Оглавление

Глава 1

Все, что нужно знать о React

Эта книга предполагает, что вы уже знаете, что такое React, и какие проблемы он может помочь решить. Скорее всего вы уже пробовали его использовать и создавали небольшие приложения, но хотите улучшить свои навыки и найти ответы на открытые вопросы.

Вы должны знать, что React поддерживается разработчиками Facebook и множеством участников JavaScript сообщества.

React - одна из самых популярных библиотек для создания пользовательских интерфейсов, которая достигает хорошей производительности благодаря оптимизированное работе с DOM.

Вместе с React мы также получаем язык разметки JSX, который заставит вас изменить свое отношение к разделению ответственности. Также с ним идет множество полезных инструментов как рендеринг на стороне сервера (server-side rendering), которое позволяет создавать Универсальные приложения (Universal web applications).

Для полноценного изучения книги вам нужны будут базовые навыки работы с терминалом для установки *npm* пакетов.

Также все примеры кода будут написаны на стандарте ES2015 языка JavaScript, поэтому вы должны уметь читать его и понимать.

В этой главе мы разберем базовые концепции React, которые необходимо понимать для эффективного использования библиотеки, но которые могут быть не так очевидны начинающим разработчикам:

- Разницу между императивным и декларативным программированием

- React компоненты и их экземпляры, а также, как React использует элементы для описания пользовательского интерфейса
- Как React меняет подход к созданию web приложений, делая упор на концепцию разделения ответственности
- Какие проблемы испытывают люди при разработке на JavaScript, и как вы можете избежать большинство распространенных ошибок при работе с React.

1.1 Декларативное программирование

Если вы будете читать документацию, статьи или блоги по React, то вы точно неоднократно встретите слово **декларативный**.

В декларативном подходе разработки на React кроется один из секретов его применимости.

Следовательно, для того, чтобы заниматься React профессионально, необходимо понимать разницу между императивным и декларативным подходами к разработке.

Один из простейших способов почувствовать эту разницу, думать об императивном программировании как о процессе описания "как что-либо должно работать а о декларативном программировани как о процессе описания "что вы хотите получить в результате".

Мы можем провести параллель с реальной жизнью, например походом в бар. В этом случае поход в бар в императивном стиле будет выглядеть следующим образом:

- Возьмите стакан с полки
- Поставте стакан под кран бочки с пивом
- Тяните ручку пока стакан не наполнен
- Передайте мне стакан

В декларативном стиле вам потребуется сказать: "Пива мне".

Следуя декларативному стилю вы предполагаете, что бармен сам знает, как налить вам пиво. Это важный принцип декларативного программирования, давайте посмотрим, как он работает на примере JavaScript.

Предположим, что нам нужно написать функцию, которая принимает единственным аргументом массив строк, а возвращает массив этих же строк, но в нижнем регистре:

```
toLowerCase(['FOO', 'BAR']) // ['foo', 'bar']
```

В императивном стиле проблема может быть решена следующим образом:

```
const toLowerCase = input => {
  const output = []
  for (let i = 0; i < input.length; i++) {
    output.push(input[i].toLowerCase())
  }
  return output
}
```

Сначала будет создан пустой массив для хранения результата работы функции. Затем мы в цикле обходим строки, переданные функции, и добавляем их в нижнем регистре в созданный ранее массив. Затем мы возвращаем созданный массив как результат работы функции.

В декларативном стиле решение может выглядеть следующим образом:

```
const toLowerCase = input => input.map(
  value => value.toLowerCase()
)
```

В данном решении исходный массив передается в функцию *map*, которая принимает аргументом функцию преобразования элементов и сама создает результирующий массив.

Нужно обратить внимание на важное различие в двух вариантах кода: в первом случае код сложнее и труднее для восприятия, в то время как во втором случае он краток и выразителен. Это лишь пример одной функции, на масштабах большой кодовой базы такая разница сильно влияет на поддерживаемость проекта.

Помимо этого в декларативном варианте нет необходимости создавать переменные для хранения данных и изменять их значение в процессе работы функции. Таким образом декларативный подход помогает избежать создания, и что важно изменения множества сущностей.

И теперь давайте посмотрим, что значит для кода на React быть декларативным.

Решим часто встречаемую web разработчикам задачу: показать карту с маркером на ней.

Решение на JavaScript (с использование Google Maps SDK) может выглядеть следующим образом:

```
const map = new google.maps.Map(document.getElementById('map'
    ), {
  zoom: 4,
  center: myLatLng,
})
const marker = new google.maps.Marker({
  position: myLatLng,
  title: 'Hello World!',
})
marker.setMap(map)
```

Это достаточно императивное решение так как в коде мы напрямую используем все инструкции для создания карты, создания маркера и их объединению.

В свою очередь реализация на React выглядела бы следующим образом:

```
<Gmaps zoom={4} center={myLatLng}>
  <Marker position={myLatLng} Hello world! />
</Gmaps>
```

В данном случае мы лишь описывает, как должен выглядеть компонент, но не описывает детально все шаги для достижения этого результата.

1.2 Элементы React

В этой книге мы предполагаем, что вы уже знакомы с компонентами React и имеете представление о том, как их создавать и использовать. Но если вы хотите использовать React эффективнее вы должны знать еще об одной сущности: **Элемент (the Element)**

Независимо от того, используете ли вы *createClass*, наследуете *React.Component* или создаете компонент-функцию, вы создаете компонент. React управляет всеми компонентами во время исполнения программы и для каждого компонента может быть создано множество его экземпляров в один момент времени.

Как было сказано выше, React следует декларативной парадигме, и нет необходимости дополнительно описывать, как React должен взаимо-

действовать с DOM. Вам нужно лишь описать, что вы хотите увидеть на экране, и React сделает остальное за вас.

Если у вас есть опыт работы с другими UI библиотеками, то вы могли заметить, что многие из них работают противоположенным способом: они оставляют ответственность за обновление интерфейса на разработчике, который вынужден управлять созданием и удалением DOM элементов вручную.

Для поддержания UI в актуальном состоянии, React использует специальный тип объектов, **элемент(element)**, который описывает, что должно быть показано на экране. Это неизменяемый (immutable) объект, проще и легче компонент, и содержащий ровно ту информацию, которая необходима для отображения интерфейса.

Пример просто элемента выглядит следующим образом:

```
{
  type: Title,
  props: {
    color: 'red',
    children: 'Hello, Title!'
  }
}
```

В объекте содержатся тип элемента (*type*) и параметры. Также есть специальный опциональный атрибут *children*, в котором содержатся непосредственные потомки данного элемента.

Тип важен для элемента, так как он сообщает React, как обрабатывать данный элемент. Если тип является строкой, тогда React создает на его основе **вершину DOM** в DOM дереве, а если функцией, тогда React представляет его как **React компонент**.

DOM элементы и компоненты могут быть вложенными друг в друга для отображения дерева элементов:

```
{
  type: Title,
  props: {
    color: 'red',
    children: {
      type: 'h1',
      props: {
        children: 'Hello, H1!'
      }
    }
  }
```

```
}
```

Если тип элемента - функция, то React вызывает ее, передавая необходимые параметры, для получения нижележащих элементов. React проделывает эту операцию рекурсивно пока не получит все DOM вершины, которые React уже способен отрисовать на экране. Этот процесс называется **сверкой (reconciliation)** и используется и React Dom и React Native для построения пользовательского интерфейса.

1.3 Забудьте, что вы знали

Разработка на React принесла множество новых парадигм и приемов, сломав множество устоявшихся практик. Поэтому, если вы встречаете React впервые, вам нужно быть открытым к новым подходам в разработке.

Последние два десятилетия мы делали упор на разделении ответственности как на разделении логики и шаблонов. Мы всегда стремились писать HTML верстку и JavaScript код в разных файлах.

Чтобы помочь разработчикам достигнуть этой цели было создано множество шаблонизаторов.

Но проблема такого разделения в том, что на самом деле HTML и JavaScript все равно остаются сильно связанными. Посмотрим на небольшой пример:

```
{{#items}}
  {{#first}}
    <li><strong>{{name}}</strong></li>
  {{/first}}
  {{#link}}
    <li><a href="{{url}}">{{name}}</a></li>
  {{/link}}
{{/items}}
```

Этот пример взят с сайта **Mustache**, одного из самых популярных шаблонизаторов.

Первая строка говорит Mustache, что необходимо обойти в цикле коллекцию элементов (items). Внутри цикла есть несколько условных операторов для проверки, что параметры $\#fist$ и $\#link$ существуют. И в зависимости от существования этих параметров отображаются разные куски HTML. Переменные находятся внутри двойных фигурных скобок.

Если вам нужно отрисовать содержимое лишь нескольких переменных, то шаблонизатор может быть хорошим решением. Но все сильно меняется, если вам нужно работать со сложными структурами данных.

По факту шаблонизаторы с их Предметно-ориентированными языками (Domain-Specific Language, DSL) предоставляют некоторый набор функций, пытаясь быть похожими на полноценные языки программирования, но абсолютно не достигая их полноты.

Как показано на примере выше, шаблоны сильно зависимы от формата данных, которые приходят с уровня логики. С другой стороны JavaScript все равно взаимодействует с DOM элементами, созданными шаблонизатором, для обновления UI.

Та же проблема относится к стилям: они определены в отдельных файлах, но шаблоны ссылаются на них, а CSS стили следуют структуре разметки. Невозможно изменить одно, не сломав другое, что говорит об их сильной связности.

Классическое разделение зависимостей в web является скорее разделением технологий, а не непосредственно зависимостей. Это не плохо само по себе, но решает меньше задач, чем нам возможно хотелось бы.

React пытается сделать шаг вперед и перенести шаблоны ближе к логике. А разделение ответственности осуществляется посредством того, что React рекомендует разделять приложение на небольшие строительные блоки, **компоненты**.

Фреймворк не должен говорить вам, как разделить ваш код, поскольку каждое приложение имеет собственную структуру, и разработчики могут лучше решить, где стоит проводить линию между независимыми блоками.

Компонентно-ориентированный подход коренным образом меняет подход к созданию веб приложений, постепенно вытесняя старые методы.

Парадигма, продвигаемая React, далеко не нова и не была изобретена создателями библиотеки. Но заслуга создателей библиотеки в том, что они сделали этот подход более популярным, а также создали комфортное окружение, чтобы библиотекой смогло овладеть большое множество разработчиков с разным уровнем опыта.

Так выглядит метод *render* React компонента:

```
render() {
  return (
    <button style={{ color: 'red' }} onClick={this.
       handleClick}>
```

```
      Click me!
    </button> )
}
```

Поначалу этот код может выглядеть странным, но это ожидаемо, так как мы не привыкли использовать такой синтаксис.

Когда мы изучим его и разберемся в его возможностях, мы сможем понять его потенциал.

Использование JavaScript и для логики и для шаблонизации позволяет не только лучше разделить области ответственности кода, но и за счет большей выразительности позволяет создавать более сложные приложения.

Поэтому, даже если поначалу идея смешать в одну кучу JavaScript и HTML звучит для вас странно, стоит потратить немного времени для знакомства с React.

Один из лучших способов изучить библиотеку, сделать небольшое приложение и посмотреть как это работает. В общем случае всегда стоит быть готовым отбросить текущие навыки и изучить новые парадигмы, если это принесет пользу в будущем.

Следует обратить внимание, что есть один спорный момент относительно стилей. Основная цель этой идеи - объединить все технологии создания web приложений внутри JavaScript для еще большей гибкости в разделении ответственности на основе логики приложения.

Посмотрим на пример создания стилей внутри JavaScript:

```
var divStyle = {
  color: 'white',
  backgroundImage: 'url(' + imgUrl + ')',
  WebkitTransition: 'all', // note the capital 'W' here
  msTransition: 'all' // 'ms' is the only lowercase vendor
     prefix
};
ReactDOM.render(
  <div style={divStyle}>Hello World!</div>,
  mountNode
);
```

Такой подход называется $\#CSSinJS$, подробнее о ней мы поговорим в Главе 7.

1.4 Распространенные заблуждения

Есть распространенное мнение, что React - это большой набор технологий и инструментов, и если вы хотите использовать его, то вы вынуждены иметь дело с различными пакетными менеджерами, трансляторами кода, сборщиками бандлов и огромным множеством других библиотек.

Эта мысль настолько разошлась в массы, что даже получила собственное название **Усталость от JavaScript (JavaScript Fatigue)**.

В общем-то несложно понять, почему так происходит. Все новые репозитории и библиотеки в экосистеме React создаются с использованием новых технологий, последней версией JavaScript и продвинутыми практиками и парадигмами.

Кроме того, на GitHub появилось множество готовых заготовок проектов с десятками зависимостей для решения любой проблемы.

Очень легко начать думать, что все эти зависимости Необходимы для разработки на React, но на самом деле это далеко от правды.

Вопреки этому мнению React является очень легковесной библиотекой и может быть использован на любой странице (даже в JSFiddle) таким же образом как jQuery или Backbone, просто добавлением скрипта в HTML код страницы.

Справедливости ради стоит сказать, что необходимо добавлять на страницу два скрипта, так как React разделен на две библиотеки: $react$, в котором находится ядро с основной логикой библиотеки, и $react-dom$, в котором находятся специфичные для браузеров функции. Это было сделано для того, чтобы React мог использоваться в отличных от браузера средах, например с React Native на мобильных устройствах.

Запуск React внутри простой HTML страницы не требует дополнительных менеджеров пакетов и сложных телодвижений. Вам достаточно скачать пакет с React и разместить его у себя (или использовать *unpkg.com*). С этого момента вы можете пользоваться всеми возможностями React.

Для того, чтобы начать работу с React нужно добавить следующие две ссылки в HTML:

- https://unpkg.com/react/dist/react.min.js
- https://unpkg.com/react-dom/dist/react-dom.min.js

Если мы добавим только библиотеку React, мы не сможем использовать JSX, который не является частью языка. Но основная цель - начать работу с React с минимальным набором инструментов и расширять их по мере необходимости.

Для простого интерфейса мы можем использовать *createElement*, а транслятор JSX можно добавить лишь тогда, когда нам понадобится создать что-то более сложное.

Когда наше приложение будет становиться еще сложнее, нам может понадобиться роутер (router) для переключения страниц.

В какой-то момент нам может потребоваться загружать данные с различных API, наше приложение будет расти и нам могут потребоваться сторонние библиотеки. В этот момент можно подключить пакетный менеджер.

Затем нам скорее всего потребуется разделить приложение на различные модули, чтобы у проекта была понятная структура. В этот момент стоит задуматься о сборщике бандла.

Если следовать этим простым правилом, то усталость от JavaScript обойдет вас стороной.

Важно заметить, что любая область профессии разработчика (front end в том числе) требует непрерывного обучения. Web развивается семимильными шагами в соотвествии с растущими требования пользователей и разработчиков. По такому принципу Интернет развивается с его появления и в том числе это добавляет ему великолепия.

Также удивительным является то, что с появлением новой спецификации языка, сообщество сразу создает транслятор новых возможностей в старые версии языка, позволяя разработчиком попробовать их до того, как они будут полноценно добавлены в браузеры.

Это то, что сильно отличает экосистему JavaScript и Web от других языков программирования.

Обратная сторона медали в том, что знания и инструменты устаревают очень быстро, поэтому следует искать баланс между новыми технологиями и безопасностью работы.

Но по крайней мере разработчики React беспокоятся об удобстве разработчиков и прислушиваются к мнению сообщества. Даже несмотря на то, что React не требует глубоко изучения множества инструментов для начала работы, создатели библиотеки осознали, что многим достаточно трудно сделать первые шаги в ее изучении, и выпустили консольное приложение для создания и запуска базового проекта, чтобы максимально

упростить вход в React разработку.

Единственное требование - наличие окружения с установленным *node.js/npm*. С помощью него мы без проблем можем установить консольное приложение:

```
npm install -g create-react-app
```

После установки приложения мы можем создать новый проект, запустив это приложение и передав название проекта:

```
create-react-app hello-world
```

После этого мы можем открыть созданную папку командой *cd hello-world* и запустить командой:

```
npm start
```

Таким образом за пару шагов мы запустили полноценное React приложение с минимальным набором самых актуальных инструментов.

Мы будем использовать этот инструмент на протяжении всей книги для запуска примеров кода, которые можно найти на:

https://github.com/MicheleBertoli/react-design-patterns-and-best-practices

1.5 Заключение

В первой главе мы посмотрели на базовые концепции React, которые нам понадобятся на протяжении всей книги и о которых не стоит забывать в ежедневной разработке.

Теперь мы знаем, как писать декларативный код, и понимаем разницу между компонентами и элементами, используемыми React для отображения экземпляров компонент в браузере.

Также мы посмотрели, почему React пошел по пути объединения логики и шаблонов вместе, и как это стало большой победой для React.

Помимо этого мы узнали про явление Усталости от JavaScript и поговорили немного о ее причинах и способах ухода от нее.

И в конце мы разобрались с консольным инструментом *create−react−app*, с которым мы сможем легко и быстро начать использовать React.

Глава 2

Делаем код чище

Чтобы использовать JSX без проблем, необходимо понимать как он работает под капотом и почему его удобно использовать для создания интерфейса.

Наша цель - писать чистый и поддерживаемый JSX код, и для этого мы должны знать, как JSX транслируется в JavaScript код и какие предоставляет фичи.

В этом блоке мы разберем:

- Что такое JSX и почему мы должны его использовать
- Что такое Babel и как он используется в современной разработке
- Основные фичи JSX и его отличие от HTML
- Лучшие практики при написании кода на JSX
- Статическую проверку кода с ESLint
- Основы функционального программирования и его применение в создании React компонент

2.1 JSX

**Example of JSX

React предлагает два основных способа описания элементов: использование библиотечных JavaScript функций и использование JSX разметки, похожей на XML.

На первый взгляд может показаться, что JSX - это странная смесь из HTML и JavaScript. Но на деле, JSX разметка перед попаданием в браузер конвертируется в чистый JavaScript. Этот прием позволяет достаточно лаконично и визуально понятно описывать компоненты.

2.1.1 Babel

Чтобы использовать JSX(а также фичи ES2015) нам необходимо установить Babel.

Важно понимать, почему мы добавляем Babel в наш процесс разработки. Основная причина в желании использовать фичи языка, которые еще не доступны в браузере. Новые фичи языка часто помогают нам писать код чище и понятнее, но браузер не может их исполнить.

Решение - писать код с JSX и ES2015, а затем транслировать в ES5, который сейчас могут запускать большинство браузеров, используя Babel.

Чтобы использовать Babel, его необходимо установить:

```
npm install --global babel-cli
```

Чтобы не загромождать npm пакетами систему, можно установить babel в конкретный проект и использовать через npm скрипты. Но для учебных целей установим его глобально.

После установки мы можем транслировать любой JavaScript файл:

```
babel source.js -o output.js
```

Одна из сильных сторон Babel - возможность его гибкой конфигурации. Babel всего лишь транслирует один файл в другой, а чтобы были произведены изменения содержимого, нужно сконфигурировать этот процесс.

Для Babel уже создано множество пресетов, в том числе и для JSX и ES2015. Чтобы установить их, необходимо выполнить:

```
npm install --global babel-preset-es2015 babel-preset-react
```

а также добавить в домашнюю директорию (или в папку с проектом) файл .babelrc с содержимым:

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

С этого момента мы можем спокойно использовать все фичи ES2015 и JSX, а потом запускать в браузере транслированный Babel'ем код.

2.1.2 Hello, World

Посмотрим на простейший пример создание элемента в React.

Мы можем создать *div* элемент с помощью метода *createElement* библиотеки React:

```
React.createElement('div')
```

Также мы можем создать его, используя JSX:

```
<div />
```

В данном примере JSX код выглядит как HTML. Но нужно понимать одну важную вещь, оба варианта, написанные выше, эквивалентны.

На самом деле, если мы транслируем в JavaScript при помощи babel, то мы получим React.createElement('div'). Это необходимо всегда держать в голове при описании интерфейса на React.

2.1.3 DOM элементы и React компоненты

С JSX мы можем создавать и HTML элементы и React элементы. Разница лишь в том, пишем мы название элемента с заглавной буквы или нет.

Например, в JSX мы можем создать <button /> и <Button /> элементы.

В первом случае результатом трансляции будет:

```
React.createElement('button')
```

Во втором случае:

```
React.createElement(Button)
```

Разница в том, что в первом случае тип DOM элемента передается как строка, а во втором случае мы передаем название переменной, которая должна быть как минимум определена в области видимости данного кода.

2.1.4 Props

JSX очень удобен, если у DOM или React компонентов есть параметры. В XML в целом гораздо нагляднее передавать параметры элементам:

```
<img src="https://facebook.github.io/react/img/logo.svg"
   alt="React.js" />
```

Аналогичный код на JavaScript:

```
React.createElement("img", {
     src: "https://facebook.github.io/react/img/logo.svg",
     alt: "React.js"
});
```

Читаемость резко падает даже с небольшим количеством параметров.

2.1.5 Children

JSX позволяет определять дочерние элементы, чтобы создавать комплексные древовидные разметки.

Простой пример дочернего элемента - текст внутри тега:

```
<a href="https://facebook.github.io/react/">Click me!</a>
```

Этот пример будет транслирован в:

```
React.createElement(
  "a",
  { href: "https://facebook.github.io/react/" },
  "Click me!"
);
```

Если этот тег будет обернут в другой, например div, то JSX будет выглядеть как:

```
<div>
  <a href="https://facebook.github.io/react/">Click me!</a>
</div>
```

JavaScript эквивалентный этому JSX:

```
React.createElement(
  "div",
  null,
  React.createElement(
    "a",
    { href: "https://facebook.github.io/react/" },
    "Click me!"
```

```
  )
);
```

Достаточно очевидно, что чем сложнее разметка, тем больше JSX улучшает читаемость кода. Однако не следует забывать, что каждому JSX коду соответствует однозначно определенный код на JavaScript.

Так как JSX всего лишь удобный синтаксис для JavaScript, вполне логично, что в нем можно использовать JavaScript выражения.

Для того, чтобы сделать это, выражение должно быть обернуто в фигурные скобки:

```
<div>
  Hello, {variable}.
  I'm a {function()}.
</div>
```

Или например в параметрах элемента:

```
<a href={this.makeHref()}>Click me!</a>
```

2.1.6 Differences with HTML

Мы посмотрели, чем похожи JSX и HTML. Теперь посмотрим в чем они отличаются и в чем причины этих различий.

Аттрибуты

Мы должны помнить, что JSX не стандарт языка, и транслируется в JavaScript. Из-за этого некоторые атрибуты не доступны для использования.

Например, вместо атрибута class мы вынуждены использовать className, а вместо for использовать htmlFor:

```
<label className="awesome-label" htmlFor="name" />
```

Причина этого в том, что слова class и for зарезервированы в языке JavaScript.

Стили

Стили - пример значительных различий между HTML и JSX. Подробнее мы посмотрим на них в одной из следующих глав.

Сейчас отметим, что через JSX в атрибуте style не поддерживается CSS строка. Вместо нее React ожидает JavaScript объект, в котором имена стилей переданы в camelCase:

```
<div style={{ backgroundColor: 'red' }} />
```

Root

Стоит отметить важное отличие JSX от HTML, которое заключается в том, что нельзя создать несколько элементов на одном уровне без оборачивания другим элементом:

```
<div />
<div />
```

Этот пример вызовет следующую ошибку:
Adjacent JSX elements must be wrapped in an enclosing tag
Проблема решается добавлением общего элемента:

```
<div>
  <div />
  <div />
</div>
```

Это происходит из-за того, что каждый элемент в JSX транслируется в React.createElement в JavaScript, а в JavaScript нельзя вернуть из функции результат работы двух последовательных вызовов какой-либо функции.

Сейчас React предоставляет возможность использовать пустой тег, чтобы отрисовывать несколько элементов на одном уровне:

```
render() {
  return (
    <>
      <ChildA />
      <ChildB />
      <ChildC />
    </>
  );
}
```

Spaces

Есть еще одна мелочь, которая может вводить в ступор новичков, которая заключается в различной обработке пробельных символов в HTML

и JSX.

Например, рассмотрим следующий пример кода, который корректен и в HTML и в JSX:

```
<div>
  <span>foo</span>
  bar
  <span>baz</span>
</div>
```

Если открыть этот кусочек напрямую в браузере как HTML файл, то мы увидим foo bar baz. А если мы добавим его в JSX, то после отрисовки будет foobarbaz.

Это происходит из-за того, что JSX воспримет три строки внутри div, как три дочерние элементы и проигнорирует пробельные символы, а после отрисовке все три дочерних элемента будут отрисованы один за другим.

Основной способ исправить это, добавить еще дочерние элементы, которые будут также являться дочерними элементами:

```
<div>
  <span>foo</span>
  {' '}
  bar
  {' '}
  <span>baz</span>
</div>
```

Таким образом мы добавляем пустые строки, которые являются JavaScript выражениями, чтобы заставить компилятор добавить новые дочерние элементы.

Boolean атрибуты

Также есть небольшое отличие в использовании boolean атрибутов в JSX. Если передать какой-либо атрибут без значения, то JSX поймет, что это boolean атрибут со значением true:

```
<button disabled />

React.createElement("button", { disabled: true });
```

Но в отличие от HTML, чтобы передать атрибут со значением false, необходимо сделать это в явном виде. Если не передать атрибут совсем,

то он не попадет в передаваемый объект с атрибутами, и при дальнейшей попытке использования может быть получен undefined вместо false, что приводит к потенциальным ошибкам:

```
<button disabled={false} />

React.createElement("button", { disabled: false });
```

Эта особенность может вводить в заблуждение, так как в HTML принято отсутствие атрибута считать как false значение для этого атрибута. В React следует всегда явным образом указывать значение boolean атрибутов.

2.1.7 Spread атрибут

Важная особенность JSX - **spread атрибуты**, которая приходит из стандарта ECMASript (https://github.com/sebmarkbage/ecmascript-re st-spread) и очень удобно в случае, когда нам нужно передать все атрибуты JavaScript объекта в параметры элементу.

Распространенная практика - избежать передачи объекта дочерним элементам по ссылке во избежании ошибок, связанных с изменяемостью таких объектов.

В качестве примера можем посмотреть на код:

```
const foo = { id: 'bar' }
return <div {...foo} />
```

который будет транслирован в следующий JSX код:

```
var foo = { id: 'bar' };
return React.createElement('div', foo);
```

2.1.8 Шабоны JavaScript

Мы начали с предположения, что одно из преимуществ использования шаблонов внутри наших компонент вместо использования сторонних библиотек шаблонов(прим.пер. видимо имеются ввиду библиотеки-шаблонизаторы как handlebars) в использовании всей мощи языка JavaScript внутри шаблонов.

Spread оператор один из примеров использования JavaScript внутри JSX. Но в целом любое JavaScript выражение может быть использовано

как атрибут элемента, для этого достаточно обернуть его в фигурные скобки:

```
<button disabled={errors.length} />
```

2.1.9 Основные паттерны

Мы разобрались с тем, как работает JSX. Теперь мы можем можем подумать детальнее, как использовать JSX, следуя полезным соглашениям и практикам.

Многострочный JSX код

Начнем с простого примера. Одна из причин использовать JSX вместо React.createElement - наглядность XML-like синтаксиса, а также потому что такая структура из отрывающих и закрывающих тегов идеально подходит для описания древовидных структур.

Например, если у нас есть JSX код с множеством вложенных элементов, мы должны предпочитать многострочную запись JSX кода однострочной:

```
<div>
  <Header />
  <div>
    <Main content={...} />
  </div>
</div>
```

Такой вариант гораздо предпочтительнее, чем:

```
<div><Header /><div><Main content={...} /></div></div>
```

Однако, если дочерний элемент - не React элемент, а текст или переменная, то разумнее будет записать весь тег в одной строке:

```
<div>
  <Alert>{message}</Alert>
  <Button>Close</Button>
</div>
```

Также рекомендуется оборачивать все JSX блоки в круглые скобки. Это необходимо делать, чтобы не было проблем с автоматической вставкой точки с запятой. Проблемы могут возникнуть, если например, немного не аккуратно вернуть. JSX разметку из функции.

Следующий пример отработает корректно, так как return и div находятся на одной линии:

```
return <div />
```

Но следующий уже отработает непредвиденным образом:

```
return
     <div />
```

Проблема кроется в том, что во втором случае JSX код будет транслирован в следующий JavaScript код:

```
return;
React.createElement("div", null);
```

Чтобы избежать таких проблем, рекомендуется всегда оборачивать многострочные JSX элементы в круглые скобки:

```
return (
  <div />
)
```

Multi-properties

Небольшой проблемой является также множество атрибутов у элемента. Если записывать атрибуты в одну строку, то она может начать занимать много места в ширину и быть неудобной для чтения.

Поэтому стоит стараться писать каждый атрибут в новой строке, а закрывающий тег выравнивать с открывающим (прим.пер. также хороший вариант - оставлять закрывающий тег на одной строке с последним атрибутом):

```
<button
  foo="bar"
  veryLongPropertyName="baz"
  onSomething={this.handleSomething}
/>
```

Условные операторы

Все гораздо интереснее с использование **условий** в JSX, например, если нужно отрисовать какой-то компонент только при каком-то условии. С

одной стороны возможность использовать JavaScript внутри JSX - большой плюс, но с другой стороны у нас появляется множество вариантов использования условий и нужно знать об их плюсах и минусах.

Предположим, что нам нужно отрисовать кнопку logout только для авторизованного пользователя. Простой вариант решения этой проблемы может выглядеть следующим образом:

```
let button
if (isLoggedIn) {
  button = <LogoutButton />
}
return <div>{button}</div>
```

Это работает, но этот вариант плохо читается, если у нас есть множество условий и множество элементов.

Также мы можем использовать ленивую проверку логических выражений в JavaScript:

```
<div>
  {isLoggedIn && <LoginButton />}
</div>
```

Это работает, так как в случае false в isLoggedIn JavaScript не будет проверять остальное выражение, а если true, тогда вызовется createElement для LoginButton и результат ее работы вернется как результат всего выражения.

Если мы хотим, чтобы в условии была также альтернативная ветка, например чтобы показать разные кнопки для авторизованного и неавторизованного пользователей, то мы может использовать if...else в JavaScript:

```
let button
if (isLoggedIn) {
  button = <LogoutButton />
} else {
  button = <LoginButton />
}
return <div>{button}</div>
```

Помимо этого мы можем использовать тернарный оператор, чтобы сделать код компактнее:

```
<div>
  {isLoggedIn ? <LogoutButton /> : <LoginButton />}
</div>
```

Можно найти множество примеров использования тернарных операторов в известных репозиториях, например в Redux(https://github.com/reactjs/redux/blob/mas
world/src/components/List.js#L25), где он используется для показа разного текста в зависимости от того, грузятся ли данные по сети:

```
<button [...]>
  {isFetching ? 'Loading...' : 'Load More'}
</button>
```

Посмотрим, что произойдет, если логическое выражение будет становиться сложнее и в нем будут задействованы разные переменные и логические операции:

```
<div>
  {dataIsReady && (isAdmin || userHasPermissions) &&
    <SecretData />
}
</div>
```

Это решение все еще может быть неплохим, но читаемость его начинает падать. Для решения этой проблемы можно создать вспомогательную функцию внутри компонента и использовать ее название для пояснения логики, сокрытой в теле:

```
canShowSecretData() {
  const { dataIsReady, isAdmin, userHasPermissions } = this.
     props
  return dataIsReady && (isAdmin || userHasPermissions)
}
<div>
  {this.canShowSecretData() && <SecretData />}
</div>
```

Код стал более читаемым, и даже если вернуться к нему через полгода, достаточно будет прочесть название функции и опустить чтение логики внутри.

Если вы не любите использовать функцию, то ее можно заменить геттеро, который сделат код более элегантным (прим.пер. а может быть и нет...):

```
get canShowSecretData() {
  const { dataIsReady, isAdmin, userHasPermissions } = this.
     props
  return dataIsReady && (isAdmin || userHasPermissions)
}
<div>
```

```
  {this.canShowSecretData && <SecretData />}
</div>
```

То же самое относится к вычисляемым значениям. Например, если у нас есть два значения, валюта и стоимость, и нам нужно объединить их в одну строку, то мы можем вынести эту операцию в отдельный метод.

```
getPrice() {
  return `${this.props.currency}${this.props.value}`
}
<div>{this.getPrice()}</div>
```

Это также удобнее тестировать, если внутри этого метода есть дополнительная логика.

То же самое можно сделать с помощью геттеров аналогично предыдущим примерам:

```
get price() {
  return `${this.props.currency}${this.props.value}`
}
<div>{this.price}</div>
```

Есть еще множество решений проблемы ветвления внутри React компонент, которое требует использования сторонних библиотек. В общем случае стоит аккуратно вносить новые зависимости в проект, так как они могут увеличить размер бандла, принести проблему на смене версий и увеличить порог входа в проект, но на что не пойдешь ради увеличения читаемости кода (прим.пер. используемые тут библиотеки довольно просты и имеет смысл реализовать их самостоятельно в учебных целях).

Первый вариант - использовать библиотеку render-if, которую можно установить командой:

```
npm install --save render-if
```

Мы можем легко использовать ее в проекте по аналогии с примером ниже:

```
const { dataIsReady, isAdmin, userHasPermissions } = this.
    props
const canShowSecretData = renderIf(
  dataIsReady && (isAdmin || userHasPermissions)
)
<div>
  {canShowSecretData(<SecretData />)}
</div>
```

Мы оборачиваем наше условии внутри renderIf функции.

Результатом вызова функции renderIf является функция, которая принимает аргументом React элемент, который она вернет при вызове в случае истинности условия.

Самое главное, что мы не должны забывать в данном контексте, это то, что компоненты должны оставаться простыми и глупыми настолько насколько это возможно. Иначе в этих компонентах будет сложно разбираться, править баги и расширять.

Чтобы сделать компонент чище, мы можем вынести из него логику в Компонент более высокого порядка (Higher-Order Component, HOC) с библиотекой react-only-if. Компоненты выского порядка мы рассмотрем далее в Главе 4, сейчас для нас важно только то, что HOC это функция, которая принимает аргументом компонент, расширяет его или изменяет его поведения и возвращает при вызове.

Данная библиотека устанавливается командой:

```
npm install --save react-only-if
```

После установки мы можем использовать ее внутри нашего приложения следующим образом:

```
const SecretDataOnlyIf = onlyIf(
  ({ dataIsReady, isAdmin, userHasPermissions }) => {
    return dataIsReady && (isAdmin || userHasPermissions)
  }
)(SecretData)
<div>
  <SecretDataOnlyIf
    dataIsReady={...}
    isAdmin={...}
    userHasPermissions={...}
  />
</div>
```

Как можно заметить, в этом случае внутри исходного компонента нет логики совсем, что повышает его тестируемость.

Мы передаем условие как параметр в функцию onlyIf, которая меняет поведение нашего компонента таким образом, чтобы оно отображалось только в случае истинности логического выражения.

Циклы

Отображения списков элементов - очень распространенная операция. И в данном случае JavaScript показывает себя с хорошей стороны.

Если мы поместим внутрь JSX массив с React элементами, то все они будут отрисованы на одном уровне вложенности. В целом для нас не важно, как будет получен этот массив, главное, чтобы как и любое другое выражение, было помещено в фигурные скобки.

Самым распространенным способом создать массив элементов является использование операций над множествами объектов языка JavaScript:

```
<ul>
  {users.map(user =><li>{user.name}</li>)}
</ul>
```

Этот пример прост, но показывает большую гибкость использования JSX и JavaScript для генерации HTML.

Control statements

Условные операторы и циклы часто используются для описания верстки и, возможно, вам может показаться, что вносить в JSX блоки JavaScript кода для таких базовых операция - не лучшая практика. Но JSX был разработан лишь как инструмент генерации элементов, оставляя работы с логикой программы на JavaScript.

В общем и целом не стоит держать большой объем логики внутри компонент, но тем не менее время от времени нам нужно скрывать или показывать элементы в зависимости от состояния или итерировать коллекции объектов для отображения списков.

Если вы чувствуете, что JSX должен также позволять использовать условия и циклы, и это сделает код более читаемым, то вы можете попробовать библиотеку: jsx-control-statements.

Эта библиотека не приносит никакого нового функционала в JSX и являешься лишь синтаксическим сахаром, который компилируется в JavaScript.

Прежде всего нам нужно добавить ее в проет:

```
npm install --save jsx-control-statements
```

Также его необходимо добавить в .babelrc, чтобы babel знал, что у нас появились новые правила компиляции:

```
"plugins": ["jsx-control-statements"]
```

С этого момента мы можем использовать новый синтакс и babel будет транслировать его вместе со стандартным JSX.

Условный оператор, написанный с использование этого плагина, будет выглядеть следующим образом:

```
<If condition={this.canShowSecretData}>
  <SecretData />
</If>
```

Этот код будет транслирован в обычный тернарный оператор в JavaScript:

```
{canShowSecretData ? <SecretData /> : null}
```

Для ситуации, когда нам нужно иметь возможность выбрать элемент из нескольких в зависимости от различных условий, в данной библиотеке есть компонент Choose:

```
<Choose>
  <When condition={...}>
    <span>if</span>
  </When>
  <When condition={...}>
    <span>else if</span>
  </When>
  <Otherwise>
    <span>else</span>
  </Otherwise>
</Choose>
```

Не стоит забывать, что компоненты Choose, When и Otherwise не являются React компонентами в привычном для нас понимании, это всего лишь синтаксис, который будет скомпилирован в большой набор тернарных операторов.

Также есть специальный компонент For (который также будет скомпилирован в JavaScript код) для работы с коллекциями объектов:

```
<ul>
  <For each="user" of={this.props.users}>
    <li>{user.name}</li>
  </For>
</ul>
```

Тут тоже никакой магии, после компиляции этот блок превратится в вызов метода map.

Если вы используете какой-либо линтер, то он может ругаться в последнем случае, так как переменная user по сути не определена. Это происходит из-за того, что объявление этой переменной будет сгенерировано после компиляции.

Если вы используете eslint, то для исключения данной ошибки проверки кода можно использовать библиотеку eslint- plugin-jsx-control-statements.

Если у вас еще нет опыта использования линтеров, то не беспокойтесь, они будут разобраны чуть позже.

Sub-rendering

В общем случае стоит стараться делать компоненты маленькими и простыми насколько это возможно, но тем не менее компоненты могут начать разбухать, особенно если разработка идет итеративно и функционал наращивается понемногу на каждой итерации.

Что мы можем сделать, если наши методы отображения компонент становятся слишком большими. Один из вариантов - разделить большой метод render на небольшие функции внутри одного компонента:

```
renderUserMenu() {
    // JSX for user menu
}
renderAdminMenu() {
    // JSX for admin menu
}
render() {
  return (
    <div>
      <h1>Welcome back!</h1>
      {this.userExists && this.renderUserMenu()}
      {this.userIsAdmin && this.renderAdminMenu()}
    </div>
  )
}
```

Это далеко не идеальное решение, но на практике, если нет возможности разделить компонент на более мелкие, это позволяет сохранять метод render чище.

Теперь мы должны начать чувствовать себя посвободнее в использовании JSX. Можно перейти к вопросу, как следовать единому стилю кода внутри всего проекта.

2.2 ESLint

Мы всегда пытаемся писать лучший код, на который способны, но все равно время от времени делаем ошибки и тратим часы на их поиск и исправление.

К счастью для нас, есть инструменты, которые позволяют искать ошибки еще на стадии написания кода. Такие инструменты не скажут, делает ли код то, что должен, но как минимум помогут избежать синтаксических ошибок.

Если вы пришли из мира языков со статической типизацией, таких как C#, то вы привыкли, что множество синтаксических ошибок можно обнаружить в IDE во время написания кода.

Дуглас Крокфорд (Douglas Crockford) сделал линтинг (статическую проверку кода) популярным в мире JavaScript с инструментом JSLint(первый релиз в 2002). Дальше этот инструмент перерос в JSHint, а затем в ESLint, став основным инструментом статической проверки кода в JavaScript мире в целом и в React разработке в частности.

ESLint - инструмент с открытым исходным кодом, вышедший в 2013 году и быстро набравший популярность за счет гибкости в настройке и расширении.

В мире JavaScript, где постоянно меняются библиотеки и подходы, возможность гибкой конфигурации стала одной из главных причин быстрого распространения ESLint.

Помимо этого сейчас мы транслируем код с помощью babel и используем новые возможности языка и сторонние расширения, такие как JSX. Плюс ESLint в том, что он позволяет дописывать расширения для проверки любого нового синтаксиса.

Помимо этого, так как ESLint позволяет создавать правила для проверки синтаксиса, у нас появляется возможность определить единый стиль кода внутри больших команд.

