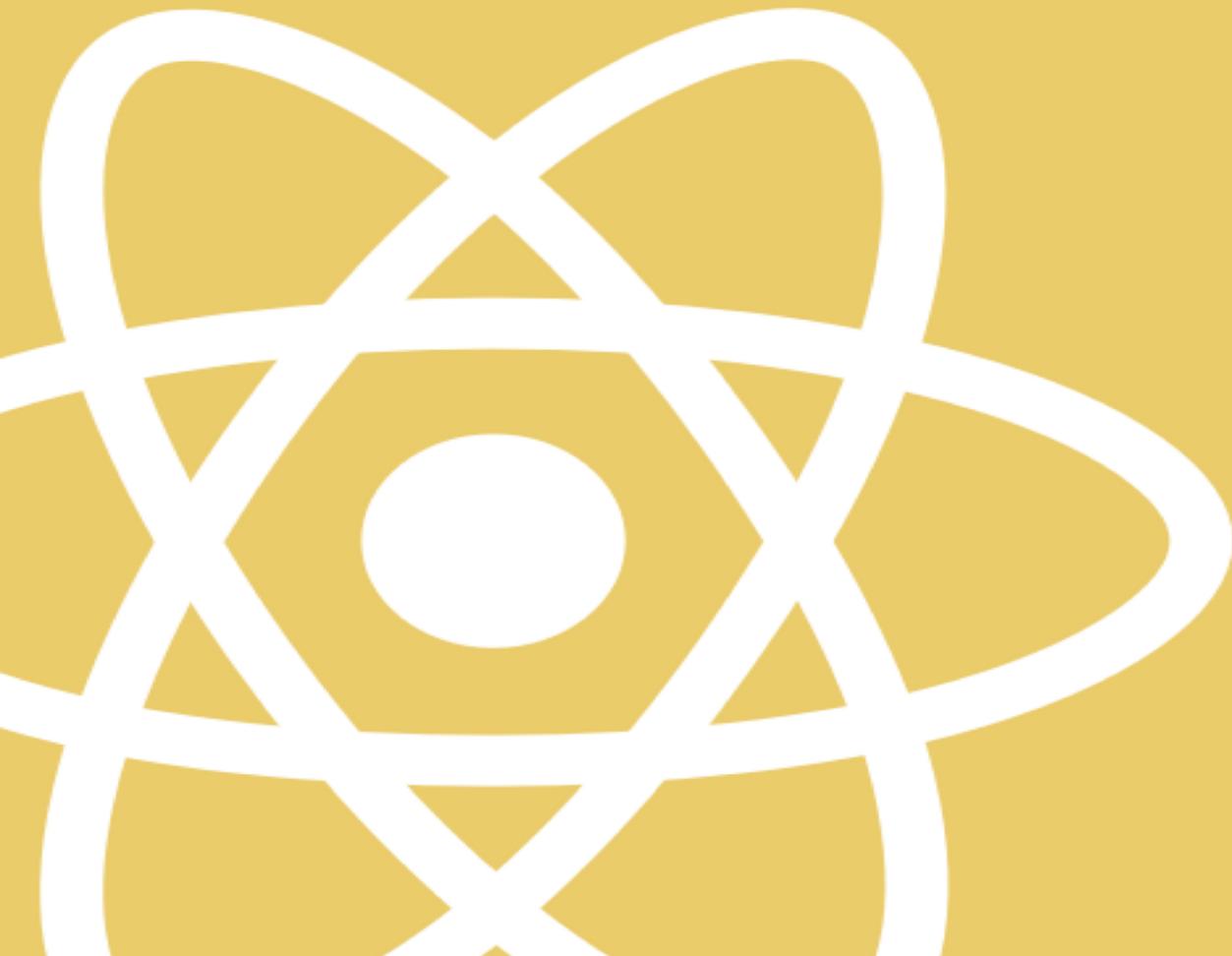


by robin wieruch

Путь к изучению React



Путь к изучению React

Ваше путешествие по освоению обычного, но прагматичного React

Robin Wieruch и Alexey Pyltsyn

Эта книга предназначена для продажи на <http://leanpub.com/the-road-to-learn-react-russian>

Эта версия была опубликована на 2018-09-08



Это книга с [Leanpub book](#). Leanpub позволяет авторам и издателям участвовать в так называемом [Lean Publishing](#) - процессе, при котором электронная книга становится доступна читателям ещё до её завершения. Это помогает собрать отзывы и пожелания для скорейшего улучшения книги. Мы призываем авторов публиковать свои работы как можно раньше и чаще, постепенно улучшая качество и объём материала. Тем более, что с нашими удобными инструментами этот процесс превращается в удовольствие.

