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.

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.

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.

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.

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.

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>
```

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.

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);
```

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} />
```

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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} />
  }
}
```

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.

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.

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)
  }
)
```

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.

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.

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.

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).

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 HoC. Это распространенная практика начинать название с *with*, если HoC 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.

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>
  )
}
```

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.

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.