как и для Jest, чтобы писать ES2015 код и JSX, нам потребуется Babel с парой плагинов:

```
npm install --save-dev babel-register babel-preset-es2015
   babel-preset-react
```

Теперь, после установки Mocha и Babel, мы можем добавить скрипт для запуска тестов:

```
"scripts": {
  "test": "mocha --compilers js:babel-register"
},
```

Мы говорим *npm*, что для выполнения команды *test* необходимо запустить *mocha* с флагом *compilers* (для предварительного прогона исходного кода через *Babel*).

Теперь добавим React и ReactDOM:

```
npm install --save react react-dom
```

А также *TestUtils*, который позволяет нам отрисовывать компоненты в тестовом окружении:

```
npm install --save-dev react-addons-test-utils
```

Базовый инструментарий для работы с Mocha готов, но для того, чтобы привести все в соответствие с Jest, нам понадобится еще паратройка библиотек.

Первая из них - *chai*, которая позволяет писать проверки в том же стиле, что и в Jest. Вторая - *chai-spies*, с которой мы можем проводить шпионские операции для проверки функций, таких как *onClick*.

И последняя библиотека *jsdom* позволяет нам создавать откреплённый DOM, чтобы TestUtils могли отрисовывать компоненты без реального браузера:

```
npm install --save-dev chai chai-spies jsdom
```

Теперь мы готовы приступить к написанию тестов, для чего можем использовать созданный ранее файл *button.js*. Мы уже реализовали компонент, поэтому мы не будем следовать TDD, но сейчас наша главная задача состоит в том, чтобы увидеть разницу между двумя тестовыми фреймворками.

Mocha ожидает, что тесты будут находиться в директории *test*, поэтому мы можем создать ее и файл *button.spec.js* внутри нее.

В начале файла импортируем все необходимые зависимости:

```
import chai from 'chai'
import spies from 'chai-spies'
import { jsdom } from 'jsdom'
import React from 'react'
import TestUtils from 'react-addons-test-utils'
import Button from '../button'
```

Как вы можете заметить, после тестов с Jest, необходимо импортировать больше различных библиотек. Это связано с тем, что Mocha предоставляет вам самим выбрать, какие вспомогательные инструменты вам нужны.

Далее необходимо указать библиотеке *chai* использовать *spies*:

```
chai.use(spies)
```

Сразу вытащим пару функция из *chai*, которые потребуются нам далее в тестах:

```
const { expect, spy } = chai
```

Далее мы создадим экземпляр *jsdom* и установим его как DOM для отрисовки компонентов:

```
global.document = jsdom('')
global.window = document.defaultView
```

И вот теперь мы можем создать первый тест. Обычно с Mocha (прим.пер. как и с Jest на самом деле) используется две функции для написания

тестов: *describe*, которая описывает набор тестов, и *it*, внутри которой непосредственно описываются тесты.

В данном случае мы описываем поведение кнопки:

```
describe('Button', () => {
```

И затем мы создаем первый тест, в котором проверяем, что у компонента правильные тип и текст:

```
it('renders with text', () => {
```

Создадим переменную с текстом, которую будем использовать для проверки текста в кнопке:

```
const text = 'text'
```

Далее сделаем поверхностную отрисовку компонента, как мы делали это до этого:

```
const renderer = TestUtils.createRenderer()
renderer.render(<Button text={text} />)
const button = renderer.getRenderOutput()
```

И в конце выполним проверки:

```
expect(button.type).to.equal('button')
expect(button.props.children).to.equal(text)
```

Как вы можете заметить, есть небольшие синтаксические различия. Вместо функциил *toBe* появилась *to.equal* из библиотеки *chai*. Но результат одинаковый: сравнение двух значений.