2.2.1 Установка

Прежде всего нам нужно установить ESLint:

```
npm install --global eslint
```

После этого мы можем запустить его следующей командой:

```
eslint source.js
```

На выходе мы получим информацию об ошибках внутри файла с кодом.

Но при первом запуске мы не должны увидеть никаких ошибок, потому ESLint не содержит никаких правил проверки по умолчанию.

2.2.2 Настройка

Для настройки ESLint используется файл *.eslintrc* в домашней директории проета или пользователя.

Начнем с простого и запретим использовать символ точки с запятой. Для этого добавим в .eslintrc следующий JSON:

```
{
  "rules": {
    "semi": [2, "never"]
  }
}
```

Эта запись определенно требует некоторых пояснений: "*semi*" - название правила(rule), а [2,"*never*"] - его значение.

В ESLint каждому правилу можно задать один из 3 уровней строгости:

- **off (0): Правило выключено**
- **warn (1): Правило предупреждения**
- **error (2): Правило ошибки**

Таким образом, указав в нашем примере значение 2, мы говорим, что ESLint, при срабатывании этого правила, должен ругаться о наличии ошибки в коде.

Второй параметр ("*never*") говорит о том, что точка с запятой никогда не должна использоваться внутри проекта.

ESLint и его плагины хорошо документированы и можно всегда посмотреть, как именно работают правила, и что значат их аргументы.

Теперь создадим файл index.js со следующим содержимым:

```
var foo = 'bar';
```

(Мы используем *var* так как ESLint еще не знает о том, что мы собираемся использовать ES2015).

Если мы запустим $eslintindex.js$, то получим следующее сообщение:

Extra semicolon (semi)

Заработало, теперь мы можем добавлять правила и следовать (или не следовать) им внутри проекта.

Мы можем добавлять все правила вручную или включить рекомендованный набор правил одной строкой в .eslintrc:

```
{
  "extends": "eslint:recommended"
}
```

После этого каждое отдельное правило может быть изменено по необходимости вручную.

После применения рекомендованных правил мы должны перестать получать ошибку о наличии точки с запятой (т.к. это правило не входит в рекомендованные), но должны получить ошибку, что переменная foo объявлена, но никогда не используется.

Правило $no - unused - vars$ очень полезно для сохранения чистоты кодовой базы.

Вспомним, что мы хотим писать код на ES2015, но если мы изменим исходный код на следующий:

```
const foo = 'bar'
```

То получим следующую ошибку:

Parsing error: The keyword 'const' is reserved

Для того, чтобы включить поддержку ES2015, мы должны добавить информацию об этом в .eslintrc:

```
"parserOptions": {
  "ecmaVersion": 6,
}
```

Также для нас будет полезно указать, что мы используем JSX:

```
"parserOptions": {
  "ecmaVersion": 6,
  "ecmaFeatures": {
    "jsx": true
  }
},
```

Если вы уже писали приложения на React, но не использовали линтер, то хорошим упражнением будет добавить ESLint в уже существующий проект и исправить все ошибки, которые будут найдены.

Использование ESLint из консоли конечно позволяет нам проверять код, но еще одним плюсом этого инструмента является то, что он поддерживается большинством современных редакторов и IDE (SublimeText, Atom и множество других).

Очень часто есть сильный соблазн просто проигнорировать все, что говорит нам ESLint и залить код в общий репозиторий. Чтобы избежать этого, имеет смысл добавить линтер как один из шагов в процесс сборки приложения, в этом случае код с ошибками просто не попадет в продакшн.

Еще вариант, добавить линтер на этап создания пулл реквета. В этом случае код с ошибками не попадет даже на ревью кода.

2.2.3 React плагин

Как было сказано раньше, одна из сильнейших сторон ESLint - его расширяемость плагинами. Самый важный плагин для нас eslint-plugin-react.

В целом ESLint может работать с JSX без дополнительных плагинов. Но мы хотим больше, например хранить наши компоненты в одном определенном стиле.

Прежде всего нам нужно установить плагин:

```
npm install --global eslint-plugin-react
```

После этого мы можем добавить его в наш файл с настройками:

```
"plugins": [
  "react"
]
```

По аналогии с самим ESLint мы можем добавить набор рекомендованных настроек для react плагина:

```
"extends": [
  "eslint:recommended",
  "plugin:react/recommended"
],
```

С этого момента, если что-то будет не так с нашими компонентами, например мы попробуем передать один и тот же параметр дважды, ESLint будет предупреждать об ошибке:

```
<Foo bar bar />
```

И соответствующее сообщение об ошибке:

No duplicate props allowed (react/jsx-no-duplicate-props)

Есть множество правил, которые мы можем использовать в нашем проекте. Давайте посмотрим, как некоторые из них могут улучшить нашу жизнь.

Одна из проблем, которую может помочь решить для нас ESLint, - это одинаковый размер отступов в JSX верстке. Это поможет нам сохранить единый стиль внутри всего приложения и не держать постоянно в голове точный размер отступов.

Для того, чтобы включить эту проверку, достаточно добавить следующее правило:

```
"rules": {
  "react/jsx-indent": [2, 2]
}
```

Первая 2 означает, что ESLint будет говорить об ошибке, если сработает это правило, вторая 2 говорит от том, что размер отступа для каждого компонента должен быть из двух пробелов. Также можно сказать, что отступов не должно быть вовсе, заменив вторую 2 на 0.

Создайте файл с содержимым вида:

```
<div>
<div />
</div>
```

И вы получите ошибку:

Expected indentation of 2 space characters but found 0 (react/jsx-indent)

Аналогичным образом мы можем заставить всех использовать одинаковый отступ для всех параметров элементов:

```
"react/jsx-indent-props": [2, 2]
```

Также часто возникают вопросы, на которые у каждого разработчика может найтись свое мнение отличное от других. Например, какой максимальной длины должны быть строки кода? Или когда считать, что у элемента слишком много параметров? С ESLint и правилом jsx-max-props-per-line можно выбрать единое значение для всех, чтобы не возвращаться к этому вопросу на каждом ревью кода.

React плагин для ESLint позволяет проверять не только JSX, но и сами компоненты.

Например, мы можем договориться определять PropTypes в алфавитном порядке. Есть правило, чтобы проверить, что используются только объявленные в PropTypes параметры. Или правило для проверки того, что все компоненты без состояния объявлены в функциональном стиле. А также множество других правил.

2.2.4 Airbnb configuration

Мы уже посмотрели, как ESLint помогает искать ошибки и следовать единому стилю кода. Также мы увидели, насколько ESLint открыт для настройки и расширения.

Можно сделать шаг дальше.

Через атрибут extends в файле настроек ESLint можно подключить конфигурации сторонних организаций, а затем уже добавлять свои правила поверх них.

Одна из самых известных конфигураций для ESLint в мире React была создана в стенах компании Airbnb. Разработчики этой компании создали набор правил, чтобы следовать единому стилю кода среди всех разработчиков, и каждый желающий может подключить этот набор правил в свой проект.

Для того, чтобы сделать это необходимо добавить несколько зависимостей:

```
npm install --global eslint-config-airbnbeslint@^2.9.0 eslint
    -plugin-jsx-a11y@^1.2.0 eslint-plugin-import@^1.7.0 eslint
    -plugin-react@^5.0.1
```

А затем добавить новые настройки в .eslintrc:

```
{
  "extends": "airbnb"
}
```

Это один из самых простых и распространенных способов начать работать с ESLint.

2.3 Основы функционального программирования

Кроме исправления JSX и использования линтера есть еще один способ сделать код чище: следовать **Функциональному (Functional Programming, FP)** стилю.

Функциональное программирование - парадигма декларативного программирования, сосредоточенная на минимизации побочных эффектов (side effects) и сохранении данных неизменяемыми (immutable).

Следующая часть не ставит своей целью полностью раскрыть обширную тему функционального программирования, но мы можем посмотреть на некоторые концепции, которые часто используются в React.

2.3.1 Объект первого класса

В JavaScript функции являются *объектами первого класса (first-class objects)*, т.е. они могут быть присвоены переменным как значение и переданы как аргументы другим функциям.

Это позволяет нам ввести концепцию **Функций высшего порядка (Higher-order Functions, HoF)**. Такой функцией мы будем назвать функцию, которая принимает аргументом функцию (и возможно другие аргументы) и возвращает новую функцию. Возвращаемая функция чаще всего расширяет функционал изначальной функции.

Посмотрим на простой пример, функцию сложения двух чисел, которую мы обернем функцией, логирующей все аргументы функции и вызывающей изначальную функцию сложения:

```
const add = (x, y) => x + y

const log = func => (...args) => {
  console.log(...args)
  return func(...args)
}
const logAdd = log(add)
```

Функции высшего порядка часто используется в React разработке. Один из самых распространенных паттернов использования этого приема - **Компоненты высшего порядка (Higher-order Components, HoC)**. Подробнее на компоненты высшего порядка мы посмотрим в Главе 4.

2.3.2 Purity

Важный аспект функционального программирования - чистые функции. С ними вы будете встречаться очень часто, особенно в таких библиотеках как Redux.

Главное отличие чистой функции - отсутствие побочных эффектов.

К примеру, если функция меняет состояние приложения, меняет значение переменных во внешней области видимости переменных или меняет значение переменных переданных по ссылке, то функция не является чистой.

Чистые функции значительно проще тестировать так как при вызовах с одинаковыми аргументами функция возвращает одинаковый результат.

Простой пример чистой функции - функция сложения:

```
const add = (x, y) => x + y
```

Она может быть запущена множество раз с одними и те ме же аргументами, но в каждый раз она вернет одинаковый результат.

В следующем примере функция перестает быть чистой:

```
let x = 0
const add = y => (x = x + y)
```

Если мы вызовем $add(1)$ дважды, то сначала мы получим 1, а затем 2. Причина в том, что работа функции зависит от переменной во внешней области видимости, которая еще и редактируется внутри этой функции.

2.3.3 Immutability

Мы разобрались с чистыми функциями, которые не изменяют состояние приложения, но что если в функцию приходит сложный объект, в котором мы хотим что-либо изменить?

FP говорит нам о том, что в этом случае следует создать новый объект с измененным значением и вернуть его, не трогая исходный объект.

Рассмотрим пример:

```
const add3 = arr => arr.push(3)
const myArr = [1, 2]
add3(myArr) // [1, 2, 3]
add3(myArr) // [1, 2, 3, 3]
```

Эта функция изменяет состояние исходного массива, что противоречит парадигме неизменяемости. Если вызвать эту функцию несколько раз с одним и тем же исходными массивом, то мы будем получать разные значения на выходе.

Мы можем исправить эту функцию, чтобы она стала immutable с помощью метода concat, который возвращает новый массив, оставляя исходный без изменений:

```
const add3 = arr => arr.concat(3)
const myArr = [1, 2]
const result1 = add3(myArr) // [1, 2, 3]
const result2 = add3(myArr) // [1, 2, 3]
```

Или используя спред оператор:

```
const add3 = arr => [...arr, 3]
```

Сколько бы мы ни вызывали этот метод, исходный массив остается нетронутым.

2.3.4 Каррирование

Каррирование (Curring) - еще одна распространенная техника функционального программирования, которая заключается в конвертации функции от многих переменных в функцию одной переменной, которая возвращает другую функцию.

Посмотрим на примере функции *add* как это работает на практике. Вместо исходной функции от двух аргументов:

```
const add = (x, y) => x + y
```

Мы можем определить каррированную функцию:

```
const add = x => y => x + y
```

Такую функцию мы можем использовать следующим образом:

```
const add1 = add(1)
add1(2) // 3
add1(3) // 4
```

Таким способом мы можем заключить первый аргумент внутри переменной $add1$ и использовать ее множество раз без явной передачи.

2.3.5 Композиция

Композиция (Composition) - еще один прием, который позволяет сохранять функции небольшими и тестируемыми.

Рассмотрим следующий пример:

```
const add = (x, y) => x + y
const square = x => x * x
```

Эти функции могут быть скомбинированы вместе, чтобы создать новую функцию, которая складывает два числа и возвращает их квадрат:

```
const addAndSquare = (x, y) => square(add(x, y))
```

Следуя этому простому приему можно сохранять функции небольшими и читаемыми, а также упростить их тестирование.

2.3.6 FP и пользовательские интерфейсы

Последний шаг - понять, как функциональное программирование помогает нам строить пользовательский интерфейс.

Мы можем думать о UI как о функции, которая принимает аргументом состояние приложения, а возвращает интерфейс приложения:

$$UI = f(state)$$

Мы хотим ожидать, что эта функция будет чистой, т.е. для одного и того же состояния приложения она будет возвращать всегда одинаковое представление для пользователя.

В React мы будем будем рассматривать компоненты как функции и комбинировать из этих функций пользовательский интерфейс.

Есть много схожего между функциональным программирование и созданием пользовательского интерфейса с React. И чем больше мы будем смотреть на разработку с React как на функциональное программирование, тем лучше будет наш код.

2.4 Заключение

В этой главе мы детально разобрали строение JSX и как с помощью него создавать компоненты.

Также мы разобрались со статической проверкой кода средствами ESLint и его плагинов. Это позволит нам быстрее находить потенциальные проблемы в коде.

И в конце мы рассмотрели базовые концепции функционального программирования, которые часто встречаются в React.

Используя все выше сказанное, можно сделать код значительно чище и более тестируемым. Настало время сделать еще один шаг вперед и разобраться, как сделать действительно переиспользуемые компоненты.

Глава 3

Создаем переиспользуемые компоненты

Для того чтобы создавать действительно переиспользуемые компоненты мы должны разобраться, какие в целом способы создания компонент предоставляет React и какой в каком случае стоит выбирать. Также относительно недавно в React появился новая способ определения компонент с помощью **безстейтовых функций (stateless function)**

Один из способов изучения - рассмотреть множество примеров. Следуя по этому пути, мы начнем с примера компонента, который имеет узкую специализацию, и превратим его в переиспользуемый.

В этой главе мы рассмотрим:

- Различные способы создания React компонент и когда какие из них следует использовать
- Что такое stateless компоненты и чем они отличаются от stateful компонент
- Как работает state компонент и когда следует избежать его использование
- Почему важно определять prop types для каждого компоненты и как с помощью них создавать динамически документацию с помощью **React Docgen**
- Примеры создания универсальных компонент из узкоспециализированных

- Как мы можем задокументировать нашу коллекцию универсальных компонент с помощью **React Storybook**

3.1 Создание классов

Давайте начнем детально разбираться с возможностями определения компонент, которые предоставляет React.

3.1.1 Фабрика createClass

Если открыть документацию React, то первый способ создания компонента, который мы найдем, будет *React.createClass*.

Попробуем создать с его помощью простой пример:

```
const Button = React.createClass({
  render() {
    return <button />
  },
})
```

Мы создали простую кнопку, которую можем использовать внутри других компонент нашего приложения.

Мы даже можем заменить JSX внутри этого компонента на обычный JavaScript:

```
const Button = React.createClass({
  render() {
    return React.createElement('button')
  },
})
```

Теперь нам не придется использовать babel для запуска этого кода, что значит, что мы можем открыть его напрямую в браузере.

3.1.2 Наследование React.Component

Следующий подход - использовать классы ES2015. Ключевое слово *class* уже неплохо поддерживается браузерами, но все равно в этом случае уже лучше транслировать код с babel.

Создадим ту же кнопку, но уже с использование JavaScript классов:

```
class Button extends React.Component {
  render() {
    return <button />
  }
}
```

Этот способ создание компонент появился с версии React 0.13 и разработчики Facebook настаивают на том, что использоваться должен именно он. Активный участник React сообщества Ден Абрамов (Dan Abramov) в защиту преимущества ES2015 классов перед createClass высказал:

"ES6 classes: better the devil that's standardized (Классы ES6: лучше тот дьявол, который стандартизирован)"

Таким разработчики библиотеки React ратуют за использование стандартных классов ES2015 вместо createClass фабрики.

3.1.3 Главные отличия

За исключением синтаксических различий есть значительные различия, которые стоит держать в голове. Давайте взглянем на них, чтобы вы могли осознанно выбирать между ними во время разработки проекта.

Props

Первое различие, которое мы рассмотрим, заключается в том, как мы определяем параметры, которые получает компонент и как мы задаем для них стандартные значение.

Как работает передача параметров компоненту мы рассмотрим чуть позже, сейчас сконцентрируемся на том, как они определяются.

В *createClass* мы определяется параметры, которые можно передать компоненту, в поле *propTypes* объекта, передаваемого этой функции, а значения по умолчанию передаем с помощью функции $getDefaultProps$:

```
const Button = React.createClass({
  propTypes: {
    text: React.PropTypes.string,
  },
  getDefaultProps() {
    return {
      text: 'Click me!',
    }
  },
  render() {
```

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

Чтобы получить тот же результат с помощью JavaScript классов, нам нужно будет немного поменять структуру:

```
class Button extends React.Component {
  render() {
    return <button>{this.props.text}</button>
  }
}
Button.propTypes = {
  text: React.PropTypes.string,
}
Button.defaultProps = {
  text: 'Click me!',
}
```

Мы вынуждены определить $propTypes$ и $defaultProps$ вне класса, так как **Class Properties** еще не являются частью стандарта языка.

Когда нам нужно определить параметры по умолчанию, нам нужно было возвращать их из специальной функции, с JavaScript классами нам достаточно определить их в параметры класса.

Основной плюс в том, что с использованием классов, мы избавились от специфичных для React функций, таких как $getDefaultProps$.

State

Еще одно различие заключается в том, как мы можем определить начальное состояние компонента.

Аналогично с определением параметров в $createClass$ мы используем функция, а в ES2015 классе атрибут экземпляра класса.

Посмотрим на пример:

```
const Button = React.createClass({
  getInitialState() {
    return {
      text: 'Click me!',
    }
  },
  render() {
    return <button>{this.state.text}</button>
  },
})
```

Метод *getInitialState* должен вернуть объект с начальным состоянием для каждого элемента.

В случае с классом мы должны определить начальное состояние в поле state экземпляра класса, и происходит это в момент вызова конструктора:

```
class Button extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      text: 'Click me!',
    }
  }
  render() {
    return <button>{this.state.text}</button>
  }
}
```

Два этих способа эквиваленты, единственное, в последнем случае JavaScript классы позволяют нам избавиться от специфичного метода React API.

В ES2015 мы должны также вызвать конструктор родительского класса, чтобы он был проинициализирован и мы могли работать с *this*.

Autobinding

У *createClass* есть очень удобная фича, которая скрывает механизм работы JavaScript и может вводить в заблуждение начинающих разработчиков. Эта фича позволяет привязывать к компоненту обработчики событий, ожидая, что в момент вызова этого обработчика в *this* попадет сам компонент.

Подробно об обработчиках событий мы поговорим позднее в *Главе 6*. Сейчас нас интересует только то, как эти обработчики привязываются к компонентам.

Посмотрим на пример:

```
const Button = React.createClass({
  handleClick() {
    console.log(this)
  },
  render() {
    return <button onClick={this.handleClick} />
  },
```

```
})
```

При использовании *createClass* мы можем спокойно использовать *this*, что позволяет нам использовать другие методы компонента, такие как *this.setState()*.

Посмотрим, насколько отличается поведение *this* при наследовании *React.Component* и как нам добиться такого же поведения.

Мы можем создать компонент следующим образом:

```
class Button extends React.Component {
  handleClick() {
    console.log(this)
  }
  render() {
    return <button onClick={this.handleClick} />
  }
}
```

Если мы вызовем этот код, то результатом нажатия на кнопку будет *null*. Это связано с тем, что наша функция передается обработчику событий и мы теряем связь с текущим компонентом.

Это не значит, что мы не можем использовать обработчики событий от слова совсем, но нам придется связывать с компонентом вручную.

Разберемся какие есть возможности связывания функций и компонент.

Возможно вы уже знаете, что стрелочные функции ES2015 автоматически связываются с текущим *this* контекста, где эта функция создается.

Посмотрим на пример стрелочной функции:

```
() => this.setState()
```

Если мы транслируем этот код с помощью Babel, то получим:

```
var _this = this;
(function () {
  return _this.setState();
});
```

Как вы уже можете догадаться, одно из решений проблемы **автоматического связывания** - использование стрелочных функций. Посмотрим, как это работает:

```
class Button extends React.Component {
  handleClick() {
    console.log(this)
```

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

Этот код будет работать корректно. Но у данного варианта есть проблемы с производительностью, чтобы понять почему, нужно разобраться, как этот код работает.

Создание стрелочной функции внутри *render* метода влечет непредвиденный посторонний эффект. Эта функция пересоздается на каждом вызове *render*, что может происходить довольно часто.

Помимо того, что мы каждый раз пересоздаем не самый легкий объект функции, мы также передаем этот объект дочерним элементам, заставляя их пересоздаваться.

Лучшим решением будет привязывать эти функции к компоненту в момент вызова конструктора. В этом случае функция и пересоздаваться не будет и будет иметь нужный нам контекст:

```
class Button extends React.Component {
  constructor(props) {
    super(props)
    this.handleClick = this.handleClick.bind(this)
  }
  handleClick() {
    console.log(this)
  }
  render() {
    return <button onClick={this.handleClick} />
  }
}
```

Проблема решена (Прим.пер. помимо этого сейчас можно определить стрелочную функция как параметр класса, тогда она не будет каждый раз пересоздаваться, но и контекст будет иметь правильный):

```
class Button extends React.Component {
  handleClick = () => {
    console.log(this)
  }
  render() {
    return <button onClick={this.handleClick} />
  }
}
```

3.1.4 Stateless functional components

Есть еще один способ создать компонент, который несколько отличается от первых двух.

Этот способ появился в **React 0.14** и принес возможности сделать код еще чище.

Чтобы определить компонент, нам достаточно определить функцию, которая будет возвращать React элемент:

```
() => <button />
```

Благодаря стрелочной функции этот код минималистичен и понятен. Само собой внутри этой функции можно использовать JSX, иначе бы это, наверно, не имело смысла.

Props and context

Компонент, который не может получить параметры от родительского элемента, в большинстве случаев будет бесполезен. В новом способе определения компонент параметры передаются в первом параметре самой функции:

```
props => <button>{props.text}</button>
```

Помимо этого мы можем использовать возможности деструктуризации ES2015:

```
({ text }) => <button>{text}</button>
```

Также мы можем определить, какие параметры компонента может принимать, по аналогии с классами через *propTypes* атрибут самой функции:

```
const Button = ({ text }) => <button>{text}</button>
Button.propTypes = {
  text: React.PropTypes.string,
}
```

Также компоненты-функции получают вторым аргументом (*context*):

```
(props, context) => (
  <button>{context.currency}{props.value}</button>
)
```

Ключевое слово this

Главное отличие создания компоненты-функции от других способов определения компонент в том, что в них *this* не ссылается на сам компонент.

Следствием этого является невозможность управлять жизненным циклом компонента и использовать такие методы как *setState*.

State

Как можно понять из названия (stateless), такие компоненты не обладают внутренним состоянием.

Все что делает такая компоненты, это принимает параметры и контекст в аргументах функции и возвращает React элемент.

Это напоминает нам о функциональном программировании, в разрезе которого мы можем смотреть на компоненты-функции как на функции React элементов от параметров и контекста.

Lifecycle

Компоненты-функции не предоставляют никаких возможностей по отслеживанию методов жизненного цикла, таких как *componentDidMount*. Все, что не касается непосредственно генерации JSX должно обрабатываться в родительских элементах.

Refs and event handlers

Не смотря на то, что мы не можем получить ссылку на сам элемент, мы все же можем получить ссылки на элементы, которые создаются внутри компонент-функции. Сделать это можно следующим образом:

```
() => {
  let input
  const onClick = () => input.focus()
  return (
    <div>
      <input ref={el => (input = el)} />
      <button onClick={onClick}>Focus</button>
    </div>
  )
}
```

Отсутствие ссылки на компонент

Еще одно отличие компонент-функций заключается в том, что если мы создадим экземпляр такого компонента с помощью *ReactTestUtils*, мы не получим никакой ссылки на созданный элемент (подробнее о тестировании и отладке мы поговорим в Главе 10).

Например:

```
const Button = React.createClass({
  render() {
    return <button />
  },
})
const component = ReactTestUtils.renderIntoDocument(<Button
   />)
```

В этом случае переменная component содержит ссылку на созданный элемент.

```
const Button = () => <button />
const component = ReactTestUtils.renderIntoDocument(<Button
   />)
```

А в этом случае переменная component будет иметь значение *null*. Для того, чтобы обойти это ограничение, можно обернуть компонент в другой элемент, например *div*:

```
const component = ReactTestUtils.renderIntoDocument(<div><
   Button/></div>)
```

Производительность

Основное, что стоит держать в голове относительно производительности компонент-функций то, что они легковеснее полноценных компонент и лучше поддаются оптимизации внутри самой библиотеки, о чем говорят разработчики Facebook.

Помимо этого есть нюанс, что в жизненном цикле компонента отсутствует метод *shouldComponentUpdate*, что не дает возможность сказать Reacrt'у, что компонент не нужно повторно вызывать метод render. Это на самая большая проблема, но стоит держать ее в голове.

3.2 The state

Мы разобрались с тем, как создавать компоненты в React. Теперь мы можем пойти дальше и посмотреть детально на управление состоянием компонент.

Также мы должны понять, когда использование компонент-функций приоритетнее полнофункциональных компонент, и как это влияет на архитектуру наших компонент.

3.2.1 Сторонние библиотеки

Прежде всего мы должны понять, почему мы вообще должны рассматривать управление состоянием внутри наших компонент.

На данный момент большинство учебных материалов и шаблонов приложений на React уже содержат сторонние библиотеки для управления состоянием, такие как **Redux** и **MobX**.

К сожалению, это может натолкнуть людей на мысль, что невозможно написать приложение с наличием хоть сколько-то сложного состояние только средствами React, что конечно же не так.

Это приводит к тому, что многие начинающие разработчики изучают React и Redux вместе и плохо представляют как управлять состоянием приложения средствами React.

В этой части мы рассмотрим детально, как управлять состоянием в компонентах React, и в каких случаях мы можем обойтись без использования сторонних библиотек.

3.2.2 Как это работает

Не смотря на различия в разных способах создания компонент, все компоненты, созданные функцией *createClass* или наследование *React.Component* могут обладать внутренним состоянием, которое может быть изменено с помощью функции *setState*.

После каждого изменения состояние компоненты React вызывает метод *render*, чтобы перестроить элементы в соответствии с новым состояние. Поэтому часто говорят, что React компоненты похожи на конечный автомат (state machine).

После вызова метода *setState* с новым состоянием (или его частью),

объект переданный функции сливается с текущим состоянием компонента. Например, если у нас есть следующее состояние:

```
this.state = {
  text: 'Click me!',
}
```

И мы вызываем *setState* со следующим объектом:

```
this.setState({
  cliked: true,
})
```

На выходе мы получим новое состояние компонента:

```
{
  cliked: true,
  text: 'Click me!',
}
```

После каждого изменения состояние React сам вызывает метод *render*, поэтому от нас не требуется никаких дополнительных движений для обновления элемента.

Однако, если нам нужно совершить какие-либо действия сразу после обновления состояния, то есть возможность передать функцию обратного вызова вторым аргументом функции *setState*:

```
this.setState(
  {
    clicked: true,
  },
  () => {
    console.log('the state is now', this.state)
  }
)
```

3.2.3 Асинхронность

Функцию *setState* стоит всегда рассматривать как асинхронную. Как сказано в официальной документации:

> There is no guarantee of synchronous operation of calls to setState... (Нет никаких гарантий в синхронной обработке вызова setState)

На деле это выливается в то, что если мы, например, распечатаем состояние компонента сразу после вызова *setState*, то увидим состояние, которое было перед вызовом *setState*:

```
handleClick() {
  this.setState({
    clicked: true,
  })
  console.log('the state is now', this.state)
}
render() {
  return <button onClick={this.handleClick}>Click me!</button
     >
}
```

Если у компонента не было состояния до вызова *setState*, то в консоль будет напечатано: the state is now *null*.

Причина этого - оптимизации, которые проводит React при перерисовки компонент. Асинхронность вызова *setState* позволяет ему откладывать вызов при нехватке ресурсов и объединять при возможности множество вызовов в один.

Но если мы совсем слегка поменяем наш код:

```
handleClick() {
  setTimeout(() => {
    this.setState({
      clicked: true,
    })
    console.log('the state is now', this.state)
  })
}
```

Результат будет уже совершенно другим:

the state is now Object $\{clicked : true\}$

Это то, что мы хотели бы ожидать в первом случае. Но в данном случае React не видит никаких возможностей для оптимизации, поэтому состояние обновляется мгновенно.

В данном случае *setTimeout* использовался для примера различного поведения обновления состояния. Писать обработчики событий в таком стиле и надеяться на синхронность *setState* крайне не рекомендуется.

3.2.4 Reacrt lumberjack

Если мы можем рассматривать компонент как конечный автомат, то мы можем пойти дальше и реализовать не только изменение состояния компонент, но также и откат этого состояния к предыдущим значениям, что может быть очень полезно при отладке.

Такой функционал уже реализован в библиотеке *react- lumberjack* Райаном Флоренсом (Ryan Florence), создателем очень известной библиотеки *react-router*.

Использовать библиотеку очень просто, главное не стоит забывать выключать ее на проде. Ее можно установить как и множество других библиотек через *npm* или добавить прямую ссылку на *unpkg.com*:

```
<script src="https://unpkg.com/react-lumberjack@1.0.0"></
    script>
```

После установки библиотеки наше приложение продолжает работать как раньше.

Но если мы хотим отменить изменения в состоянии приложения, то достаточно набрать в консоли:

```
Lumberjack.back()
```

Если после этого мы хотим снова вернуться к состоянию до отката, то нужно набрать в консоли:

```
Lumberjack.forward()
```

Таким нехитрым образом мы можем гулять по состояниям назад и вперед.

Но стоит помнить, что это достаточно экспериментальная библиотека и она может или исчезнуть вовсе после какого-нибудь изменения React API или наоборот стать частью React Developer Tools.

3.2.5 Использование state

Теперь, когда мы узнали, как хранится и изменяется состояние компонент, мы можем подумать о там, какие данные стоит или не стоит хранить в компонентах.

Нам нужно выработать некоторые правила, которые бы помогли нам в большинстве случаев понимать, должен ли компонент быть с внутренним состоянием или нет.

Прежде всего стоит запомнить, что внутри компонент стоит хранить минимально необходимое количество данных.

Например, если у нас есть текстовое поле, значение которого должно меняться когда пользователь нажимает на кнопку, то нам не стоит хранить весь текст поля во внутреннем состоянии компонента, достаточно хранить лишь boolean значение факта нажатия кнопки.

Таким образом, если мы можем посчитать какое-то значение на основе данных состояния, то это значение хранить внутри компоненты не стоит.

Во-вторых, во внутреннем состоянии компонента стоит хранить только те данные, которые мы собираемся менять по какому-либо событию, и при изменении которых требуется пересоздание элементов.

Пример таких данных - флаг *isClicked*, который меняется при нажатии пользователя, или любое поле ввода данных.

В общем случае, во внутреннем состоянии следует хранить данные, которых будет необходимо и достаточно для восстановления работоспособности сайта. Сюда же могу войти текущая страница, на которой находится пользователь, выбранная вкладка, введенные фильтры.

Также принять решение относительно каких-то данных поможет знание того, какому количеству внешних или дочерних элементов требуются эти данные.

Если данные требуются большому количеству компонент, то скорее всего стоит задуматься о специальном хранилище данных на уровне всего приложения, например Redux.

Сейчас посмотрим на случаи, когда мы должны избежать использования внутреннего состояния компонент для хранения данных.

Derivables

Если какое-либо значение, которое требуется для отображения компонента, может быть посчитано на основе *props*, то это не нужно сохранять в *state*.

Например, если мы получаем валюту и цену из *props* и всегда показываем их вместе, то могли бы предположить, что логично поместить их в *state*:

```
class Price extends React.Component {
  constructor(props) {
    super(props)
```

```
    this.state = {
      price: `${props.currency}${props.value}`
    }
  }
  render() {
    return <div>{this.state.price}</div>
  }
}
```

Но это будет работать только если мы будем создавать экземпляры этого компонента со статичными параметрами, например:

```
<Price currency="\textdollar" value="100" />
```

Проблема в том, что если валюта или цена изменится в течение жизни компонента, то это никак не отразится на отображении, так как конструктор вызывается только один раз в момент создания элемента.

Вместо этого нам следует использовать *props* для вычисления значение.

Помимо этого, как говорилось ранее, эти расчеты можно вынести в отдельную вспомогательную функцию:

```
getPrice() {
  return `${this.props.currency}${this.props.value}`
}
```

The render method

Нужно помнить, что любое изменение состояния компонента влечет перерисовку элемента, поэтому в *state* должны быть только те данные, которые используются для отрисовки элемента.

Например, если нам нужно хранить подписку на API сервера или ссылку на работающий таймер, но ни первое ни второе не влияет на отображение компонента, то стоит хранить их в отдельном модуле.

Следующий пример плохо кода показывает, как не стоит хранить данные в *state*. Данные сохраняются в *state*, но при этом не используются в *render* функции, что влечет к излишним обновлениям компонента:

```
componentDidMount() {
  this.setState({
    request: API.get(...)
  })
}
componentWillUnmount() {
```

```
  this.state.request.abort()
}
```

Такого использования *state* следует избегать и лучше хранить данные, которые не затрагивают отображение, в отдельном модуле (прим.пер. SOLID и дядя Боб вам в помощь).

Еще одним распространенным решение является хранение таких данных внутри экземпляра компонента, но не в *state*, а в обычном поле. Это хорошо работает, так как компонента все еще остается обычным JavaScript классом:

```
componentDidMount() {
  this.request = API.get(...)
}
componentWillUnmount() {
  this.request.abort()
}
```

В этом случае данные сохранены в экземпляре класса и никак не затрагивают метод *render* и перерисовку элемента.

Следующий пример кода от Дена Абармова кратко поясняет, что не нужно хранить в *state*:

3.3 Prop types

Мы ставим перед собой цель, создавать переиспользуемые компоненты, поэтому нам нужно описывать интерфейс этих компонент удобным для пользователей этих компонент образом.

React из коробки позволяет нам описывать, какие параметры ожидает компонент и простые правила валидации к ним. Привила позволяют указывать, какого типа данных должны быть параметры, а также являются ли параметры обязательными. Помимо этого React позволяет определять пользовательские функции проверки параметров.

Посмотрим на простой пример:

```
const Button = ({ text }) => <button>{text}</button>
Button.propTypes = {
  text: React.PropTypes.string,
}
```

В этом примере мы создаем простую компонент-функцию, которая должна получать один параметр *text*, который должен быть типа *string*.

```
function shouldIKeepSomethingInReactState() {
  if (canICalculateItFromProps()) {
    // Don't duplicate data from props in state.
    // Calculate what you can in render() method.
    return false;
  }
  if (!amIUsingItInRenderMethod()) {
    // Don't keep something in the state
    // if you don't use it for rendering.
    // For example, API subscriptions are
    // better off as custom private fields
    // or variables in external modules.
    return false;
  }
  // You can use React state for this!
  return true;
}
```

Теперь каждый, кто попытается воспользоваться данным компонентом, сможет легко понять, что ему нужно передавать даже без детального чтения кода.

Но что если компонент вообще не сможет работать, если ему не передать определенный параметр? В этом случае этот параметр можно указать как обязательный:

```
Button.propTypes = {
  text: React.PropTypes.string.isRequired,
}
```

В этом случае кто-либо, кто создаст такой компонент, не передав обязательный параметр, получат следующую ошибку:

> **Failed prop type: Required prop ‘text‘ was not specified in ‘Button‘.**

Важно понимать, что предупреждение об ошибке мы получим только

в режиме разработки. В релизной сборке валидация с *propTypes* выключается в целях улучшения производительности.

React предоставляет возможность проверки множества типов параметров: от чисел до массивов и компонент.

Если мы хотим, чтобы компонент работал с данными разных типов, передаваемых через один параметр, то можем использовать функцию **oneOf**, которая позволяет указать список разных типов для одного параметра.

Также стоит стараться передавать через параметры только примитивные типы, так как они лучше поддаются отладке и терстированию.

Передача примитивов также позволяет быстрее находить разбухающие интерфейсы у компонент. Если компонент начинает требовать все больше и больше параметров, то возможно он содержит больше логики чем должен и нарушает принцип единственности ответственности.

Если мы замечаем, что компонент получает множество параметров, которые слабо связаны логикой приложение, то можно попробовать разделить этот компонент на два независимых.

Однако нам все равно достаточно часто приходится передавать компонентам объекты. В этом случае для валидации следует использовать функцию *shape*.

Функция *shape* позволяет определить параметр типа объект, а также определить тип всех полей, которые должны быть у этого объекта, которые в свою очередь тоже могут быть объектами.

Например, если мы создадим компонент $Profile$, который ожидает объект с именем и фамилией пользователя, то мы можем создать следующие *propTypes*:

```
const Profile = ({ user }) =>(
  <div>{user.name} {user.surname}</div>
)
Profile.propTypes = {
  user: React.PropTypes.shape({
    name: React.PropTypes.string.isRequired,
    surname: React.PropTypes.string,
  }).isRequired,
}
```

Если же ни одного из стандартных методов валидации React нам не подходят, то мы можем определить собственную функцию проверки:

```
user: React.PropTypes.shape({
```

```
  age: (props, propName) => {
    if (!(props[propName] > 0 && props[propName] < 100)) {
      return new Error(`${propName} must be between 1 and
          99`)
    }
    return null
  },
})
```

Например, в примере выше мы проверяем, что возраст находится в определенном числовом промежутке. Если же возраст выйдет за этот промежуток, то мы увидим соответствующую ошибку в консоли.

3.3.1 React Docgen

Хорошо описанные *propTypes* уже значительное облегчение жизни тем, кто будет пользоваться нашими компонентами. Но мы можем пойти дальше и еще больше упростить их использование.

Когда количество компонентов значительно возрастает, то появляется проблема поиска необходимого компонента среди многих, особенно для новых членов проекта.

Но если мы поддерживали *propTypes* в хорошем состоянии, то мы можем автоматически создавать из них документацию.

Для этого мы можем использовать библиотеку $react - docgen$, которую можно установить следующей командой:

```
npm install --global react-docgen
```

React Docgen проходит по файлу с компонентом и достает, необходимую для него информацию, из *propTypes* и комментариев.

Например, если у нас есть компонент:

```
const Button = ({ text }) => <button>{text}</button>
Button.propTypes = {
  text: React.PropTypes.string,
}
```

И мы запустим:

```
react-docgen button.js
```

Мы получим следующий результат;

```
{
  "description": "",
```

```
  "methods": [],
  "props": {
    "text": {
      "type": {
        "name": "string"
      },
      "required": false,
      "description": ""
    }
  }
}
```

Этот JSON представляет собой описание интерфейса компонента. Как вы видите, в него попали поля и их типы, описанные в *propTypes*.

Также мы можем добавить комментарий к нашему компоненту:

```
/**
 * A generic button with text.
 */
const Button = ({ text }) => <button>{text}</button>
Button.propTypes = {
  /**
   * The text of the button.
   */
  text: React.PropTypes.string,
}
```

Если мы снова запустим Docgen, то получим:

```
{
  "description": "A generic button with text.",
  "methods": [],
  "props": {
    "text": {
      "type": {
        "name": "string"
      },
      "required": false,
      "description": "The text of the button."
    }
  }
}
```

Теперь, с этим описанием интерфейса в формате JSON мы можем создать документацию и использовать ее внутри команды.

Результат работы Docgen имеет простой формат, поэтому не составляет никакого труда создать страницу с документацией на его основе.

Один из ярких примеров использования React Docgen - документация библиотеки *MaterialUI*, где вся документация создана на основе исходного кода библиотеки.

3.4 Переиспользуемые компоненты

Мы уже хорошо разобрались с тем как создавать компоненты, как использовать внутреннее состояние компонент и как сделать их переиспользуемыми с помощью *propTypes*.

Давайте теперь, вооружившись всеми полученными знаниями, попробуем сделать из непереиспользуемыех компонент переиспользуемые.

Предположим, что у нас есть компонент, который загружает список постов через API сервера и отображает их на экране.

Это упрощенный пример, но он хорошо подходит, чтобы показать все этапы создания переиспользуемого компонента.

Создадим класс посредством его наследования от React.Component:

```
class PostList extends React.Component
```

Затем создадим конструктор и добавим загрузку данных в *componentDidMount*:

```
constructor(props) {
  super(props)
  this.state = {
    posts: [],
  }
}
componentDidMount() {
  Posts.fetch().then(posts => {
    this.setState({ posts })
  })
}
```

Во *state* компонента только одно поле *posts*, в котором мы будем хранить посты и которое инициализируется пустым массивом.