© 2018 Robin Wieruch и Alexey Pyltsyn

ТВИТНИТЕ ЭТУ КНИГУ!

Пожалуйста, помогите Robin Wieruch и Alexey Pyltsyn в распространении информации о книге в [Twitter](#)!

Предлагаемый твит для этой книги:

Я собираюсь изучать #ReactJs вместе с «Путь к изучению React» от @rwieruch
Присоединяйтесь к моему путешествию ➡ <https://roadtoreact.com>

Предлагаемый хештег для этой книги [#ReactJs](#).

Узнайте, что другие люди пишут об этой книге, кликнув на ссылку на страницу поиска по этому хештегу в Twitter:

[#ReactJs](#)

Оглавление

Предисловие	i
Об авторе	ii
Отзывы	iii
Образование для детей	v
Часто задаваемые вопросы	vi
Журнал изменений	viii
Как читать книгу?	x
Вызов	xi
Об этом переводе	xiii
Небольшая предыстория	xiv
Немного о переводчике	xv
Немного о редакторе	xvi
Соглашения, принятые в переводе	xvii
В добрый путь	xviii
Введение в React	1
Привет, меня зовут React	2
Требования	4
node и npm	6
Установка	8
Установка без конфигурации	10
Введение в JSX	14
const и let в ES6	17
ReactDOM	19
Горячая перезагрузка	21
Комплексный JavaScript в JSX	23
Стрелочные функции ES6	27
Классы ES6	29

ОГЛАВЛЕНИЕ

Основы React	33
Внутреннее состояние компонента	34
Инициализация объектов ES6	37
Однонаправленный поток данных	39
Привязки	44
Обработчик событий	49
Взаимодействия с формами и событиями	54
Деструктуризация ES6	62
Контролируемые компоненты	65
Разделение компонентов	67
Компонуемые компоненты	71
Повторно используемые компоненты	73
Объявления компонентов	76
Стилизация компонентов	80
Получение реальных данных с API	87
Методы жизненного цикла	88
Получение данных	91
Оператор расширения ES6	96
Отрисовка по условию	100
Поиск на стороне клиента и на стороне сервера	103
Получение данных с разбивкой на страницы	108
Кеш клиента	113
Обработка ошибок	120
Использование Axios вместо Fetch	124
Организация и тестирование кода	130
Модули ES6: импорт и экспорт	131
Организация кода с помощью модулей в ES6	135
Тестирование снимками с помощью Jest	140
Модульное тестирование с помощью Enzyme	147
Интерфейс компонента с помощью PropTypes	150
Отладка с помощью инструментов разработчика React	155
Продвинутые React-компоненты	158
Ссылка на DOM-элемент	159
Загрузка	163
Компоненты высшего порядка	167
Продвинутая сортировка	172
Управление состоянием в React и за его пределами	185
Подъём состояния	186
Пересмотр: setState()	193
Укрощение состояния	198

ОГЛАВЛЕНИЕ

Заключительные шаги к развертыванию в продакшене	201
Команда eject	202
Деплой приложения	203
Краткий обзор	204

Предисловие

Путь к изучению React (*The Road to learn React*) научит вас основам React. В ходе изучения вы создадите реальное приложение, используя обычный React без сложного инструментария. Вам будет объяснено всё, начиная от настройки проекта и заканчивая развертыванием его на сервере. Книга содержит дополнительные ссылки для изучения и упражнения в конце каждого раздела глав. После прочтения книги вы сможете создавать собственные приложения на React. Материал книги постоянно обновляется мною и сообществом.

В книге *Путь к изучению React* я хочу изложить основы, прежде чем вы начнёте погружение в обширную экосистему React. В книге мало внимания уделяется инструментам и внешнему управлению состоянием, но зато есть много информации о React. Она объясняет общие концепции, типовые решения и лучшие практики, используемые при разработке React-приложений для работы в реальном мире.