Не забудем закрыть скобки для первого теста:

```
})
```

В следующем тесте мы будем проверять, что вызывается функция обратного вызова *onChange*:

```
it('fires the onClick callback', () => {
```

Создадим мок функцию с помощью *spy*, аналогично тому, как до этого создавали при помощи *jest.fn*:

```
const onClick = spy()
```

Отрисовываем компонент в откреплённый DOM при помощи *TestUtils*:

```
const tree = TestUtils.renderIntoDocument(
  <Button onClick={onClick} />
)
```

А с помощью *tree* мы можем найти нужный элемент в дереве:

```
const button = TestUtils.findRenderedDOMComponentWithTag(
  tree,
  'button'
)
```

Следущий шаг - симуляция нажатия кнопки:

```
TestUtils.Simulate.click(button)
```

И последним шагом можно проверить, была ли вызвана функция:

```
expect(onClick).to.be.called()
```

И снова, хоть синтаксис немного и поменялся, но общая идея осталась той же, мы просто проверяем у функции, была ли она вызвана.

Теперь, если мы запустим *npm test* в корневой директории, то увидим следующее сообщение:

```
Button
  renders with text
  fires the onClick callback

2 passing (847ms)
```

Это означает, что наш тест выполнился успешно, а мы готовы использовать Mocha для тестирования наших компонентов.

1.4 JavaScript инструменты для тестирования React

На данный момент вы должны понимать, как тестировать компоненты при помощи Jest и Mocha, а также плюсы и минусы обоих подходов.

Также вы познакомились с TestUtils и узнали о двух способах отрисовки компонент: поверхностном и в откреплённый DOM.

Однако вы могли заметить, что с TestUtils не всегда легко получить доступ к нужным элементам и их параметрам.

По этой причине разработчики из *AirBnb* создали Enzyme, тестовый фреймворк, который работает поверх TestUtils и урощает работу с отрисованными компонентами.

API Enzyme приятно и схоже с jQuery, а также предоставляет удобные методы для работы с компонентами, их состоянием и параметрами.

Давайте посмотрим, как будут выглядеть уже написанные нами тесты, если мы перепишем их с Enzyme.

Давайте вернемся к проекту, где мы писали тесты с Jest, и добавим в него Enzyme:

```
npm install --save-dev enzyme
```

Теперь откроем файл *button.spec.js* и поправим в нем импорты следующим образом:

```
import React from 'react'
import { shallow } from 'enzyme'
import Button from './button'
```

Как вы можете увидеть, вместо *TestUtils* мы импортируем *shallow* из Enzyme. Из названия можно понять, что она выполняет поверхностную отрисовку компонента, но также обладает дополнительными возможностями.

Прежде всего, Enzyme позволяет эмулировать события даже при поверхностной отрисовке компонентов, чего мы не могли сделать с *TestUtils*. И помимо этого, *shallow* из Enzyme возвращает не просто React элемент, а **обертку (ShallowWrapper)** над ним, специальный объект с дополнительными параметрами и методами, которые мы разберем чуть дальше.

Давайте начнем с теста, который называется **renders with text**. Первая строка, где мы определяем переменную с текстом, остается той же самой:

```
const text = 'text'
```

Поверхностная отрисовка компонента становится интуитивно понятнее и выразительнее. Три строки кода с использованием *TestUtils* мы можем заменить одной:

```
const button = shallow(<Button text={text} />)
```

Объект *button* представляет собой обертку над React элементом со вспомогательными методами, которые мы можем использовать для выполнения проверок:

```
expect(button.type()).toBe('button')
expect(button.text()).toBe(text)
```

Теперь, вместо проверки параметров React элемента, названия которых могут измениться, мы используем библиотечные функции, которые абстрагируют внутри себя поиск нужных параметров (прим.пер. особенно если библиотека вовремя обновляется).

Функция *type* проверяет тип элемента, а функиця *text*, соответственно, текст внутри элемента. В нашем случае, это тот текст, который мы передали через параметры.

Теперь весь тест должен выглядеть следующим образом:

```
test('renders with text', () => {
  const text = 'text'
  const button = shallow(<Button text={text} />)

  expect(button.type()).toBe('button')
  expect(button.text()).toBe(text)
})
```

Теперь тест выглядит лаконичнее и чище чем до этого.