В *componentDidMount* вызывается API сервера, для получения списка постов. По окончанию запроса посты сохраняются в *state* компонента с помощью метода *setState*.

Это распространенный паттерн загрузки данных, другие варианты детальнее мы рассмотрим в Главе 5.

Posts - это вспомогательный класс, который содержит логику общения с сервером. Сейчас для нас важно только то, что этот класс имеет

метод $fetch$, который возвращает $Promise$, который при успешном выполнении вернет список постов.

Теперь мы можем отобразить список постов:

```
render() {
  return (
    <ul>
      {this.state.posts.map(post => (
        <li key={post.id}>
          <h1>{post.title}</h1>
          {post.excerpt && <p>{post.excerpt}</p>}
        </li> ))}
    </ul>
  )
}
```

Внутри метода *render* мы обходим все посты и для каждого создаем элемент $< li >$.

Мы полагаем, что у поста всегда есть поле *title* и безусловно показываем его внутри $< h1 >$. Поле же *post.excerpt* мы считаем необязательным и отображаем только при наличии.

Теперь представим другой компонент. Пусть он отображает список пользователей, которые получает из *props*, а не из собственного состояния:

```
const UserList = ({ users }) => (
  <ul>
    {users.map(user => (
      <li key={user.id}>
        <h1>{user.username}</h1>
        {user.bio && <p>{user.bio}</p>}
      </li>
      )
    )}
  </ul>
)
```

Данный компонент отображает список пользователей очень похожим на отображение постов способом.

Отличие в том, что теперь вместо *title* отображается *username* и опциональное поле теперь *bio* пользователя вместо *excerpt* поста.

Дублирующийся код как правило считаются плохим звоночком, так что давайте разбираться как React позволяет следовать правилу **Не повторяйся (Don't Repeat Yourself, DRY)**. Прежде всего мы можем

создать отдельный компонент *List*, в который вынесем логику отображения списков, отделив ее от самих данных. Главным требованием является возможность передать ключи полей по которым мы сможем получить данные, чтобы иметь возможность брать нужные данные из разных типов объектов.

Чтобы сделать это, мы определим два параметра: *titleKey* для передачи ключа, по которому мы получим значение для обязательного поля, и *textKey* для передачи ключа опционального поля.

Параметры нашего нового компонента будут выглядеть следующим образом:

```
List.propTypes = {
  collection: React.PropTypes.array,
  textKey: React.PropTypes.string,
  titleKey: React.PropTypes.string,
}
```

Так как *List* не будет обладать своим собственным состоянием, мы можем его создать как компонент-функцию:

```
const List = ({ collection, textKey, titleKey }) => (
  <ul>
    {collection.map(item =>
      <Item
        key={item.id}
        text={item[textKey]}
        title={item[titleKey]}
      />
    )}
  </ul>
)
```

Компонент *List* получает через *props* коллекцию объектов, проходит по ним и преобразует в элементы *Item*, который мы скоро реализуем. Также в дочерний элемент мы передаем *text* и *title*, который получаем с помощью полученных ключей из элементов коллекции.

Компонент *Item* будет максимально простым и чистым:

```
const Item = ({ text, title }) => (
  <li>
    <h1>{title}</h1>
    {text && <p>{text}</p>}
  </li>
)
```

```
Item.propTypes = {
  text: React.PropTypes.string,
  title: React.PropTypes.string,
}
```

Таким образом мы создали два компонента с достаточно простыми интерфейсами, чтобы с их помощью отображать пользователей, посты или что-либо еще. При этом такие небольшие компоненты очень удобны в поддержке и тестировании.

Отлично, теперь мы можем переписать наши исходные компоненты с использованием новых вспомогательных компонент.

Метод *render* компонента *PostsList* будет выглядеть следующим образом:

```
render() {
  return (
    <List
      collection={this.state.posts}
      textKey="excerpt"
      titleKey="title"
    />
  )
}
```

А компонент-функция *UserList* следующим:

```
const UserList = ({ users }) => (
  <List
    collection={users}
    textKey="bio"
    titleKey="username"
  />
)
```

Таким образом мы из узкоспециализированных компонент получили базовые компоненты, которые могут быть переиспользованы в будущем.

Мы также можем использовать *react – docgen* для генерации документации для полученных нами компонент.

Также теперь в случае необходимости расширения данного отображения, которое пока состоит из двух текстовых полей, нам будет достаточно поменять его в одном компоненте *Item*, а не в множестве узкоспециализированных компонент.

Например, если нам понадобится в случае слишком длинной строки урезать ее и показывать троеточие, нам будет достаточно добавить эту

логику внутрь одного компонента.

3.5 Living style guides

Использование переиспользуемых компонент с простыми интерфейсами - хороший способ сократить количество дублирующегося кода в проекте, но это не единственная причина, чтобы сосредоточиться на переиспользуемости.

Если вы создаете простые и понятные компоненты с чистыми интерфейсами, которые хорошо отделены абстракциями от конкретных данных, то эти компоненты можно объединить в библиотеку компонент и использовать за пределами команды. Такая библиотека будет представлять из себя набор готовых к использованию блоков, которыми можно будет поделиться с другими командами, дизайнерами или выложить ее в open source.

Очень часто новым членам команды может быть сложно понять, какие компоненты уже есть, а какие нужно реализовать. Решением этой проблемы может быть создание Style guide'a, которое бы позволило распространять не только сами компоненты, но и примеры их использования.

По сути style guide - это собранное визуальное представление всех единичных компонентов, которые уже реализованы в проекте. Это очень удобный способ сохранять единый стиль всех компонент среди множества разработчиков разного уровня.

К сожалению, создание style guide'a не всегда является простой задачей, так как из-за меняющихся требований может появиться множество дублирующихся компонент с небольшими отличиями, решающие какие-то локальные проблемы. Тем не менее React позволяет без значительных усилий создавать такой род документации, что может окупить немало времени в будущем.

Но не только React может вам помочь создать библиотеку визуальных компонентов из кода самих компонент. Есть инструменты, которые помогают решить эту проблему, один из которых $react - storybook$.

React Storybook изолирует компоненты, предоставляя вам возможность создавать компоненты без запуска всего приложения, что помогает в тестировании и разработке.

Как видно из названия библиотеки React Storybook позволяет созда-

вать истории для отображения разных состояний компонента. Например, если вы пишите TO-DO приложение, то вы можете создать две истории для отображения выбранного и невыбранного состояний элемента.

Давайте попробуем применить эту библиотеку к примеру с компонентом *List*. Прежде всего нам нужно установить Storybook:

Теперь мы можем начать создавать истории.

В нашем примере компонент *Item* требует обязательный параметр *title* и опциональный *text*, в этом случае мы можем создать как минимум две истории.

Обычно истории хранят в директории *stories* внутри проекта, но в целом никто не запрещает использовать любую удобную для вас директорию.

Внутри этой директории можно создать специально файл для каждой из компонент.

В нашем случае создадим файл *list.js*. В этом файле обязательно необходимо добавить импорт основной функции библиотеки:

```
import { storiesOf } from '@kadira/storybook'
```

Дальше мы можем создать истории следующим образом:

```
storiesOf('List', module)
  .add('without text field', () => (
    <List collection={posts} titleKey="title" />
  ))
```

Функция *storiesOf* принимает аргументом название компонента и позволяет добавить для него множество историй. Каждая история включает в себя описание и функцию, которая создает необходимый компонент.

В нашем случае *posts* может быть объектом вида:

```
const posts = [
  {
    id: 1,
    title: 'Create Apps with No Configuration',
  },
  {
    id: 2,
    title: 'Mixins Considered Harmful',
  },
]
```

Перед запуском Storybook и создания нашей визуальной коллекции нам необходимо настроить библиотеку. Для этого необходимо создать директорию *.storybook*.

Внутри этой директории нам нужно создать файл *config.js* для загрузки наших историй:

```
import { configure } from '@kadira/storybook'
function loadStories() {
  require('../src/stories/list')
}
configure(loadStories, module)
```

Сначала мы загружаем функцию *configure* из библиотеки, а затем описываем функцию для загрузки историй по путям к их файлам.

И последний шаг, если мы хотим, чтобы storybook с нашими компонентами был доступен из браузера, мы можем добавить специальный скрипт в package.json:

```
"storybook": "start-storybook -p 9001"
```

Теперь мы можем запустить storybook:

```
npm run storybook
```

И открыть в браузере *http* : *//localhost* : 9001.

Теперь мы можем увидеть интерфейс storybook. Слева находится список наших историй. Если мы нажмем на историю, то справа увидим соответствующий ей компонент.

Отлично, теперь у нас есть визуальная документация наших компонент, чтобы все члены команды, в том числе продуктовые менеджеры и дизайнеры, имели представление о существующей базе готовых компонентов.

В завершение мы можем создать еще одну историю.

Наш лист умеет отображать элементы с *title* и *text*, поэтому добавим второй атрибут в список постов:

```
const posts = [
  {
    id: 1,
    title: 'Create Apps with No Configuration',
    excerpt: 'Create React App is a new officially supported
       ...',
  },
  {
    id: 2,
```

```
    title: 'Mixins Considered Harmful',
    excerpt: '"How do I share the code between several...',
  }
]
```

После этого добавим еще одну историю, где передадим оба параметра:

```
.add('with text field', () => (
  <List collection={posts} titleKey="title" textKey="excerpt"
      />
))
```

Если мы сейчас вернемся в браузер, то увидим, что наша страница автоматически обновилась, и добавилась вторая история.

Таким образом для сложных компонентов мы можем добавить любое количество историй, чтобы показать все состояния, в которых они могут находиться.

3.6 Заключение

Поздравляю, мы разобрались с созданием переиспользуемых компонент.

В этой главе мы детально посмотрели как можно создавать компоненты и в чем различие между компонент-функциями и компонентами с внутренним состоянием. Также мы изучили как осуществляется изменение состояния React компонент и как это приводит к перерисовке компонента. Помимо этого мы посмотрели как описывать параметры компонента и как это помогает в совместной работе надо общими элементами.

И в конце мы посмотрели на примеры превращения узкоспециализированных компонент в переиспользуемые посредством вынесения общей логики в достаточно абстрактные базовые компоненты.

Теперь пришло время посмотреть на разные техники комбинации компонентов между собой.

Глава 4

Собираем все в кучу

В предыдущей главе мы разобрались, как создавать переиспользуемые компоненты. Теперь мы можем поговорить о том, как заставить это компоненты эффективно взаимодействовать друг с другом.

Сильной стороной React является то, что он позволяет создавать сложные интерфейсы комбинирование маленьких, тестируемых компонент. Этот подход позволяет контролировать каждый аспект приложения.

В этой главе мы рассмотрим самые распространенные паттерны и инструменты для комбинирования компонент.

Мы обсудим следующие вопросы:

- Как компоненты коммуницируют друг с другом посредство передачи *props* дочерним элементам
- Как паттерн Контейнер и Представление помогает писать более поддерживаемый код
- Проблему, которую пытались решить миксины (mixins), но не смогли
- Улучшение структуры приложения с Компонентами Высшего порядка
- Библиотеку recompose и ее встроенные функции
- Как мы можем взаимодействовать с контекстом и как избежать сильной связности компонентов с ним

- Паттерн Function as a Child и какую пользу он может принести

4.1 Взаимодействие компонентов

Переиспользуемые компоненты могут использоваться внутри множества других компонент в процессе разработки вашего приложения.

Небольшие компоненты с простым интерфейсом могут составлять более сложные компоненты, которые в свою очередь являются частью еще более сложных компонент и приложения в целом.

Мы уже неоднократно видели, как в React объединяются компоненты. Для этого достаточно описать структуру из вложенных компонент внутри метода *render*:

```
const Profile = ({ user }) => (
  <div>
    <Picture profileImageUrl={user.profileImageUrl} />
    <UserName name={user.name} screenName={user.screenName}
       />
  </div>
)
Profile.propTypes = {
  user: React.PropTypes.object,
}
```

Например вы можете создать компонент $Profile$ путем комбинирования компонентов $Picture$ для отображения изображения профиля и $UserName$ для имени пользователя.

Таким образом вам требуется всего нескольких строчек кода для добавления новых блоков интерфейса.

После того как вы объединили компоненты как на примере выше, вы можете передавать между ними данные, используя *props*.

Props - основной способ передачи данных от родительских компонент дочерним в React.

Когда компонент передает данные другому компоненту, он является **Владельцем (Owner)** этого компоненты, не зависимо от иерархической принадлежности каждого из них.

Например, в последнем примере $Profile$ не является непосредственным родителем $Picture$ (между ними еще тег div), но $Profile$ является владельцем $Picture$, так как передает ему данные через параметры

(прим.пер. далее я все равно буду называть такие компоненты родительскими, просто в чуть более обобщенном значение).

4.1.1 Children

Есть специальный параметр **children**, который передается от родительского компонента дочерним и доступен в методе render.

В документации React говорится, что это *непрозрачный (opaque)* параметр, так как он не несет никакой информации о том, что именно внутри него содержится.

Вложенные компоненты, определенные в методе *render*, обычно получают параметры через атрибуты в JSX (или через второй аргумент метода *createElement*).

Также компонент можно определить с вложенными компонентами, в этом случае они будут доступны для него через параметр *children*.

Представим, что у нас есть компонент *Button*, у которого есть параметр *text*, отвечающий за текст на кнопке:

```
const Button = ({ text }) => (
  <button className="btn">{text}</button>
)
Button.propTypes = {
  text: React.PropTypes.string,
}
```

Этот компонент можно использовать следующим образом:

```
<Button text="Click me!" />
```

Теперь предположим, что мы хотим использовать ту же самую кнопку с тем же className, но отображать внутри нее что-то более сложное чем просто текст.

Что если мы хотим, чтобы у нас были кнопки с текстом, кнопки с изображением и кнопки с текстом и заголовком?

В множестве случаем достаточным решением будет добавить множество параметров в компонент *Button* или создать специализированные компоненты, например *IconButton*.

Однако, если мы понимаем, что *Button* всего лишь обертка, которая должна отображать любое содержимое, то мы можем использовать параметр *children*.

Мы можем легко поправить предыдущий вариант *Button*, чтобы иметь возможность отображать любое содержимое:

```
const Button = ({ children }) => (
  <button className="btn">{children}</button>
)
Button.propTypes = {
  children: React.PropTypes.array,
}
```

Теперь мы можем использовать любые компоненты внутри *Button*, они будут подставлены вместо *children* в JSX.

Например мы можем создать кнопку с изображением и текстом внутри:

```
<Button>
  <img src="..." alt="..." />
  <span>Click me!</span>
</Button>
```

В этом случае мы получим следующий HTML код:

```
<button className="btn">
  <img src="..." alt="..." />
  <span>Click me!</span>
</button>
```

Это очень удобный способ, чтобы позволить компонентам принимать любые дочерние элементы и оборачивать их предопределенным образом.

Как вы могли заметить в предыдущем примере, мы определили параметр *children* как массив, что значит, что можно передать любое количество элементов.

Но если мы передадим только один элемент, например:

```
<Button>
  <span>Click me!</span>
</Button>
```

то получим следующую ошибку:

> **Failed prop type: Invalid prop ‘children‘ of type ‘object‘ supplied to ‘Button‘, expected ‘array‘.** (Неверный тип параметра: неверный тип параметра ’children’ с типом ’объект’, переданный компоненту *Button*; ожидается ’массив’)

Это происходим из-за того, что в случае с передачей одиночного элемента React оптимизирует выделение памяти и используем сам элемент вместо создания массива с одним элементом.

Мы можем легко это поправить, указав в propTypes не только массив, но и одиночный элемент:

```
Button.propTypes = {
  children: React.PropTypes.oneOfType([
    React.PropTypes.array,
    React.PropTypes.element,
  ]),
}
```

4.2 Паттерн Контейнер и Представление

В этой главе мы рассмотрим паттерн, который поможет сделать наш код еще чище и более поддерживаемым.

Как правило React компоненты представляют из себя сочетание **логики** и **отображения**.

Под логикой мы понимаем все, что не относится к UI, т.е. такие вещи как обращения к API сервера, преобразование данных и обработку событий.

А под представлением наоборот, ту часть, которая отвечает за создание элементов для UI. Прежде всего это содержимое метода *render*.

В React есть простой и мощный паттерн, **Контейнер и Представление (Container and Presentational)**, который помогает разделить по отдельным компонентам две эти составляющие.

Заодно посмотрим в этой главе, какая еще польза, помимо переиспользуемости компонент, может быть от разделения логики и представления.

Как всегда начнем изучения паттерна с примера, в котором он используется.

Предположим, у нас есть компонент, который получает из API геолокации долготу и широту, а затем отображает их на экране.

Для начала создадим файл *geolocation.js* и определим в нем компонент *Geolocation*:

```
class Geolocation extends React.Component
```

В этом компоненте создадим конструктор для определения начального состояния и привязки обработчиков событий:

```
constructor(props) {
  super(props)
```

```
  this.state = {
    latitude: null,
    longitude: null,
  }
  this.handleSuccess = this.handleSuccess.bind(this)
}
```

Теперь в *componentDidMount* мы можем осуществить вызов API:

```
componentDidMount() {
  if (navigator.geolocation){
    navigator.geolocation.getCurrentPosition(this.
        handleSuccess)
  }
}
```

После того, как компонент получит данные от сервера, их можно сохранить во внутреннее состояние компонента:

```
handleSuccess({ coords }) {
  this.setState({
    latitude: coords.latitude,
    longitude: coords.longitude,
  })
}
```

И в конце концов мы можем отобразить высоту и широту на экране через метод *render*:

```
render() {
  return (
    <div>
      <div>Latitude: {this.state.latitude}</div>
      <div>Longitude: {this.state.longitude}</div>
    </div>
  )
}
```

Важно заметить, что после первой отрисовки компонента значение долготы и широты равно *null*, так как запрос на данные асинхронен и лишь инициализируется в *componentDidMount*. В реальном проекте вы скорее всего захотите в этот момент показывать какой-то индикатор загрузки, что можно сделать с помощью условных операторов, подробно разобранных в Главе 2.

Но в целом в этом компоненте нет никаких проблем и он прекрасно работает.

Предположим, что мы работаем с дизайнером над UI составляющей компонента. Не было бы это хорошей идеей, создать компонент, состоящий только из UI части, чтобы иметь возможность быстрее обсудить ее с дизайнером.

Если мы отделим представление от логики, то мы сможем без проблем добавить его в документацию на основе **Storybook**, как мы делали в одной из предыдущих глав.

Как вы уже можете догадаться, использование паттерна Контейнер и Представление предполагает разделение компонента на два, с более четкой зоной ответственности у каждого.

Если быть более точным, то в Контейнере находится вся логика и работа с данными, а в Представлении создание элементов и минимум логики. Чаще всего компонент представления может быть выражен компонент-функцией.

Но это не значит, что в Представлении не может быть состояния вообще. В некоторых случаях, например при создании полей ввода, может быть уместнее хранить состояние в компоненте представления.

В случае нашего примера мы отображаем на экране лишь широту и долготу, поэтому мы воспользуемся компонент-функцией для создания Представления.

Для начала переименуем наш компонент *Geolocation* в *GeolocationContainer*:

```
class GeolocationContainer extends React.Component
```

А также переименуем файл, содержащий этот компонент, из *geolocation.js* в *geolocation − container.js*.

Такой вариант наименования не высечен в камне, но является наиболее распространенным в сообществе React. К компоненты Контейнера мы добавляем в конце *Container*, а компоненте Представления оставляем оригинальное имя.

Также нам нужно изменить реализацию метода *render*, заменив все содержимое отрисовкой одного компонента:

```
render() {
  return (
    <Geolocation {...this.state} />
  )
}
```

Таким образом, вместо отрисовки HTML элементов мы просто отображаем компонент Представления и передаем в него свое состояние.

В **состоянии** нашего компонента высота и широта, которые по умолчанию имеют значение *null* и меняются на координаты пользователя через **обратный вызов (calback)** после вызова API.

Чтобы передать состояние целиком, мы используем спред оператор (spread operator), который избавляет нас от необходимости указывать параметры один за другим вручную.

Теперь создадим файл *geolocation.js*, в котором создадим компонент Отображения:

```
const Geolocation = ({ latitude, longitude }) => (
  <div>
    <div>Latitude: {latitude}</div>
    <div>Longitude: {longitude}</div>
  </div>
)
```

Компонент-функции - очень лаконичный способ описания интерфейса. Чистые функции однозначно отображают состояние в набор элементов.

В нашем случае компонент принимает через *state* долготу и широту и отображет их внутри *div* элементов.

Мы хотим следовать лучшим практикам, поэтому определим необходимый и достаточный интерфейс для этого компонента:

```
Geolocation.propTypes = {
  latitude: React.PropTypes.number,
  longitude: React.PropTypes.number,
}
```

Следуя паттерну Контейнер и Представление, мы сорздаем глупые компоненты, которые потом можно использовать в Style guide с искусственными данными.

Если в нашем приложении в другом месте будет предполагаться такой же визуальный компонент, то нам не придется создавать компонент с нуля. Нам будет достаточно создать новый Контейнер для существующего представления, например если нам нужно будет загрузить координаты из другого сервиса.

Также другим членам команды будет проще расширить логику в контейнере, например добавить обработку ошибок, не затрагивая представления.

Также можно создать временный компонент с отображением отладочной информации для скорейшей реализации логики.

Также такой подход позволяет разделить создание компонента между разными людьми, что особенно полезно в случае больших команд и итеративных процессов разработки.

Это очень простой в использовании и полезный на практике паттерн. В большой команде он способен значительно увеличить скорость разработки и поддерживаемость написанного кода.

Но с другой стороны, использование этого паттерна без явной необходимости может значительно увеличить количество файлов и размер кодовой базы.

Не стоит начинать делить все компоненты на два сломя голову. Чаще всего стоит начинать рефакторить компонент посредством разделения логики и представления, когда они начинают быть сильно связанными.

Например в нашем примере, мы предположили, что у нас может появиться другой источник данных, для которого мы и будем создавать отдельный компонент.

Не всегда можно однозначно понять, что должно быть в Контейнере, а что в Представлении. Следующий список утверждений должен помочь вам в сложной ситуации:

Компонент Контейнера:

- Сосредоточен больше на поведении
- Отображает компонент Представления
- Выполняет асинхронные запросы к серверу и преобразует данные
- Определяет обработчики событий
- Создаются как наследуемые от React.Component классы

Компонент Представления:

- Сконцентрированы на визуальной составляющей
- Отображают HTML разметку (и другие компоненты)
- Получают данные от родительского компонента через *props*
- Часто определяются через компонент-функции без состояния

4.3 Mixins

Компоненты отлично служат цели достижения переиспользуемости кода, но что если у нас появляется множество различных компонент, которые должны обладать общими чертами?

Очевидно, мы не хотим дублировать код, к счастью React предоставляет специальный инструмент для решения этой проблемы: **примеси (mixins)**.

В общем и целом примеси не рекомендуются к использованию, но все равно стоит знать, какие проблемы они решают и какие есть альтернативы.

Также есть не нулевая вероятность, что вас может занести на проект с кучей старого кода, где могут во всю применяться примеси, поэтому быть готовым к такому повороту лишним не будет.

Начать стоит с того, что примеси работают только с *createClass*, что является одной из причин предания их забвению.

Предположим, что вы используете *createClass* и понимаете, что вам нужно написать один и тот же код в разные компоненты.

Например, вам нужно подписаться на событие изменения размера экрана и выполнять по нему какой-то код.

Собственно его можно написать один раз и передавать через примеси в любые компоненты. Посмотрим на примере кода.

Точкой соприкосновения компонента и примеси обычно выбирается *state*. Мы можем выделить в *state* конкретное поле и использовать его и из компонента и из примеси. В остальном примесь описывается как обычный самостоятельный компонент.

Определим в нашей примеси начальное состояние с помощью метода *getInitialState*, в котором будет одно поле *innerWidth*:

```
getInitialState() {
  return {
    innerWidth: window.innerWidth,
  }
},
```

Теперь мы можем начать отслеживать изменения размера экрана, для чего подпишемся на соответсвующее событие:

```
componentDidMount() {
  window.addEventListener('resize', this.handleResize)
},
```

Также мы хотим удалить этот обработчик события перед удалением компонента, чтобы избежать накопления неиспользуемых обработчиков в объекте *window*:

```
componentWillUnmount() {
  window.removeEventListener('resize', this.handleResize)
},
```

И осталось только создать функцию, которая будет вызваться на каждом изменении размера экрана.

В этой функции мы будем обновлять значение поля $innerWidth$ в *state* актуальным значением, так что любой компонент, который использует эту примесь, будет перерисован как после собственного $setState$:

```
handleResize() {
  this.setState({
    innerWidth: window.innerWidth,
  })
},
```

Как видно из примера, создание примеси почти не отличается от создания обычного компонента.

Чтобы использовать эту примесь вместо с компонентом, достаточно добавить ее в массив *mixins* внутри компонета:

```
const MyComponent = React.createClass({
  mixins: [WindowResize],
    render() {
      console.log('window.innerWidth', this.state.innerWidth)
      ...
  },
})
```

С этого момента значение $innerWidth$ будет доступно не только в примеси, но и в компоненте, который будет перерисовывать при каждом обновлении состояния из примеси.

Само собой мы можем использовать одну и ту же примесь в множестве компонент, также и внутри одного компонента может использоваться сразу множество примесей.

Очень полезной особенностью примесей является то, что они обладают одинаковым с компонентами жизненным циклом, а также возможностью задать состояние по умолчанию.

Например, если мы используем $WindowResize$ в компоненте, в котором уже есть $componentDidMount$, то никаких коллизий не произойдет,

и оба метода выполнятся.

Теперь посмотрим, в чем проблемы примесей и почему от них отказались. А в следующей части разберемся, как достигнуть такого же результата другими средствами.

Во первых примеси часто используют внутренние функции для взаимодействия с компонентом.

Например, наша примесь $WindowResize$ может ожидать, что функция обратного вызова $handleResize$ будет реализована внутри компонента, что даст возможность разработчикам большую свободу в обработке изменения размера экрана.

Или наоборот, примесь хочет получать данные из компонента и дергает специальный метод, что-то вроде $getInnerWidth$. Само собой этот метод тоже должен быть реализован внутри компонента.

К сожалению, нет никакой возможности получить точный список методов, которые должны быть реализованы внутри компонента при добавлении примеси.

Такой подход очень сильно ухудшает поддерживаемость кода. Если компонент использует множество примесей, то при их удалении или изменении очень сложно выделить код, которые также может быть удален или требует модификации.

Также частая проблема - конфликты имен. Очень часто примеси могут начать требовать функции или атрибуты с одинаковыми названиями. React без проблем разделяет вызовы методов жизненного цикла компонент, но совсем ничего не может сделать с вызовами пользовательских функций.

Таким образом примесям остается использовать внутреннее состояние компонент, что не очень хорошо, так как мы пытаемся наоборот сократить его использование с целью повышения переиспользуемости.

Помимо этого, может начать складываться ситуация, когда одни примеси начинают зависеть от других. Например, мы можем создать еще одну примесь **ResponsiveMixin**, которая будет скрывать некоторые элементы с экрана в зависимости от текущего размера экрана, который мы получаем из примеси $WindowResize$.

Такая тесная связь примесей значительно усложняет отладку приложения и его масштабируемость.

4.4 Компоненты высшего порядка

В прошлой части мы посмотрели, как примеси помогают избежать дублирования кода при создании общего для компонент функционала, и какие проблемы это приносит.

Когда мы говорили о функциональном программировании в Главе 2, мы упоминали концепцию **Функций высшего порядка (Higher-order Functions, HoFs)**. Такая функция принимает аргументом другую функцию и возвращает ее с измененным поведением.

Посмотрим, можем ли мы применить этот подход к React компонентам и достигнуть цели переиспользования функционала множеством компонент.

В случае применения данной концепции к компонентам React они станут называться **Компонентами высшего порядка (Higher-order Components, HoCs)**

Структура любого HoC выглядит следущим образом:

```
const HoC = Component => EnhancedComponent
```

Компонент высшего порядка - это функция, которая принимает аргументом React компонент и возвращает его с расширенным функционалом.

Давайте начнем с простого примера, чтобы как это все выглядит на практике.

Предположим, что вам по какой-то причине необходимо добавить к множеству компонент один и тот же *className*. Никто не запрещает обойти все компоненты и в каждом поправить метод *render*, а можно создать один HoC, который решит нашу проблему:

```
const withClassName = Component => props => (
  <Component {...props} className="my-class" />
)
```

Если вы впервые встречаете эту концепцию, может быть не очевидно, как работает этот код, поэтому давайте детально разбираться, что тут происходит.

Мы определили функцию *withClassName*, которая принимает аргументом компонент *Component* и возвращает другую функцию.

Эта созданная функция есть обыкновенная компонент-функция, которая принимает аргументом параметры *props* и возвращает компонент

Component, передавая ему с помощью спред оператора все параметры и в дополнение к ним параметр *className* со значением "*my − class*".

Чаще всего HoC передают параметры дальше через спред оператор. Это делается для того, чтобы HoC меньше зависел от изменения API компонента, а также чтобы только добавлять поведение и минимально затрагивать поведение самого компонента.

Это очень простой пример, который скорее всего никогда не пригодился бы в реальном проекте, но на нем мы посмотрели как выглядит HoC и как его можно создать.

Теперь посмотрим, как *withClassName* можно использовать с другими компонентами.

Прежде всего создадим компонент, который принимает в параметрах *className* и добавляет его к *div* элементу:

```
const MyComponent = ({ className }) => (
  <div className={className} />
)
MyComponent.propTypes = {
  className: React.PropTypes.string,
}
```

Но вместо того, чтобы использовать этот компонент напрямую, мы передадим его созданному ранее HoC'у, и по сути получим новый компонент:

```
const MyComponentWithClassName = withClassName(MyComponent)
```

Оборачивая наш компонент в *withClassName*, мы гарантируем получение компонентом параметра *className*.

Давайте теперь попробуем сделать что-то более впечатляющее и переделаем примесь *WindowResize* из предыдущей части в HoC, чтобы снова иметь возможность переиспользоввать ее в сферическом проекте в вакууме.

Напомним, что эта примесь создавала обработчик для отслеживания изменения размера экрана и сохраняла актуальное значение в поле *innerWidth* внутри состояния компонента.

Основная проблема была в том, что примесь использовала *state* компонента, чтобы передавать ему актуальные данные.

Это не очень хорошее поведение, так как могут возникнуть конфликты имен внутри состояния компонента.

Прежде всего создадим функцию, которая принимает аргументом компонент:

```
const withInnerWidth = Component => (
  class extends React.Component { ... }
)
```

Возможно вы обратили наименование НоС. Это распространенная практика начинать название с *with*, если НоС расширяет параметры, которые передаются компоненту.

Помимо этого, *withInnerWidth* будет возвращать компонент-класс, а не компонент-функцию, так как нам потребуется использовать внутреннее состояние и методы жизненного цикла.

Посмотрим, как будет выглядеть возвращенный класс.

В конструкторе мы определим начальное состояние и привяжем функцию обработчика событий к создаваемому экземпляру класса:

```
constructor(props) {
  super(props)
  this.state = {
    innerWidth: window.innerWidth,
  }
  this.handleResize = this.handleResize.bind(this)
}
```

Добавление и удаление обработчиков события изменения размера экрана и обновление внутреннего состояния аналогично уже реализованному в примеси:

```
componentDidMount() {
  window.addEventListener('resize', this.handleResize)
}
componentWillUnmount() {
  window.removeEventListener('resize', this.handleResize)
}
handleResize() {
  this.setState({
    innerWidth: window.innerWidth,
  })
}
```

И в конце нам нужно реализовать метод *render*, в котором мы должны отобразить изначальный компонент, передавая ему новые данные:

```
render() {
  return <Component {...this.props} {...this.state} />
}
```

Можно обратить внимание, что мы через спред оператор передаем не только параметры, но также и внутреннее состояние.

По сути мы аналогично примеси храним *innerWidth* внутри состояния, но передаем его не в *state* изначального параметра, а в его *props*.

Как мы уже говорили в Главе 3, использование параметров чаще всего предпочтительнее состояния в разрезе повышения переиспользуемости компонент.

Теперь мы можем без проблем обернуть любой компонент, который ожидает параметр *innerWidth* (или не ожидает, но зачем тогда все это..) в *withInnerWidth* HoC.

Создадим для примера компонент, который получает параметр *innerWidth* и просто выводит его значение на экран:

```
const MyComponent = ({ innerWidth }) => {
  console.log('window.innerWidth', innerWidth)
  ...
}
MyComponent.propTypes = {
  innerWidth: React.PropTypes.number,
}
```

Который мы можем теперь обернуть функцией *withInnerWidth* следующим образом:

```
const MyComponentWithInnerWidth = withInnerWidth(MyComponent)
```

Есть несколько преимуществ использования этого подхода перед примесями: прежде всего мы не затрагиваем внутреннее состояние исходного компонента, а также не требуем (и не ожидаем) от него реализации каких-либо специфичных методов.

Это значит, что и исходный компонент и компонент высшего порядка не связаны, что позволяет переиспользовать их независимо друг от друга в дальнейшем.

Также передача данных через параметры позволяет уменьшить количество логики внутри исходного компонента, что упрощает его использование внутри Style Guide.

В этом случае нам достаточно создать компонент с разными размера экрана, которые мы поддерживаем внутри приложения.

Т.е. мы без проблем можем передать конкретное число в через параметры так:

```
<MyComponent innerWidth={320} />
```

Или так:

```
<MyComponent innerWidth={960} />
```

4.5 Recompose

В предыдущей главе мы познакомились с компонентами высшего порядка и на примерах посмотрели, как они работают.

Есть библиотека, которая называется **recompose**, которая предоставляет набор полезных HoC, а также удобный способ их комбинировать.

Библиотечные HoC представляют из себя простые вспомогательные компоненты, которые помогают вынести часть логики из компонент, что конечно же делает их проще и более переиспользуемыми (прим. пер. за питоном не ходи, чтобы найти здесь самое переиспользуемое слово...).

Предположим, что наш компонент получает объект с данными пользователя из API, и у этого объекта есть множество атрибутов.

Получение сложного объекта компонентом в общем случае считается не самой лучшей практикой. Если компонент получает сложный объект, то скорее всего он знает о структуре этого объекта (или его части), а это ведет к том, что в случае изменения структуры этого объекта компонент будет сломан.

Будет гораздо лучше, если необходимые данные будут переданы в виде отдельных параметров с примитивными значениями.

Пусть у нас есть компонент $Profile$, в котором мы хотим отобразить $username$ и age:

```
const Profile = ({ user }) => (
  <div>
    <div>Username: {user.username}</div>
    <div>Age: {user.age}</div>
  </div>
)
Profile.propTypes = {
  user: React.PropTypes.object,
}
```

Если мы хотим изменить интерфейс компонента, чтобы получать одиночные параметры вместо полного объекта, мы можем воспользоваться $flattenProp$ HoC из библиотеки recompose.

Посмотрим, как это работает.

Для начала поправим сам компонент, чтобы получать в нем одиночные параметры:

```
const Profile = ({ username, age }) => (
  <div>
    <div>Username: {username}</div>
    <div>Age: {age}</div>
  </div>
)
Profile.propTypes = {
  username: React.PropTypes.string,
  age: React.PropTypes.number,
}
```

Теперь обернем его в *flattenProp* HoC:

```
const ProfileWithFlattenUser = flattenProp('user')(Profile)
```

Вы можете заметить, что мы используем этот HoC немного не так как предыдущие. Сами HoC также могут зависеть от некоторых параметров, тогда обычно сначала передают их, а потом уже компонент. В общем случае такие HoC имеют следующую структуру:

```
const HoC = args => Component => EnhancedComponent
```

За счет этого мы можем разделить создание конкретного HoC с определенным набором параметром и использование его с копонентами:

```
const withFlattenUser = flattenProp('user')
const ProfileWithFlattenUser = withFlattenUser(Profile)
```

Уже неплохо. Но сейчас параметры компонента завязаны на то, что это именно данные о пользователе. Давайте сделаем их более обобщенными.

В этих целях мы можем использовать *renameProp* HoC из recompose и обновить компонент следующим образом:

```
const Profile = ({ name, age }) => (
  <div>
    <div>Name: {name}</div>
    <div>Age: {age}</div>
  </div>
)
Profile.propTypes = {
  name: React.PropTypes.string,
  age: React.PropTypes.number,
}
```

Теперь мы можем приметь оба HoC (один для выделения простых параметров из объекта *user* и второй для их переименования) к компоненту. Но множество вложенных вызовов функций будет ужасно читаться.

Тут нам на помощь приходит функция *compose* библиотеки *recompose*.

Она делает очень простую вещь, принимает множество компонент высшего порядка и возвращает функцию (по сути тоже HoC), которая может применить их к какому-либо компоненту:

```
const enhance = compose(
  flattenProp('user'),
  renameProp('username', 'name'),
  withInnerWidth
)
```

Как можете увидеть, функция *compose* значительно улучшает читаемость кода.

Мы можем объединить множество HoC, чтобы сохранить изначальный компонент настолько простым, насколько это возможно.

Но и тут важно не переусердствовать, так как каждое добавление слоя абстракции потенциально может принести проблем, а конкретно в данном случае, множество вложенных HoC могут сказаться на производительности.

Нужно держать в голове, что добавляя каждый новый HoC, вы добавляете еще один метод *render*, еще одну пачку методов жизненного цикла и выделяете на это память.

Если у вас появляются глубокие вложенные компоненты высшего порядка, то стоит задуматься, возможно у вас что-то поломалось в структуре самого приложения.

4.5.1 Context

Также компоненты высшего порядка очень удобны в работе с контекстом.

Контекст (Context) - инструмент библиотеки React, который используется во множестве библиотек, хотя был задокументирован значительно позже своего появления.

Документация до сих пор рекомендует при возможности не использовать контекст, так как он еще находится в стадии эксперимента и его API может в будущем измениться.

Однако этот инструмент очень полезен в случаях, когда нам нужно передать данные ниже по дереву элементов, но при этом не передавать их через каждый уровень в *props*.

Компоненты высшего порядка и контекст образуют очень мощную связку, так как позволяют передавать данные ниже по дереву, но при этом избежать сильной связи между компонентами и API контекста.

Схема проста - HoC получает данные из контекста, преобразует в *props* и передает компоненту.

В этом случае компонент ничего не знает о существовании контекста и может быть переиспользован в любом месте приложения.

Помимо этого, в случае изменения API контекста, нам не придется исправлять все компоненты, нужно будет лишь поправить необходимые HoC.

В библиотеки recompose есть специальный метод, который делает процесс извлечения данных из контекста понятным и одинаковым для всех компонент.

Предположим, что у вас есть компонент *Price*, который вы используете для отображения валюты и величины.

Контекст часто используется для того, чтобы передавать общие настройки приложения всем компонентам, валюта может быть одной из таких настроек.

Давайте начнем с компонента, который сам работает с контекстом, и шаг за шагом переделаем его в более универсальный:

```
const Price = ({ value }, { currency }) => (
  <div>{currency}{value}</div>
)
Price.propTypes = {
  value: React.PropTypes.number,
}
Price.contextTypes = {
  currency: React.PropTypes.string,
}
```

У нас есть компонент-функция, которая принимает значение как параметр, а валюту вторым аргументом из контекста.

Также для обоих параметром мы определили типы (prop types и context types).

Как видим, его переиспользуемость сильно ограничивается потребностью в родительском элементе с *currency* в контексте.

Например, мы не сможем без проблем использовать его в Style guide, так как не сможем передать валюту через параметры.

Прежде всего поменяем компонент так, чтобы он получал оба значения через параметры:

```
const Price = ({ currency, value }) => (
  <div>{currency}{value}</div>
)
Price.propTypes = {
  currency: React.PropTypes.string,
  value: React.PropTypes.number,
}
```

Конечно, нельзя просто так взять и заменить старый компонент новым, так как нет родительского элемента, который бы передал в параметрах валюту.

Но мы можем создать специальный HoC, чтобы перенести в параметры компонента данные из контекста.

Мы будем использовать функцию *getContext* из recompose, но ничего не мешает вам написать собственную реализацию с нуля.

Создадим отдельно сам HoC с помощью *getContext*, таким образом его можно будет переиспользовать множество раз:

```
const withCurrency = getContext({
  currency: React.PropTypes.string
})
```

И мы можем применить его к нашему компоненту:

```
const PriceWithCurrency = withCurrency(Price)
```

Теперь мы можем заменить старый компонент *Prive* новым, и компонент будет работать без явной привязки к контексту.

Для нас это большая победа, так как нам совсем не пришлось изменять родительские компоненты, но теперь мы меньше завязаны на Context API, которое может измениться, и наш компонент стал гибче в использовании.