Вы научитесь создавать собственное приложение на React, которое содержит такие возможности, как постраничная навигация, кеширование на стороне клиента и такие пользовательские взаимодействия, как поиск и сортировка. Кроме того, в процессе чтения книги вы перейдёте с JavaScript ES5 на JavaScript ES6. Я надеюсь, что эта книга отразит мой энтузиазм по поводу React и JavaScript и поможет вам начать их изучение.

Об авторе

Робин Вирух (Robin Wieruch) — немецкий разработчик программного обеспечения и веб-приложений, который занимается обучением и преподаванием программирования JavaScript. После получения степени магистра в области информатики, он продолжает учиться каждый день. Его опыт в мире стартапа, где он много использовал JavaScript во время работы и свободного времени. Это дало ему возможность учить других людей этим темам.

В течение нескольких лет Робин тесно сотрудничал с большой командой инженеров в компании [Small improvements¹](#) над работой крупномасштабного приложения. Компания создаёт SaaS-продукт, позволяющий клиентам создавать культуру обратной связи в своей компании. Под капотом приложение на фронтенде работало на JavaScript, в качестве бекенда использовался язык Java. На фронтенде первая итерация была написана на Java с использованием Wicket Framework и jQuery. Когда первое поколение SPA стало популярным, компания перешла на Angular 1.x для фронтенда приложения. После использования Angular в течение более двух лет стало ясно, что Angular не лучшее решение для интенсивного взаимодействия с состоянием в те дни. Именно поэтому компания сделала окончательный переход на React и Redux, что позволило приложению успешно работать в больших масштабах.

Во время своей работы в компании Робин регулярно писал статьи о веб-разработке на своём сайте. Он получал отличные отзывы от людей о своих статьях, что в конечном счёте позволило улучшить его стиль письма и обучения. Статья за статьёй, Робин вырос в своей способности учить других. В его первой статье было слишком много лишнего, что усложнило её понимание для студентов, но со временем он улучшил эту статью, сосредоточив внимание на преподавании только одной темы.

В настоящее время Робин занимается собственным делом, обучая других людей. Он считает своей полноценной деятельностью видеть, как его студенты процветают, давая им чёткие цели и короткий цикл обратной связи. Это то, чему бы вы научились, работая в компании, предоставляющей обратную связь, не так ли? Но без достаточной практики, он не смог бы обучать других. Именно поэтому он уделяет своё оставшееся время на программирование. Вы можете найти более подробную информацию о Робине и о способах его поддержки и работы с ним на его [сайте²](#).

¹<https://www.small-improvements.com/>

²<https://www.robinwieruch.de/about>

Отзы́вы

Есть много [отзывов³](#), [оценок⁴](#) и [отзывов⁵](#) о книге, подтверждающие её качество. Я так горжусь этим, потому что я никогда не ожидал такой мощной обратной связи. Если вам понравится данная книга, я бы с удовольствием хотел получить вашу оценку или обзор. Это помогает мне рассказать о книге. Ниже приведён краткий отрывок из этих хороших отзывов:

Мухаммад Кашиф (Muhammad Kashif)⁶: «”Путь к изучению React” – это уникальная книга, которую я рекомендую любому студенту или профессиональному, интересующемуся обучением основам React до продвинутого уровня. Она наполнена содержательными советами и методами, которые трудно найти в другом месте, и замечательное использование примеров и ссылок на примеры проблем, у меня есть 17-летний опыт разработки веб-приложений и десктопных приложений, и до чтения этой книги у меня были проблемы с обучением React, но эта книга действует как магия».

Андре Варгас (Andre Vargas)⁷: «”Путь к изучению React” Робина Вируха – такая потрясающая книга! Большинство из того, что я узнал о React и даже ES6, было получено с помощью неё!»

Николас Хант-Уокер (Nicholas Hunt-Walker), инструктор Python в школе программирования в Сиэтле⁸: «Это одна из самых хорошо написанных и содержательных книг по программированию, с которой я когда-либо работал. Твёрдое введение в React и ES6».

Остин Грин (Austin Green)⁹: «Спасибо, очень понравилась книга. Идеальное сочетание для изучения React, ES6 и концепций программирования высшего уровня».

Николь Фергюсон (Nicole Ferguson)¹⁰ : «В эти выходные я прохожу курс обучения Робина Вируха, и я почти чувствую вину за то, что так много веселья».