Теперь поправим тест, который проверяет работу события *onClick*. Снова, первая строчка остается той же:

```
const onClick = jest.fn()
```

Мы можем также использовать мок функции из Jest для проверки срабатывания обработчиков событий.

Мы можем заменить строку, где мы использовали *renderIntoDocument* для отрисовки компонента в откреплленный DOM следующей:

```
const button = shallow(<Button onClick={onClick} />)
```

Нам не нужно использовать *findRenderedDOMComponentWithTag* для поиска кнопки, так как *shallow* и так возвращает ссылку на нее.

Синтаксис для вызова эмуляции события немного отличается от *TestUtils*, но все также интуитивно понятен:

```
button.simulate('click')
```

У каждой обертки есть метод *simulate*, который принимает имя события и допольнительные аргументы, которые в данном случае нам не нужны, но мы их разберем, когда будем разбираться с тестированием форм.

Проверка, была ли вызвана функция, остается той же:

```
expect(onClick).toBeCalled()
```

Весь код теста выглядит следующим образом:

```
test('fires the onClick callback', () => {
  const onClick = jest.fn()
  const button = shallow(<Button onClick={onClick} />)
```

```
  button.simulate('click')
  expect(onClick).toBeCalled()
})
```

Переход на Enzyme предельно прост, и при этом значительно улучшает читаемость кода.

Библиотека предоставляет множество полезных методом, таких как поиск вложенных элементов или поиск элементов по имени класса.

Есть методы для выполнения проверок параметров у компонентов, а также установка конкретных состояния или контекста.

Помимо поверхностной отрисовки, которой в случае Enzyme хватает для большинства случаем, библиотека предоставляет метод *mount*, который отрисовывает компонент в DOM.

1.5 Пример тестов из реального мира

На данный момент мы разобрались, как настроить тестовое окружение и посмотрели на разные тестовые фреймворки. Пришло время посмотреть на тестирование компонентов из реального мира.

Компонент *Button* из предыдущего примера был великолепен, и мы должны стремиться сохранять компоненты настолько простыми, насколько это возможно. Но порой нам приходится реализовывать какую-либо логику внутри компонентов, а также хранить состояние, что несколько усложняет задачу тестирования.

На этот раз мы собираемся протестировать компонент **TodoTextInput** из примера Redux **TodoMVC**:

```
https://github.com/reactjs/redux/blob/master/examples/
   todomvc/src/components/TodoTextInput.js
```

Вы можете скопировать его в ваш *Jest* проект.

Это отличный пример для написания тестов, так как у компонента есть несколько параметров, его имя класса (прим.пер. я буду называть именем класса *className* для CSS, чтобы не путать с ключевым словом *class* JavaScript) меняется в соответствии с полученными параметрами, а также у него есть три обработчика с небольшим количеством логики, которую также стоит протестировать.

TodoMVC - пример создания *стандартного* приложения при помощи различных фреймворков для их сравнения, что должно помочь разработчика в выборе между ними.

В результате у нас есть простенькое приложение, в котором можно добавлять задачи (to-do) и отмечать их выполнение. Для наших целей мы возьмем компонент, который отвечает за поле ввода для создания и редактирования задач.

Имеет смысл, сначала пробежаться по коду самого компонента, чтобы понимать, что именно мы собираемся тестировать.

Начинается компонент с определения соответствующего ему класса:

```
class TodoTextInput extends Component
```

В данном случае *propTypes* определены при помощи свойства класса:

```
static propTypes = {
  onSave: PropTypes.func.isRequired,
  text: PropTypes.string,
  placeholder: PropTypes.string,
  editing: PropTypes.bool,
  newTodo: PropTypes.bool
}
```

Для того, чтобы свойства класса поддерживались с Babel, необходимо добавить еще один плагин:

```
  npm install --save-dev babel-plugin-transform-class-
     properties
```

И затем добавить этот плагин к остальным плагина в *.babelrc*:

```
"plugins": ["transform-class-properties"]
```

Состояние компонента также определено через свойство класса:

```
state = {
  text: this.props.text || ''
}
```

Значение по умолчанию может быть пустой строкой или установлено из параметра *text (this.props.test)*.