4.6 Функция как Потомок

Есть еще один паттерн в React, о котором точно стоит знать, он называется **Функция как Потомок (Function as Child)**

Чаще всего вместе с ним вспоминают библиотеку react-motion, о которой мы подробнее поговорим в Главе 6.

Основная идея здесь заключается в том, что мы вместо того, чтобы передавать компоненту дочерние элементы, передаем ему функцию, которая сможет создать ему дочерний элемент, и в аргументах которой этот элемент сможет передать ей данные.

Посмотрим, как это выглядит:

```
const FunctionAsChild = ({ children }) => children()
FunctionAsChild.propTypes = {
  children: React.PropTypes.func.isRequired,
}
```

Как вы видите, компонент *FunctionAsChild* смотрит на параметр *children* как на функцию. И вместо того, чтобы использовать его внутри JSX, вызывает его.

Этот компонент может быть использован следующим образом:

```
<FunctionAsChild>
  {() => <div>Hello, World!</div>}
</FunctionAsChild>
```

В общем-то и весь паттерн. Мы передаем в компонент *FunctionAsChild* функцию, которая создает текст "Hello, World!"внутри тега *div*. Эта функцию будет вызвана внутри метода *render* компонента *FunctionAsChild*.

Смысл этот подход начинает обретать тогда, когда этой функции через аргументы будут передаваться какие-либо данные.

Создадим компонент *Name*, который ожидает в *children* функцию и передает ей строку 'World':

```
const Name = ({ children }) => children('World')
Name.propTypes = {
  children: React.PropTypes.func.isRequired,
}
```

Воспользоваться этим компонентом можно следующим образом:

```
<Name>
  {name => <div>Hello, {name}!</div>}
</Name>
```

На экране будет тот же 'Hello, World!', но на этот раз имя передается не из компонента, где функция создается, а из компонента, в котором она вызывается.

Мы разобрались, как работает этот прием, давайте посмотрим, какую пользу он приносит.

Первый плюс в том, что мы можем обернуть компоненты, передавая им переменные во время исполнения программы, в отличие от фиксированных параметром, как мы делали в HoC.

Хороший пример - компонент *Fetch*, который загружает данные из сети и передает их функции *children*:

```
<Fetch url="...">
  {data => <List data={data} />}
</Fetch>
```

Во вторых, этот подход позволяет избежать использования предустановленных имен параметров в *children*. Так как в компонент приходит функция, то их может определить разработчик, который использует этот компонент.

И также, что не менее важно, такой компонент очень удобен для переиспользования, так как он не делает никаких предположений относительно того, как будут выглядеть дочерние компоненты.

Таким образом, компонент, использующий паттерн Функция как Потомок, может быть использован в разных частях приложения с разными дочерними компонентами.

4.7 Заключение

В этой главе мы научились комбинировать наши переиспользуемые компоненты и выстраивать между ними эффективную коммуникацию.

Определение минимального и понятного интерфейса компонента через *props* - отличный способ сделать компоненты менее связными друг с другом.

Потом мы посмотрели на самые распространенные паттерны комбинирования в React.

Первым был паттерн Контейнер и Представление, который помогает отделить логику работы компонента от его отображения и создавать узкоспециализированные компоненты, которые следуют принципу единственности ответственности.

Мы посмотрели, как React предлагает решить проблему использования общего кода между компонентами с помощью примесей. К сожале-

ния, этот подход помимо решения проблемы, приносит множество новых, а также негативно сказывается на поддерживаемости приложения.

Один из способов достижения той же цели - использование компонент высшего порядка (HoC), которые являясь функцией, принимают аргументом компонент и возвращают его с расширенным функционалом.

Библиотека recompose предлагает множество удобных в использовании HoC, а также удобный способ их комбинирования, что позволяет вынести еще больше логики из наших компонент.

Также мы научились использовать контекст без сильной привязки к нему компонент за счет использования HoC.

И в конце мы разобрались, как связывать компоненты динамически с помощью паттерна Функция как Потомок.

Теперь пришло время, чтобы поговорить о загрузке данных из сети и об однонаправленном потоке данных.

Глава 5

Загрузка данных

Цель этой главы - рассмотреть различные способы загрузки данных в React приложении.

Чтобы лучше понимать, как работать с загружаемыми данными, нам нужно будет разобраться как в целом распространяются данные по дереву компонентов в React.

Важно понимать, как родительские компоненты могут коммуницировать с потомками. А также как несвязанные между собой напрямую потомки могут передавать друг другу данные.

Мы посмотрим на конкретные примеры загрузки данных и улучшение структуры компонент, которые загружают данные, с HoC.

И в конце мы посмотрим на удобные библиотеки, такие как react-refetch, которые могут сохранить нам много времени, предоставляя ядро работы с сетью.

В этой главе мы рассмотрим следующие пункты:

- Как Однонаправленный поток данных в React упрощает для понимания структуру приложения
- Как дочерние элементы могут взаимодействовать с родителем через функции обратного вызова
- Как множество дочерних компонент могут делить данные между собой через общий родительский элемент
- Как создать универсальный HoC, с помощью которого можно будет загружать данные из любого API

- Как работает библиотека react-refetch, и как она может упростить работу с сетевыми запросами в нашем приложении

5.1 Поток данных

В последних двух главах мы разбирались, как создавать переиспользуемые компоненты и как эффективно их комбинировать.

Теперь мы поговорим о том, как выстроить правильный поток данных (data flow) между множеством компонент внутри нашего приложения.

React использует очень интересный паттерн, чтобы распространять данные от коренных элементов к дочерним. Этот паттерн обычно называют **Однонаправленный поток данных (Unidirectional Data Flow)**, и в этой части мы посмотрим на него детальнее.

Как видно из названия данные в React компонентах передаются в одном направлении, от корневых элементов к дочерним. У этого подхода есть множество преимуществ, так как это упрощает поведение компонент и их взаимоотношения, делая код более предсказуемым и поддерживаемым.

Каждый компонент получает данные от родительского компонента в виде параметров, которые не должен модифицировать. Также каждый компонент может при необходимости иметь собственное состояние. На основе состояния и полученных параметров компоненты может создать новые данные и передать их дальше по дереву элементов.

Во всех примерах, которые мы видели на текущий момент, данные передавались только от родительских компонент дочерним.

Однако, что делать, если появилась необходимость передать данные от дочернего элемента родительскому?, или родительский элемент должен быть обновлен при изменении состояния дочернего?, или два дочерних элемента хотят передать данные друг другу? Мы ответим на все эти вопросы в процессе разбора примеров из жизни.

Мы начнем с простого компонента, у которого нет потомков, и шаг за шагом преобразуем его в компонент чистый и структурированный.

Мы должны посмотреть, какие паттерны на каждом из шагов преобразования компонента подходят больше всего.

Давайте погрузимся в создание компонента счетчика *Counter*, который имеет две кнопки для увеличения и уменьшения счетчика и значению 0 по умолчанию.

Начнем с создания класса, который наследует *React.Component*:

```
class Counter extends React.Component
```

В конструкторе мы зададим начальное значение счетчика и привяжем к компоненту обработчики событий:

```
constructor(props) {
  super(props)
  this.state = {
    counter: 0,
  }
  this.handleDecrement = this.handleDecrement.bind(this)
  this.handleIncrement = this.handleIncrement.bind(this)
}
```

Обработчики событий будут также просты, им достаточно изменять состояние компонента, увеличивая или уменьшая значение счетчика:

```
handleDecrement() {
  this.setState({
    counter: this.state.counter - 1,
  })
}
handleIncrement() {
  this.setState({
    counter: this.state.counter + 1,
  })
}
```

И в конце нам нужно создать метод *render*, в котором мы будем отображать текущее состояние счетчика и кнопки для его изменения:

```
render() {
  return (
    <div>
      <h1>{this.state.counter}</h1>
      <button onClick={this.handleDecrement}>-</button>
      <button onClick={this.handleIncrement}>+</button>
    </div>
  )
}
```

5.1.1 Взаимодействие потомка с родителем (callbacks)

В общем и целом этот компонент работает, но он делает несколько вещей:

- Содержит во внутреннем состоянии счетчик
- Отвечает за отображение данных
- Содержит логику по увеличению и уменьшению счетчика

Стоит стремиться к тому, чтобы компоненты оставались небольшими и отвечающими за определенную вещь. Это улучшает поддерживаемость приложения и увеличивает шансы пережить изменения требований с меньшей кровью.

Предположим, что нам нужны такие же кнопки плюса и минуса в другом блоке приложения.

Было бы прекрасно, если бы мы смогли переиспользовать кнопки, которые созданы внутри компонента *Counter*, но встает вопрос: если мы вынесем кнопки за границы компонента, то как узнать, когда они были нажаты, чтобы изменить состояние счетчика?

Посмотрим, как мы можем это сделать.

Создадим компонент *Buttons*, который будет отображать необходимые нам кнопки, но вместо того, чтобы определять функции для обработки событий нажатия внутри этого компонента, он будет ожидать их из параметров:

```
const Buttons = ({ onDecrement, onIncrement }) => (
  <div>
    <button onClick={onDecrement}>-</button>
    <button onClick={onIncrement}>+</button>
  </div>
)
Buttons.propTypes = {
  onDecrement: React.PropTypes.func,
  onIncrement: React.PropTypes.func,
}
```

Это обычная компонент-функция, которая передает кнопкам через параметр *onClick* функции, которые получает из параметров.

Теперь мы можем интегрировать этот компонент с кнопками в наш компонент *Counter*:

```
render() {
  return (
    <div>
      <h1>{this.state.counter}</h1>
      <Buttons
```

```
          onDecrement={this.handleDecrement}
          onIncrement={this.handleIncrement}
        />
      </div>
    )
}
```

Как видно, в компоненте *Counter* меняется только блок с кнопками на новый компонент, которому через параметры передаются обработчики событий.

Компонент с кнопками теперь сам по себе ничего не знает о том, кто его использует и лишь уведомляет о нажатиях.

Таким образом, если нам нужно передавать какие-либо данные из дочернего компонента в родительский, то мы можем передать потомку функцию для обратного вызова и реализовать всю остальную логику внутри родительского компонента.

5.1.2 Общий предок

Теперь компонент *Counter* выглядит уже гораздо лучше. Осталось вынести из него часть, отвечающую за отображение.

Чтобы сделать это, мы можем создать компонент *Display*, который будет получать значение и выводить его на экран:

```
const Display = ({ counter }) => <h1>{counter}</h1>
Display.propTypes = {
  counter: React.PropTypes.number,
}
```

Так как нам не нужно хранить состояние внутри этого компонента, мы можем использовать компонент-функцию. Также стоит сказать, что конкретно в этом примере не так много смысла выносить отображение одного $h1$ элемента в отдельный компонент, но в общем случае у вас тут могут быть еще стили, логика смены цвета в зависимости от значение и так далее.

В общем случае, мы должны стремиться делать компоненты так, чтобы они не знали о том, кто именно источник данных, в этом случае их можно будет значительно проще переиспользовать в разных частях приложения.

Теперь мы можем заменить старую разметку в компоненте *Counter* новым компонентом *Display*:

```
render() {
  return (
    <div>
      <Display counter={this.state.counter} />
      <Buttons
        onDecrement={this.handleDecrement}
        onIncrement={this.handleIncrement}
      />
    </div>
  )
}
```

Как вы можете видеть, два дочерних компонента (*Display* и *Buttons*) коммуницируют посредством общего предка, компонента *Counter*.

Когда на компонент *Buttons* кто-либо кликает, он через функцию обратного вызова уведомляет об этом компонент *Counter*, который обновляет данные и передает их компоненту *Display*. Это очень распространенный и эффективный паттерн в React, который позволяет работать с общими данными компонентам, которые не обладают прямой связью.

Данные распространяются от родительских компонентов к дочерним, но последние через функции обратного вызова могут попросить родительский компонент изменить состояние и вызвать тем самым перерисовку других компонентов.

Таким образом, если у нас есть данные, который нужны двум или более компонентам, мы должны найти общего для них родительский компонент и хранить состояние там. В этом случае этот компонент сможет через параметры передавать данные в актуальном состоянии всем дочерним компонентам.

5.2 Загрузка данных

В предыдущей части мы посмотрели, как мы можем передавать данные между компонентами.

Теперь можно разобраться, как в React осуществляется загрузка данных из сети, и в какой части приложения расположить логику загрузки данных.

Примеры в этой главе для http запросов мы будем использовать функцию $fetch$, которая является современной альтернативой функции $XMLHttpRequest$.

На данный момент функция $fetch$ нативно поддерживается толь-

ко браузерами Chrome и FireFox, поэтому если вы хотите поддерживать другие браузеры, вы должны использовать **полифил (polyfill)** от GitHub:

```
https://github.com/github/fetch
```

Мы также будем использовать публичное API GitHub для использования его внутри нашего приложения. Например, мы можем использовать сервис для получения списка **гистов (gists)** пользователя:

```
https://api.github.com/users/:username/gists
```

Гисты - небольшие кусочки кода, которыми пользователи могут делиться между собой через систему GitHub.

Первым компонентом, который мы создадим, будет простой компонент со списком гистов пользователя *gaearon (Dan Abramov)*.

Приступим. Для начала создадим соответствующий класс:

```
class Gists extends React.Component
```

В конструкторе мы создадим начальное состояние, состоящие из пустого списка гистов:

```
constructor(props) {
  super(props)

  this.state = { gists: [] }
}
```

Есть два метода жизненного цикла компонента, в которых можно осуществлять загрузку данных: $componentWillMount$ и $componentDidMount$.

Первый срабатывает перед первой отрисовкой компонента, а второй сразу после окончания монтирования компонента.

Звучит разумным просто использовать первый, так как мы хотим начать загрузку как можно раньше, но есть нюанс.

По факту метод $componentWillMount$ вызывается и на клиенте и на сервере в случае отрисовки на стороне сервера (server-side rendering).

Детальнее об отрисовке на стороне сервера мы поговорим в Главе 8. Сейчас отметим, что вызов асинхронного API в момент отрисовки на стороне сервера может привести к непредсказуемому результату.

Поэтому мы будем использовать $componentDidMount$, чтобы быть уверенными, что вызов API произойдет только на клиенте.

Также стоит учесть, что в реальном проекте вы скорее всего захотите показывать индикатор загрузки во время вызова API. Сделать это можно одним из способов, описанных в Главе 2, в этой главе мы их опустим.

Как мы сказали раньше, мы хотим загрузить список гистов пользователя gaearon функцией *fetch*:

```
componentDidMount() {
  fetch('https://api.github.com/users/gaearon/gists')
    .then(response => response.json())
    .then(gists => this.setState({ gists }))
}
```

Этот код требует некоторых пояснений. Когда срабатывает метод *componentDidMount*, мы вызываем функцию *fetch* адресом нужного нам сервиса.

Функция *fetch* возвращает *Promise*, который в случае успешного выполнения возвращает объект *response* с результатом запроса. Затем из этого объекта с помощью функции *json* можно получить данные в формате JSON.

Затем этот json можно сохранить во внутреннее состояние компонента, чтобы он был доступен из метода *render*:

```
render() {
  return (
    <ul>
      {this.state.gists.map(gist => (
        <li key={gist.id}>{gist.description}</li>
      ))}
    </ul>
  )
}
```

В методе *render* мы просто обходим список гистов и оборачиваем описание каждого из них в тег <li>.

Вы могли обратить внимание на атрибут *key* элементов <li>. Это делается в целях улучшения производительности и будет разобрано подробнее в конце книги.

Если вы удалите этот атрибут, то получите предупреждение в консоли разработчика, но приложение продолжит работать.

Компонент работает, но он пока далек от идеального. Как мы уже увидели в предыдущих главах, мы можем как минимум разнести логику и отображение в разные компоненты, что сделает их проще и более тестируемыми.

Но для нас сейчас более важно то, что мы скорее всего хотим загружать данные из разных участков нашего приложения, но не хотим дублировать соответствующий код в каждый компонент.

Основной способ, который мы можем использовать для переиспользования какой-либо логики в множестве компонент, это компоненты высшего порядка (HoC).

В данном случае HoC будет загружать данные из сети и передавать дочерним компонентам через параметры.

Давайте посмотрим, как это может выглядеть.

Как мы уже знаем, HoC - это функция, которая принимает аргументом компонент (и возможно какие-то параметры) и возвращает его с расширенным функционалом.

Мы будем использовать частичное применение (partial application) для того, чтобы первым вызовом передать параметры, а компонент уже вторым:

```
const withData = url => Component => (...)
```

Мы назвали HoC *withData*, так как он будет передавать данные в параметре *data*.

Функция принимает аргументом url, по которому нужно загрузить данные, и компонент, которому эти данные нужно передать.

Реализация этого HoC будет очень похожа на сам компонент. Разница будет лишь в том, что url теперь приходит в виде аргумента функции, и в методе *render* мы отрисовыываем дочерний компонент.

Функция *withData* будет возвращать класс, который наследует React.Component:

```
class extends React.Component
```

В конструкторе мы определим начальное состояние с пустым списком данным:

```
constructor(props) {
  super(props)

  this.state = { data: [] }
}
```

Заметим, что мы заменили *gists* внутри состояния на *data*, так как мы хотим создать универсальный HoC и нам незачем привязываться к конкретному названию (прим. пер. однако он будет работать только со списками, я бы заменил пустой массив на null, но кто я такой чтобы идти против творца).

Аналогично исходному компоненту, мы инициируем загрузку данных внутри метода *componentDidMount* и сохраняем внутри состояния после успешной загрузки:

```
componentDidMount() {
  fetch(url)
    .then(response => response.json())
    .then(data => this.setState({ data }))
}
```

Важный нюанс, url не забит гвоздями внутри этого компонента, а приходит как параметр HoC. Это основа переиспользуемости этого компонента внутри приложения.

И в конце мы отображаем полученный в аргументах компонент, передавая ему все параметры и новые данные:

```
render() {
  return <Component {...this.props} {...this.state} />
}
```

Собственно, HoC готов. Теперь мы можем обернуть им любой компонент, чтобы передать ему данные из любого сетевого сервиса.

Давайте посмотрим, как сделать это.

Для начала создадим глупый компонент, который получает данные и отображает их аналогично изначальному компоненту:

```
const List = ({ data: gists }) => (
  <ul>
    {gists.map(gist => (
      <li key={gist.id}>{gist.description}</li>
    ))}
  </ul>
)

List.propTypes = {
  data: React.PropTypes.array,
}
```

Мы можем использовать компонент-функцию, так как мы не собираемся хранить внутри компонента какие-либо данные или определять обработчики событий.

Параметр, который мы получаем из HoC, называется *data*, что не очень удобно для использования внутри компонента, но мы можем без проблем его переименовать благодаря возможностям ES2015.

Теперь мы можем посмотреть, как мы можем воспользоваться нашим HoC *withData*, для того, чтобы передать данные новому компоненту *List*.

Благодаря частичному использованию функции мы можем сначала создать HoC с нужным url, а потом использовать для любого компонента:

```
const withGists = withData(
  'https://api.github.com/users/gaearon/gists'
)
```

И в конце концов мы можем обернуть им наш новый компонент, чтобы создать новый:

```
const ListWithGists = withGists(List)
```

Теперь мы можем добавить расширенный компонент в любое место приложения, и он будет работать.

Наш HoC *withData* великолепен, но он может загружать данные только из статических url, когда в реальности загрузка данных может зависеть от различных параметров, которые бы можно было передать через *props*.

К сожалению, *props* не доступны в момент создания HoC, но они доступны внутри метода *componentDidMount* в момент вызова сетевого запроса.

Что мы можем сделать, так это научить работать наш HoC с двумя типами URL: строкой, как это работает сейчас, и функцией, которая будет создавать url в зависимости от пришедших параметров.

Для того, чтобы сделать это, достаточно поправить метод *componentDidMount*:

```
componentDidMount() {
  const endpoint = typeof url === 'function'
    ? url(this.props)
    : url

  fetch(endpoint)
    .then(response => response.json())
    .then(data => this.setState({ data }))
}
```

Таким образом, если к нам пришла строка, то мы используем его как раньше, если же функция, то мы передаем ей параметры, чтобы получить url.

Мы можем использовать обновленный HoC следующим образом:

```
const withGists = withData(
  props => `https://api.github.com/users/${props.username}/
    gists`
)
```

И имя пользователя мы можем передать через параметры компонента *ListWithGists*:

```
<ListWithGists username="gaearon" />
```

5.3 React-refetch

Теперь наш HoC работает как задумывался, и мы можем использовать его в любой части приложения.

Вопрос, что нам делать, если нам нужно больше возможностей в работе с сетью?

Например, что если мы хотим сделать POST запрос на сервер, или перезагрузить данные в случае изменения параметров?

Или мы не хотим осуществлять загрузку в *componentDidMount* и хотим сделать ее ленивой.

Конечно мы можем реализовать все сами, но есть уже готовая библиотека, в которой реализовано множество инструментов для осуществления разных сценариев работы с сетью.

Библиотека называется $react-refetch$, и она поддерживается разработчиками из *Heroku*.

Давайте посмотрим, как мы можем использовать эту библиотеку, чтобы заменить наш HoC.

В прошлой главе мы создали компонент-функцию *List*, которая принимает список гистов черзе параметры и выводит описание каждого на экран:

```
const List = ({ data: gists }) => (
  <ul>
    {gists.map(gist => (
      <li key={gist.id}>{gist.description}</li>
    ))}
  </ul>
)

List.propTypes = {
  data: React.PropTypes.array,
}
```

Оборачивая этот компонент в *withData* HoC, мы можем передать данные через параметры прозрачным для компонента образом.

С библиотекой react-refetch мы можем сделать то же самое. Но для начала нам нужно ее установить:

```
npm install react-refetch --save
```

Затем мы импортируем функцию *connect* из этой библиотеки в наш модуль:

```
import { connect } from 'react-refetch'
```

А затем мы оборачиваем наш компонент в HoC *connect*. Мы снова воспользуемся частичным применением для создания HoC функции и дальнейшего ее переиспользования:

```
const connectWithGists = connect(({ username }) => ({
  gists: `https://api.github.com/users/${username}/gists`,
}))
```

Разберемся в этом коде.

Мы используем функцию *connect*, которой передаем функцию для создания url. При вызове нашей функции, *connect* передаст ей *porps* (и *context*), что позволит динамически создавать url, основываясь на текущих параметрах компонента.

Наша функция должна вернуть объект, в котором ключами будут идентификаторы запросов, а значениями их url'ы.

В данном случае URL может быть не только строкой. В дальнейшем мы рассмотрим, как добавить к нему различные параметры.

Сейчас мы расширяем компонент *List* функцией, которую только что создали:

```
const ListWithGists = connectWithGists(List)
```

Но нам теперь нужно немного поправить исходный компонент, чтобы он работал с новым HoC.

Прежде всего, параметр больше не называется *data*, теперь компонент должен ожидать параметр *gists*.

По сути react-refetch будет использовать для ключей идентификаторы, которые мы использовали в объекте с url'ами.

Также параметр *gists* не содержит непосредственно данные, он является объектом типа *PromiseState*.

PromiseState - это синхронное представление *Promise* объекта. У него есть множество удобных свойств, такие как *pending* (ожидание) или $fulfilled$ (выполнено), которые могут быть использованы для отображения индикатора загрузки или списка объектов.

Также есть свойство *rejected* (ошибка) для обработки ошибок.

После окончания запроса, данные для отображения можно получить через свойство *value*:

```
const List = ({ gists }) => (
  gists.fulfilled && (
    <ul>
      {gists.value.map(gist => (
        <li key={gist.id}>{gist.description}</li>
      ))}
    </ul>
  )
)
```

В момент отрисовки компонента, мы проверяем, что запрос уже выполнен, получаем данные через *gists.value* и отрисовываем на экране.

Все остальное остается неизменным.

Также нам нужно обновить *propTypes*, так как у нас изменились и название параметра и его тип:

```
List.propTypes = {
  gists: React.PropTypes.object,
}
```

Теперь, мы можем расширить функционал нашего проекта с помощью этой библиотеки.

Например, мы можем добавить кнопку, чтобы лайкнуть новый гист.

Давайте начнем с интерфейса, а потом добавим вызовы к серверу с react-refetch.

Задача компонента *List* состоит в отображении гистов, и мы не хотим добавлять ему еще больше ответственности, поэтому вынесем каждый гист в отдельный компонент.

Мы создадим новый компонент *Gist* для отображения каждого конкретного гиста, который будем использовать внутри *List*:

```
const List = ({ gists }) => (
  gists.fulfilled && (
    <ul>
      {gists.value.map(gist => (
        <Gist key={gist.id} {...gist} />
      ))}
    </ul>
  )
)
```

Мы просто заменяем тег <li> на компонент *Gist* и через спред оператор передаем ему гист. Компонент *Gist* в этом случае получает данные не одиночным объектом, а отдельными параметрами, что упрощает его тестирование.

Gist мы сделаем компонент-функцией, так как нам не нужно хранить внутри него какие-либо данные.

Компонент будет как и раньше отображать описание, но помимо этого, мы добавим еще одну кнопку с текстом '+1', к которой в дальнейшем добавим логики:

```
const Gist = ({ description }) => (
  <li>
    {description}
    <button>+1</button>
  </li>
)

Gist.propTypes = {
  description: React.PropTypes.string,
}
```

Для лайка гиста мы будем использовать следующий URL:

```
https://api.github.com/gists/:id/star?access_token=:
    access_token
```

Здесь нам нужен идентификатор конкретного гиста : *id* и токен доступа (access_token).

Есть несколько способов получить токен доступа, они хорошо описаны в документации GitHub.

Они выходят за рамки обсуждения в этой книги, поэтому оставим их для самостоятельного изучения.

Следующий шаг - добавить обработчик события *onClick* на кнопку, в котором будем выполнять сетевой запрос.

Как мы видели прежде, функция *connect* принимает аргументом функцию и возвращает объект с описанием сетевых запросов.

Если значение в этом объекте типа строка, то данные будут грузиться непосредственно в момент получения параметров. Если же по ключу в этом объекте лежит функция, то запрос может быть выполнен позднее, например по событию внутри компонента.

Посмотрим, как нам добавить новый вызов:

```
const token = 'access_token=123'
```

```
const connectWithStar = connect(({ id }) => ({
  star: () => ({
    starResponse: {
      url: `https://api.github.com/gists/${id}/star?${token
          }`,
      method: 'PUT',
    },
  }),
}))
```

Таким образом мы создаем НоС *connectWithStar*, который использует *id* из параметров для лайка соответствующего гиста.

Затем мы определяем объект с описанием запроса, в котором ключ *star*, а значение опять же функция, которая возвращает объект для запроса. В этом случае *starResponse* уже на простая строка, так как нам нужно добавить еще метод сетевого запроса.

Мы должны сделать это, так как по умолчанию библиотека выполняет HTTP GET запросы, а если нам нужно выполнить POST или PUT запрос, мы должны указать это явно.

Теперь мы можем обернуть наш компонент в этот НоС:

```
const GistWithStar = connectWithStar(Gist)
```

И в конце концов мы можем добавить выполнение этого запроса на нажатие кнопки:

```
const Gist = ({ description, star }) => (
  <li>
    {description}
    <button onClick={star}>+1</button>
  </li>
)

Gist.propTypes = {
  description: React.PropTypes.string,
  star: React.PropTypes.func,
}
```

Здесь все просто, мы определили функцию для вызова сервиса по ключу *star*, который теперь пришел в компонент через *props*. При нажатии на кнопку эта функция вызывается и соответственно совершается сетевой запрос.

Такой подход позволяет уменьшить потребность в хранении состояния внутри наших компонент. Помимо этого компоненту не нужно бес-

покоиться об обработчике нажатия, так как он получает его от родительского компонента (из HoC).

Это позволяет нам тестировать отображение и логику загрузки отдельно. А также изменить реализацию компонента, который предоставляет данные, не затрагивая при этом основной компонент.

5.4 Закючение

Подошла к концу глава о загрузке данных. Мы разобрались, как получать и отправлять данные через сетевые запросы.

Мы посмотрели, как в React осуществлен односторонний поток данных, и почему такой подход упрощает разработку приложений.

Также мы рассмотрели основные паттерны передачи данных между родительскими и дочерними компонентами через функции обратного вызова. И научились передавать данные между компонентами, которые не связанны напрямую.

Во второй половине главы мы написали небольшой компонент, который загружает данные из API GitHub. А за счет использования HoC мы сделали его переиспользуемым.

Таким образом мы освоили уже множество способов вынесения логики из компонент, что позволяет нам улучшать их тестируемость.

И в конце мы познакомились с библиотекой react-refetch, которая реализует основные паттерны работы с сетью, и помогает уменьшить количество необходимых велосипедов.

В следующей главе мы будем разбираться, как эффективно работать с React в среде браузера.

Глава 6

Пишем Код для Браузера

Есть множество операций, которые мы можем выполнять только в связке с браузером. Например, мы можем попросить пользователя ввести данные через форму, поэтому стоит рассмотреть, какие есть техники для обработки таких

Мы можем использовать **Неконтролируемые компоненты**, которые сами управляют своим внутренним состоянием, или **Контролируемые**, в которых мы, как разработчики, берем управление на себя.

В этой главе мы также посмотрим, как React работает с событиями (events) и реализует некоторые продвинутые техники для предоставления консистентного интерфейса между разными браузерами.

После событий мы разберем атрибут ref, который позволяет получить ссылку на элементы, лежащие ниже в дереве элементов. Это мощный инструмент, который следует использовать с осторожностью, так как он ломает некоторые договоренности, которые упрощают разработку на React.

После этого мы разберемся, как работать с анимациями, используя расширения React и сторонние библиотеки, такие как **react-motion**. И в конце посмотрим, насколько легко в React использовать SVG, и как мы можем динамические создавать иконки для нашего приложения.

В этой главе мы разберем следующие вопросы:

- Использование различных подходов для создания форм
- Отслеживание событий DOM и создание собственных обработчиков

- Способ выполнения императивных операций над DOM элементами через параметр ref
- Создание простых анимаций, которые будут работать в различных браузерах
- Способ создания SVG в React

6.1 Формы

После того, как мы начали делать приложение, нам очень быстро может понадобиться взаимодействовать с пользователем. Если мы хотим попросить пользователя какие-либо данные, то формы одно из самых распространенных решений этой проблемы.

Из-за особенностей работы React и его декларативной парадигмы работа с формами может показаться на первый взгляд совсем не тривиальной, но когда мы разберемся с этим вопросом, все станет проще.

6.1.1 Неконтролируемые компоненты

Давайте начнем с простого примера кода, состоящего из поля ввода и кнопки отправки:

```
const Uncontrolled = () => (
  <form>
    <input type="text" />
    <button>Submit</button>
  </form>
)
```

Если мы запустим этот код, то получим форму с полем ввода, в которое мы можем что-то ввести, и кнопкой. Это пример Неконтролируемого Компонента, так как мы не передаем ему данные и оставляем на него управление его состоянием.

Чаще всего мы хотим что-то сделать в момент нажатия на кнопку. Например, мы можем послать данные по сети.

Мы можем сделать это, просто добавив обработчик *onChange* (об обработчиках событий мы поговорим подробнее дальше в этой главе).

Давайте посмотрим, как это сделать.

Для начала создадим класс для компонента, так как нам нужны дополнительные функции:

```
class Uncontrolled extends React.Component
```

В конструкторе класса привяжем обработчик события к экземпляру компонента:

```
constructor(props) {
  super(props)
  this.handleChange = this.handleChange.bind(this)
}
```

Затем определим собственно сам обработчик:

```
handleChange({ target }) {
  console.log(target.value)
}
```

Обработчик события получает объект события (event object), в поле *target* которого находится элемент, который это событие создал. В данном случае нас интересует значение (value) этого элемента. Мы начнем с малого шага и пока что просто будем печатать данные в консоль, но в дальнейшем мы будем сохранять их в state.

И компоненту не хватает только функции *render*:

```
render() {
  return (
    <form>
      <input type="text" onChange={this.handleChange} />
      <button>Submit</button>
    </form>
  )
}
```

Если мы запустим этот код в браузере и начнем набирать в поле ввода слова **React**, то увидим в консоли что-то вида:

```
R
Re
Rea
Reac
React
```

Обработчик *handleChange* вызывается после каждого изменения в поле ввода. Наш следующий шаг - сохранить это значение внутри компонента, чтобы использовать при нажатии кнопки пользователем.

Мы изменим реализацию обработчика событий, чтобы вместо печати в консоль он сохранял значение в *state* компонента:

```
handleChange({ target }) {
  this.setState({
    value: target.value,
  })
}
```

Отслеживание нажатия на кнопку для отправки формы очень похоже на отслеживание изменения поля ввода. В обоих случаях браузером создается событие, которое можно перехватить.

Поэтому нам нужно добавить второй обработчик события:

```
constructor(props) {
  super(props)
  this.state = {
    value: '',
  }

  this.handleChange = this.handleChange.bind(this)
  this.handleSubmit = this.handleSubmit.bind(this)
}
```

Также добавим значение по умолчанию для сохраняемого значения на случай, если кнопка будет нажата до начала ввода значения.

Добавим обработчик *handleSubmit*, который сейчас будет печатать данные в консоль. В реальном проекте здесь может быть отправка данных в сеть:

```
handleSubmit(e) {
  e.preventDefault()

  console.log(this.state.value)
}
```

Этот обработчик крайне прост: он печатает текущее значение в консоль. Также он вызывает метод $e.preventDefault$, чтобы предотвратить стандартное поведение браузера на отправку формы.

Отлично, этот код прекрасно работает, но что если у нас больше полей? Предположим, что у нас есть десяток полей для ввода.

Начнем с того, что попытаемся создать обработчики вручную, а потом посмотрим, как мы можем это дело оптимизировать.

Давайте создадим компонент, где будут два поля ввода для имени и фамилии. Мы можем использовать класс *Uncontrolled*, поправив в нем

нужные блоки. Начнем с конструктора:

```
constructor(props) {
  super(props)
  this.state = {
    firstName: '',
    lastName: '',
  }

  this.handleChangeFirstName = this.handleChangeFirstName.
     bind(this)
  this.handleChangeLastName = this.handleChangeLastName.bind(
     this)
  this.handleSubmit = this.handleSubmit.bind(this)
}
```

Мы создаем два поля для данных и обработчик для кождого из них. Уже можно заметить не лучшую расширяемость этого решения, но перед тем, как мы создадим что-то более гибкое, посмотрим на реализацию этих обработчиков:

```
handleChangeFirstName({ target }) {
  this.setState({
    firstName: target.value,
  })
}

handleChangeLastName({ target }) {
  this.setState({
    lastName: target.value,
  })
}
```

Также немного поправим обработчик отправки формы, чтобы он выводил в консоль новые данные:

```
handleSubmit(e) {
  e.preventDefault()

  console.log(`${this.state.firstName} ${this.state.lastName
     }`)
}
```

И в конце мы описываем элементы формы в методе *render*:

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
```

```
        <input type="text" onChange={this.handleChangeFirstName
            } />
        <input type="text" onChange={this.handleChangeLastName}
            />
        <button>Submit</button>
      </form>
    )
}
```

Теперь у нас есть отправная точка. Мы можем запустить приложение, ввести в первое поле ввода строку **Dan**, во вторую **Abramov**, после чего увидеть имя и фамилию в консоли при нажатии на кнопку отправки формы.

Компонент работает, но текущий подход требует значительных доработок. Если мы продолжим добавлять поля сейчас, то нам понадобится написать множество шаблонного кода, чего мы конечно хотим избежать.

Посмотрим, как мы можем это исправить.

Простоя идея, использовать один обработчик событий, чтобы мы могли использовать его для множества полей ввода.

Давайте вернемся к конструктору и заменим обработчики ввода одним:

```
constructor(props) {
  super(props)
  this.state = {
    firstName: '',
    lastName: '',
  }

  this.handleChange = this.handleChange.bind(this)
  this.handleSubmit = this.handleSubmit.bind(this)
}
```

Мы все еще хотим задавать дефолтные значения для полей ввода, далее мы рассмотрим как передавать эти значения форме.

Теперь мы можем переделать функцию *onChange*, чтобы она работала с разными полями ввода:

```
handleChange({ target }) {
  this.setState({
    [target.name]: target.value,
  })
}
```

Как уже говорилось, в параметре *target* находится представление поля ввода, поэтому мы можем использовать из него не только значение, но и имя.

Теперь нам нужно передать полям ввода имена, чтобы данные сохранялись корректно. Мо можем сделать это в методе *render*:

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <input
        type="text"
        name="firstName"
        onChange={this.handleChange}
      />
      <input
        type="text"
        name="lastName"
        onChange={this.handleChange}
      />
      <button>Submit</button>
    </form>
  )
}
```

Теперь мы можем добавить множество полей ввода, используя один обработчик событий для всех полей.

6.1.2 Контролируемые компоненты

Следующий шаг, который мы хотим сделать, это предзаполнить поля ввода значениями, которые мы можем получить от сервера или из параметров компонента.

Чтобы полностью понять идею, мы снова начнем с простого компонента и будем постепенно его улучшать.

Первый пример показывает предзаполненное значение внутри поля ввода:

```
const Controlled = () => (
  <form>
    <input type="text" value="Hello React" />
    <button>Submit</button>
  </form>
)
```

Если мы запустим этот пример в браузере, то увидим, что в поле ввода находится переданное нами значение. Но при попытке ввода значение меняться не будет; что бы мы ни делали, поле ввода будет сохранять константное значение.

Все дело в том, что в React мы сами определяем то, что хотим видеть на экране, поэтому, передавая константное значение полю вводу, мы получаем константное значение на экране. Очевидно, что мы ожидаем другого поведения от поля ввода.

Если в этот момент мы откроем консоль браузера, то React подскажет нам, что мы делаем что-то не так:

```
You provided a ‘value‘ prop to a form field without an ‘
   onChange‘ handler. This will render a read-only field
```

(Полю формы передан параметр 'value', но не передан параметр 'onChange'. Это приводит к создания поля без возможности ввода.)

И он чертовски прав.

Сейчас, для того чтобы добавить значение по умолчанию и иметь возможность редактирования поля ввода, мы можем добавить параметр $defaultValue$:

```
const Controlled = () => (
  <form>
    <input type="text" defaultValue="Hello React" />
    <button>Submit</button>
  </form>
)
```

Таким образом поле ввода получит значение по умолчанию и его можно будет редактировать. Но сейчас мы можем задать значение компонента только в момент создания, а потом он будет управлять собой сам. В целом, это уже решает проблему с начальным значением, но все еще не дает полностью контролировать состояние поля ввода.

Для того, чтобы перенести управление состоянием поля ввода, создадим компонент собственным состоянием, для чего заменим компонент-функцию на класс:

```
class Controlled extends React.Component
```

Как всегда мы начнем с создания конструктора, где определим начальное состояние компонента (в данном случае это будут начальные значение полей ввода) и обработчики событий.