Каран (Karan)¹¹: «Закончил чтение “Путь к изучению React”. Лучшая книга для новичка в мире React и JS. Элегантное ознакомление с ES. Респект! :)»

Эрик Приоу (Eric Priou)¹²: «”Путь к изучению React” Робина Вируха – обязательное чтение. Чистота и краткость для изучения React и JavaScript».

Начинающий разработчик: «Я только что закончил книгу как неопытный разработчик, спасибо за работу над ней. Мне было легко изучать её, и я уверен, что в ближайшие дни

³<https://roadtoreact.com/>

⁴<https://www.goodreads.com/book/show/37503118-the-road-to-learn-react>

⁵<https://www.amazon.com/dp/B077HJFCQX>

⁶<https://twitter.com/appsdevpk/status/848625244956901376>

⁷<https://twitter.com/andrevar66/status/853789166987038720>

⁸<https://twitter.com/nhuntwalker/status/845730837823840256>

⁹<https://twitter.com/AustinGreen/status/845321540627521536>

¹⁰<https://twitter.com/nicoleffe/status/833488391148822528>

¹¹<https://twitter.com/kvss1992/status/889197346344493056>

¹²<https://twitter.com/erixtekila/status/840875459730657283>

я начну разработку нового приложения с нуля. Книга была намного лучше официального учебного введения React.js, который я пробовал раньше (и не мог закончить из-за отсутствия подробностей). Упражнения в каждой главе были очень полезными».

Студент: «Лучшая книга, чтобы начать изучать ReactJS. Разработка проекта движется вместе с изучаемыми концепциями, которые помогают понять предмет. Я нашёл “Code and learn” как лучший способ освоить программирование, и эта книга точно следует этому.

Томас Локни (Thomas Lockney)¹³: «Довольно солидное введение в React, которое не пытается быть всеобъемлющим. Я просто хотел понять, что это такое, и данная книга дала мне именно это. Я не следил за всеми маленькими сносками, чтобы узнать о новых возможностях ES6, которые я пропустил (“Я бы не сказал, что я скучал по этому, Боб” — [отсылка¹⁴](#) из сериала “Офисное пространство”). Но я уверен, что те из вас, кто останавливался и прилежно следил за ними, вероятно, узнают намного больше, чем то, чему учит книга.».

¹³<https://www.goodreads.com/review/show/1880673388>

¹⁴<http://quotegeek.com/quotes-from-movies/office-space/4848/>

Образование для детей

Эта книга должна позволить каждому изучить React. Однако не все имеют привилегии использовать эти ресурсы, потому что не все получают образование на английском языке в первую очередь. Таким образом, я хочу использовать данный проект для поддержки проектов, которые учат детей английскому языку в развивающихся странах.

- 1. April to 18. April, 2017, [Giving Back, By Learning React¹⁵](https://www.robinwieruch.de/giving-back-by-learning-react/)

¹⁵<https://www.robinwieruch.de/giving-back-by-learning-react/>

Часто задаваемые вопросы

Как получать обновления У меня есть два канала связи, где я делаюсь обновлениями о моём контенте. Вы можете [подписаться на обновления по электронной почте](#)¹⁶ или [следовать за мной в Twitter](#)¹⁷. Независимо от канала, моя цель состоит в том, чтобы делиться только качественным контентом. Вы никогда не получите спама. Как только вы получите обновление об изменении книги, вы можете загрузить её новую версию.

Используется ли последняя версия React? Книга всегда актуализируется при обновлении версии React. Обычно книги устаревают вскоре после их выхода. Поскольку эта самоизданная книга, я могу обновлять её, когда захочу.

Рассматривается ли Redux? Нет. Поэтому я написал вторую книгу. «Путь к изучению React» должна дать вам прочную основу, прежде чем вы погрузитесь в продвинутые темы. Реализация примера приложения в книге покажет, что Redux не требуется в создании приложения в React. После того, как вы прочтёте книгу, вы сможете реализовать надёжное приложение без Redux. Затем вы можете прочитать мою вторую книгу, чтобы узнать [Redux](#)¹⁸.

Используется ли JavaScript ES6? Да. Но не волнуйтесь. Всё будет хорошо, если вы знакомы только с JavaScript ES5. Все возможности JavaScript ES6, которые я описываю во время изучения React, я иллюстрирую примерами на ES5 в книге. Объясняется каждая особенность по ходу дела. Эта книга не только учит React, но и всем полезным возможностям JavaScript ES6 в контексте React.