Далее идут три обработчика событий, каждый из которых представляет из себя стрелочную функцию (поэтому нет необходимости прикреплять их к экземплярам класса в конструкторе), которая также сохранена как свойство класса.

Первый из них - обработчик окончания ввода (*submit*):

```
handleSubmit = e => {
  const text = e.target.value.trim()
  if (e.which === 13) {
```

```
    this.props.onSave(text)
    if (this.props.newTodo) {
      this.setState({ text: '' })
    }
  }
}
```

Функция получает объект события, проверяет, что была нажата клавиша *Enter (13)*, убирает пробелы из введенной строки и сохраняет при помощи функции *this.props.onSave*. Если параметр *newTodo* - *true*, то сбрасывает поле ввода для создания новой задачи.

Следующий обработчик будет отслеживать изменения в поле ввода:

```
handleChange = e => {
  this.setState({ text: e.target.value })
}
```

Помимо того, что этот обработчик также определен через свойство класса, можно отметить, что оно сохраняет значение контролируемого поля ввода внутри состояния компонента.

И последний обработчик для отслеживания момента, когда пользователь убирает фокус с поля ввода (*blur*):

```
handleBlur = e => {
  if (!this.props.newTodo) {
    this.props.onSave(e.target.value)
  }
}
```

Он вызывает функцию *onSave*, если значение параметра *newTodo* равно *false*.

И в конце находится метод *render*, в котором определен элемент *input* со всеми этими параметрами:

```
render() {
  return (
    <input className={
      classnames({
        edit: this.props.editing,
        'new-todo': this.props.newTodo
      })}
      type="text"
      placeholder={this.props.placeholder}
      autoFocus="true"
      value={this.state.text}
      onBlur={this.handleBlur}
```

```
        onChange={this.handleChange}
        onKeyDown={this.handleSubmit} />
    )
}
```

Для применения нужного имени класса используется функция *classnames*, которая создана *Джедом Уотсоном (Jed Watson)* и удобна для расчета имени класса, которое зависит от различных логических выражений.

Также установлены несколько статических атрибутов (*type* и *autofocus*), через атрибут *text* передается текст для управления значением поля ввода, и через соответствующие атрибуты переданы обработчики событий.

Перед тем, как начать, стоит понять, что именно мы собираемся тестировать и почему. Глядя на этот компонент, несложно выделить наиболее важные для покрытия тестами части. В данном случае, вы можете думать о данном компоненте как о коде, пришедшем в наследство от других команд (legacy code), или как о коде, который вы можете найти в новой компании.

Следущий список отражает функционал компонента, который в большей или меньшей степени подходит для покрытия тестами:

- Состояние компонента проинициализировано значением, пришедшим в параметрах
- Параметр *placeholder* корректно передается в поле ввода
- Применяется правильное имя класса
- Состояние компонента изменяется при вводе данных пользователем
- Функция *onSave* вызывается корректно для различных состояний и условий

Пришло время начать писать код. Мы начнем с того, что создадим файл *TodoTextInput.spec.js* со следующими импортами:

```
import React from 'react'
import { shallow } from 'enzyme'
import TodoTextInput from './TodoTextInput'
```

Мы импортируем сам React, функцию *shallow* из Enzyme и компонент, который будет тестировать. Также создадим функцию, которую будет передавать в параметр *onSave* в некоторых тестах:

```
const noop = () => {}
```

Теперь мы можем создать первый тест, в котором проверим, что значение по умолчанию устанавливается из полученных параметров:

```
test('sets the text prop as value', () => {
  const text = 'text'
  const wrapper = shallow(
    <TodoTextInput text={text} onSave={noop} />
  )
  expect(wrapper.prop('value')).toBe(text)
})
```

Здесь все просто: создаем текстовую переменную, а затем выполняем поверхностную отрисовку компонента с передачей в него этой переменной. Также мы передаем функцию *noop* в параметр *onSave*, так как этот параметр является необходимым для компонента.