В данном случае для полей ввода мы будем использовать один обработчик событий, который был написан для примера Неконтролируемого компонента ранее:

```
constructor(props) {
  super(props)

  this.state = {
    firstName: 'Dan',
    lastName: 'Abramov',
  }

  this.handleChange = this.handleChange.bind(this)
  this.handleSubmit = this.handleSubmit.bind(this)
}
```

Обработчики событий такие же, как в предыдущем примере:

```
handleChange({ target }) {
  this.setState({
    [target.name]: target.value,
  })
}

handleSubmit(e) {
  e.preventDefault()
  console.log(`${this.state.firstName} ${this.state.lastName
     }`)
}
```

Важное изменение происходим в методе *render*, где мы будем использовать параметр *value* полей ввода для передачи актуального значение:

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <input
        type="text"
        name="firstName"
        value={this.state.firstName}
        onChange={this.handleChange}
      />
      <input
        type="text"
        name="lastName"
        value={this.state.lastName}
        onChange={this.handleChange}
```

```
        />
        <button>Submit</button>
      </form>
    )
}
```

Когда компонент отрисовывается в первый раз, React использует начальное значение состояние компонента как значение полей ввода.

Когда пользователь вводит что-либо, вызывается обработчик *handleChange*, который обновляет состояние компонента. После изменения состояние компонент перерисовывается и отображает поля ввода с уже новыми значениями.

Теперь у нас есть способ полного контроля над данными полей ввода, такой паттерн называется **Контролируемым Компонентом (Controlled Component)**.

6.1.3 JSON схема

Теперь мы знаем, как работают формы в React, и можно задуматься о том, чтобы автоматизировать создание форм, уменьшить количество шаблонного кода и сделать код чище.

Одно из возможных вариантов - использование библиотеки react-jsonschema-form, которая поддерживается mozilla-services. Для установки библиотеки нужно выполнить команду:

```
npm install --save react-jsonschema-form
```

После установки библиотеки мы можем добавить импорт в файл с нашим компонентом:

```
import Form from 'react-jsonschema-form'
```

Также мы должны определить схему формы следующим образом:

```
const schema = {
  type: 'object',
  properties: {
    firstName: { type: 'string', default: 'Dan' },
    lastName: { type: 'string', default: 'Abramov' },
  },
}
```

В этой книге мы не будем углубляться в формат JSON Schema, но самое главное здесь то, что мы можем создать конфигурацию формы вместо непосредственного создания HTML элементов.

Как вы можете увидеть в примере, мы установили тип схемы *object*, у которого есть два параметра, $firstName$ и $lastName$, каждый из которых типа *string* и со своим значением по умолчанию.

Если после этого мы передадим объект схемы компоненту $Form$, который мы импортировали из библиотеки, то форма будет создана автоматически.

Давайте как всегда начнем с простого использования данный библиотеки и будем постепенно улучшать код:

```
const JSONSchemaForm = () => (
  <Form schema={schema} />
)
```

Теперь, если мы запустим код, то увидим форму с полями, которые мы определили в схеме, и кнопкой отправки.

Теперь мы хотим как-то отслеживать момент отправки формы для выполнения каких-либо действий с данными полей ввода.

Прежде всего нам нужно создать обработчик событий, поэтому переделаем компонент-функцию в класс:

```
class JSONSchemaForm extends React.Component
```

В конструкторе мы привяжем обработчик к экземпляру этого класса:

```
constructor(props) {
  super(props)

  this.handleSubmit = this.handleSubmit.bind(this)
}
```

В этом примере мы просто печатаем данные в консоль, но в реальном коде вы скорее всего захотите выполнить с ними какие-то действия, например отправить на сервер.

Обработчик событий *handleSubmit* получает объект с полем $formData$, в котором находятся все имена и значения полей ввода:

```
handleSubmit({ formData }) {
  console.log(formData)
}
```

И в итоге метод *render* будет выглядеть следующим образом:

```
render() {
  return (
    <Form schema={schema} onSubmit={this.handleSubmit} />
  )
}
```

Здесь объект *schema* - это объект схемы, который мы создали ранее. Этот объект может быть объявлен статически, как в данном примере, собран из параметров компонента или получен от сервера.

Остается только передать обработчик *onSubmit* компонента *Form*, и работающая форма готова.

Помимо этого можно передать обработчики событий *onChange* для отслеживания всех изменений данных в форму и *onError* для перехвата попыток отправить неправильные данные.

6.2 События

Работа **событий (events)** несколько отличается во всех браузерах. React пытается дать разработчику общий интерфейс для работы с событиями в любых браузерах, скрывая различия в работе событий за единой абстракцией. Это великолепная возможность React, которая позволяет перестать писать отдельные обработчики событий для каждого браузера.

Для реализации такого интерфейса React вводит новый концепт **Синтетический Событий (Synthetic Event)**. Синтетическое событие - это объект, созданный на основе оригинального события, но имеющий всегда одну и ту же структуру независимо от структуры события, из которого он был создан.

Для прикрепления к элементу обработчика события мы можем использовать простое соглашение, которое напоминает способ привязки обработчиков событий к DOM элементам. Мы просто используем приставку *on* и название события в ГорбатомРегистре (например, *onKeyDown*).

Распространенный способ объявления обработчиков событий заключается в использовании названия названия события в ГорбатомРегистре и добавлении префикса *handle* (например, *handleKeyDown*).

Мы увидели этот паттерн в полной мере в предыдущем примере, когда создавали обработчик для события *onChange*.

Давайте разберемся, как можно организовать обработку множества событий внутри компонента.

Мы собираемся создать простою кнопку и начнем с создания соответствующего класса:

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

В конструкторе сделаем привязку обработчика событий к экземпляру класса:

```
constructor(props) {
  super(props)
  this.handleClick = this.handleClick.bind(this)
}
```

И определим сам обработчик событий:

```
handleClick(syntheticEvent) {
  console.log(syntheticEvent instanceof MouseEvent)
  console.log(syntheticEvent.nativeEvent instanceof
     MouseEvent)
}
```

Как можно увидеть, мы делаем простую вещь: проверяем тип объекта события, который мы получаем от React, а также, переданного вместе с ним, объекта нативного события. В первом случае мы ожидаем $false$, а во втором $true$.

Скорее всего объект нативного события вам никогда не понадобится, но стоит помнить, что в случае необходимости вы можете им воспользоваться. И в конце нам нужно определить метод $render$, где мы создадим кнопку с атрибутом $onClick$, в который передадим только что созданный обработчик событий:

```
render() {
  return (
    <button onClick={this.handleClick}>Click me!</button>
  )
}
```

Теперь представим, что мы хотим добавить второй обработчик событий, который срабатывает при двойных кликах. Для того, чтобы сделать это, можно просто добавить отдельный обработчик и передать кнопке через атрибут $onDoubleClick$:

```
<button
  onClick={this.handleClick}
  onDoubleClick={this.handleDoubleClick}>
    Click me!
</button>
```

Помните, что нужно всегда стараться писать меньше шаблонного и дублирующегося кода. Поэтому достаточно распространенный подход, писать **один обработчик событий** на компонент, который может вызывать необходимые методы в зависимости от типа события.

Этот подход хорошо описал Майкл Чен (Michael Chan) в своей коллекции паттернов:

http://reactpatterns.com/#event-switch

Прежде всего поправим конструктор, потому что теперь мы будем использовать один общий обработчик событий:

```
constructor(props) {
  super(props)

  this.handleEvent = this.handleEvent.bind(this)
}
```

Далее реализуем сам обработчик событий:

```
handleEvent(event) {
  switch (event.type) {
    case 'click':
      console.log('clicked')
      break
    case 'dblclick':
      console.log('double clicked')
      break
    default:
      console.log('unhandled', event.type)
  }
}
```

Общий обработчик событий получает объект события и в зависимости от его типа совершает нужное действие, чаще всего вызывает соответствующий метод.

И осталось только передать новый обработчик событий кнопке в атрибутах *onClick* и *onDoubleClick*:

```
render() {
  return (
    <button
      onClick={this.handleEvent}
      onDoubleClick={this.handleEvent}
    >
      Click me!
    </button>
  )
}
```

С этого момента, если нам нужно будет начать обрабатывать еще одно событие, то вместо добавления еще одного метода будет достаточно добавить еще один case в switch.

Есть еще два важных нюанса, касающиеся событий в React: Синтетические События переиспользуются и есть **глобальный обработчик событий**.

Из первого вытекает то, что мы не можем хранить синтетические события для дальнейшего переиспользования, так как React перезапишет все его значения сразу после окончания обработки этого события. Это очень выгодно для производительности приложения, так как помогает избежать создания множества объектов, но добавляет нам головной боли если мы хотим сами сохранить этот объект для дальнейшего использования. Для решения этой проблемы React предоставляет метод синтетических событий *persist*, который позволяет сделать последние персистентными, чтобы иметь возможность хранить их.

Наличие глобального обработчика событий вытекает из способа, которым React привязывает обработчики событий к DOM элементам.

Когда мы используем *on∗* атрибуты, мы описываем, какое поведение мы ожидаем, но React не привязывает эти обработчики событий непосредственно к DOM элементам.

Вместо этого React прикрепляет к корневому элементу один глобальный обработчик событий, который прослушивает все события за счет **всплытия событий**. Когда интересующее нас событие создается браузером, React вызывает соответствующий обработчик в нужном компоненте. Этот подход называется **делегацией событий** и используется для оптимизации используемой памяти и производительности.

6.3 Refs

Одна из причин популярности React - его декларативность. Это значит, что вы просто описываете, что должно быть на экране, а React берет на себя всю работу с браузером. Эта возможность делает React очень простым для понимания с одной стороны и мощным инструментом с другой.

Однако, в некоторых случаях вам может понадобиться получить доступ к внутренним DOM элементам для выполнения императивных операций. В общем случае стоит избегать использование такой возможности, так как чаще всего для достижения того же результата найдется другой способ, который лучше соответствует стилю React.

Предположим, что мы хотим создать форму, где будет поле ввода

и кнопка, и при нажатии на кнопку фокус должен переходить на поле ввода.

Первое, что приходит в голову, использовать метод *focus* элемента поля ввода непосредственно в DOM дереве.

Давайте создадим компонент *Focus*, в конструкторе которого привяжем метод *handleClick* к экземпляру этого класса:

```
class Focus extends React.Component
```

Мы будем ожидать нажатия по кнопке, чтобы перенести фокус на поле ввода:

```
constructor(props) {
  super(props)

  this.handleClick = this.handleClick.bind(this)
}
```

Затем реализуем сам метод *handleClick*:

```
handleClick() {
  this.element.focus()
}
```

Как вы можете увидеть, мы обращаемся к атрибуту *element* и вызываем на нем метод *focus*.

Чтобы понять, откуда приходит этот атрибут, нужно посмотреть на метод render:

```
render() {
  return (
    <form>
      <input
        type="text"
        ref={element => (this.element = element)}
      />
      <button onClick={this.handleClick}>Focus</button>
    </form>
  )
}
```

Собственно здесь и ответ. Мы создаем форму с полем ввода внутри и передаем функцию через параметр *ref*.

Функция, которую мы передаем, вызывается сразу после монтирования компонента, а параметр *element* ссылается на соответствующий

элемент в DOM дереве. Важно помнить, что когда компонент будет размонтирован, эта же функция будет вызвана с параметром *null*. Это сделано для освобождения памяти и отсутствия утечек памяти.

Все что мы делаем в этой функции, сохраняем ссылку на элемент для дальнейшего использования (как в *handleClick* в нашем примере). Если запустить пример в браузере, то при нажатии на кнопку, как и ожидается, фокус будет переходить на поле ввода.

! Как мы уже упоминали ранее, в общем случае стоит избегать использования ссылок на DOM элементы, так как это делает код более императивным, что уменьшает его читаемость и поддерживаемость.

Один из случаев, когда мы вынуждены обращаться к этому методу без других альтернатив, это интеграция с другими императивными библиотеками, такими как jQuery.

Также важно знать, что если мы используем атрибут ref на другом React компоненте, то получаем внутри функции обратного вызова не DOM элемент, а ссылку на экземпляр этого компонента. Эта возможность расширяет наши возможности, так как дает доступ к состоянию дочерних элементов, но также влечет опасности, поэтому следует по возможности избегать ее использования.

Посмотрим на пример использования этой возможности, для чего создадим два компонента:

- Первый компонент - простое контролируемое поле ввода. В этом компоненте мы создадим метод *reset*, который будет сбрасывать значение поля ввода к пустой строке.
- Второй компонент - форма с предыдущим полем ввода и кнопкой, по нажатию на которую вызывается метод *reset* и поле ввода очищается.

Начнем с создания поля ввода:

```
class Input extends React.Component
```

В конструкторе мы определим начальное значение поля ввода (пустая строка) и привяжем методы *onChange*, который нужен для контроля поля ввода, и *reset* к экземпляру этого класса:

```
constructor(props) {
  super(props)
```

```
    this.state = {
      value: '',
    }

    this.reset = this.reset.bind(this)
    this.handleChange = this.handleChange.bind(this)
}
```

Функция *reset* просто меняет текущее значение поля ввода на пустую строку:

```
reset() {
  this.setState({
    value: '',
  })
}
```

Метод *handleChange* нужен для синхронизации состояния компонента и значение поля ввода:

```
handleChange({ target }) {
  this.setState({
    value: target.value,
  })
}
```

И в методе *render* мы определяем элемент *input* с контролируемым компонентом значением:

```
render() {
  return (
    <input
      type="text"
      value={this.state.value}
      onChange={this.handleChange}
    />
  )
}
```

Теперь мы можем создать компонент *Reset*, который использует предыдущий компонент и вызывает метод *reset* по нажатию на кнопку:

```
class Reset extends React.Component
```

В конструкторе мы как всегда привяжем обработчик событий к экземпляру этого класса:

```
constructor(props) {
  super(props)
```

```
  this.handleClick = this.handleClick.bind(this)
}
```

Самое интересное находится внутри метода *handleClick*, где мы можем вызывать метод *reset* компонента с полем ввода:

```
handleClick() {
  this.element.reset()
}
```

И в конце мы определяем метод *render*:

```
render() {
  return (
    <form>
      <Input ref={element => (this.element = element)} />
      <button onClick={this.handleClick}>Reset</button>
    </form>
  )
}
```

Как вы можете увидеть использование атрибута ref выглядит одинаково и для элементов DOM дерева и для других компонентов.

Таким образом мы можем получать доступ к методам дочерних компонентов. С одной стороны это очень полезная возможность, с другой стоит использовать ее крайне осторожно, так как это значительно усложняет рефакторинг. Например, если мы захотим переименовать метод *reset*, то нам придется найти все родительские компоненты, которые используют этот компонент, и поправить их тоже.

React прекрасен своим функциональным подходом и декларативным API, но также позволяет напрямую обращаться к нижележащим элементам DOM дерева и компонентам для реализации сложных сценариев взаимодействия с пользователем.

6.4 Анимации

Когда мы задумываемся о UI и браузере, мы также не должны забывать об анимации.

Анимированные UI гораздо дружелюбнее статических, а также это хороший способ показать пользователю, что что-то произошло или требует его участия.

Эта глава не ставит своей целью, научить создавать красивые анимации и UI; здесь мы поговорим о базовых инструментах создания анимаций и распространенные решения для анимирования React компонент.

Для библиотек UI в целом и для React в частности очень важно предоставлять удобный способ для создания и управлением анимациями. В React есть расширение, которое позволяет нам создавать анимации в декларативном стиле и называется *react-addons-css-transition-group*.

Давайте посмотрим, как мы можем создать простой эффект плавного появления элемента(fade-in) для текстового поля, сначала с использованием этого расширения, а затем с помощью сторонней библиотеки *react-motion*, которая делает процесс создания анимаций еще проще.

Для того, чтобы начать работать с этим расширением, нам нужно его установить:

```
npm install --save react-addons-css-transition-group
```

После этого мы можем импортировать компонент $CSSTransitionGroup$:

```
import CSSTransitionGroup from 'react-addons-css-transition-
    group'
```

После этого мы оборачиваем компонент, к которому хотим применить анимации, следующим блоком:

```
const Transition = () => (
  <CSSTransitionGroup
    transitionName="fade"
    transitionAppear
    transitionAppearTimeout={500}
  >
    <h1>Hello React</h1>
  </CSSTransitionGroup>
)
```

Здесь используется несколько параметров, которые определенно требуют объяснения.

Сначала мы определяем $transitionName$. $ReactCSSTransitionGroup$ добавляет класс, указанный в этом параметру, к дочернему элементу, что позволяет использовать CSS переходы(CSS transitions) для создания анимаций.

Но одного класса недостаточно для создания правильных анимаций, поэтому $CSSTransitionGroup$ добавляет множество классов в соответствии с состоянием анимации.

В данном случае, передавая атрибут $transitionAppear$, мы говорим компоненту, что мы хотим анимировать дочерние компоненты, когда они появятся на экране.

Таким образом, компоненту будет присвоен класс $fade - appear$ (где fade взят из параметра $transitionName$) сразу после отображения на экране.

Сразу после этого компоненту будет присвоен класс $fade - appear - active$, что позволяет вызвать анимацию из начального состояния в следующее из CSS.

Также мы передаем атрибут $transitionAppearTimeout$, чтобы указать продолжительность анимации. React не будет удалять компонент из DOM дерева до завершения анимации.

Далее нам нужно определить CSS для работы нашей анимации.

В начальном состоянии нам нужно сделать элемент прозрачным:

```
.fade-appear {
  opacity: 0.01;
}
```

Затем мы определяем анимацию перехода с помощью второго класса. Анимация сработает сразу после того, как элемент получит этот класс:

```
.fade-appear.fade-appear-active {
  opacity: 1;
  transition: opacity .5s ease-in;
}
```

Таким образом мы создали эффект появления элемента, который длится $500ms$.

Выглядит довольно просто, и такой подход позволяет создавать сложные анимации, состоящие из множества состояний.

Например, классы $* - enter$ и $* - enter - active$ добавляются, когда новый элемент добавляется в $CSSTransitionGroup$.

При удалении элементов также добавляются соответствующие классы.

6.4.1 React motion

По мере роста сложности анимаций, когда одни анимации начинают зависеть от других, или нам нужно поведение элементов схожее с физическими объектами, мы понимаем, что группы переходов становится недо-

статочно. В этот момент мы можем задуматься об использовании сторонних библиотек.

Одна из самых используемых библиотек для создания анимаций - $react-motion$, которую поддерживает Ченг Лу(Cheng Lou). Эта библиотека предоставляет чистый и простой API для создания анимаций.

Для ее использования нужно ее установить:

```
npm install --save react-motion
```

После установки библиотеки, мы можем импортировать из нее компонент **Motion** и функцию **spring**. Компонент потребуется для оборачивания других компонентов, которые мы хотим анимировать, а функция spring может интерполировать значение из начального в заданное конечное:

```
import { Motion, spring } from 'react-motion'
```

Посмотрим на пример кода:

```
const Transition = () => (
  <Motion
    defaultStyle={{ opacity: 0.01 }}
    style={{ opacity: spring(1) }}
  >
    {interpolatingStyle => (
      <h1 style={interpolatingStyle}>Hello React</h1>
    )}
  </Motion>
)
```

Здесь есть несколько интересных вещей.

Вы можете заметить, что здесь используется паттерн Функция как Потомок (в Главе 4 он разобран подробно), который очень удобен для передачи компоненту параметров, которые определяются во время исполнения программы.

Также мы можем увидеть, что у компонента $Motion$ есть два атрибута, первый из которых $defaultStyle$, определяющий начальное состояние.

В этом примере мы устанавливаем прозрачность в 0.01, чтобы компонент был скрыт в начальный момент времени.

В атрибуте *style* мы определяем конечное состояние, но вместо того, чтобы установить непосредственное значение, мы используем функцию spring, чтобы значение плавно менялось от начального значения к конечному.

На каждой итерации функции spring создается новое значение, соответствующее данному моменту времени, которое передается через аргумент *interpolatingStyle* дочерней функции.

Эта библиотека умеет делать еще множество полезных вещей, но на первом шаге этого должно быть достаточно для знакомства с ее основами.

Сравнение подходов $CSSTransitionGroup$ и $react - motion$ может быть интересным занятием с целью выбрать наиболее подходящий для вашего проекта.

6.5 Векторная графика, SVG

И в завершение поговорим еще об одном не менее важном инструменте, который доступен нам в браузере и позволяет создавать масштабируемые иконки и графики, а именно об **SVG (Scalable Vector Graphics, Масштабируемая векторная графика)**

SVG - это инструмент декларативного описания векторов, что очень хорошо коррелирует с подходами React.

Возможно, вы привыкли создавать иконки посредством использования специальных шрифтов, но у этого способа есть известные проблемы. Их не так удобно позиционировать с помощью CSS и они могут выглядеть не лучшим образом в различных браузерах. Поэтому мы рекомендуем отдать предпочтение SVG для создания иконок.

Для React нет принципиальной разницы между отрисовкой тега *div* и элемента SVG, что очень хорошо работает в наших целях.

Также плюсом SVG является возможность его редактирования во время исполнения посредством JavaScript и CSS, что заставляет его выглядеть еще лучше внутри функционального подхода React.

Таким образом, если мы смотрим на компоненты как на функции от передаваемых компонентам параметрам, то мы можем легко представить, как создать компонент для отображения SVG элементов и как управлять им посредством передачи различных параметров.

Таким образом, стандартным способом работы с SVG элементами в React является оборачивание их в компоненты.

Давайте посмотрим на пример, в котором мы отображаем синий круг с помощью SVG элемента, обернутого в React компонент:

```
const Circle = ({ x, y, radius, fill }) => (
```

```
  <svg>
    <circle cx={x} cy={y} r={radius} fill={fill} />
  </svg>
)
```

Как вы можете увидеть, мы можем использовать компонент-функцию без внутреннего состояния, который принимает все необходимые для отображения круга параметры и создает SVG элемент.

Таким образом, компонент *Circle* - лишь шаблон для отображения круга, который мы можем использовать множество раз внутри нашего приложения.

Также определим типы его параметров:

```
Circle.propTypes = {
  x: React.PropTypes.number,
  y: React.PropTypes.number,
  radius: React.PropTypes.number,
  fill: React.PropTypes.string,
}
```

Не стоит пренебрегать указанием типов параметров компонента, так как при дальнейшем его использовании можно будет без чтения кода понять, какие и какого типа параметры нужно передать компоненту.

Этот компонент мы можем использовать следующим образом:

```
<Circle x={20} y={20} radius={20} fill="blue" />
```

Мы можем использовать все возможности React и установить часть значений по умолчанию, тогда, даже если мы не передадим параметры, мы получим какой-то результат.

Например, мы можем установить цвет круга по умолчанию:

```
Circle.defaultProps = {
  fill: 'red',
}
```

Это очень удобно, когда мы создаем UI компоненты, которые будут потом использоваться другими членами команды, так как в этом случае им не придется пересоздавать SVG элемент с нуля, чтобы использовать с другими параметрами.

Хотя иногда может быть наоборот удобнее зафиксировать некоторые параметры, чтобы сделать компонент более узкоспециализированным.

Например, мы можем создать узкоспециализированный компонент *RedCircle* для создания красных кругов:

```
const RedCircle = ({ x, y, radius }) => (
  <Circle x={x} y={y} radius={radius} fill="red" />
)
```

В данном компоненты мы просто отображаем предыдущий компонент *Circle* с зафиксированным цветом, передавая остальные параметры без изменений.

Соответственно у этого компонента будет следующий интерфейс:

```
RedCircle.propTypes = {
  x: React.PropTypes.number,
  y: React.PropTypes.number,
  radius: React.PropTypes.number,
}
```

Как мы видим, за счет декларативной структуры самого SVG мы получаем очень органичный способ их использования внутри React компонент.

6.6 Заключение

В этой главе мы разобрали несколько вопросов, которые появляются, когда мы начинаем иметь дело с браузерами. Мы разобрались с созданием форм, обработкой событий, созданием анимаций и SVG графики.

React предоставляет для нас все возможности для создания web приложений в декларативном стиле.

Однако, он также позволяет работать напрямую с элементами DOM дерева в императивном стиле, что очень удобно, если нам нужно интегрировать приложение с существующими библиотеками, созданными в императивном стиле.

В следующе главе мы детально посмотрим на CSS и внутренние(inline) стили, а также разберемся, как писать CSS в JavaScript.

Глава 7

Делаем Компоненты красивыми

Наш путь по лучшим практикам и паттернам React пришел к моменту, когда мы хотим сделать наши компоненты красивее. Для этого мы разберемся с вопросом, почему обычный CSS не всегда является наилучшим вариантом для стилизации компонент, и какие есть альтернативы.

Мы начнем со встроенных стилей, библиотеки Radium, CSS модулей и Styled Compopnents, а затем детально разберем волшебство CSS в JavaScript.

Вопрос стилизации в React стоит очень горячо и вызывает множество споров, поэтому эта глава требует непредвзятости и готовности оценить плюсы и минусы различных инструментов.

В этой главе мы рассмотрим следующие вопросы:

- Общие проблемы масштабируемости CSS
- Встроенные стили в React и их отрицательные стороны
- Как библиотека Radium помогает исправить проблемы встроенных стилей
- Как настроить проект с нуля для работы с Webpack и CSS Модулей
- Возможности CSS Модулей и их преимущества перед глобальными стилями
- Современный подход к стилизации компонентов с библиотекой Styled Components

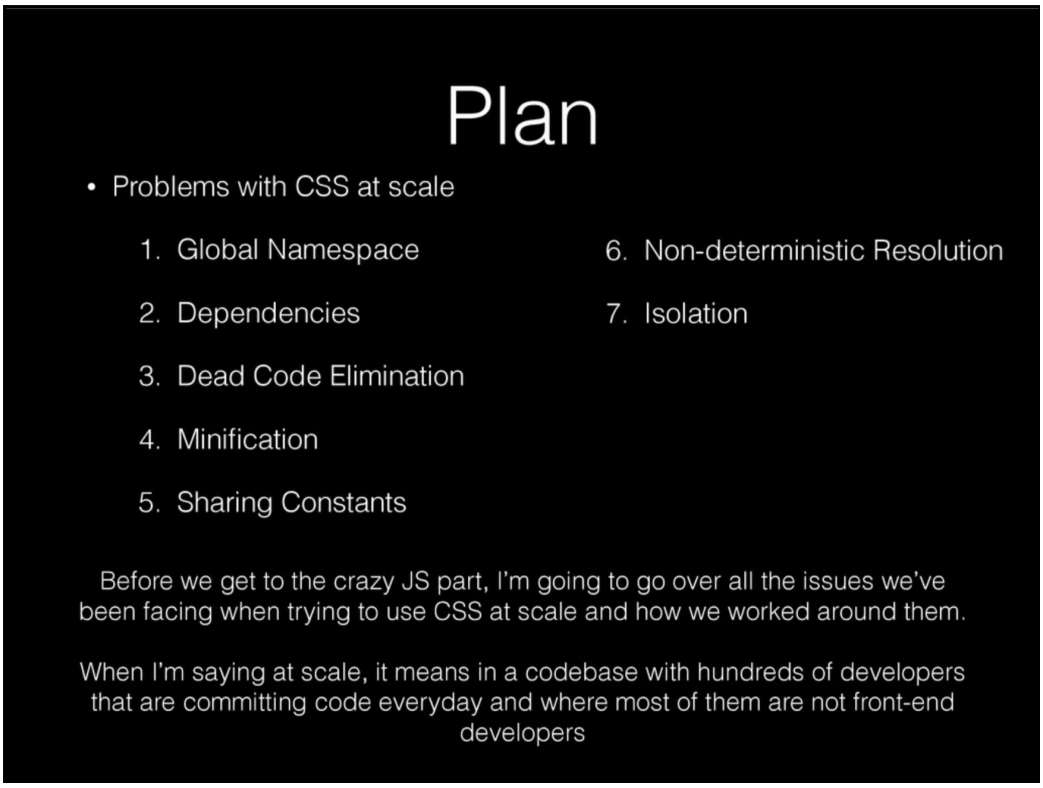
7.1 CSS in JS

Многие знают, что для стилизации в React переломный момент произошел в Ноябре 2014 года, когда *Кристофер Шедо (Christopher Chedeau)* выступил на конференции NationJS.

Кристофер, также известный в интернете как *Vjeux*, работает в Facebook и способствует развитию React. На этой конференции он рассказал о проблемах с CSS при масштабировании приложений, которые они встретили в Facebook.

Важно знать об этих проблемах, так как многие из них часто встречаются на практике, а также будет легче понять необходимость разных концепций, таких как **встроенные стили** и **локальные классы**.

На слайде вы можете увидеть один из слайдов с этой презентации со списком главных проблем с CSS:



Первая и хорошо известная проблема CSS заключается в глобальных селекторах. Не важно, как вы организовали свой код, использовали ли

пространства имен или BEM методологию, в конце все равно стили окажутся в одном глобальном пространстве имен. И это ошибка не только идеологически, это ведет к множеству ошибок и ухудшает поддерживаемость большой кодовой базы. Когда мы работаем в больших командах, не так просто знать о существовании конкретного класса или стилизованного элемента, что ведет к добавлению большего количества классов вместо переиспользования существующих.

Следующая проблема относится к определению зависимостей. На деле бывает очень сложно понять, от каких именно стилей зависит конкретный компонент. Так как стили глобальны, они могут использоваться из любых компонентов, а любой компонент может использовать любые стили. В таких условиях может быть очень легко потерять контроль.

Фронтенд разработчики привыкли использовать препроцессоры для разделения CSS на модули, но в итоге для браузера все равно генерируется большой CSS бандл. Так как объем кода CSS быстро растет, мы получаем еще одну проблему с **удалением неиспользуемого кода (dead code elimination)**. Так как сложно понять, какой стиль к какому компоненту относится, задача удаления неиспользуемого кода становится еще сложнее. Также, если учесть каскадную природу CSS, удаление любого селектора или правила может привести к непредсказуемым последствиям в браузере.

Минификация селекторов и имен классов также добавляет головной боли и в CSS и в JavaScript приложение. Хотя на первый взгляд эта задача кажется простой, на деле все усложняется, когда классы добавляются во время исполнения программы или высчитываются на клиенте.

Отсутствие возможности минифицировать CSS сильно сказывается на производительности, так как может значительно сказаться на размере CSS файлов.

Также в рамках обычного CSS нетривиально создать константы, общие для CSS и JavaScript. Например для случаев, когда мы хотим знать высоту заголовка, чтобы рассчитать расположение элементов относительно него.

Обычно, JavaScript API используется для получения нужных значений, однако было бы значительно оптимальней использовать общие константы и избежать дорогих вычислений во время исполнения программы. Это и есть пятая проблема, которую Vjeux и другие разработчики Facebook попытались решить.

Шестая проблема заключается в недетерминированный обработке фай-

лов CSS. По факту, в CSS важен порядок обработки файлов с исходниками, поэтому, если файлы грузятся по требованию, то сохранение их порядка не гарантируется, что ведет к применению неверных стилей к компонентам.

Предположим, что мы хотим оптимизировать загрузку CSS и загружать стили для конкретной страницу только тогда, когда пользователь открывает эту страницу. Если стили, которые относятся к последней странице, содержат правила, которые затрагивают остальное приложение, то отображение всего приложения может измениться. Например, если пользователь вернется по истории к предыдущей странице, она может несколько отличаться от того, что было до этого.

Очень сложно контролировать различные комбинации стилей, правил и путей, но возможность загружать CSS по мере необходимости критична для производительности приложения.

И седьмая проблема, о которой говорит Кристофер Шедо, связана с изоляцией компонент. В CSS очень сложно добиться изоляции файлов или компонентов между собой. Так как селекторы глобальны, они могут быть легко перезаписаны. Это нетривиальная задача, определить финальные стили компонента по примененным к нему классам, так как любой компонент может быть затронут любыми стилями, находящимися в приложении.

Я рекомендую посмотреть это выступление, если вы хотите узнать больше о проблемах масштабируемости CSS. Даже если этот вопрос выглядит сложным и противоречивыми, стоит открыто подходить к нему, чтобы найти решение, которое лучше всего подойдет в вашем случае:

https://vimeo.com/116209150

В заключении этого выступления было сказано, что для решения этих проблем масштабирования CSS в Facebook остановились на использовании *встроенных стилей (inline styles)*.

В следующей части мы рассмотрим, как использовать встроенные стили в React, и какие есть плюсы и минусы у этого подхода.

7.2 Встроенные стили

Документация React советует разработчикам использовать встроенные стили для стилизации компонент. Это выглядит странно, так как за про-

шедшие годы мы усвоили, что разделение ответственности это хорошо, и мы не должны смешивать разметку и CSS.

React пытается изменить взгляд на разделение ответственности с привычного разделения технологий на разделение компонент. Разделение разметки, стилей и логики на разные файлы, которые сильно связаны и не могут работать по отдельности, иллюзия. Даже если это делает структуру чище, это не приносит реальной выгоды.

В React мы комбинируем компоненты, которые являются базовыми блоками для создания приложения. Мы можем переносить блоки по всему приложению и, независимо от того, где используются компоненты, они должны предоставлять одинаковую логику и отображение.

Это одна из причин, почему объединение стилей с компонентом с помощью встроенных стилей может иметь смысл в React.

Прежде всего посмотрим, как вообще использовать встроенные стили в React компонентах. Создадим кнопку с текстом **Click me!** и изменим у нее цвета фона и текста:

```
const style = {
  color: 'palevioletred',
  backgroundColor: 'papayawhip',
}

const Button = () => <button style={style}>Click me!</button>
```

Как вы видите, использовать встроенные стили очень просто. Нам достаточно создать объект, в котором будут пары ключей и значений как в обычном CSS.

Единственный нюанс, правила с дефисом в названии должны быть записаны в горбатом регистре, а значения передаваться как строки, то есть в кавычках.

Есть несколько отличий при использовании вендорных префиксов. Например, если мы хотим определить переход (transition) в **webkit**, мы должны использовать атрибут WebkitTransition, который начинается с заглавной буквы. Это правило работает для всех вендорных префиксов кроме **ms**, которой должен быть в нижнем регистре.

Помимо этого, числа могут использовать без кавычек и единиц измерения, тогда они будут считаться пикселями.

Следующий фрагмент стилей устанавливает высоту в $100px$:

```
const style = {
  height: 100,
```

```
}
```

Встроенные стили не только прекрасно работают, но и позволяют делать вещи, которые сложно сделать в CSS. Например, мы можем пересчитать значения стилей на клиенте во время исполнения, что мы увидим в следующим примере.

Предположим, что мы хотим создать поле ввода, размер шрифта в котором будет зависеть от его значения. То есть если значение поля будет равно 24, то и размер шрифта должен быть 24 пикселя. Сделать это с помощью CSS невозможно, также нужно приложить значительные усилия, чтобы сделать это в JavaScript.

Посмотрим, как легко сделать это со встроенными стилями.

Нам нужно будет хранить состояние компонента, поэтому создадим для него класс:

```
class FontSize extends React.Component
```

В конструкторе класса определим начальное значение компонента, а также привяжем обработчик событий ввода к экземпляру этого класса:

```
constructor(props) {
  super(props)

  this.state = {
    value: 16,
  }

  this.handleChange = this.handleChange.bind(this)
}
```

Мы создадим простой обработчик, который будет только обновлять состояние компонента в соответствии с вводимыми пользователем данными:

```
handleChange({ target }) {
  this.setState({
    value: Number(target.value),
  })
}
```

И в конце мы создаем поле ввода с числовым типом, значение которого контролируется нашим компонентом через значение состояния и обработчик событий ввода.

Также мы передадим в атрибут *style* этого поля ввода объект с актуальным значение размера шрифта. Как говорилось выше название пра-

вило должно быть в горбатом регистре, то есть мы должны определить параметр $fontSize$:

```
render() {
  return (
    <input
      type="number"
      value={this.state.value}
      onChange={this.handleChange}
      style={{ fontSize: this.state.value }}
    />
  )
}
```

Как мы видим, при изменении значения поля ввода обновляется состояние компонента, что влечет за собой перерисовку компонента. В момент перерисовки значение размера шрифта берется из состояния компонента, которое берется из состояния. Таким образом размер шрифта меняется вслед за значением поля ввода.

Как и у любого другого решения у встроенных стилей есть свои плюсы и минусы. И в данном случае последних не мало.

Например, со встроенными стилями невозможно использовать псевдоклассы (такие как : $hover$) и псевдоэлементы, что является большим ограничением, если вы хотите создать интерактивный и анимированный UI.

Есть множество костылей (workarounds), которые вы можете использовать для обхода этих ограничений. Например вы можете обычные элементы вместо псевоэлеменов, но для симуляции поведения CSS придется использовать CSS, что не оптимально.

То же самое относится и к **Медиа запросам (Media queries)**, которые нельзя определить с помощью встроенных стилей, что затрудняет создание адаптивного интерфейса. Также, так как стили передаются через JavaScript объект, во встроенных стилях невозможно использовать style fallbacks:

```
display: -webkit-flex;
display: flex;
```

Все из-за того, что JavaScript объекты не могут содержать два атрибута с одинаковым именем. По этой причине использовать style fallbacks невозможно, хотя было бы хорошо иметь возможность их использовать при необходимости.

Еще одна возможность CSS, которую невозможно использовать через встроенные стили, это **Анимации**. Основной костыль (workaround) в этом случае, определить анимации глобально и использовать их внутри атрибута элементов *style*.

После использования встроенных стилей для предопределения стилей из CSS мы вынуждены использовать ключевое слово !*important*, что является плохой практикой, так как предотвращает применение других стилей к элементу.

И самое ужасное, что происходит при использовании встроенных стилей, это значительное усложнение отладки приложения. Мы вынуждены использовать названия классов для поиска элементов через DevTools браузера в целях отладки и проверки, какие стили были применены.

При использовании встроенных стилей все созданные стили окажутся в атрибуте *tyle* созданного HTML элемента, что усложняет их отладку.

Например, кнопка из предыдущего примера будет отображена в следующий элемент:

```
<button style="color: palevioletred; background-color:
    papayawhip;">Click me!</button>
```

Один такой такой элемент несложно прочитать, но представьте, что будут сотни таких элементов стилей. В этот момент это начинает превращаться в проблему.

Если вы отлаживаете список таких элементов, в котором у каждого элемента своя копия стилей, то при изменении одного элемента в браузере вы увидите, что меняется только элемент, который вы редактируете, а все соседние элементы остаются неизменными.

И помимо всего этого, если вы рендерите приложение на стороне сервера (подробнее мы поговорим об этом в Главе 8), то при использовании встроенных стилей размер страницы будет значительно больше.

Алгоритмы сжатия могут достаточно сильно сжать получившийся HTML, так как в нем будет много повторяющихся частей, в некоторых случаях загрузка критичной части CSS может быть даже хорошей идеей, но в целом мы должны стремиться избежать этого.

Таким образом мы приходим к тому, что встроенные стили создают проблем больше чем решают.

По этим причинам сообщество создало другие инструменты для решения проблем встроенных стилей, не теряя при этом объединения стилей с компонентами.

После выступления Кристофера Шедо множество разработчиков задумались о проблеме встроенных стилей и начали искать новые решения для использования CSS в JavaScript.

Автор книги изучил все из них и опубликовал репозиторий, в котором создал простую кнопку с помощью каждого из этих методов:

https://github.com/MicheleBertoli/css-in-js

В начале их было две или три, но сейчас насчитывается уже больше 40.

В следующих частях мы разберем самые популярные из них.

7.3 Radium

Одна из первых библиотек, созданных для решения проблемы, обозначенной выше, **Radium**. Она была создана в стенах *Formidable Labs* и до сих пор обладает популярностью.

В этой части мы посмотрим, как работает библиотека Radium, какие проблемы решает, и почему стоит использовать ее вместе с React для стилизации компонентов.

Мы собираемся создать простую кнопку, похожую на одну из уже созданных ранее в этой главе.

Мы начнем с кнопки без стилей, а потом начнем добавлять простое оформление, псевдоклассы и медиа запросы для того, чтобы разобраться в основных возможностях библиотеки.

Создадим кнопку:

```
const Button = () => <button>Click me!</button>
```

Далее установим библиотеку Radium:

```
npm install --save radium
```

После этого мы можем импортировать библиотеку и обернуть наш компонент следующим образом:

```
import radium from 'radium'

const Button = () => <button>Click me!</button>

export default radium(Button)
```

Функция *radium* - **Компонент высшего порядка (HOC)** (Подробнее об этом паттерне в Главе 4), который расширяет возможности нашего компонента и возвращает новый компонент.

Если мы запустим компонент прямо сейчас, то не заметим никаких изменений, так как еще не применили никакие стили.

Давайте начнем с того, что добавим простых стилей, таких как цвет фона, размер, отступы и немного других параметров.

Как мы уже видели в предыдущей части, встроенные стили в JavaScript определяются с помощью объектов с параметрами CSS в горбатом регистре:

```
const styles = {
  backgroundColor: '#ff0000',
  width: 320,
  padding: 20,
  borderRadius: 5,
  border: 'none',
  outline: 'none',
}
```

Этот пример CSS не отличается от предыдущих примеров встроенных стилей и после применения к компоненту мы увидим такой же результат в браузере:

```
const Button = () => <button style={styles}>Click me!</button
    >
```

И соответствующий код в браузере:

```
<button data-radium="true" style="background-color: rgb(255,
    0, 0); width: 320px; padding: 20px; border-radius: 5px;
    border: none; outline: none;">Click me!</button>
```

Единственное отличие здесь в том, что появился новый атрибут *data–radium* со значение *true*.

Мы уже видели, что с помощью встроенных стилей нельзя использовать псевдоклассы. Давайте разбираться, как Radium позволяет решить эту проблему.

Если нам нужно добавить какой-либо псевдокласс, то достаточно добавить его как атрибут в объект со стилями, остальное Radium сделает сам. Добавим в наш объект со стилями псевдокласс *:hover*:

```
const styles = {
  backgroundColor: '#ff0000',
  width: 320,
```

```
  padding: 20,
  borderRadius: 5,
  border: 'none',
  outline: 'none',
  ':hover': {
    color: '#fff',
  },
}
```

Если вы запустите код с этими стилями, то вы увидите, что наведение мыши делает текст белым. То есть мы можем использовать псевдоклассы.

Однако, если вы запустите DevTools и вручную выставите флаг *:hover*, то ничего не произойдет.

Причина того, что вы можете видеть этот эффект, но не можете его эмулировать при помощи CSS в том, что Radium использует JavaScript для добавления и удаления этого эффекта, определенного в объекте со стилями.

Если вы откроете DevTools и наведете мышь на элемент, то увидите, что цвет добавляется к элементу динамически:

```
<button data-radium="true" style="background-color: rgb(255,
    0, 0); width: 320px; padding: 20px; border-radius: 5px;
    border: none; outline: none; color: rgb(255, 255, 255);">
    Click me!</button>
```

По сути дела Radium добавляет обработчики для событий, которые могут помочь эмулировать работу псевдоклассов.

После срабатывания события Radium меняет состояние компонента, который перерисовывается с новыми стилями. Это может показаться странным поначалу, но у этого способа нет реальных отрицательных сторон и разница в производительности несущественна.

Мы можем добавить новый псевдокласс, например *:active*, и он также будет прекрасно работать:

```
const styles = {
  backgroundColor: '#ff0000',
  width: 320,
  padding: 20,
  borderRadius: 5,
  border: 'none',
  outline: 'none',
  ':hover': {
    color: '#fff',
```

```
  },
  ':active': {
    position: 'relative',
    top: 2,
  },
}
```

Еще одна критическая возможность Radium - это поддержка Медиа запросов. Медиа запросы важны при создании отзывчивых сайтов, и Radium снова использует JavaScript для поддержки этой возможности в нашем приложении.

Давайте посмотрим, как это работает: подход очень похож; нам достаточно добавить новый атрибут в объект со стилями и передать в нем стили, которые должны быть применены в случае срабатывания медиа запроса:

```
const styles = {
  backgroundColor: '#ff0000',
  width: 320,
  padding: 20,
  borderRadius: 5,
  border: 'none',
  outline: 'none',
  ':hover': {
    color: '#fff',
  },
  ':active': {
    position: 'relative',
    top: 2,
  },
  '@media (max-width: 480px)': {
    width: 160,
  },
}
```

Но для того, чтобы Медиа запросы заработали, нужно обернуть наше приложение в компонент *StyleRoot* из библиотеки Radium.

Для того, чтобы Медиа запросы работали корректно, особенно с отрисовкой на стороне сервера, Radium добавляет правила, относящиеся к Медиа запросом, в элемент DOM дерева, указывая все стили как !*important*.

Это сделано для того, чтобы избежать мерцания разных стилей, которое может произойти перед тем, как библиотека поймет, какие из стилей соответствуют текущим Медиа запросам. Реализация этого механиз-

ма внутри специального компонента позволяет браузеру выполнять его обыкновенные задачи.

Таким образом, мы импортируем компонент *StyleRoot*:

```
import { StyleRoot } from 'radium'
```

И оборачиваем им наше приложение:

```
class App extends Component {
  render() {
    return (
      <StyleRoot>
      </StyleRoot>
    )
  }
}
```

В результате, если вы запустите приложение, то можете увидеть, что Radium добавляет стиль в DOM:

```
<style>@media (max-width: 480px){ .rmq-1d8d7428{width: 160px
    !important;}}</style>
```

А класс *rmq-1d8d7428* добавляется к кнопке автоматически:

```
<button class="rmq-1d8d7428" data-radium="true" style="
    background-color: rgb(255, 0, 0); width: 320px; padding:
    20px; border-radius: 5px; border: none; outline: none;">
    Click me!</button>
```

Если вы теперь измените размер окна, то увидите, что размер кнопки становится меньше для меньших экранов, чего мы и ожидали.

7.4 CSS Модули

Если вы чувствуете, что встроенные стили по тем или иным причинам не подходят для вашего проекта, но все еще хотите хранить стили как можно ближе к компонентам, **CSS Модули** могут помочь с этой проблемой.

7.4.1 Webpack

Перед тем как мы погрузимся в CSS Модули и начнем разбираться, как они работают, важно понять, как они были созданы и какие инструменты обеспечивают их работоспособность.

В Главе 2 мы рассматривали, как можно писать код на ES2015 и как его транслировать с помощью Babel. Но по мере роста приложения вы можете также захотеть разделить кодовую базу на разные модули.

Для того, чтобы иметь возможность разделить проект на небольшие модули, которые можно будет импортировать по необходимости, а также возможность создать один большой бандл для браузера, вы можете использовать такие инструменты как **Browserify** и **Webpack**. Эти инструменты называются **бандлерами модулей** и предназначены для того, чтобы собрать все зависимости вашего проекта в один бандл, который может быть запущен в браузерах, в которых на момент написания текста концепций о модулях не завезли.

Webpack особенно популярен в среде React благодаря его системе загрузчиков. По факту, вы можете загрузить в бандл любые ресурсы помимо JavaScript, если для них создан специальные загрузчики. Это могут быть JSON файлы, изображения и прочие ресурсы вашего приложения.

В мае 2015 года *Марк Далгиш (Mark Dalgleish)*, один из создателей CSS Модулей, заключил, что CSS могут быть импортированы в Webpack бандл, что дало начало развитию этой идеи.

Он подумал о том, что если стили импортируются локально в компоненты, то и все имена классов также могут быть в локальной области видимости. Эта идея подробно раскрыта в статье *The end of global CSS*:

```
https://medium.com/seek-ui-engineering/the-end-of-global-css
    -90d2a4a06284
```

7.4.2 Настройка проекта

В этой главе мы разберемся, как создать простой Webpack проект с использованием Babel, для трансляции JavaScript кода для предыдущих версий языка, и CSS Модулями для загрузки локальных CSS в бандл. Также мы пройдем по возможностям CSS Модулей и проблемам, которые они решают.

Прежде всего нам нужно создать проект. Для этого откройте пустую папку и выполните команду:

```
npm init
```

В результате чего будет создан файл *package.json* с настройками по умолчанию.

Далее добавим необходимые зависимости: Webpack и *webpack-dev-server*; которые мы будем использовать для локального запуска приложения и сборки бандла на лету:

```
npm install --save-dev webpack webpack-dev-server
```

После установки Webpack мы можем установить Babel и его загрузчик. В данном случае Babel будет использовать внутри самого Webpack для трансляции ES2015 кода на предыдущие версии языка.