Как получить доступ к исходному коду проектов и сериям скринкастов? Если вы купили один из расширенных пакетов, который даёт вам доступ к исходному коду проектов, серии скринкастов или любой другой пакет, вы должны найти их в своей [панели курсов](#)¹⁹. Если вы купили курс где-то ещё, отличной от [официальной «Road to React»](#)²⁰, вам необходимо создать учётную запись на данной платформе, перейти на страницу администратора и обратиться ко мне с одним из шаблонов электронной почты. После этого я могу зачислить вас на курс. Если вы не купили один из расширенных пакетов, вы можете в любое время связаться с ним для обновления, чтобы получить доступ к исходному коду проектов и сериям скринкастов.

Как я могу получить помощь во время чтения книги? У книги есть [канал в Slack](#)²¹ для людей, которые читают её. Вы можете присоединиться к каналу, чтобы получить помощь или помочь другим. В конце концов, помочь другим может улучшить ваши знания.

¹⁶<https://www.getrevue.co/profile/rwieruch>

¹⁷<https://twitter.com/rwieruch>

¹⁸https://roadtoreact.com/course-details?courseId=TAMING_THE_STATE

¹⁹<https://roadtoreact.com/my-courses>

²⁰<https://roadtoreact.com>

²¹<https://slack-the-road-to-learn-react.wieruch.com/>

Есть ли площадка для решения проблем? Если у вас возникли проблемы, присоединитесь к каналу Slack. Кроме того, вы можете посмотреть на [открытые ишью на GitHub²²](#). Возможно, ваша проблема уже встречалась, и вы можете найти её решение. Если ваша проблема не была упомянута, не стесняйтесь открывать новую проблему, где вы можете объяснить её суть, необязательно приложить скриншот и предоставить другие подробности (например, страница книги, версия node). В конце концов, я пытаюсь внести все исправления в следующие выпуски книги.

Есть ли гарантия возврата денег? Да, есть 100% гарантия возврата денег в течение двух месяцев, если вы не считаете, что курс обучения вам подходит. Просьба связаться со мной, чтобы получить возврат средств.

Как поддержать проект? Если вам нравится мой контент, вы можете [поддержать меня²³](#). Кроме того, я был бы признателен, если бы вы распространяли информацию об этой книге после того, как прочитаете и полюбите её. А ещё я с удовольствием хотел бы видеть вас моим патроном в Patreon²⁴.

Какова ваша мотивация для написания книги? Я хочу учить данной теме в единообразной форме. Вы часто находите материал в интернете, который больше не обновляется с тех пор, как был опубликован, или обучает только небольшой части темы. Когда вы узнаете что-то новое, люди изо всех сил пытаются найти последовательные и актуальные ресурсы для учёбы. Я хочу дать вам этот систематический и современный опыт обучения. Кроме того, я надеюсь, что смогу поддержать меньшинства в своих проектах, предоставив им контент бесплатно или [оказывая другого рода воздействия²⁵](#). Кроме того, в последнее время я обнаружил, что доволен собой, когда обучаю других программированию. Для меня это важная деятельность, поэтому я предпочитаю её любой другой работе от 9 до 5 в любой компании. Вот почему я надеюсь продолжить этот путь в будущем.

Есть призыв к действию? Да. Я хочу, чтобы вы на мгновение подумали о человеке, который хорошо бы разбирался в React. Такой человек мог бы уже показать свой интерес, а может быть, в середине изучения React или, возможно, ещё изъявил о желании узнать React. Познакомьтесь с этим человеком и поделитесь книгой. Это много значило бы для меня. Эта книга предназначена для других.