Далее мы выполняем проверку того, что значение в полученном элементе идентично значению переменной. Теперь, если мы запустим тест, то должны получить следующий результат:

```
PASS  ./TodoTextInput.spec.js
  sets the text prop as value (10ms)
Test Suites:  1 passed, 1 total
Tests:        1 passed, 1 total
Snapshots:    0 total
Time:         1.384s
Ran all test suites.
```

Великолепно, продолжим писать тесты. Следующий тест будет очень похож на предыдущий за тем исключением, что мы будем проверять значение свойства *placeholder*:

```
test('uses the placeholder prop', () => {
  const placeholder = 'placeholder'
  const wrapper = shallow(
    <TodoTextInput placeholder={placeholder} onSave={noop} />
  )
  expect(wrapper.prop('placeholder')).toBe(placeholder)
})
```

Можно запустить тесты, оба должны светить зеленым светом.

Попробуем написать что-нибудь поинтереснее. Например, проверим, что имя класса меняется в соответствии полученным параметрам:

```
test('applies the right class names', () => {
  const wrapper = shallow(
```

```
    <TodoTextInput editing newTodo onSave={noop} />
  )
  expect(wrapper.hasClass('edit new-todo')).toBe(true)
})
```

В этом тесте мы добавили компоненту два параметра (*editing* и *newTodo*), а затем проверили, что соответствующие классы добавились в имя класса.

Было бы лучше проверить каждый из классов отдельно, но идея должна быть понятна.

Следующий тест будет несколько сложнее, потому что теперь мы хотим проверить реакцию компонента на событие нажатия клавиши (key down).

Проверим, что при нажатии клавиши *Enter*, вызывается функция *onSave* с текущим значением поля ввода:

```
test('fires onSave on enter', () => {
  const onSave = jest.fn()
  const value = 'value'
  const wrapper = shallow(<TodoTextInput onSave={onSave} />)

  wrapper.simulate('keydown', { target: { value }, which: 13
     })

  expect(onSave).toHaveBeenCalledWith(value)
})
```

Сначала мы создаем мок функцию при помощи *jest.fn()*, далее создаем переменную для установки значения поля ввода и отрисовываем компонент. После этого мы симулируем событие *keydown* с кодом клавиши *Enter (13)*.

У объекта события есть два параметра: *target*, который отражает элемент, инициировавший событие, и *which*, в котором находится код нажатой клавиши.

И в конце выполняем проверку, что функция *onSave* была вызвана со значением поля ввода.

Теперь *npm test* должен сказать, что 4 теста прошли успешно.

С помощью теста, похожего на предыдущий, мы можем проверить, что при нажатии отличной от *Enter* клавиши функция *onSave* не вызывается:

```
test('does not fire onSave on key down', () => {
  const onSave = jest.fn()
```

```
  const wrapper = shallow(<TodoTextInput onSave={onSave} />)

  wrapper.simulate('keydown', { target: { value: '' } })

  expect(onSave).not.toBeCalled()
})
```

Тест очень похож на предыдущий за исключением выполняемый проверки, в которой на этот раз используется параметр *.not*. Как можно догадаться, таким образом мы говорим, что ожидаем *false* из вызова функции *toBeCalled*.

Как вы могли заметить, синтаксис вызова проверок очень похож на разговорный язык.

У нас уже есть 5 зеленых тестов, двигаемся к следующему:

```
test('clears the value after save if new', () => {
  const value = 'value'
  const wrapper = shallow(<TodoTextInput newTodo onSave={noop
     } />)

  wrapper.simulate('keydown', { target: { value }, which: 13
     })

  expect(wrapper.prop('value')).toBe('')
})
```

Отличие на этот раз в том, что мы передали параметр *newTodo*, который заставляет сбрасываться значение поля ввода по нажатию клавиши *Enter*.