```
npm install  --save-dev babel-loader babel-core babel-
   preset-es2015 babel-preset-react
```

И в конце мы устанавливаем загрузчик стилей и загрузчик CSS, которые нужны для работы CSS Модулей:

```
npm install --save-dev style-loader CSS-loader
```

Еще одна вещь, которую мы можем сделать для упрощения нашей жизни, это добавить плагин *html-webpack-plugin*, который может создать HTML страницу для запуска нашего приложения. Нам не придется добавлять отдельных файлов, так как этому плагину необходим только файл конфигурации Webpack:

```
npm install --save-dev html-webpack-plugin
```

Также не забудем добавить *react* и *react-dom*, которые будем использовать в проекте:

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

Собственно все зависимости установлены, теперь нам нужно их настроить, чтобы все работало вместе.

Для начала добавим скрипт в *package.json*, который будет запускать *webpack-dev-server*, чтобы развернуть наше приложение:

```
"scripts": {
  "start": "webpack-dev-server"
},
```

Для того, чтобы Webpack понимал, как работать с разными типами зависимостей, которые мы используем в проекте, требуется создать файл *webpack.config.js*, который экспортирует объект:

```
module.exports = { }
```

Объект, который экспортируется в этом файле, отвечает за настройки сборки бандла и может обладать различными параметрами, которые будут зависеть от размера и возможностей проекта.

Мы хотим сохранить пример простым насколько это возможно, поэтому мы добавим всего три атрибута.

Первым параметром мы укажем путь к главному файлу проекта, иначе говоря точку входа в приложение:

```
entry: './index.js',
```

Следующий атрибут - *module*, в котором настраивается, как Webpack будет загружать внешние зависимости. Внутри него, через атрибут *loaders*, мы передаем загрузчики для каждого из типа файлов:

```
module: {
  loaders: [
    {
      test: /\.js$/,
      exclude: /(node_modules|bower_components)/,
      loader: 'babel',
      query: {
        presets: ['es2015', 'react'],
      }
    },
    {
      test: /\.css$/,
      loader: 'style!css?modules',
    },
  ],
},
```

Мы указываем, что файлы, название которых соответствует регулярному выражению *.js*, должны быть загружены с помощью *babel − loader*, за счет чего JavaScript код транспилируется и загружается в бандл.

Вы можете заметить, что мы также указали пресеты *es*2015 и *react* для Babel. Как мы уже говорили в Главе 2, с помощью пресетов можно настроить Babel, чтобы он понимал, с каким типом синтаксиса мы сейчас работаем (например JSX).

Второй загрузчик указывает Webpack, что делать с CSS файлами. Этот загрузчик использует *css − loader* с флагом *modules* для активации CSS Модулей.

Результат трансформации стилей передается загрузчику *style*, который добавляет стили в заголовок страницы.

Наконец, мы добавим HTML плагин для автоматического создания страницы с тегом *script*, в который будет передаваться наш бандл:

```
const HtmlWebpackPlugin = require('html-webpack-plugin')
```

```
...
plugins: [new HtmlWebpackPlugin()]
```

На этом мы завершаем с настройкой Webpack. Теперь, если мы запустим *npmstart* в терминале, и откроем в браузере страницу `http://localhost:8080`, мы должны увидеть следующую страницу:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Webpack App</title>
  </head>
<body>
  <script type="text/javascript" src="bundle.js"></script></
     body>
</html>
```

7.4.3 CSS локальной области видимости

Пришло время создать приложение, которое будет состоять из одной кнопки, аналогичную тем, которые мы использовали в предыдущих примерах. На этом примере мы сможем показать все возможности CSS Модулей.

Создадим файл *index.js*, который мы указали в конфигурации Webpack. В нем импортируем *React* и *ReactDOM*:

```
import React from 'react'
import ReactDOM from 'react-dom'
```

Теперь создадим простую кнопку. Как всегда начнем с кнопки без стилей, которые будем добавлять далее шаг за шагом:

```
const Button = () => <button>Click me!</button>
```

И в конце отрисуем компонент внутри DOM дерева:

```
ReactDOM.render(<Button />, document.body)
```

Обратим внимание, что отрисовывать компонент внутри *body* - плохая практика. Но мы пойдем на это для упрощения примера.

Теперь предположим, что мы хотим добавить на эту кнопку немного стилей: цвет фона, размер и т.д.

Создадим обыкновенный CSS файл с названием *index.css* и добавим туда следующий класс:

```
.button {
  background-color: #ff0000;
  width: 320px;
  padding: 20px;
  border-radius: 5px;
  border: none;
  outline: none;
}
```

Ранее мы сказали, что с помощью CSS Модулей мы можем импортировать CSS файлы в JavaScript; давайте посмотрим, как это работает.

В файле *index.js*, в котором мы создали кнопку, добавим следующую строку:

```
import styles from './index.css'
```

Результатом вызова *import* будет объект, атрибутами которого являются классы, определенные в *index.css* файле.

Если мы вызовем *console.log*(*styles*), то мы увидим примерно следующий объект в DevTools:

```
{
  button: "_2wpxM3yizfwbWee6k0UlD4"
}
```

Таким образом, у нас есть объект, в котором атрибутами являются CSS классы, а значениями произвольные (на самом деле нет) строки. Далее мы посмотрим на то, как они формируются, а сейчас разберемся, как мы можем использовать сам объект.

Мы можем использовать этот объект, чтобы установить имя класса для нашей кнопки:

```
const Button = () => (
  <button className={styles.button}>Click me!</button>
)
```

Если мы сейчас вернемся в браузер, то увидим, что стили, определенные в файле *index.css*, применились к кнопке.

Выглядит как магия, так как, если мы откроем DevTools, то увидим, что к элементу кнопки применен класс, который мы увидели в импортированном объекте *style*:

```
<button class="_2wpxM3yizfwbWee6k0UlD4">Click me!</button>
```

Также мы можем увидеть, что аналогичное имя класса было добавлено в стили в заголовке страницы:

```
<style type="text/css">
._2wpxM3yizfwbWee6k0UlD4 {
  background-color: #ff0000;
  width: 320px;
  padding: 20px;
  border-radius: 5px;
  border: none;
  outline: none;
}
</style>
```

Собственно это то, как работает загрузка стилей с помощью CSS Модулей.

Загрузчик CSS позволяет вам импортировать CSS файлы в JavaScript, и все имена классов будут находиться в локальной области видимости того файла, куда вы его импортировали.

Как уже было сказано, значения в объекте стилей не случайны, они генерируются на основе хеша загружаемого файла и других параметров для того, чтобы они оставались уникальны внутри всего проекта.

И в конце загрузчик стилей Webpack берет результат работы трансформации CSS Модулей и вставляет его в заголовок страницы.

Это очень мощная связка, так как с одной стороны позволяет одновременно использовать все возможности CSS, а с другой иметь локальные имена классов и явные зависимости.

Как говорилось в начале главы, CSS находятся в глобальной области видимости, что затрудняет поддержку больших проектов. С CSS Модулями стили находятся в локальной области видимости, что гарантирует отсутствие пересечений в именах классов и предоставляет детерминированный результат.

Кроме того, когда мы явно импортируем стили, мы можем легко понять, какие стили используются конкретным компонентом. Это значительно упрощает удаление неиспользуемого кода, так как когда мы удаляем компонент, мы можем легко удалить весь связанный с ним CSS.

CSS Модули - это обычный CSS, т.е. у нас не возникнет никаких проблем с псевдоклассами, Медиа запросами и анимациями.

Например, мы можем добавить следующие CSS правила:

```
.button:hover {
  color: #fff;
}
```

```
.button:active {
  position: relative;
  top: 2px;
}

@media (max-width: 480px) {
  .button {
    width: 160px
  }
}
```

После трансформации в документ добавится следующий CSS код:

```
._2wpxM3yizfwbWee6k0UlD4:hover {
  color: #fff;
}

._2wpxM3yizfwbWee6k0UlD4:active {
  position: relative;
  top: 2px;
}

@media (max-width: 480px) {
  ._2wpxM3yizfwbWee6k0UlD4 {
    width: 160px
  }
}
```

Имена классов будут подставляться везде, где используется кнопка, что делает их надежными и локальными.

Как вы могли заметить, имена классов прекрасно работают, но они усложняют отладку приложения, так как сложно определить, из какого класса был получен конкретный хеш.

Но в режиме разработки мы можем добавить специальный параметр для изменения шаблона создания локальных имен классов.

Например, мы можем изменить значение загрузчика следующим образом:

```
loader: 'style!css?modules&localIdentName=[local]--[hash:
    base64:5]',
```

В данном случае $localIdentName$ - это параметр, определяющий шаблон для локальных имен классов, в котором на место $[local]$ и $[hash : base64 : 5]$ будут подставлены название оригинального класса и пятизначный хеш соответственно.

Также в шаблоне можно использовать [*path*] для подстановки пути к CSS файлу и [*name*] для подстановки имени файла.

После добавления этой настройки мы получим следующий результат в браузере:

```
<button class="button--2wpxM">Click me!</button>
```

Такой вариант уже лучше подходит для отладки.

Для релизной сборки нам не нужны длинные имена классов, так как улучшение производительности будет скорее всего важнее, поэтому мы захотим короткие имена классов и хеши.

С Webpack мы можем легко создать различные настройки для разных сборок приложения. Помимо этого для релизной сборки мы можем захотеть создать отдельный файл со стилями вместо вставки CSS в бандл. В этом случае бандл будет легковеснее, а стили могут быть закешированы в CDN.

Для того, чтобы сделать это, вам потребуется установить *extract − text − plugin* плагин, который может создать отдельный CSS файл, поместив туда все стили из CSS Модулей.

Есть еще несколько возможностей CSS Модулей, о которых стоит упоминуть.

Первая из них - ключевое слов **global**. При добавлении : *global* перед именем класса мы укажем CSS Модулям, что это имя класса нужно оставить без изменения, т.е. не превращать его в локальный.

Например, создадим следующий CSS код:

```
:global .button {
  ...
}
```

После трансформирования мы получим следующий результат:

```
.button {
  ...
}
```

Это удобно в случаях, когда вы не можете применить стили локально. Например, при подключении стороннего кода.

Еще одна великолепная возможность CSS Модулей - это **композиция (composition)**. С помощью композиции мы можем ссылаться на другие классы в том же файле или из внешних зависимостей, и все стили будут применены к элементу.

Например, создадим отдельное правило, которое будет устанавливать цвет фона в красный цвет:

```
.background-red {
  background-color: #ff0000;
}
```

После этого мы можем вставить этот класс в другой следующим образом:

```
.button {
  composes: background-red;
  width: 320px;
  padding: 20px;
  border-radius: 5px;
  border: none;
  outline: none;
}
```

В результате все правила класса *button* и классов, переданных через атрибут *composes*, будут применены к элементу.

Эта великолепная возможность работает очень интересным образом. Можно было бы ожидать, что все правила класса, на который есть ссылка, будут продублированы в месте использования ссылки, как это происходит в случае SASS @extend, но все совсем не так. На деле все скомбинированные имена классов (имя самого класса и имена классов, переданные через *composes*) применяются один за другим к компоненту в DOM.

В нашем случае мы получим следующий результат:

```
<button class="_2wpxM3yizfwbWee6k0UlD4 Sf8w9cFdQXdRV_i9dgcOq"
   >Click me!</button>
```

При этом CSS правила будут вставлены без дублированного кода:

```
.Sf8w9cFdQXdRV_i9dgcOq {
  background-color: #ff0000;
}

._2wpxM3yizfwbWee6k0UlD4 {
  width: 320px;
  padding: 20px;
  border-radius: 5px;
  border: none;
  outline: none;
}
```

7.4.4 Atomic CSS Модули

На данный момент вы должны понимать, как работает композиция имен классов в CSS Модулях. В YPlan, компании где я (прим. пер. автор книги) работал, когда начал ее создание, мы попробовали сделать шаг вперед и объединить *composes* из CSS Модулей и **Atomic CSS** (также известный как **Functional CSS**).

Atomic CSS - это по сути способ использования CSS, при котором у каждого класса есть только одно правило.

Например, мы можем создать правило для установки нижнего отступа в 0:

```
.mb0 {
  margin-bottom: 0;
}
```

Или можем создать правило, которое будет устанавливать $font-weight$ в 600:

```
.fw6 {
  font-weight: 600;
}
```

Затем мы можем применить эти классы к элементу:

```
<h2 class="mb0 fw6">Hello React</h2>
```

Этот подход достаточно противоречив, но в то же время эффективен. Начать использовать этот прием может быть достаточно сложно, так как в вашей разметке появляется множество классов, что усложняет предсказание финального результата. В каком-то смысле это очень напоминает встроенные стили, так как вы используете одно правило на класс, за тем исключением, что вы используете более короткие имена классов как прокси.

Главный аргумент против Atomic CSS заключается обычно в том, что вы переносите логику разметки из CSS в разметку, чего мы хотели бы избежать. Классы определены в CSS файле, но комбинируются они внутри компонента, и каждый раз, когда вы хотите поправить отображение компонента, вам приходится редактировать разметку.

С другой стороны, мы немного попробовали Atomic CSS и пришли к выводу, что его использования значительно ускоряет создание прототипов.

По факту, когда базовые правила уже созданы, их применение к элементам и создание новых стилей становится очень быстрым процессом. Во вторых, используя Atomic CSS, мы можем контролировать размер CSS файла, так как при создании нового компонента мы используем уже существующие классы и нам не нужно создавать новые, что улучшает производительность.

Таким образом мы попробовали исправить проблемы Atomic CSS с помощью CSS Модулей и назвали этот подход **Atomic CSS Модули (Atomic CSS Modules)**.

По существу вы начинаете с создания базовых CSS классов (таких как $mb0$), но затем вместо вставки их в разметку, вы комбинируете их с помощью CSS Модулей.

Взглянем на пример:

```
.title {
  composes: mb0 fw6;
}
```

Затем:

```
<h2 className={styles.title}>Hello React</h2>
```

Это великолепно, так как вы все еще сохраняете логику стилизации в CSS, а *composes* из CSS Модулей берет на себя работу по объединению одиночных классов и вставке результата в разметку.

В нашем примере результат может выглядеть следующим образом:

```
<h2 class="title--3JCJR mb0--21SyP fw6--1JRhZ">Hello React</
    h2>
```

Здесь классы *title*, $mb0$ и $fw6$ применяются к элементу автоматически. Мы также сохранили плюсы CSS Модулей, так как эти классы также находятся в локальной области видимости.

7.4.5 React CSS Модули

Есть еще одна библиотека, которая может помочь нам работать с CSS Модулями. Вы явно заметили, что в объекте *styles*, который мы используем для загрузки имен классов из CSS, мы используем горбатый регистр, так как JavaScript не поддерживает атрибуты, написанные через дефис.

Также, если мы попытаемся добавить в компонент имя класса, которого нет в CSS файле, мы об этом никак не узнаем, а к компоненту будет добавлен *undefined*.

Для этих и других полезных функций мы можем использовать библиотеку, которая упрощает работу с CSS Модулями.

Давайте посмотрим, как это работает, для чего вернемся в *index.js* из предыдущего примера и заменим обычные CSS Модули на React CSS Модули.

Пакет называется $react - css - modules$, и прежде всего мы должны установить его:

```
npm install --save react-css-modules
```

После установки пакета, мы можем импортировать его в нашем *index.js*:

```
import cssModules from 'react-css-modules'
```

Мы используем его как Компонент-Высшего-Порядка и передаем ему компонент *Button* и объект *styles*, которые мы загрузили из CSS:

```
const EnhancedButton = cssModules(Button, styles)
```

Теперь мы можем поправить сам компонент и убрать из него использование объекта *styles*. С React CSS Модулями мы используем параметр *styleName*, который будет трансформирован в обычный *class*.

Вся соль в том, что мы можем использовать строки для имени CSS класса (например, "*button*"):

```
const Button = () => <button styleName="button">Click me!</
    button>
```

Если мы отрисуем компонент *EnhancedButton* в браузере, то увидим, что ничего не изменилось, что говорит о том, что библиотека работает.

Если же мы заменим имя класса на несуществующее, например следующим образом:

```
const Button = () => (
  <button styleName="button1">Click me!</button>
)
```

Мы увидим в консоли браузера следующую ошибку:

Uncaught Error: "button1"CSS module is undefined.

Это очень удобно, когда кодовая база растет, и разные разработчики работают над компонентами и стилями.

7.5 Styled Components

Есть многообещающая библиотека, которая стремится исправить все проблемы, с которыми столкнулись другие библиотеки.

Различные пути были исследованы в вопросе написания CSS в JavaScript, множество решений было опробовано; пришло время для библиотеки, которая строит новое решение поверх всего изученного.

Библиотека создана и поддерживается двумя, довольно известными в JavaScript сообществе, людьми: *Гленном Маддерном (Glenn Maddern)* и *Максом Стойбергом (Max Stoiberg)*.

Эта библиотека представляет современный подход к проблеме, использует возможности ES2015 и продвинутые подходы работы с React для создания полного решения проблемы стилизации.

Давайте посмотрим, как можно создать кнопку аналогичную кнопкам из предыдущих частей, и убедимся, что все возможности CSS (такие как псевдоклассы и Медиа запросы) работают в Styled Components.

Прежде всего нам нужно установить эту библиотеку:

```
npm install --save styled-components
```

После этого мы можем импортировать ее в файл с нашим компонентом:

```
import styled from 'styled-components'
```

С этого момента мы можем использовать функцию *styled* для создания элемента с помощью вызова $styled.elementName$, где $elementName$ может быть *div*, *button* или другой валидный DOM элемент.

После этого мы можем определить стили элемента, который мы создаем. Для этого мы используем **Теговые шаблоны (Tagged Template Literals)** из ES2015. Этот способ позволяет передавать в функцию шаблонные строки без их предварительной интерполяции.

Это означает, что функция получает шаблоны с JavaScript выражениями внутри, что позволяет библиотеке использовать все возможности языка JavaScript для применения стилей к элементу.

Давайте начнем с создания кнопки с простыми стилями:

```
const Button = styled.button `
  backgroundColor: #ff0000;
  width: 320px;
  padding: 20px;
  borderRadius: 5px;
```

```
  border: none;
  outline: none;
`
```

Этот, кажущийся на первый взгляд странным, синтаксис возвращает валидный React компонент с названием *Button*, который отрисовывает элемент *button* и применяет к нему все стили из шаблона. Стили применяются к элементу следующим образом: создается уникальное имя класса, которое добавляется к элементу, а соответствующие стили добавляются в заголовок страницы.

В данном примере будет создан следующий компонент:

```
<button class="kYvFOg">Click me!</button>
```

А в заголовок будут добавлены соответствующие стили:

```
.kYvFOg {
  background-color: #ff0000;
  width: 320px;
  padding: 20px;
  border-radius: 5px;
  border: none;
  outline: none;
}
```

Плюсом этой библиотеки является то, что она поддерживает все возможности CSS, что потенциально позволяет использовать ее в реальных приложениях.

Например, она поддерживает SASS-подобный синтаксис для псевдоклассов:

```
const Button = styled.button`
  background-color: #ff0000;
  width: 320px;
  padding: 20px;
  border-radius: 5px;
  border: none;
  outline: none;
  &:hover {
    color: #fff;
  }
  &:active {
    position: relative;
    top: 2px;
  }
`
```

А также Медиа запросы:

```
const Button = styled.button‘
  background-color: #ff0000;
  width: 320px;
  padding: 20px;
  border-radius: 5px;
  border: none;
  outline: none;
  &:hover {
    color: #fff;
  }
  &:active {
    position: relative;
    top: 2px;
  }
  @media (max-width: 480px) {
    width: 160px;
  }
‘
```

На этом список возможностей данной библиотеки не исчерпывается.

Например, единожды создав кнопку, вы можете легко переопределять в ней в ней стили для переиспользования с различными параметрами.

Внутри шаблона также возможно использовать параметры компонента и менять стили соответственно.

Темы (Theming) - еще одна великолепная возможность. Обернув свой компонент в компонент ThemeProvider, вы можете передать параметры текущей темы вниз по дереву, что позволяет легко создавать различные UI, в которых между компонентами есть общая часть стилей, зависящая от текущей темы.

7.6 Заключение

В этой главе мы рассмотрели множество интересных вопросов. Мы начали с проблем масштабирования CSS, точнее с тех проблем, с которыми столкнулись разработчики Facebook.

Мы разобрались, как работают встроенные стили и почему стоит хранить стили рядом с компонентами, которые их используют. Также мы разобрали ограничения встроенных стилей.

Затем мы перешли к библиотеке Radium, которая решает главную проблему встроенных стилей, предоставляя простой интерфейс для создания CSS внутри JavaScript. Для тех же, кто считает, что встроенные стили - плохое решение, мы разобрали CSS Модули, с которыми создали проект с нуля,

Импорт CSS стилей в компоненты делает зависимости прозрачнее, а также переносит их в локальную область видимости, чтобы избежать пересечений имен классов. Также мы рассмотрели функцию *composes* из CSS Модулей, которая в совокупности с Atomic CSS позволяет создать основу для создания прототипов.

В конце мы бегло посмотрели на возможности библиотеки Styled Components, которая подает большие надежды полностью изменить подход к стилизации компонент.

Глава 8

Отрисовка приложения на стороне сервера

Следующий шаг на пути изучения React - отрисовка на стороне сервера. **Изоморфные приложения (Universal applications)** хорошо в SEO (Search Engine Optimisations), а также позволяют обмениваться знаниями между фронтендом и бекендом.

Такие приложения могут уменьшить время между началом загрузки страницы и моментом, когда пользователь начинает воспринимать информацию на ней. Однако отрисовка на стороне сервера React приложений влечет за собой дополнительные расходы, поэтому мы должны понимать, когда нам действительно это необходимо.

В этой главе мы посмотрим, как настроить отрисовку на стороне сервера, а также разберемся в плюсах и минусах этого подхода.

В этой главе мы рассмотрим следующие вопросы:

- Что такое Изоморфное приложение
- Причины, по которым мы можем захотеть включить отрисовку на стороне сервера
- Создание простого React приложения с отрисовкой на стороне сервера
- Загрузка данных при отрисовке на стороне сервера и концепция dehydration/hydration

- Использование **Next.js** от Zeith для создания React приложения, которое будет запускаться и на сервере и на клиенте

8.1 Универсальные приложения

Когда мы говорим о JavaScript web приложениях, мы обычно думаем о коде, который выполняется в браузере пользователя.

Основной подход, по которому работают SPA приложения, это передача на клиент почти пустого HTML с тегом script, в котом будет загружен код приложения. Затем этот код уже напрямую взаимодействует с DOM в браузере для отображения UI пользователю. Таким образом приложения работали последние несколько лет, таким же образом множество из них продолжают работать сейчас.

В этой книге мы уже рассмотрели, как создавать приложения с помощью React компонент и как они работают в браузере. Но чего мы еще не касались, как React может создавать те же самые компоненты на стороне сервера, что называется **Отрисовкой на стороне сервера (Server-Side Rendering, SSR)**.

Перед тем, как мы углубимся в детали, давайте поймем, что же это в принципе означает, что приложение может отрисовываться и на сервере и на клиенте. Многие годы мы были вынуждены создавать отдельные приложения для сервера и клиента: например, Django приложение, которое создает страницы на сервере, и какие-то JavaScript фреймворки, такие как Backbone или jQuery, на клиенте.

В таком подходе требуется две команды разработчиков с различными наборами навыков. Если вам требовалось передавать данные между страницей, созданной на сервере, и JavaScript приложением, то вы могли вставлять специальные переменные в тэг *script*. При использовании двух различных языков программирования не было возможности делиться общей информацией, такой как модели или отображения, между различными частями приложения.

С момента релиза Node.js в 2009 было приложено немало усилий на укрепление позиций JavaScript в бекенд разработке, где немалую роль сыграли фреймоврки для создания web приложений, такие как, например, **Express**.

Использование одного и того же языка с обеих сторон позволяет не только упростить для разработчиков переиспользование их знаний, но

также открывает различные пути обмена кодом между клиентом и сервером.

В частности, с React концепция изоморфных приложений стала очень популярной в JavaScript сообществе.

Создание **изоморфного приложения** означает разработку приложения, которые будет выглядеть одинаково и на сервере и на клиенте.

Использование одного языка программирования открывает новые возможности для переиспользования логики. Также упрощается анализ кода и уменьшается количество дублирующегося кода.

React делает еще один шаг вперед, предоставляя удобное API для отрисовки компонент на сервере, а также сам выполняет все необходимые действия, чтобы сделать страницу интерактивной (например, обработчики событий) в браузере.

Термин изоморфный не очень хорошо подходит в данном случае, потому что в случае с React приложения именно одинаковые. По этой причине один из создателе React Router, *Майкл Джексон (Michael Jackson)*, предложил более значимое название для этого паттерна: *Универсальный (Universal)*

Универсальным приложением мы будем называть приложение, которое использует одинаковый код для запуска и на сервере и на клиенте.

В этой главе мы поговорим о том, почему мы должны рассматривать создание Универсальных приложений, и как React помогает отрисовывать компоненты на стороне сервера.

8.2 Мотивация к Отрисовке на стороне сервера

SSR - отличный инструмент, но не стоит использовать его только для того, чтобы использовать его, необходимо понимать, как этот инструмент может улучшить наше приложение и какие проблемы решить.

8.2.1 SEO

Одна из главных причин для отрисовки приложения на стороне сервера - **Поисковая оптимизация (Search Engine Optimization, SEO)**.

Проблема в том, что когда мы отдаем поисковому роботу (crawler) поисковой машины пустую HTML страницу, он не может вытащить из

нее значимую информацию.

На данный момент Google уже должен уметь запускать JavaScript при сканировании страниц, но все равно есть множество ограничений, а SEO является критическим фактором для множества бизнесов.

Последние годы мы были вынуждены писать два приложения: одно для поисковых роботов, которое отрисовывалось на сервере, а второе для клиента, которое работало в браузере.

Нам приходилось делать это, так как приложения, создаваемые на сервере не обладали достаточной интерактивностью, ожидаемой пользователями, а приложения, запускаемые в браузере, не индексировались поисковыми машинами.

Поддержка двух приложений вместо одного доставляет хлопот и делает код менее гибким и открытым к изменениям.

К счастью, с React мы можем отрисовывать приложения на стороне сервера и отдавать их поисковым роботам в удобном для индексации виде.

Также это влияет на отображение ссылок на наши страницы в социальных сетях, где как правило показывается превью страниц, которыми делятся пользователи.

Например, с помощью Open Graph мы можем сказать Facebook, что для определенной страницы должны быть показаны определенные изображение и заголовок.

Но сделать это без SSR невозможно, так как поисковые машины извлекают информацию из страниц, полученных от сервера. А если наш сервер присылает по всем URL пустую страницу, то и превью всех наших страниц будет пустым.

8.2.2 Общая кодовая база

У нас не много вариантов при разработке web приложений: мы можем использовать JavaScript, или вместо него можем использовать JavaScript. Конечно, есть языки программирования, которые компилируются в JavaScript, но принципиально картину это не меняет.

Возможность использования такого же языка программирования на сервере открывает новые возможности поддержки приложений и обмена знаниями внутри компании.

Реализация общем логики для клиента и сервера на одном языке программирования упрощает внесение изменений в эту логику, так как не

придется реализовывать их дважды разными командами, что ведет к уменьшению количества ошибок в коде.

Усилия, которые необходимо затратить для поддержки одного проекта, будут значительно меньше усилий по актуализации двух различных приложений.

Возможность использования одного языка упрощает сотрудничество между командами, так как они находятся в одном информационном пространстве, что ускоряет принятие решений и внесение изменений.

8.2.3 Повышение производительности

Мы любим SPA приложения за то, что они быстры и отзывчивы, но есть одна проблема: перед тем как пользователь сможет сделать что-либо на сайте должен быть загружен и запущен бандл с приложением.

Это может не быть проблемой для пользователей с быстром интернет соединением, но для пользователей с 3G соединением время ожидания станет ощутимым. В целом это не проблема UX, но это также влияет на конверсию. Крупными e-commerce сайтами было доказано, что изменение времени загрузки в большую или меньшую сторону даже на несколько миллисекунд может значительно влиять на прибыль.

Например, если мы отдаем с сервера пустую HTML страницу с тегом *script*, которая показывает индикатор загрузки до момента, когда пользователь сможет полноценно взаимодействовать с сайтом, то восприятие производительности сайта пользователем будет искажено не в лучшую для нас сторону.

Если же мы отрисовываем приложение на стороне сервера, и пользователь видит часть контента сразу после загрузки страницы, то он с большой вероятностью останется на странице, даже если для полноценной работы ему придется ждать то же самое время, так как загрузку бандла с приложенем никто не отменял.

Таким образом мы можем значительно улучшить восприятие пользователем производительности сайта за счет того, что будем часть контента отдавать вместе со страницей.

8.2.4 Не все так просто

Очевидно, что даже с удобным API для создания Универсальных приложений, создание таких приложений будет требовать дополнительных

усилий. Таким образом, перед тем, как включать отображение на сервере, нужно убедиться, что у нас есть причины для этого, а наша команда готова это сделать.

Как мы увидим дальше, отрисовка React компонент - не единственная проблема, которую нужно решить для полноценной отрисовки на стороне сервера.

Нас ожидает настройка и поддержка сервера с роутингом, управление потоком данных, кеширование контента, чтобы быстрее отдавать данные, и множество других интересных задач на пути создания полноценного Универсального приложения.

По этим причинам я советую начинать с приложения, которое полностью работоспособно без поддержки сервера, а затем оценивать плюсы и минусы перехода на SSR.

Включать отрисовку на стороне сервера стоит при жесткой необходимости в этом. Например, если вам нужна оптимизация для поисковых машин или кастомизация отображения для социальных сетей.

Если вы осознали, что ваше приложение загружается слишком долго, а все оптимизации (подробнее об оптимизациях мы поговорим в следующей главе) уже применены, вы можете решить использовать SSR для уменьшения времени между началом загрузки страницы и первого взаимодействия пользователя с контентом.

Кристофер Пожер(Christopher Pojer), инженер Facebook, сказал в Twitter, что в приложении Instagram они используют SSR только в целях SEO. Он сказал, что для приложений с динамическим контентом, как например Instagram, использовать SSR в целях улучшения восприятия пользователя не эффективно:

$$https://twitter.com/cpojer/status/711729444323332096$$

8.3 Простой пример

Пришло время создать простое приложение с отрисовкой на стороне сервера и посмотреть, какие шаги нужно выполнить для создания Универсального приложения.

Мы создадим минимальное и простое приложение, так как наша цель - посмотреть, как работает SSR, а не предоставить комплексное решение или шаблон для проекта.

В данном разделе мы предполагаем, что вы знакомы с концепциями сборки JavaScript приложения и такими инструментами как Webpack. Также будет полезен опыт работы с Node.js, но даже без него, со знанием JavaScript, вы сможете разобраться в этой главе.

Приложение будет состоять из двух частей:

- Серверную часть, в которой мы будем использовать *Express* для создания сервера, который будет отдавать HTML страницу с отрисованным React приложением
- Клиентскую часть, где мы будем отображать приложение привычным для нас образом с помощью $react - dom$.

Обе части приложения пройдут через Babel и Webpack, поэтому мы можем спокойно использовать все возможности ES2015 и на клиенте и на сервере.

Начнем с того, что создадим пустую директорию и инициализируем в ней npm проект:

```
npm init
```

После того, как будет создан package.json, мы можем начать устанавливать зависимости. Начнем с Webpack:

```
npm install --save-dev webpack
```

Также нам понадобится Babel и пресеты для ES2015 и JSX:

```
npm install  --save-dev babel-loader babel-core babel-
   preset-es2015 babel-preset-react
```

Также нам нужно добавить пакеты для сборки серверного бандла. Webpack позволяет определить список зависимостей, которые необходимо исключить из сборки бандла. Например, при сборке серверного приложения мы не хотим добавлять весь код проекта, а только релевантную для сервера часть. Есть пакет, который упрощает этот процесс:

```
npm install --save-dev webpack-node-externals
```

Не забудем добавить в *package.json* скрипт для удобного запуска webpack из терминала:

```
"scripts": {
  "build": "webpack"
},
```

Теперь нам понадобится файл конфигурации Webpack *webpack.config.js* для управления сборкой приложения.

Начнем с импорта библиотеки для определения исключаемых файлов. Также добавим настройки для Babel загрузчика, который мы будем использовать его и на клиенте и на сервере:

```
const nodeExternals = require('webpack-node-externals')

const loaders = [{
  test: /\.js$/,
  exclude: /(node_modules|bower_components)/,
  loader: 'babel',
  query: {
    presets: ['es2015', 'react'],
  },
}]
```

В *Главе 7* мы уже видели, как в файле конфигурации создается объект конфигурации. Но помимо этого Webpack позволяет определять множество разных конфигураций внутри одного файла, что мы и сделаем для клиентского и серверного приложений.

Конфигурация клиентского приложения будет выглядеть уже знакомой:

```
const client = {
  entry: './src/client.js',

  output: {
    path: './dist/public',
    filename: 'bundle.js',
  },

  module: { loaders },
}
```

Мы говорим Webpack, что исходный код приложения находится в папке *src*, а бандл приложения должен создаваться в папке *dist*.

Также мы установили список загрузчиков, созданный ранее при помощи *babel – loader*. Выглядит простым настолько, насколько это возможно.

Серверная конфигурация будет несколько отличаться, но не должны вызвать проблем для понимания:

```
const server = {
  entry: './src/server.js',
```

```
  output: {
    path: './dist',
    filename: 'server.js',
  },

  module: { loaders },

  target: 'node',

  externals: [nodeExternals()],
}
```

Как вы можете видеть, структура та же, за исключением *externals* и *target*, но с другими путями к файлам.

С помощью параметра *target* со значением *node* мы говорим Webpack, что нужно игнорировать все встроенные в node пакеты, такие как *fs*. Через *externals* мы используем библиотеку для исключения зависимостей из сборки.

И в конце мы экспортируем конфигурации в виде массива:

```
module.exports = [client, server]
```

Конфигурация завершена. Теперь мы можем начать писать код и начнем с React приложения, с которыми мы на данный момент работали больше.

Создадим папку *src* и файл *app.js* внутри него.

В файл *app.js* добавим следующее содержимое:

```
import React from 'react'

const App = () => <div>Hello React</div>

export default App
```

Здесь никакой магии: импортируем React, создаем компонент App и экспортируем его.

Теперь создадим файл *client.js*, который будет непосредственно отрисовывать компонент *App* в DOM браузера:

```
import React from 'react'
import ReactDOM from 'react-dom'
import App from './app'

ReactDOM.render(<App />, document.getElementById('app'))
```

Этот код должен выглядеть знакомо. Мы импортируем React, ReactDom и созданный ранее компонент App, а затем просим ReactDom отрисовать этот компонент в элемент DOM с id *app*.