Следущий тест:

```
test('updates the text on change', () => {
  const value = 'value'
  const wrapper = shallow(<TodoTextInput onSave={noop} />)

  wrapper.simulate('change', { target: { value } })

  expect(wrapper.prop('value')).toBe(value)
})
```

Этот тест проверяет, что контролируемое поле ввода работает корректно. Если в вашем приложение есть формы, то вам обязательно нужны такого рода тесты.

Мы симулируем событие *change* со значением *value*, а затем проверяем, что значение поля ввода изменилось соответствующим образом.

Теперь у нас есть 7 зеленых тестов, и остался только один.

В последнем тесте мы проверим, что событие *blur* вызывает функцию обратного вызова только в том случае, если это не новый элемент:

```
test('fires onSave on blur if not new', () => {
  const onSave = jest.fn()
  const value = 'value'
  const wrapper = shallow(<TodoTextInput onSave={onSave} />)

  wrapper.simulate('blur', { target: { value } })

  expect(onSave).toHaveBeenCalledWith(value)
})
```

Все как раньше: создали мок функцию, контрольное значение, вызвали событие *blur*, проверили, что функция *onSave* была вызвана с контрольным значением.

Если мы запустим тесты теперь, то должны увидеть чуть более объемистый вывод:

```
PASS  ./TodoTextInput.spec.js
  sets the text prop as value (10ms)
  uses the placeholder prop (1ms)
  applies the right class names (1ms)
  fires onSave on enter (3ms)
  does not fire onSave on key down (1ms)
  clears the value after save if new (5ms)
  updates the text on change (1ms)
  fires onSave on blur if not new (2ms)
Test Suites:  1 passed, 1 total
Tests:        8 passed, 8 total
Snapshots:    0 total
Time:         2.271s
Ran all test suites.
```

Отличная работа, теперь компонент покрыт тестами. Теперь, если мы захотим изменить поведение компонента или добавить новые возможности, тесты покажут, где мы сломали старый функционал.

Это делает нас более уверенными в нашем коде, поэтому мы можем изменить любую строчку кода без страха, что сломаем старый функционал.

1.6 Snapshot-тестирование React компонентов

После того, как вы увидели, как много тестов нужно написать для покрытия одного компонента, вы можете подумать, что на это будет уходить слишком много времени и в этом нет смысла.

Проверка всех комбинаций текста, значения поля ввода и имени класса довольно трудоемкий процесс, который требует написания большого количество кода. Однако, в большинстве случаев, для нас важно, что содержимое компонента не претерпит неожиданных изменений от изменений в коде. Для решения этой проблемы неплохо подходит **Snapshot-тестирование (Snapshot Testing)**.

Snapshot - это по сути *снимок (picture)* компонента с определенными параметрами. Каждый раз, когда мы запускаем тесты, Jest создает снимок компонента и проверяет с эталонным на наличие изменений.

Содержимое снимка - это результат работы метода *render* пакета *react- test-renderer*, который необходимо установить:

```
npm install --save-dev react-test-renderer
```

После установки пакета, создадим новый файл *TodoTextInput–snapshot.spec.js* со следующими импортами:

```
import React from 'react'
import renderer from 'react-test-renderer'
import TodoTextInput from './TodoTextInput'
```

Импортируем React, чтобы использовать JSX, *renderer*, чтобы создавать дерево для снимков, и компонент, который мы собираемся тестировать.

Теперь мы можем создать простенький тест:

```
test('snapshots are awesome', () => {
```

В первой строке теста отрисуем компонент при помощи *renderer*:

```
const component = renderer.create(
  <TodoTextInput onSave={() => {}} />
)
```

1.7 Code coverage tools

1.8 Common testing solutions

1.8.1 Testing Higher-Order Components

1.8.2 The Page Object pattern

1.9 React Dev Tools

1.10 Error handling with React

1.11 Заключение