Теперь двинемся в сторону сервера.

Для начала создадим файл *template.js*, в котором будет функция для создания разметки страницы, которая будет отправляться пользователю:

```
export default body => `
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
  </head>
  <body>
    <div id="app">${body}</div>
    <script src="/bundle.js"></script>
  </body>
</html> `
```

Здесь создается функция, которая принимает параметр *body*, который будет содержать React приложение, и возвращает скелет страницы.

Отметим, что мы все равно должны загружать бандл с приложением на клиент, так как отрисовка на стороне сервера создает лишь каркас страницы, а все взаимодействие с пользователем, такое как обработка событий ввода, происходить все равно на клиенте.

Пришло время для создания файла *server.js*, в котором будет больше зависимостей, с которыми требуется разобраться:

```
import express from 'express'
import React from 'react'
import ReactDOM from 'react-dom/server'
import App from './app'
import template from './template'
```

Первая зависимость - библиотека *express*, которая позволяет легко создать web сервер с маршрутизацией и выдачей статических файлов.

Затем мы импортируем React и ReactDom для отрисовки нашего приложения *App*, которое мы также импортируем. Заметьте, что мы импортируем ReactDom из $react-dom/server$. И в конце мы импортируем наш шаблон страницы, созданный ранее.

Теперь создадим сервер Express приложения:

```
const app = express()
```

А также скажем серверу, где находятся статические файлы:

```
app.use(express.static('dist/public'))
```

Вы можете заметить, что мы используем тот же путь, который использовали в настройках Webpack клиентского приложения для сохранения бандла.

Теперь напишем саму логику отрисовки приложения на стороне сервера с помощью React:

```
app.get('/', (req, res) => {
  const body = ReactDOM.renderToString(<App />)
  const html = template(body)
  res.send(html)
})
```

Мы говорим Express, что хотим прослушивать путь /, и когда пользователь перейдет по нему, мы отрисуем компонент *App* в строку с помощью библиотеки ReactDOM. Здесь собственно и происходит магия серверной отрисовки React приложения.

Функция *renderToString* возвращает строковое представление DOM элементов созданного компонента App, идентичное тому, которое было бы создано на клиенте методом *render*.

Значение переменной *body* будет выглядеть примерно следующим образом:

```
<div data-reactroot="" data-reactid="1" data-react-checksum="
    982061917">Hello React</div>
```

Теперь мы можем передать это строковое представление компонента функции *template*, чтобы создать HTML страницу с уже отрисованными компонентами и вернуть ее пользователю.

И осталось только запустить приложение на нужном нам порту:

```
app.listen(3000, () => {
  console.log('Listening on port 3000')
})
```

Почти готово, осталось добавить скрипт в package.json, чтобы было удобнее запускать:

```
"scripts": {
  "build": "webpack",
  "start": "node ./dist/server"
},
```

Теперь мы можем собрать бандл командой *build*:

```
npm run build
```

И запустить сервер командой *start*:

```
npm start
```

Теперь можно открыть в браузере страницу http://localhost:3000 и увидеть результат.

Отметим, что если вы откроете Исходный код страницы (View Page Source), то увидите, что она была получена от сервера с уже отрисованным приложением, чего не произошло бы с отключенным SSR.

Также, если у вас установлено React расширение для браузера, вы сможете увидеть, что приложение было загружено на клиенте.

8.4 Пример загрузки данных

После предыдущей части вы должны понимать, как настроить работу Универсального приложения в React. В целом, это задача не требует принципиально новых знаний, и большая часть проблем, с которыми вы скорее всего встретились, относится к настройке самого проекта.

Но в большинстве реальных проектов вам будет недостаточно создания статичной разметки. Предположим, что мы хотим загрузить все гисты *Дена Абрамова* и вернуть их из созданного Express приложения пользователю.

Ранее, в Главе, 5 мы уже разбирались, как загрузить данные с помощью метода $componentDidMount$ жизненного цикла. Но этот метод не будет работать на сервере, так как компонент не монтируется к DOM и методы жизненного цикла не срабатывают.

Помимо этого обычные способы загрузки данных через методы жизненного цикла компонентов не будут работать, так как они **асинхронные**, а метод $renderToString$ - нет. Поэтому нам придется каким-либо образом грузить данные предварительно и передавать компонентам через параметры.

Давайте возьмем приложение из предыдущей части и посмотрим, как мы можем его поправить, чтобы гисты загружались во время отрисовки приложения на сервере.

Первое, что нам нужно будет сделать, это поправить $app.js$, чтобы была возможность передать через параметры гисты для отрисовки их описаний:

```
const App = ({ gists }) => (
  <ul>
    {gists.map(gist => (
      <li key={gist.id}>{gist.description}</li>
    ))}
  </ul>
)

App.propTypes = {
  gists: React.PropTypes.array,
}
```

С применением уже изученных подходов мы создаем функциональный компонент, который получает параметр *gists*, обходит его в цикле и отрисовывает список элементов.

Теперь нам нужно поправить сервер, чтобы иметь возможность загружать список гистов и передавать его компоненту.

Для использования **fetch** на сервере, мы добавим библиотеку *isomorphic–fetch*, которая реализует такую возможность. Эта библиотека может быть использована и с node и в браузере:

```
npm install --save isomorphic-fetch
```

Сначала импортируем эту библиотеку в *server.js*:

```
import fetch from 'isomorphic-fetch'
```

Вызов сетевого API, которое мы собираемся использовать, будет выглядеть следующим образом:

```
fetch('https://api.github.com/users/gaearon/gists')
  .then(response => response.json())
  .then(gists => {

  })
```

После выполнения этой функции, требуемые нам гисты будут доступны в переменной *gists* последней переданной функции. Теперь нам нужно передать их компоненту *App*.

Для этого поправим обработку сетевого пути / следующим образом:

```
app.get('/', (req, res) => {
  fetch('https://api.github.com/users/gaearon/gists')
    .then(response => response.json())
    .then(gists => {
      const body = ReactDOM.renderToString(<App gists={gists}
          />)
```

```
      const html = template(body)
      res.send(html)
    })
})
```

Собственно теперь мы сначала загружаем список гистов, а затем передаем их компоненту *App*.

После того, как компонент *App* будет отрисован, и мы получим его разметку, мы можем воспользоваться шаблоном из предыдущей части, чтобы вернуть полную страницу пользователю.

Запустим приложение следующей командой и откроем страницу http://localhost:3000 в браузере, где вы будете должны увидеть список гистов, пришедший с сервера:

```
npm run build && npm start
```

Чтобы убедиться, что гисты были отрисованы на сервере, вы можете проверить исходный код страницы:

$$view - source : http : //localhost : 3000/$$

Там вы увидите разметку и описание гистов.

Все выглядит просто и вроде даже работает, но если мы откроем DevTools, то увидим следующую ошибку:

> Cannot read property 'map' of undefined

Проблема в том, что в браузере приложение перерисовывается снова, но получить список гистов ему уже неоткуда.

На первый взгляд это может выглядеть контринтуитивно, так как мы ожидаем, что внутри React достаточно мозгов для использования уже отрисованных гистов. Но все работает немного не так, поэтому нам нужно сделать доступным список гистов также и на клиентской стороне.

Вы можете подумать, что в этом случае достаточно сделать запрос к данным еще раз из браузера. Это безусловно будет работать, но в этом случае HTTP запрос будет выполнен дважды (один раз с сервера, второй раз из браузера), чего хотелось бы избежать.

Мы уже сделали запрос к данным один раз с сервера, поэтому у нас на руках есть все данные, которые нам нужны. Типичным для такого случая решением может быть сохранение сериализованных данных на странице HTML и их десериализация в браузере.

Может звучать запутанно, но на практике все очень просто. Давайте посмотрим на пример.

Прежде всего нам потребуется поправить шаблон страницы, чтобы сохранить в нем JSON с данными гистов:

```
export default (body, gists) => `
  <!DOCTYPE html>
  <html>
    <head>
      <meta charset="UTF-8">
    </head>
    <body>
      <div id="app">${body}</div>
      <script>window.gists = ${JSON.stringify(gists)}</script
         >
      <script src="/bundle.js"></script>
    </body>
  </html>
`
```

Теперь шаблон принимает два параметра: *body* с разметкой компонента *App* как прежде и *gists* с загруженными данными. Последний сохраняется в глобальную переменную для дальнейшего использования на клиенте.

Также нужно не забыть поправить использование этого шаблона, чтобы список гистов действительно передавался в него:

```
const html = template(body, gists)
```

И в конце нам нужно забрать данные из глобальной переменной *gists* на клиенте и передать их компоненту *App*, для чего надо будет поправить файл *client.js* (прим.пер. скорее всего здесь ошибка в коде и должен быть использован JSON.parse на window.gists):

```
ReactDOM.render(
  <App gists={window.gists} />,
  document.getElementById('app')
)
```

Таким образом, мы берем данные напрямую из глобального объекта и передаем их компоненту *App*, который перерисуется с ними уже на клиенте.

И снова запустим сервер:

```
npm run build && npm start
```

Теперь при открытии страницы http://localhost:3000, ошибка появляться не будет. А также мы можем проверить через DevTool, что компонент *App* получает данные гистов через *props*.

8.5 Next.js

Мы разобрались с основами отрисовки React приложений на стороне сервера, а также создали небольшой проект, который можно брать за основу для создания других приложений.

Однако, вы можете подумать, что для создания простого Универсального приложения требуется написать слишком много шаблонного кода и разобраться в множестве инструментов.

Знайте, вы не первый, кто испытывает подобные ощущения, которые часто называют **Усталостью от JavaScript (JavaScript Fatigue)**, о чем сказано во вступлении к этой книге.

К счастью разработчики Facebook и другие компании в сообществе React работают над улучшением DX (Developers Experience) и упрощением жизни разработчиков. Скорее всего вы уже использовали $create - react - app$ для создания приложений, и понимаете насколько оно упрощает создание и запуск приложения.

На данный момент $create - react - app$ не поддерживает SSR, но есть компания *Zeit*, которая разработала инструмент **Next.js**, который значительно упрощает создание Универсальных приложений, помогает избежать медитаций над файлами конфигураций, а также уменьшает количество шаблонного кода.

Важно понимать, что использование правильных абстракций, может значительно ускорить процесс создания приложений. Но важно понимать, как все работает изнутри, прежде чем добавлять еще прослойки из инструментов. Именно поэтому мы сначала разбирались, как сделать все вручную, перед изучением Next.js.

Мы уже посмотрели, как работает SSR и как передать состояние приложения от сервера клиенту. Теперь, зная основные концепции SSR, мы можем добавить инструмент, который скроет от нас часть этого процесса и позволит писать меньше кода для достижения того же результата.

Мы создадим аналогичное приложение, в котором будут загружаться гисты Дена Абрамова, но сделаем это с помощью Next.js.

Начнем с того, что откроем пустую папку и создадим в ней проект:

```
npm init
```

После этого мы можем установить саму библиотеку:

```
npm install --save next
```

Теперь добавим в package.json скрипт для запуска этой библиотеки:

```
"scripts": {
  "dev": "next"
},
```

Великолепно, теперь мы можем перейти к созданию компонента *App*.

Next.js основан на некоторых договоренностях, одна из самых важных заключается в том, что вы можете создавать страницы, которые будут соответствовать разным URL. Страница по умолчанию - index, поэтому мы можем создать директорию *pages* и файл *index.js* внутри нее.

Добавим в этот файл следующие зависимости:

```
import React from 'react'
import fetch from 'isomorphic-fetch'
```

Мы также будем использовать $isomorphic - fetch$, так как хотим использовать функцию $fetch$ на сервере.

Теперь создадим класс для нашего приложения, который будет наследовать $React.Component$:

```
class App extends React.Component
```

В этом классе мы создадим **static async** функцию $getInitialProps$, в которой мы сможем сказать Next.js, какие данные мы хотим загружать и на клиенте и на сервере. Библиотека сделает объект, возвращенный из этой функции, доступным внутри компонента.

Ключевые слова *static* и *async* говорят о том, что метод можно будет использовать непосредственно из класса, а не через его экземпляры, и о том, что внутри функции используются асинхронные операции.

Разбор этих концепций выходит за рамки книги, но если они вас заинтересовали, вы всегда можете взглянуть на ECMAScript proposals.

Реализацию данного метода будет выглядеть следующим образом:

```
static async getInitialProps() {
  const url = 'https://api.github.com/users/gaearon/gists'
  const response = await fetch(url)
  const gists = await response.json()
  return { gists }
}
```

В этой функции мы делаем асинхронный сетевой запрос на загрузку списка гистов. Дожидаемся выполнения этой функции и возвращаем объект с данными.

Метод *render* компонента остается очень похожим на предыдущий:

```
render() {
  return (
    <ul>
      {this.props.gists.map(gist => (
        <li key={gist.id}>{gist.description}</li>
      ))}
    </ul>
  )
}
```

Единственное отличие заключается в строке *this.props.gists*, так как теперь мы используем класс.

И не стоит забывать про *PropTypes*, чтобы интерфейс компонента оставался в актуальном состоянии:

```
App.propTypes = {
  gists: React.PropTypes.array,
}
```

Теперь мы можем экспортировать компонент:

```
export default App
```

И запустить приложение из консоли:

```
npm run dev
```

Об успешном запуске должно свидетельствовать сообщение:

> Ready on http://localhost:3000

Если мы откроем эту страницу в браузере, то увидим Универсальное приложение в действии.

На этом примере вы можете увидеть, насколько Next.js упрощает создание Универсальных приложений.

Вы также можете заметить, что если вы начнете редактировать код с запущенным сервером, то страница будет автоматически обновляться. Это работает за счет того, что Node.js реализует возможность замены кода на лету (hot module replacement), что очень удобно во время разработки приложения.

Если вам понравился Node.js, никто не запрещает вам добавить им звездочку на GitHub:

https://github.com/zeit/next.js

8.6 Заключение

Приключение по отрисовке приложения на стороне сервера подошло к концу. Теперь вы должны понимать, как создать Универсальное приложение, и какие плюсы это может принести. SEO - главная причина для отрисовки приложения в большинстве случаев, но отображение ссылок в социальных сетях и производительность могу также сказать свое веское слово.

Также мы разобрались, как загрузить данные на стороне сервера и передать в браузер через сохранение внутри HTML кода.

И в конце мы посмотрели, как специальные инструменты, такие как Next.js, могут уменьшить сложность кода, которую обычно приносит SSR.

И в следующей главе мы поговорим о производительности React приложений.

Глава 9

Улучшаем Производительность Приложений

Производительность web приложения может в значительной степени влиять на пользовательский опыт и конверсию. React реализует множество подходов, чтобы уменьшить изменение DOM браузера настолько, насколько это возможно. Применение изменений к DOM - достаточно дорогостоящая операция, поэтому критически важным становится уменьшение их количества.

Однако, есть некоторое множество сценариев, в которых React не может оптимизировать процесс отрисовки компонентов. В таких случаях разработчик вынужден реализовать решение, которое поможет приложению работать плавно.

В этой главе мы разберем базовые концепции производительности React приложений, и посмотрим на API библиотеки React, которые мы можем использовать, чтобы помочь библиотеке оптимально обновлять DOM. Также мы разберем распространенные ошибки, которые приводят к существенному ухудшению производительности приложения.

На простых примерах мы разберем, какие инструменты мы можем добавить в проект, чтобы отслеживать производительность приложения и искать проблемные участки кода. Также мы посмотрим, как неизменяемые (immutable) объекты и *PureComponent* могут помочь в создании быстрого приложения.

В этой главе мы разберем следующие вопросы:

- Как работает согласование (reconciliation), и как мы можем исполь-

зовать ключи (keys) для увеличения эффективности этого процесса

- Как использование production версии React меняет его производительность
- Как использовать *PureComponent* и метод жизненного цикла *shouldComponentUpdate*
- Основные способы оптимизации приложения и ошибки, вызывающие падение производительности
- Использование неизменяемых (immutable) данных
- Полезные инструменты и библиотеки для улучшения производительности приложения

9.1 Согласование и ключи

В большинстве случаев React работает достаточно быстро, чтобы не думать о проблемах производительности приложения. React использует различные техники для оптимизации отрисовки компонентов на экране.

Для отрисовки React вызывает рекурсивно метод **render** у компонента, затем у его потомков, затем у потомков потомков и т.д. Вызов метода *render* возвращает дерево элементов, которое React использует для осознания, что нужно изменить в DOM для обновления UI.

Когда состояние компонента меняется, React снова пересоздает дерево элементов с помощью метода *render*, а затем сравнивает полученный результат с предыдущим деревом. React достаточно умен, чтобы понять, какие операции достаточно применить к DOM, чтобы отобразить ожидаемые изменения UI. Этот процесс называется **согласованием (reconciliation)** и работает под капотом React. Благодаря этому, мы можем описывать в декларативном стиле, как должны выглядеть наши компоненты, пока React делает всю остальную работу по отрисовке.

React пытается делать как можно меньше изменений в DOM, так как внесение изменений в DOM - относительно дорогостоящая операцию.

Однако, сравнение двух деревьев с элементами также не бесплатно, поэтому React делает два допущения, чтобы упростить этот процесс:

- Если два элемента обладают разным типом, они отображаются в разные деревья

- Разработчик может использовать ключи (keys) для обозначения дочерних элементов стабильными при разных вызовах метода *render*

С точки зрения разработчика второй пункт интереснее, так как предоставляет инструмент для помощи React с оптимизацией отрисовки компонентов.

Сейчас мы посмотрим на простой пример, чтобы понять, как использование ключей может значительно повлиять на производительность приложения.

Давайте создадим простой компонент, который будет отрисовывать список объектов и кнопку, по нажатию на которую будет добавляться еще один объект в список, что будет приводить к перерисовке компонента.

Мы создадим компонент с помощью класса, так как нам нужно хранить список объектов для отрисовки, а также создадим обработчик событий для кнопки:

```
class List extends React.Component
```

В конструкторе класса *List* добавим создание списка объектов для первой отрисовки, а также привяжем обработчик событий к экземпляру класса:

```
constructor(props) {
  super(props)
  this.state = {
    items: ['foo', 'bar'],
  }

  this.handleClick = this.handleClick.bind(this)
}
```

Создадим собственно сам обработчик событий, который будет добавлять новый элемент в список объектов на отрисовку:

```
handleClick() {
  this.setState({
    items: this.state.items.concat('baz'),
  })
}
```

И в конце создадим метод *render*, который будет отрисовывать объекты, сохраненные внутри компонента, а также кнопку для добавления новых:

```
render() {
  return (
    <div>
      <ul>
        {this.state.items.map(item => <li>{item}</li>)}
      </ul>
      <button onClick={this.handleClick}>+</button>
    </div>
  )
}
```

Компонент готов, и если вы добавите его в свое приложение (или создадите для этого новое с помощью *create-react-app*), то вы увидите на экране элементы **foo** и **bar**, а также кнопку *+*, по нажатию на которую добавиться элемент **baz**.

Все работает так, как мы и ожидаем, но для получения полноты картины происходящего внутри React, мы добавим еще один инструмент, который поможет собирать и отображать информацию о производительности приложения. Для его установки можно воспользоваться npm:

```
npm install --save-dev react-addons-perf
```

После установки мы можем импортировать его в файл с компонентом *List*:

```
import Perf from 'react-addons-perf
```

У объекта $Perf$ есть несколько полезных методов, которые мы можем использовать для отслеживания производительности React компонентов. С помощью методов *start*() и *stop*() мы можем соответственно начать и остановить запись информации о производительности компонента.

Есть несколько методов для отображения в консоли информации, собранной ранее. Один из самых используемых - $printWasted$, который печатает информацию о том, сколько времени было затрачено на вызовы метода *render*, которые не привели к изменению DOM. Также есть метод $Perf$, который позволяет получать информацию о том, какие операции в DOM выполняет React. Этот метод называет $printOperations$, и мы будем использовать его для оценки влияния использования Ключей на производительность приложения.

Для того, чтобы начинать и останавливать сбор информации о производительности, мы можем методы жизненного цикла $componentWillUpdate$ и $componentDidUpdate$ соответственно, в последнем мы можем также печатать результаты.

Сначала реализуем метод *componentWillUpdate*, который вызывается непосредственно перед обновлением и перерисовкой компонента:

```
componentWillUpdate() {
  Perf.start()
}
```

Как и было сказано выше, в этом методе нужно просто начать сбор информации о производительности с помощью метода *start*(). После того, как компонент обновился, мы можем остановить сбор информации и распечатать результаты:

```
componentDidUpdate() {
  Perf.stop()
  Perf.printOperations()
}
```

Как вы видите, мы останавливаем измерения и вызываем метод *printOperations*, чтобы увидеть, какие операции сделал React над DOM для добавления элемента **baz** на экран.

Если мы запустим компонент и нажмем кнопку *+*, то увидим в консоли браузера список совершенных операций в виде таблицы. Нас интересуют столбцы *Operation*, который показывает *"insert child"*, и *Payload*, в котором мы можем увидеть *"{"toIndex":2,"content":"LI"}"*

React распознал, что после добавления новой строки в список необходимо создать новый дочерний элемент *Li*, причем индекс этого элемента должен быть 2 (т.е. третий элемент).

Как вы можете заметить, вместо того, чтобы перерисовывать все компоненты, React высчитал минимально количество операций, которых будет достаточно для обновления DOM. Этот механизм работает прекрасно, и его достаточно для большинства ситуаций, которые могут возникнуть.

Однако есть случаи, когда у React оказывается немного недостаточно мозгов для выбора оптимального набора операций. В некотором множестве таких случаем на помощь может придти использование Ключей. Если мы немного поправим пример выше и сделаем добавление **baz** не в конец списка, а в начало, то мы увидим совсем другую картину.

Для того, чтобы добавить **baz** в начало списка, мы можем использовать метод *unshift*, но его проблема в том, что он не создает новый массив, а мутирует существующей. Для того, чтобы создать копию массива, мы можем использовать метод *slice* без аргументов (прим.пер. или использовать спред оператор, привет из 2019):

```
handleClick() {
  const items = this.state.items.slice()

  items.unshift('baz')
  this.setState({
    items,
  })
}
```

Таким образом мы копируем массив, добавляем в его начало строку **baz** и сохраняем обратно в state, вызывая тем самым перерисовку.

Теперь, когда мы запустим компонент снова, то снова увидим элементы **foo** и **bar**, но после нажатия на кнопку *+*, строка **baz** будет добавлена уже в начало списка.

Все работает так, как мы ожидаем, но если мы снова откроем DevTools, то увидим, что на этот раз React совершил множество операций над DOM, а именно:

- Заменил текст первого элемента на **baz**
- Заменил текст первого элемента на **foo**
- Добавил в конец списка новый элемент с текстом **bar** и индексом 2

Таким образом React вместо того, чтобы добавить один новый дочерний элемент в начало списка, изменил оба существующих и добавил новый элемент в конец.

Это происходит из-за того, что React после сравнения первых элементов до и после перерисовки увидел в них разницу и поправил DOM в соответсвии с этими изменениями, затем то же самое произошло со вторыми элементами, а в конце React увидел, что в списке появился новый элемент и добавил соответствующий ему элемент.

Конкретно в этом примере это не приводит к видимым потерям производительности, но в реальном приложении могут отображаться списки из сотен элементов, полная перерисовка которых может привести к существенному замедлению приложения.

Но каждая проблемы имеет свое решение, и в данном случае мы можем собственно использовать **Ключи**, которые помогут React понимать, какие из элементов обновились, добавились или удалились.

Для использования ключей достаточно добавить уникальный атрибут *key* каждому из элементов списка. Важно, чтобы эти ключи не менялись для каждого элемента после перерисовки, так как React будет использовать их для сопоставления элементов до и после перерисовки.

Например, мы можем изменить метод *render* следующим образом:

```
render() {
  return (
    <div>
      <ul>
        {this.state.items.map(item => <li key={item}>{item}</
            li>)}
      </ul>
      <button onClick={this.handleClick}>+</button>
    </div>
  )
}
```

После того как мы установили значение поля *key* для каждого из элементов в значение этих элементов, мы можем запустить приложение заново. Сразу же мы можем отметить, что функционально поведение компонента не изменилось: есть список из двух элемента и кнопка, по нажатию на которую добавляется новый элемент в начало списка.

Однако, если мы откроем DevTool, то увидим, что теперь снова всего одна операция над DOM, причем это операция добавления дочернего элемента по индексу 0.

Таким образом мы помогли React правильно рассчитать минимальное количество операций, которых будет достаточно для обновления DOM. С помощью этого простого метода мы можем избежать значительных падений производительности при отображении больших списков элементов.

Также отметим, что если мы забудем добавить ключи там, где это требуется, React вежливо напомнит нам об этом:

Each child in an array or iterator should have a unique "key"prop. Check the render method of ‘List‘.

Это сообщение очень удобно, так как говорит о том, какой именно компонент нам необходимо поправить.

Также, если вы используете **Eslint**, о котором мы говорили в Главе 2, и добавили правило *jsx-key* для *eslint-plugin-react*, то получите аналогичную ошибку на этапе статического анализа кода.

9.2 Техники оптимизации

Важно заметить, что во всех примерах в этой книге мы используем или приложение, созданное с помощью *create-react-app*, или приложение, созданное с нуля, но всегда с версией React для разработки (development version).

Использование версии React для разработки очень удобно для написания кода и отладки, но нужно понимать, что все проверки и уведомления стоят потерь производительности приложения, которых хотелось бы избежать в финальной сборке приложения.

Таким образом, первая оптимизация, которую мы должны сделать, это собрать приложение при установленной переменной окружения NODE_ENV в *production*. В случае использования **Webpack** это всего лишь вопрос использования *DefinePlugin*:

```
new webpack.DefinePlugin({
  'process.env': {
    NODE_ENV: JSON.stringify('production')
  }
}),
```

Дальнейшим улучшением нашей сборки может быть минимизация кода, после которой бандл приложения будет меньше весить, а значит быстрее загружаться по сети. Для этого мы можем добавить соответствующий плагин в список плагинов **Webpack**:

```
new webpack.optimize.UglifyJsPlugin()
```

Если мы запустим приложение и поймем, что некоторые его части все еще выполняются медленно, то мы можем начать применять другие методы оптимизации React приложения.

Важно сказать, что не стоит оптимизировать приложение, пока нет измерений его производительности и понимания, что является узким горлышком. Преждевременная оптимизация приложений как правило ведет к излишней сложности последних, чего мы конечно хотели бы избежать.

Помимо этого, в очередной раз напомним, что React, независимо от нашего кода, уже применяет различные методы оптимизации, так что в большинстве случаев нам вообще не стоит об этом задумываться.

Однако в некоторых случаях, когда из коробки React работает недостаточно быстро, и мы хотим ему помочь, мы можем приказать React остановить процесс согласования для частей дерева элементов.

9.2.1 shouldComponentUpdate

Первое знакомство с процессом согласования в React может стать некоторой болью для многих разработчиков. Интуитивно мы ожидаем, что если с компонентами ничего не происходит, то метод *render* для них не вызывается повторно. К сожалению это далеко от истины.

На самом деле, для того, чтобы правильно оценить все изменения, которые необходимо выполнить в DOM, React вызывает метод *render* для все компонентов, а затем сравнивает результат с предыдущим состоянием дерева элементов.

Если ничего не изменилось, то никаких изменений не будет применено к DOM, но если компоненты сами по себе выполняют тяжелые (с точки зрения вычислений) операции, то они могут замедлять работу всего приложения, даже если изменений в DOM при этом совершаться не будет.

React не может сам оценить, какие компоненты стоит обновлять, а какие нет, но мы можем реализовать специальный метод, чтобы помочь ему в этом вопросе.

Этот метод называется *shouldComponentUpdate*, и если он вернет *false* в процессе обновления дерева, то для него и всех его дочерних элеметов метод *render* вызван не будет.

Например, если вы просто добавите следующий код в созданный ранее *List*:

```
shouldComponentUpdate() {
  return false
}
```

То при нажатии на кнопку + вы не увидите никаких изменений, несмотря на то, что состояние приложения меняется. Это происходит из-за того, что мы явно сказали React, что обновлять компонент не нужно.

Просто возвращать из этого метода *false* с небольшой вероятностью принесет пользу, но ничто не запрещает нам проверить не были ли изменены параметры или состояние этого компонента.

Например, в случае с компонентом *List* мы можем проверить, не был ли изменен список с отображаемыми элементами и вернуть соответствующее значение.

Для того, чтобы осуществить эту проверку, мы можем использовать два параметра, которые React передает в вызов этого метода: *nextProps*

и *nextState*, которые отвечают за будущие параметры и состояние этого компонента соответственно.

В нашем случае реализацию может выглядеть следующим образом:

```
shouldComponentUpdate(nextProps, nextState) {
  return this.state.items !== nextState.items
}
```

Мы возвращаем *true* только в том случае, если список элементов для отображения не изменился, а *false* в остальных случаях. Предположим, что компонент *List* может быть использован внутри другого компонента, который сам по себе изменяется довольно часто, но при этом не влияет на отображение компонента *List*. В этом случае с помощью метода *shouldComponentUpdate* мы можем сказать React, что не нужно обновлять компонент *List* и его потомков.

Реализация проверки всех параметров компонента и его состояния на наличие изменений может ввести в уныние. А поддержка сложных реализаций метода *shouldComponentUpdate* будет отнимать много времени, особенно при часто меняющихся требованиях.

В этих целях React предоставляет уже готовый компонент, который осуществляет сравнение всех параметров и состояния в методе *shouldComponentUpdate*.

Использовать этот компонент очень просто, для этого достаточно наследовать наши компоненты от *React.PureComponent* вместо *React.Component*.

Важно заметить, что данный компонент сравнивает только ссылки объектов в параметрах и состоянии, но не осуществляет глубокое сравнение, что иногда может привести к неожиданным результатам.

Данный подход хорошо работает при использовании неизменяемых структур данных, о которых мы поговорим далее. Также стоит отметить, что глубокое сравнение сложных объектов иногда может занимать больше времени чем их отрисовка.

Поэтому использовать *PureComponent* стоит только в том случае, если есть проверенный факт потери производительности и компоненты занимают слишком много времени для отрисовки.

9.2.2 Функциональные компоненты

Еще один неочевидный факт о React заключается в том, что функциональные компоненты не дают никакой выгоды в плане производительности приложения.

Легко подумать, что они должны работать быстрее, так как не создаются экземпляры классов, нет внутреннего состояние и обработчиков событий, но все это вносит вклад в улучшение производительности как минимум второго порядка.

На данный момент во многих ситуациях функциональные компоненты работают даже хуже, так как для них нельзя реализовать функцию *shouldComponentUpdate*, которая может значительно уменьшить затраты вычислительных ресурсов.

(Прим.пер. на данный момент функциональные компоненты уже начинают править балом, для них есть hook api, кеширование и, скорее всего, различные внутренние оптимизации)

9.3 Распространенные решения

Мы уже рассмотрели, как *PureComponent* может помочь нам оптимизировать отрисовку React компонентов. Если направить его в правильное русло, можно значительно улучшить производительность приложения. Но еще раз напомним, что использовать этот компонент стоит, когда есть обнаруженные и измеренные проблемы производительности.

Есть некоторое множество неочевидных моментов, когда использование *PureComponent* не дает нам ожидаемый результат. Чаще всего это происходит тогда, когда изменяются параметры или состояние компонента, а мы думаем, что они этого делать не должны. Иногда бывает довольно сложно определить, что именно привело компонент к перерисовки, или какой компонент следует оптимизировать с помощью *PureComponent*.

В этой части мы рассмотрим инструменты и подходы для исправления проблем с перерисовкой компонентов. Также мы посмотрим, как можно разделить большой компонент, чтобы улучшить производительность.

9.3.1 Почему ты обновился?

Понять, должен ли компонент обновляться или нет, можно разными способами. Один из простейших, установить специальную библиотеку, которая будет предоставлять нам такую информацию автоматически.

Прежде всего, установим библиотеку **why-did-you-update**:

```
npm install --save-dev why-did-you-update
```

И добавим следующий кусочек кода сразу после импорта React:

```
if (process.env.NODE_ENV !== 'production') {
  const { whyDidYouUpdate } = require('why-did-you-update')
  whyDidYouUpdate(React)
}
```

Таким образом, мы просто говорим, что если находимся в процессе разработки, то нужно импортировать эту библиотеку и применить ее к React. Не стоит пускать ее в сборку, которая уйдет к пользователям.

Теперь, если мы вернемся к примеру компонента *List* из предыдущей главы и немного его поправим, то мы сможем увидеть работу этой библиотеки в действии.

Для начала нам нужно поправить метод *render*:

```
render() {
  return (
    <div>
      <ul>
        {this.state.items.map(item => (
          <Item key={item} item={item} />
        ))}
      </ul>
      <button onClick={this.handleClick}>+</button>
    </div>
  )
}
```

Мы заменим отрисовку простых элементов внутри *map* на другой компонент *Item*, который скоро создадим. Этому компоненту мы будем передавать элементы списка для отрисовки, а также параметр *key*, чтобы React мог понять, какие из элементов уже существовали перед обновлением.

Реализуем этот компонент с помощью *React.Component*:

```
class Item extends React.Component
```

На данный момент компонент *Item* будет реализовывать только метод *render* для отрисовки элементов списка:

```
render() {
  return (
    <li>{this.props.item}</li>
  )
}
```

Также поправим использование Perf, так как сейчас нас больше интересует время, потраченное на вызовы методов *render*, которые не привели к обновлению DOM:

```
componentDidUpdate() {
  Perf.stop()
  Perf.printWasted()
}
```

Отлично, теперь мы можем запустить этот компонент. На экране мы увидим то же самое, что и раньше: элементы **foo**, **bar** и кнопку *+*. Самое интересное начинается в консоли после нажатия на вышеупомянутую кнопку.

Прежде всего вывод из библиотеки **whyDidYouUpdate**, которая сообщает, какие перерисовки можно было избежать:

```
Item.props
  Value did not change. Avoidable re-render!
  before Object {item: "foo"}
  after  Object {item: "foo"}
Item.props
  Value did not change. Avoidable re-render!
  before Object {item: "bar"}
  after  Object {item: "bar"}
```

Эти данные наглядно показывают, что даже если React не перерисовывает элементы **foo** и **bar**, соответствующие им методы render все равно вызываются. Это очень полезная информация, которую часто не так легко получить.

Далее можно увидеть вывод в консоль расширения Perf, который показывает, сколько времени было затрачено на работу методов *render*, но не вызвавших изменений в DOM.

Теперь мы легко можем поправить эту проблему заменой наследования класса *Item* с *extends React.Component* на:

```
class Item extends React.PureComponent
```

Если мы снова запустим приложение и нажмем кнопку *+*, то убедимся, что устрашающие сообщения в консоли больше не появляются. Это означает, что компонент *Item* больше не перерисовывается без изменений параметров.

Конкретно в этом случае мы можем и не получить большой выгоды в производительности, но если вы представите себе приложение, которое

отрисовывает списки из сотен элементов, то там любая лишняя перерисовка всего списка может вызывать замедление работы программы.

9.3.2 Создание функций внутри метода *render*

Теперь давайте попробуем добавить новые возможности в компонент *List*, как если бы мы работали с реальным приложением, и посмотрим, как мы можем свести на ноль всю пользу от использования *PureComponent*.

Например, мы хотим добавить обработчик нажатий на каждый элемент списка и выводить в консоль содержимое элемента в случае его срабатывания.

Это немного далековато от чего-то реально полезного для пользователя, но вы можете легко представить, что по нажатию на элемент списка происходит любое другое событие, например открытие нового окна с детальной информацией по объекту.

Для этого мы добавим пару изменение в методы *render* компонентов *List* и *Item*.

Начнем с первого из них:

```
render() {
  return (
    <div>
      <ul>
        {this.state.items.map(item => (
          <Item
            key={item}
            item={item}
            onClick={() => console.log(item)}
          />
        ))}
      </ul>
      <button onClick={this.handleClick}>+</button>
    </div>
  )
}
```

Теперь параметр *onClick* мы передаем функцию, которая печатает содержимое элемента в консоль, компоненту *Item*.

Теперь осталось только передать эту функцию в элемент *<li>* внутри компонента *Item*:

```
render() {
  return (
```

```
      <li onClick={this.props.onClick}>
        {this.props.item}
      </li>
    )
}
```

Мы все еще используем *PureComponent* и ожидаем, что после добавления элемента **baz**, метод *render* для уже существующих элементов списка вызываться повторно не будут.

К сожалению, если мы запустим приложение, то увидим несколько новых сообщений в консоли: первое от библиотеки *whyDidYouUpdate*, которая говорит, что есть потенциально лишние вызовы метода *render*, так как функция *onClick* всегда одна и та же:

```
Item.props.onClick
  Value is a function. Possibly avoidable re-render?
  before onClick() {
    return console.log(item);
  }
  after  onClick() {
    return console.log(item);
  }
Item.props.onClick
  Value is a function. Possibly avoidable re-render?
  before onClick() {
    return console.log(item);
  }
  after  onClick() {
    return console.log(item);
  }
```

Второе от расширения Perf, которое говорит, что мы теряем время на отрисовке компонента *List* > *Item*

Причина, по которой в компонент *Item* передается каждый раз новая функция, в том, что стрелочная функция, которая создается в каждом вызове метода *render* компонента *List*, возвращает каждый раз новый экземпляр функции, даже если реализация этой функции не изменилась. Это очень распространенная ошибка, которую достаточно просто поправить.

К сожалению, мы не можем вынести реализацию обработчика событий за метод *render*, так как для каждого из элементов списка нам нужно выводить в консоль разные данные. Так что создание обработчика внутри цикла отрисовки списка выглядит вполне логичным решением.

Но мы можем перенести логику печати внутрь самого компонента *Item*, который знает о том, по какому именно компоненту было произведено нажатие.

Давайте посмотрим, как мы можем реализовать компонент *Item*, который будем наследовать от *PureComponent*:

```
class Item extends React.PureComponent
```

В конструкторе мы привяжем к экземпляру самого класса обработчик событий, который будет частью реализации класса *Item*:

```
constructor(props) {
  super(props)

  this.handleClick = this.handleClick.bind(this)
}
```

Внутри функции *handleClick* мы будем вызывать функции *onClick*, которую получаем из параметров, и передавать в нее содержимое текущего элемента:

```
handleClick() {
  this.props.onClick(this.props.item)
}
```

Теперь мы можем использовать в методе *render* обработчик, созданный внутри класса:

```
render() {
  return (
    <li onClick={this.handleClick}>
      {this.props.item}
    </li>
  )
}
```

Осталось только изменить метод *render* класса *List* так, чтобы он не создавал новые экземпляры функций при каждом вызове:

```
render() {
  return (
    <div>
      <ul>
        {this.state.items.map(item => (
          <Item
            key={item}
            item={item}
```

```
            onClick={console.log}
          />
        ))}
      </ul>
      <button onClick={this.handleClick}>+</button>
    </div>
  )
}
```

Как вы можете увидеть, мы передаем функцию, которую хотели использовать (в данном случае *console.log*), которая вызывается внутри каждого дочернего компонента с нужными параметрами. Но теперь мы используем одну и ту же функцию внутри класса *List*, поэтому ее не нужно пересоздавать для каждого дочернего элемента в отдельности.

Если мы запустим приложение и нажмем кнопку *+*, то убедимся, что теперь в консоли снова нет сообщений о лишних вызовах метода *render*.

Также, если мы кликнем по какому-либо элементу в списке, то увидим соответствующее сообщение в консоли.

Как сказал *Ден Абрамов*, использование стрелочных функций внутри метода *render* не является проблемой само по себе, но мы должны быть осторожны и проверять, что мы не вызываем этим лишние перерисовки компонентов.

9.3.3 Параметры константы

Давайте продолжим расширять наш список и посмотрим, что еще может произойти при добавлении новых возможностей.

Рассмотрим еще одну распространенную ошибку, которая снижает эффективность *PureComponent*.

Предположим, что элементы нашего списка должны мочь получать через параметры список доступных для них статусов. Такую возможность можно реализовать разными способами, но сейчас остановимся на варианте, в котором мы просто укажем доступные статусы в JSX.

Изменим метод *render* компонента *List* следующим образом:

```
render() {
  return (
    <div>
      <ul>
        {this.state.items.map(item => (
          <Item
```

```
            key={item}
            item={item}
            onClick={console.log}
            statuses={['open', 'close']}
          />
        ))}
      </ul>
      <button onClick={this.handleClick}>+</button>
    </div>
  )
}
```

Параметры *key*, *item* и *onClick* мы уже разбирали ранее. Сейчас мы добавили параметр *statuses*, в котором собственно передаем массив с возможными статусами для элемента.

Теперь, если мы снова запустим приложение и нажмем кнопку +, то увидим следующее сообщение:

```
Item.props.statuses
  Value did not change. Avoidable re-render!
  before ["open", "close"]
  after  ["open", "close"]
Item.props.statuses
  Value did not change. Avoidable re-render!
  before ["open", "close"]
  after  ["open", "close"]
```

Эти сообщения говорят о том, что несмотря на то, что содержимое массивов со статусами остается одним и тем же, экземпляры этих массивов создаются заново в каждом вызове метода *render*.

Причина в том, что все объекты в JavaScript при создании возвращают новый экземпляр (это не относится к примитивным типам), т.е. два отдельно созданных объекта (даже с одинаковым содержимом) всегда имеют разные адреса в памяти:

```
[] === []
  false
```

Помимо этого в консоли можно увидеть информацию о времени, потраченном на выполнение методов *render*, не вызвавших изменений в DOM.

Основное, что мы можем сделать в данной ситуации, это вынести создание этого массива в отдельную константу, которая будет инициализирована единожды, а затем использовать ее для всех элементов списка:

```
const statuses = ['open', 'close']
...
render() {
  return (
    <div>
      <ul>
        {this.state.items.map(item => (
          <Item
            key={item}
            item={item}
            onClick={console.log}
            statuses={statuses}
          />
        ))}
      </ul>
      <button onClick={this.handleClick}>+</button>
    </div>
  )
}
```

Если мы снова запустим приложение, то увидим, что сообщения в консоли пропали, что говорит о том, что элементы снова не перерисовываются без необходимости при добавлении новых элементов.

9.3.4 Рефакторинг и хороший дизайн

В последней части данной части мы посмотрим, как можно поправить существующий компонент (или создать новый в лучшем стиле) и улучшить при этом производительность приложения.

Слабый архитектурные решения всегда приводят к проблемам. В случае с React, например, неправильный выбор места хранения состояния внутри дерева элементов может привести к тому, что компоненты будут отрисовывать больше, чем требуется.

Как мы уже говорили ранее, от того, что какой-то компонент отрисует себя лишний раз, скорее всего хуже не станет. Проблемы обычно начинаются, когда большие списки элементов начинают дружно перерисовываться по любому чиху из вне.

Компонент, который мы сейчас создадим, похож на предыдущий пример в том смысле, что это будет еще одно приложение со списком элементов, но теперь в нем появится форма, которая позволит пользователю добавлять новые элементы.

Как всегда мы начнем с простого примера и будет его шаг за шагом улучшать.

Мы назовем компонент *Todos*, и он будет наследовать *extends React.Component*:

```
class Todos extends React.Component
```

В конструкторе мы определим начальное состояние и привяжем обработчики событий к экземпляру класса:

```
constructor(props) {
  super(props)

  this.state = {
    items: ['foo', 'bar'],
    value: '',
  }

  this.handleChange = this.handleChange.bind(this)
  this.handleClick = this.handleClick.bind(this)
}
```

У состояния есть два атрибута:

- **items**: Массив с парой предустановленных элементов, в который будут добавляться новые значения
- **value**: Поле, в которое будет сохраняться текущее состояние ввода пользователя.

Вы уже можете догадаться, как будут выглядеть обработчики событий. Создадим метод *handleChange*, который будет вызываться каждый раз, когда пользователь меняет значение в поле ввода:

```
 handleChange({ target }) {
  this.setState({
    value: target.value,
  })
}
```

Мы уже обсуждали в Главе 6, что такой обработчик событий получает объект события (event), у которого есть поле *target*, через которое собственно и передается новое значение из поля ввода. Все, что нам остается сделать, это сохранить это значение внутри состояния компонента.

И еще один обработчик *handleClick*, который будет вызываться, когда пользователь нажмет кнопку добавления нового элемента в список:

```
handleClick() {
  const items = this.state.items.slice()
  items.unshift(this.state.value)

  this.setState({
    items,
  })
}
```

Этот обработчик похож на тот, который у нас уже был, с той лишь разностью, что он добавляет не константную, но введенную пользователем строку.

И остается только определить метод *render*:

```
render() {
  return (
    <div>
      <ul>
        {this.state.items.map(item => <li key={item}>{item}</
            li>)}
      </ul>
      <div>
        <input
          type="text"
          value={this.state.value}
          onChange={this.handleChange}
        />
        <button onClick={this.handleClick}>+</button>
      </div>
    </div>
  )
}
```

В данном методе мы отрисовываем список элементов из состояния компонента, сопоставляя каждому из них элемент *<li>*. Далее идет поле ввода, значение которого сохраняется в состоянии по ключу *value* и кнопка для добавления нового элемента.

Теперь мы можем запустить этот компонент и убедиться, что он работает, т.е. отображаются все элементы, меняется значения ввода при наборе текста и добавляются новые элементы по нажатию на кнопку.

В общем и целом этот компонент будет работать, и даже не будет значительно замедлять приложение, но только пока вы не решите добавить пару сотен элементов. В этом случае, вы сможете заметить, что ввод каждого символа начинает выполняться с ощутимой задержкой.

Это происходит из-за того, что React вызывает метод *render* после каждого изменения состояния, т.е. после каждого ввода со стороны пользователя, и каждый раз заново отрисовывает сотни элементов.

Мы понимаем, что после каждого введенного символа обновляться в DOM должно только это поле ввода, но для, чтобы это понял React, ему приходится вызвать метод *render* у всех дочерних элементов и сравнить полученный результат с предыдущим состоянием дерева элементов.

Теперь, если мы посмотрим на состояние компонента, то можем подумать о том, что хранить в одном месте и состояние поля ввода и список элементов было не лучшей идеей.

В общем случае стоит стремиться к тому, что каждый компонент должен выполнять одну, а в идеале только одну, функцию.

Основное решение в данной ситуации - разделить компонент на два, каждый со своей частью состояния и, соответственно, зоной ответственности.

Для того, чтобы сделать это, нам потребуется общий родительский компонент для новых компонентов, так компоненты все таки связаны. Мы не хотим перерисовывать весь список элементов после каждого ввода пользователя, но мы точно хотим обновить список после добавления нового элемента в список элементов.

Сначала мы поправим компонент *Todos*, чтобы он хранил только список элементов.

Затем мы создадим отдельные компоненты для отрисовки списка элементов и отображения формы для ввода пользователя.

Начнем с компонента *Todos*:

```
class Todos extends React.Component
```

Теперь в состоянии мы оставим только один атрибут *items*, где будет тот же самый список элементов. Также в этом компоненте останется только один обработчик событий, который будет добавлять новые элементы в состояние:

```
constructor(props) {
  super(props)

  this.state = {
    items: ['foo', 'bar'],
  }

  this.handleSubmit = this.handleSubmit.bind(this)
```

```
}
```

Реализация обработчика событий значительно не меняется:

```
 render() {
  return (
    <div>
      <List items={this.state.items} />
      <Form onSubmit={this.handleSubmit} />
    </div>
  )
}
```

Компонент *List* будет получать список элементов для отрисовки через параметры. Компонент *Form* в свою очередь будет получать через параметры функцию для добавления новых элементов.

Теперь мы можем создать дочерние компоненты *List* и *Form*. Первый из них мы можем собрать из кода предыдущей версии компонента *Todos*.

Также, чтобы компонент *List* перерисовывался только при изменении получаемого списка, мы будем наследовать его от компонента *PureComponent*:

```
class List extends React.PureComponent
```

В методе *render* мы будем просто отрисовывать список элементов:

```
render() {
  return (
    <ul>
      {this.props.items.map(item => <li key={item}>{item}</li
          >)}
    </ul>
  )
}
```

Теперь создадим компонент *Form*, в котором нам нужно будет сохранять состояние ввода пользователя, а также обрабатывать добавление новых элементов через получаемую в параметрах функцию. Этот компонент мы также будет наследовать от *PureComponent*:

```
class Form extends React.PureComponent
```

В конструкторе мы определим начальное значение поля ввода и привяжем обработчик ввода к экземпляру класса:

```
constructor(props) {
  super(props)

  this.state = {
```

```
    value: '',
  }

  this.handleChange = this.handleChange.bind(this)
}
```

Реализация обработчика остается неизменной относительно аналогичного из компонента *Todos* до начала разделения на более мелкие компоненты:

```
handleChange({ target }) {
  this.setState({
    value: target.value,
  })
}
```

И в методе *render* нам остается только отрисовать форму ввода и кнопку для сохранения элемнетов:

```
render() {
  return (
    <div>
      <input
        type="text"
        value={this.state.value}
        onChange={this.handleChange}
      />
      <button
        onClick={() => this.props.onSubmit(this.state.value)}
      >+</button>
    </div>
  )
}
```

Таким образом мы получили поле ввода с кнопкой. Стоит также заметить, что мы создаем стрелочную функцию прямо в методе *render*, но здесь мы можем спокойно это делать, так как нет дочерних элементов, которых это будет заставлять лишний раз перерисовываться (наследников компонента *PureComponent* или компонентов с похожими возможностями).

Готово. Теперь, если вы запустите приложение, то увидите, что визуально оно не изменилось, но изменения в поле ввода не заставляются перерисовываться весь список, что значительно увеличивает производительность даже на больших списках.

Мы решили проблему перерисовки конкретного списка в конкретном приложении. Отрисовка списков очень часто является узким горлышком в производительности приложений с пользовательским интерфейсом в целом и React приложений в частности, поэтому всегда проверяйте, что заставляет перерисовываться списки в ваших приложениях.

9.4 Инструменты и библиотеки

В этой части мы рассмотрим несколько полезных инструментов и библиотек, которые можно добавить в проект, чтобы эффективнее отслеживать и решать проблемы с производительностью.

9.4.1 Неизменяемые объекты

Как мы уже видели, один из самых мощных инструментов оптимизации заключается в использовании метода *shouldComponentUpdate* или компонента *PureComponent*.

Единственная проблема в том, что *PureComponent* использует поверхностное *shallow* сравнение, т.е. только ссылки на объекты. Поэтому если мы изменим какое-либо поле объекта, передаваемого такому объекту через параметры, не создавая при этом новый объект, то компонент решит, что никаких изменений не было и перерисовываться не будет.

В общем случае, поверхностное сравнение не может распознать изменение отдельных полей внутри объекта, что может приводить к отображению устаревшей информации в DOM дереве.

Один из способов решения этой проблемы - использование **неизменяемых данных (immutable data)**, т.е. таких данных, которые после своего создания не могут быть изменены.

Например, мы можем обновить состояние компонента следующим образом:

```
const obj = this.state.obj
obj.foo = 'bar'
this.setState({ obj })
```

Несмотря на то, что поле *foo* объекта меняется, сам объект (в смысле ссылка на него) остается тем же. Если такой код отработает внутри компонента, который наследуется от *PureComponent*, то перерисовки не произойдет.

Вместо этого мы можем создавать новый экземпляр объекта при каждом обновлении состояния:

```
const obj = Object.assign({}, this.state.obj,
  { foo: 'bar' })

this.setState({ obj })
```

В этом случае мы получаем новый объект, в который копируются все поля из *this.state.obj*, а также в поле *foo* устанавливается значение *bar*. Так как создается новый объект, ссылка на него будет отличаться от исходного, а поверхностное сравнение сможет обнаружить изменения.

С помощью ES2015 и Babel можно реализовать ту же идею через spread оператор:

```
const obj = { ...this.state.obj, foo: 'bar' }
this.setState({ obj })
```

Такой вариант записи лаконичнее и реализует аналогичную функциональность, но требует дополнительных инструментов, чтобы быть запущенным в браузере.

Также есть библиотека *immutable.js*, которая предоставляет вспомогательные функции для упрощения работы с неизменяемыми объектами, но ее использование требует небольшого изучения API этой библиотеки.

9.4.2 Инструменты мониторинга

Мы уже рассматривали, как мы можем использовать *Perf* для отслеживания производительности приложения.

В нашем примере мы использовали методы жизненного цикла компонента для запуска и остановки записи подсчета производительности приложения:

```
componentWillUpdate() {
  Perf.start()
}

componentDidUpdate() {
  Perf.stop()
  Perf.printOperations()
}
```

После вызова метода *stop* мы вызываем метод *printOperations*, чтобы распечатать в консоль браузера, какие операции в DOM выполнил React для применения всех необходимых изменений.

Этот инструмент очень удобен, но нам приходится использовать методы жизненного цикла и загрязнять кодовую базу для измерения производительности.

Лучшим вариантом для нас было бы иметь возможность измерять производительность компонентов, не модифицирую при этом сами компоненты. И такую возможность предоставляет для нас расширение *chrome-react-perf* для Chrome.

Его можно установить, перейдя в браузере по следующей ссылке.

Данное расширение добавляет новую вкладку в DevTools, с помощью которой мы можем запускать и останавливать *Perf* без необходимости вносить изменения в код приложения.

Еще один инструмент, который помогает нам облегчить сбор информации о производительности приложения, - *react-perf-tool*. Данный инструмент представляет из себя компонент, который мы можем добавить в наше приложение, для того, чтобы получить удобный интерфейс работы с *Perf* прямо внутри окна браузера.

Данный компонент отрисовывает небольшую консоль внизу страницы, с которой мы можем запускать и останавливать сбор информации о производительности. Помимо этого *react-perf-tool* печатает данные не в виде таблицы, а в виде графа, на котором наглядно видно, какие компоненты расходуют больше вычислительной мощности в сравнении с остальными.

9.4.3 Плагины Babel

Список планинов Babel открывает **React constant elements transformer (прим.пер. на всякий случай я не буду называть его как Неизменяемых элементов React преобразователь)**. Данный плагин ищет в коде все элементы, отрисовка которых не зависит от передаваемых им параметров, и выносит их из методов render (или функциональных компонентов), в которых они используются, чтобы уменьшить количество вызовов функции *createElement*.

Для того, чтобы начать им пользоваться, прежде всего необходимо его установить:

```
npm install --save-dev babel-plugin-transform-react-
    constant-elements
```

А затем нам необходимо добавить его в файл конфигурации Babel *.babelrc*:

```
{
  "plugins": ["transform-react-constant-elements"]
}
```

Еще один плагин, который вы потенциально можете захотеть поставить, - **React inline elements transform**. Он заменяет все объявления JSX (или вызовы *createElement*) на более производительные аналоги.

Для установки плагину необходимо выполнить:

```
npm install --save-dev babel-plugin-transform-react-inline-
    elements
```

А затем также добавить в файл конфигурации Babel:

```
{
  "plugins": ["transform-react-inline-elements"]
}
```

Оба плагина должны использовать только в финальный сборке, которая уйдет пользователям, так как изменения, которые они применяют к коду, могут значительно усложнить отладку приложения.

9.5 Заключение

На этом наш экскурс в вопросы производительности React приложений подходи к концу, и теперь мы можем оптимизировать наши продукты так, чтобы они оставляли лучшее впечатление у пользователей.

В этой главе мы разобрались в процессе согласования и как React ищет минимальное количество операций в DOM для применения всех изменений. А также мы посмотрели, как мы можем ему в этом помочь при помощи использования ключей.

Также всегда используйте финальную (production) сборку приложения для измерения производительности при помощи *Perf*. Это поможет вам точнее определить, какие части приложения необходимо оптимизировать.

После того, как вы нашли узкое горлышко, вы можете применить один из способов исправления проблем с производительностью, рассмотренных в этой главе. Например, вы можете использовать компонент

PureComponent и неизменяемы объекты для сокращения количества вызовов методов *render*.

Избегайте распространенных ошибок, связанных с использованием *PureCompoentn*, так как создание обработчиков событий внутри метода *render* или использования константных объектов.

Мы также немного переосмыслили подход к проектированию компонентов так, чтобы они показывали большую производительность. Но в общем и целом, как всегда, нужно стремиться сохранять компоненты небольшими и выполняющими одну, каждый свою, конкретную задачу.

И в конце мы поговорили о неизменяемых объектах и увидели, почему важно не вносить изменения в существующие объекты, чтобы не потерять пользу от использования методов *shouldComponentUpdate* и *shallowCompare*. И в конце мы посмотрели на несколько интересных инструментов, которые могу помочь вам с улучшением производительности приложений.

В следующий главе мы встретимся лицом к лицу с тестированием и отладкой.

Глава 10

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

10.1 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.

10.2 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 открепленный (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.

10.3 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 откреплленный 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.

10.4 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.

10.5 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.

10.6 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={() => {}} />
)
```

В результате мы получаем экземпляр объекта, у которого есть специальный метод *toJSON*. Вызовем этот метод следующей строкой:

```
const tree = component.toJSON()
```

Таким образом мы получаем json со снимком элемента, который будет использоваться Jest для будущих сравнений.

Если мы распечатаем *tree*, то получим примерно следующий результат:

```
{ type: 'input',
    props:
      { className: '',
        type: 'text',
        placeholder: undefined,
        autoFocus: 'true',
        value: '',
        onBlur: [Function],
        onChange: [Function],
        onKeyDown: [Function] },
    children: null }
```

И в конце выполняем проверку, что снимок соответствует сохраненному ранее:

```
expect(tree).toMatchSnapshot()
```

После первого запуска *npm test* снимок будет сохранен в директорию *__snapshots__*.

Каждый файл в этой директории содержит снимок компонента. Если мы заглянем внутрь одного из таких файлов, то найдем вполне читаемое представление React компонента:

```
exports[`test snapshots are awesome 1`] = `
<input
  autoFocus="true"
  className=""
  onBlur={[Function]}
  onChange={[Function]}
  onKeyDown={[Function]}
  placeholder={undefined}
  type="text"
  value="" />
`;
```

Теперь, если мы вернемся в тест и добавим параметр *editing* в компонент, а затем запустим тест снова, то получим следующий результат:

```
FAIL  ./TodoTextInput-snapshot.spec.js
        snapshots are awesome
        expect(value).toMatchSnapshot()
        Received value does not match stored snapshot 1.
```

```
- Snapshot
+ Received
@@ -1,8 +1,8 @@
 <input
    autoFocus="true"
-   className=""
+   className="edit"
    onBlur={[Function]}
    onChange={[Function]}
    onKeyDown={[Function]}
    placeholder={undefined}
    type="text"
```

Здесь мы можем увидеть, что текущий снимок отличается от сохраненного. В данном случае, в сохраненном снимке *className* был пустым, а теперь содержит строку *edit*.

Чуть ниже можно найти следующее сообщение:

```
Inspect your code changes or run with npm test -- -u to
   update them.
```

Снимками очень удобно пользоваться. Теперь, после обнаружения различий в сохраненном и полученном снимках, у вас есть два выбора: исправлять код или обновить сохраненные снимки, в последнем случае достаточно запустить команду:

```
npm test -- -u
```

Как вы можете видеть, тестирование с помощью снимков значительно упрощает и ускоряет процесс написания тестов, так как можно проверять сразу весь компонент.

10.7 Покрытие кода тестами

Есть множество причин для написания тестов, и некоторые из них мы уже обсудили в предыдущей части. Но главной причиной остается повышение надежности кодовой базы.

По этой причине, я очень скептически отношусь к различным подсчетами количества тестов, количества строк кода или процента покрытия тестами. Я рекомендую не концентрироваться на численных показателях тестов, а думать об их реальной пользе.

Однако, в некоторых случаях имеет смысл отслеживать численные показатели покрытия тестами. В больших проектах, это может помочь

находить файлы, которые протестированы недостаточно хорошо или не протестированы совсем.

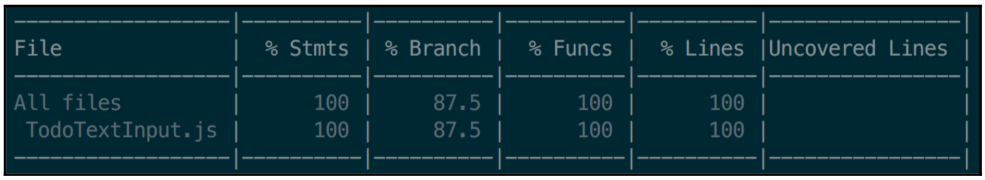
Как уже говорилось, Jest предоставляет все возможности для запуска тестов, и конечно он предоставляет возможность собирать статистику о покрытии тестами кода.

Jest использует Istanbul, одну из самых популярных библиотек для подсчета покрытия кода тестами (если вы используете Mocha, вы можете установить ее вручную).

Для того, чтобы собрать статистику покрытия кода тестами с помощью Jest, достаточно добавить флаг `--coverage` к *npm test*. Также есть вариант, добавить блок с конфигурацией в *package.json*, где установить флаг *collectCoverage* в *true*:

```
"jest": {
  "collectCoverage": true
}
```

Если вы запустите тесты теперь, то увидите в консоли таблицу со статистикой покрытия кода тестами.



Как вы видите, наш файл практически полностью покрыт тестами. В первой колонке процент покрытых выражений, во второй условных переходов, в третьей протестированных функций и в четвертой строчек кода. И в последней колонке должны быть перечислены номера строк кода, которые еще не покрыты тестами, но в нашем случае таких нет.

В наших тестах мы не достигли максимума только по критерию покрытия условных переходов. На самом деле я оставил один из них не протестированным специально, чтобы мы могли сделать это сейчас.

Если вы откроете файл *TodoTextInput.js* и посмотрите в функцию *onBlur*, вы найдете условный переход (хоть и без ветки *else*):

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

В случае, если это новая задача, функция *onSave* вызываться не должна.

А мы протестировали только тот случай, когда эта функция вызывается для существующей задачи. Но для того, чтобы убедиться, что все работает корректно, часто имеет смысл протестировать все возможные переходы.

Давайте откроем файл *TodoTextInput.spec.js* и добавим еще один тест:

```
test('does not fire onSave on blur if new', () => {
  const onSave = jest.fn()
  const wrapper = shallow(
    <TodoTextInput newTodo onSave={onSave} />
  )
  wrapper.simulate('blur')
  expect(onSave).not.toBeCalled()
})
```

Тест очень похож на предыдущий за тем исключением, что мы передаем параметр *newTodo* и проверяем, что функцию *onSave* не вызывается.

Теперь *npm test* должен показывать 100% покрытия по всем критериям.

10.8 Распространенные подходы к созданию тестов

В последней части главы о тестировании мы рассмотрим несколько распространенных шаблонов в тестировании React компонентов.

Вы уже познакомились с основами тестирования, однако иногда возникают ситуации, когда сложно понять, с какой стороны стоит начать тестирование кода. Это может произойти с **Компонентами высшего порядка (Higher-Order Components, HoC)**.

10.8.1 Тестирование компонентов высшего порядка

Как мы уже видели в предыдущих главах Компоненты высшего порядка предназначены для переиспользования кода между различными компонентами. HoC - это функция, которая принимает компонент и возвращает его с расширенным функционалом.

Тестирование таких компонентов может быть не так очевидно, как тестирование простых компонентов, поэтому имеет смысл разобрать пару распространенных решений вместе.

Попробуем протестировать HoC *withData*, который мы создали в Главе 5. Единственный нюанс, мы немного поправим способ загрузки данных.

Функция *withData* имеет следующую структуру:

```
const withData = URL => Component => (...)
```

Эта функция принимает url, по которому необходимо загрузить данные, и передает загруженные данные компоненту. URL может быть функцией, которая получает параметры компонента, или строкой.

Функция *withData* возвращает класс, определенный следующим способом:

```
class extends React.Component
```

В конструкторе создается заготовка для скачиваемых данных:

```
constructor(props) {
  super(props)

  this.state = { data: [] }
}
```

Данные загружаются в методе жизненного цикла *componentDidMount*:

```
componentDidMount() {
  const endpoint = typeof url === 'function'
    ? url(this.props)
    : url
  getJSON(endpoint).then(data => this.setState({ data }))
}
```

Как вы можете видеть, код немного отличается от того, что было в Главе 5: вместо использования *fetch* мы используем *getJSON*. Мы сделали это для того, чтобы вы увидели, как делать моки (mocks) для внешних модулей.

Это хорошая практика, оборачивать сторонние библиотеки и абстрактные API в отдельные модули, так как при тестировании это позволяет абстрагировать сам компонент от его зависимостей.

Функция *getJSON* импортируется в начале файла:

```
import getJSON from './get-json'
```

Эта функция возвращает промис (прим.пер. Promise; у меня рука не поднимется называть их обещ*ниями) на загрузку данных из сети.

И в методе *render* мы отрисовываем компонент, передавая в него все параметры и текущее состояние:

```
render() {
  return <Component {...this.props} {...this.state} />
}
```

Есть множество вещей, которые мы можем покрыть в данном случае тестами. Начнем с чего-то простого, например проверим, что параметры, полученные расширенным компонентом, корректно передаются целевому.

Затем мы можем проверить, что запрос отправляется на корректный адрес в обоих случаях: когда была передана строка с URL, или когда была передана функция для ее создания.

И самое главное, нужно проверить, что целевой компонент получает данные, которые возвращает функция *getJSON*.

Создадим файл для тестов с необходимыми зависимостями:

```
import React from 'react'
import { shallow, mount } from 'enzyme'
import withData from './with-data'
import getJSON from './get-json'
```

Мы импортируем и *shallow* и *mount* из Enzyme, так как для тестирования простых вещей нам не требуется полностью отрисованный DOM. Но как только мы собираемся использовать методы жизненного цикла компонентов, нам потребуется полная отрисовка.

Далее создадим пару вспомогательных переменных, которые пригодятся нам в тестах:

```
const data = 'data'
const List = () => <div />
```

Первую из них мы будем использовать для проверки, что загруженные данные передаются целевому компоненту, а вторая будет минималистичным вариантом целевого компонента.

Создание почти пустого компонента для тестирования HoC распространенная практика, так как нам требуется компонент, который мы будем расширять.

Далее следуем самая сложная часть данных тестов:

```
jest.mock('./get-json', () => (
  jest.fn(() => ({ then: callback => callback(data) }))
))
```

Как уже было сказано, мы используем внешний модуль для загрузки данных. Одна из вещей, которую мы хотим избежать, это загрузка реальных данных; помимо этого мы не хотим, чтобы наши тесты начали падать, если сломается какая-либо из внешних библиотек. К счастью с помощью Jest очень удобно изолировать и подменять внешние зависимости для конкретного теста.

При помощи *jest.mock* мы говорим Jest, что хотим заменить внешний модуль другой реализацией, которую должны передать вторым аргументом. В данном случае наша реализация содержит мок функцию *jest.fn*, которая возвращает объект похожий на промис, но синхронный. У этого объекта есть метод *then*, который получает функцию обратного вызова и вызывает ее с ранее созданными мок данными.

Начиная с этого момента мы можем тестировать наш HoC без опасений, что внешний модуль может повлиять на результат работы тестов.

Теперь мы можем приступить к созданию самих тестов. В первом из них проверим, что параметры, которые получает HoC, корректно передаются целевому компоненту:

```
test('passes the props to the component', () => {
  const ListWithGists = withData()(List)
  const username = 'gaearon'

  const wrapper = shallow(<ListWithGists username={username}
     />)

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

Все должно быть понятно, но давайте вместе посмотрим на этот код. Сначала, мы создаем компонент *ListWithGists* посредством расширения компонента *List* с HoC *withData*. Затем мы создаем переменную *username*, которую будем передавать как параметр компоненту *ListWithGists*, и выполняем поверхностную отрисовку.

И в конце мы проверяем, что у отрисованного элемента есть параметр *username* и он совпадает с заданным. Можно запустить *npm test* и убедиться, что тест показывает зеленый свет.

Дальше наши тесты будут требовать отрисовки компонента в DOM.

Сначала проверим, что передача строки с URL и функции для ее получения работают корректно. Начнем со статической строки:

```
test('uses the string url', () => {
  const url = 'https://api.github.com/users/gaearon/gists'
  const withGists = withData(url)
  const ListWithGists = withGists(List)

  mount(<ListWithGists />)

  expect(getJSON).toHaveBeenCalledWith(url)
})
```

В данном случае мы определили URL, а затем с помощью частичного применения создали новую функцию, которую дальше используем для расширения компонента *List*.

Затем мы монтируем данный компонент в DOM и проверяем, что функция *getJSON* была вызвана с заданным URL.

Теперь проверим, что работает вызов функции для получения url:

```
test('uses the function url', () => {
  const url = jest.fn(props => (
    `https://api.github.com/users/${props.username}/gists`
  )
  const withGists = withData(url)
  const ListWithGists = withGists(List)
  const props = { username: 'gaearon' }

  mount(<ListWithGists {...props} />)

  expect(url).toHaveBeenCalledWith(props)
  expect(getJSON).toHaveBeenCalledWith(
    'https://api.github.com/users/gaearon/gists'
  )
})
```

Сначала мы создаем мок функцию (что позволит проверить факт вызова), которая возвращает url. Создаем компонент *ListWithGists*, расширяющий компонент *List*, и параметры для этого компонента.

Затем мы делаем две проверки:

- В первой проверяем, что функция для создания url была вызвана с заданными аргументами
- И во второй проверяем, что функция *getJSON* была вызвана с правильным адрессом

И в последнем тесте проверим, что то, что возвращает функция *getJSON*, передается целевому компоненту:

```
test('passes the data to the component', () => {
  const ListWithGists = withData()(List)

  const wrapper = mount(<ListWithGists />)

  expect(wrapper.prop('data')).toEqual(data)
})
```

Как всегда создаем расширенную версию компонента *List* и монтируем его к DOM. Затем мы ищем компонент *List* в том, что отрисовалось (прим.пер. в коде выше этого нет, постараться не забыть, сверить с кодом на api.github.com), и проверяем, что он получил параметр *data* со значением, которое возвращает мок функция.

Если мы запустим *npm test*, то должны увидеть, что все 4 теста проходят успешно.

Вот мы и разобрались, как можно тестировать компоненты высшего порядка в Jest и как изолировать наши тесты от внешних зависимостей.

10.8.2 Паттерн Page Object

Давайте посмотрим на еще один распространенный способ тестирования компонентов, который пригодится в тех случаях, когда дерево компонентов становится сложнее и появляется множество вложенных дочерних элементов.

В этих целях воспользуемся формой (Controlled form), созданной в Главе 6Ж

```
class Controlled extends React.Component
```

Для начала вспомним, что этот компонент из себя представляет, а затем поговорим о его тестировании.

В конструкторе мы инициализируем начальное состояние и привязываем обработчики событий к экземпляру класса:

```
constructor(props) {
  super(props)

  this.state = {
    firstName: 'Dan',
    lastName: 'Abramov',
```

```
  }

  this.handleChange = this.handleChange.bind(this)
  this.handleSubmit = this.handleSubmit.bind(this)
}
```

Обработчик *handleChange* обновляет состояние при вводе данных:

```
handleChange({ target }) {
  this.setState({
    [target.name]: target.value,
  })
}
```

Далее идет обработчик *handleSubmit*, в котором сначала вызывается функция *preventDefault* (чтобы браузер не выполнял никаких действий по отправке формы), а затем вызывается функция из параметров *onSubmit*, в которую передаются введенные данные.

Этой функции не было в предыдущей реализации, но мы добавим ее, чтобы показать, как правильно тестировать такой компонент:

```
handleSubmit(e) {
  e.preventDefault()

  this.props.onSubmit(
    `${this.state.firstName} ${this.state.lastName}`
  )
}
```

И в конце идет метод *render*, в котором отрисовываются все поля ввода с нужными обработчиками событий:

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <input
        type="text"
        name="firstName"
        value={this.state.firstName}
        onChange={this.handleChange}
      />
      <input
        type="text"
        name="lastName"
        value={this.state.lastName}
        onChange={this.handleChange}
      />
```

```
        <button>Submit</button>
      </form>
    )
}
```

Основной сценарий, который видится логичным для тестирования, это ввод каких-либо данных в форму, отправка (submit) формы и проверка, что функция *onSumbit* вызвалась с нужными параметрами.

Вы уже должны представлять, как реализовать такой тестовый сценарий с помощью Enzyme, поэтому давайте быстро пройдем по реализации:

```
test('submits the form', () => {
```

Прежде всего создадим мок функцию *onSubmit*. Затем смонтируем компонент и получим ссылку на его обертку:

```
const onSubmit = jest.fn()
const wrapper = shallow(<Controlled onSubmit={onSubmit} />)
```

Далее с помощью полученной ссылки найдем поле ввода и вызовем для него *change* событие, для обновления его значения:

```
const firstName = wrapper.find('[name="firstName"]')
firstName.simulate(
  'change',
  { target: { name: 'firstName', value: 'Christopher' } }
)
```

Проделаем то же самое со вторым полем ввода:

```
const lastName = wrapper.find('[name="lastName"]')
lastName.simulate(
  'change',
  { target: { name: 'lastName', value: 'Chedeau' } }
)
```

После обновления полей ввода можно вызвать событие *submit* на саму форму:

```
const form = wrapper.find('form')
form.simulate('submit', { preventDefault: () => {} })
```

И затем проверить, что мок функция была вызвана с нужным значением:

```
expect(onSubmit).toHaveBeenCalledWith('Christopher Chedeau')
```

И не забудем закрыть блок с тестом:

```
})
```

Можно запустить *npm test* и убедится, что он проходит. В общем и целом тест работает, но если мы посмотрим на него чуть пристальнее, то сможем обнаружить потенциальные проблемы.

Прежде всего бросается в глаза то, что код для заполнения полей в значительной степени дублируется, код очень многословный и сильно завязан на конкретную разметку.

В случае, когда у нас есть множество тестов, изменение разметки приведет к необходимости менять множество участков тестирующего кода. Будет здорово, если мы избавимся от дублирующегося кода и вынесем логику поиска полей ввода в отдельное место.

Собственно, тут и приходит на помощь паттерн **Page Object**. Основная идея которого заключается в создании специального класса *Page*, который будет инкапсулировать логику по выборке и заполнению полей отрисованного компонента.

Стоит отметить, что в большинстве случаев следование правилу **Не повторяй себя (Don't Repeat Yourself, DRY)** при создании тестов является не самой хорошей практикой, так как появляется риск добавления излишней сложности, но в данном случае он того стоит.

Посмотрим, как наш тест может быть улучшен при помощи данного паттерна.

Прежде всего нам нужно создать класс *Page*:

```
class Page
```

В конструкторе мы будем получать *wrapper* из Enzyme для сохранения и дальнейшего использования:

```
constructor(wrapper) {
  this.wrapper = wrapper
}
```

Затем мы создадим обобщенную функцию *fill*, которая будет принимать название поля с новым значение и вызывать соответствующее *change* событие:

```
fill(name, value) {
  const field = this.wrapper.find(`[name="${name}"]`)
  field.simulate('change', { target: { name, value } })
}
```

Также создадим метод *submit*, в который вынесем логику по поиску формы и вызова одноименного события:

```
submit() {
  const form = this.wrapper.find('form')
  form.simulate('submit', { preventDefault() {} })
}
```

Теперь мы можем переписать тест следующим образом:

```
test('submits the form with the page object', () => {
  const onSubmit = jest.fn()
  const wrapper = shallow(<Controlled onSubmit={onSubmit} />)

  const page = new Page(wrapper)

  page.fill('firstName', 'Christopher')
  page.fill('lastName', 'Chedeau')
  page.submit()

  expect(onSubmit).toHaveBeenCalledWith('Christopher Chedeau'
    )
})
```

Как вы можете увидеть, мы создали экземпляр класса *Page* и используем его для заполнения и отправки формы.

С паттерном *Page Object* код выглядит значительно чище. Теперь, если что-либо поменяется в коде самого компонента, скорее всего нам будет достаточно поправить только класс *Page* в соответствии с этими изменениями.

10.9 React Dev Tools

Если тестов становится недостаточно, и появляется желание проверить приложение во время его работы в браузере, то можно воспользоваться React Developer Tools.

Вы можете установить как расширение для Chrome, для чего можно воспользоваться следующей ссылкой:

> https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en

После установки расширения в Chrome Dev Tools появится новая вкладка **React**, где вы сможете проверить, какие компоненты были отрисованы, какие параметры они получили и какой состояние в каждом из них на данный момент.

Параметры и состояние можно не только посмотреть, но и отредактировать, после чего UI будет обновлен. Это позволяет очень быстро проверять, как компоненты ведут себя при получении определенных параметров.

В общем и целом этот инструмент уже обязателен к использованию, но в последних версиях появилась еще одна полезная возможность, для включения которой нужно выбрать флажок **Trace React Updates**.

Когда эта возможность включена, мы можем видеть, какие именно компоненты обновляются, когда происходит какое-то событие. Обновленные компоненты подсвечиваются цветными прямоугольниками, что может помочь при поиске участков подлежащих оптимизации.

10.10 Обработка ошибок с React

Даже если мы пишем превосходный код и покрываем его тестами, ошибки все равно будут возникать. Различные браузеры и окружения, реальные данные пользователей, слишком много параметров которые мы не можем контролировать, поэтому время от времени наш код все равно будет падать. Как разработчики, мы должны просто принять этот факт и смириться с ним.

Лучше, что мы можем сделать, когда ошибка в коде возникает, это:

- Сообщить об этом пользователям, объяснить, что произошло и что они должны делать
- Собрать полезную информацию об ошибке и состоянии приложения, чтобы воспроизвести ее и исправить

Обработка ошибок в React на первый взгляд может показаться контринтуитивным.

Предположим, что у нас есть два компонента. Первый:

```
const Nice => <div>Nice</div>
```

И второй:

```
const Evil => (
  <div>
    Evil
    {this.does.not.exist}
  </div>
)
```

При отрисовке следующего компонента *App* в DOM мы можем ожидать различных вещей:

```
const App = () => (
  <div>
    <Nice />
    <Evil />
    <Nice />
  </div>
)
```

Например, мы можем ожидать, что один компонент *Nice* будет отрисован, а затем отрисовка прекратится из-за ошибки в *Evil*. Или мы можем ожидать, что оба компонента *Nice* будут отрисованы, а компонент *Evil* - нет. Но на самом деле на экране не будет отрисовано ничего.

В React, если какой-то из компонентов выбросил ошибку, будет остановлена отрисовка всего дерева. Это было сделано в целях безопасности и сохранения консистентности данных.

Будет ли это здорово, иметь возможность отрисовывать дерево компонентов даже при падении какой-либо из его внутренних частей? По сути единственный способ добиться этого, это вручную оборачивать методы *render* в блок *try...catch*. В общем случае это очень плохая практика и нужно ее избегать; но она может быть полезна в определенных случаях для тестирования.

Есть библиотека, которая называется *react-component-errors*, которая оборачивает все методы компонента в блок *try...catch*, что позволяет им не ломать отрисовку приложения при падении.

Такой подход в значительной степени сказывается на производительности приложения, но если вы осознанно идете на этот риск, вы можете попробовать эту библиотеку.

Установить библиотеку можно при помощи команды:

```
npm install --save react-component-errors
```

Далее необходимо импортировать ее в файле с компонентом:

```
import wrapReactLifecycleMethods from 'react-component-errors
    '
```

И затем мы можем добавить декоратор к классу следующим образом:

```
@wrapReactLifecycleMethods
class MyComponents extends React.Component
```

Библиотека не только предотвращает остановку отрисовки всего приложения при падении одного из компонентов, но также предоставляет возможность создания собственного обработчика ошибок, чтобы иметь возможность получить полезную информацию, когда ошибка уже произошла.

Для этого нам необходимо импортировать объект *config* из библиотеки:

```
import { config } from 'react-component-errors'
```

А затем определить собственны обработчик событий следующим образом:

```
config.errorHandler = errorReport => { ... }
```

Функция, которую мы определили как *errorHandler*, получает отчет об ошибке с полезной информацией, которую мы можем использовать для воспроизведения и исправления ошибки.

Отчет об ошибке, помимо всего прочего содержит название компонента и название функции, в которой произошла ошибка. Также в этом отчете приходят все параметры, которые были получены компонентом. Эта информация может очень сильно упростить воспроизведение ошибки.

Но еще раз напомним, что в общем случае следует избегать такого подхода, так как он ведет к множеству проблем в работе приложения. Как минимум, такое поведение приложения не должно быть включено для реальных пользователей.

10.11 Заключение

В этой главе мы поговорили о пользе тестирования и фреймворках, которые вы можете в этих целях использовать. Jest уже содержит все, что вам нужно для создания тестов. Если же вы хотите настроить тестовый фреймворк под специфичные нужды, вы можете использовать Mocha.

С помощью *TestUtils* вы можете отрисовывать компоненты вне браузера, а Enzyme позволяет получить удобный доступ к результатам отрисовки внутри тестов. Также мы разобрались, как использовать в тестах моки и выполнять проверки.

Мы научились создавать тесты на основе снимков компонентов (Snapshot Testing), а также проверять процент покрытия кодовой базы тестами.

Также важно держать в голове распространенные решения для тестирования сложных компонентов, таких как компоненты высшего порядка или формы.

И в конце мы посмотрели, как React Developer Tools помогает в отладке, и как работать с обработкой ошибок с React.