²²<https://github.com/rwieruch/the-road-to-learn-react/issues>

²³<https://www.robinwieruch.de/about/>

²⁴<https://www.patreon.com/rwieruch>

²⁵<https://www.robinwieruch.de/giving-back-by-learning-react/>

Журнал изменений

10 января 2017:

- пулреквест со второй версией книги²⁶
- более дружелюбный для начинающих
- На 37% больше контента
- 30% улучшенного контента
- 13 улучшенных и новых тем
- 140 страниц учебного материала
- + интерактивный курс книги на educative.io²⁷

8 марта 2017:

- пулреквест с третьей версией книги²⁸
- На 20% больше контента
- 25% улучшенного контента
- 9 новых тем
- 170 страниц учебного материала

15 апреля 2017:

- обновление до React 15.5

5 июля 2017:

- обновление до node 8.1.3
- обновление до npm 5.0.4
- обновление до create-react-app 1.3.3

17 октября 2017:

- обновление до node 8.3.0
- обновление до npm 5.5.1
- обновление до create-react-app 1.4.1

²⁶<https://github.com/rwieruch/the-road-to-learn-react/pull/18>

²⁷<https://www.educative.io/collection/5740745361195008/5676830073815040>

²⁸<https://github.com/rwieruch/the-road-to-learn-react/pull/34>

- обновление до React 16
- [пулреквест с четвёртой версией книги²⁹](#)
- На 15% больше контента
- 15% улучшенного контента
- 3 новые темы (Привязки, Обработки событий, Обработка ошибок)
- 190+ страниц учебного материала
- [более 9 проектов с исходным кодом³⁰](#)

17 февраля 2018:

- обновление до node 8.9.4
- обновление до прм 5.6.0
- обновление до create-react-app 1.5.1
- [пулреквест с пятой версией книги³¹](#)
- больше путей для обучения
- дополнительный материал для чтения
- 1 новая тема (Axios вместо Fetch)

²⁹<https://github.com/rwieruch/the-road-to-learn-react/pull/72>

³⁰https://roadtoreact.com/course-details?courseId=THE_ROAD_TO_LEARN.REACT

³¹<https://github.com/the-road-to-learn-react/the-road-to-learn-react/pull/105>

Как читать книгу?

Книга — это моя попытка научить React, пока вы будете писать приложение. Это практическое руководство по изучению React, а не справочник по React. Вы напишите приложение Hacker News, которое взаимодействует с реальным API. Среди нескольких интересных тем, эта книга охватывает управление состоянием в React, кеширование и взаимодействие (сортировка и поиск). В процессе вы узнаете лучшие практики и шаблоны в React.

Кроме того, в книге представлен переход от использования JavaScript ES5 в пользу JavaScript ES6. React охватывает множество возможностей JavaScript ES6, и я хочу показать вам, как вы можете их использовать.

В целом, каждая глава книги будет основываться на предыдущей главе. Каждая глава научит вас чему-то новому. Не торопитесь изучать материал книги. Лучше пошагово усваивать новый материал. Вы можете использовать свои собственные реализации и узнать больше о теме. В конце каждой главы я даю дополнительные ресурсы для чтения и упражнения. Если вы действительно хотите изучить React, я настоятельно рекомендую прочитать дополнительные ресурсы для изучения и попрактиковаться на упражнениях. После того, как вы прочтёте главу, возьмите паузу в обучении, прежде чем снова продолжить.

В конце книги, у вас будет законченное приложение на React, которое можно использовать в продакшене. Я очень хочу увидеть ваши результаты, поэтому, пожалуйста, напишите мне, когда закончите изучение книги. Последняя глава книги предоставит вам несколько вариантов продолжения вашего путешествия по React. В общем, вы найдёте много соответствующих тем, связанных с React на [моём личном сайте](#)³².

Так как вы читаете книгу, я думаю, вы новичок в React. Это прекрасно. В конце концов, я надеюсь получить ваши отзывы для дальнейшего улучшения материала, чтобы каждый мог изучить React. Вы можете оказать непосредственное участие на [GitHub](#)³³ или написать мне в [Twitter](#)³⁴.

³²<https://www.robinwieruch.de/>

³³<https://github.com/rwieruch/the-road-to-learn-react>

³⁴<https://twitter.com/rwieruch>

ВЫЗОВ

Лично я много пишу о своем обучении. Вот как я стал тем, кто я есть. Вы преподаете тему во всей красе, только когда сами изучили её. Поскольку преподавание помогло мне в моей карьере, я хочу, чтобы вы испытали тот же эффект. Но сначала вам следует сформировать привычку учиться. Мой вызов для этой книги заключается в следующем: учите других тому, чему вы учитесь, читая данную книгу. Несколько советов, как вы могли бы достичь этого:

- Напишите в блоге о конкретной теме из книги. Речь идет не о копировании и вставке материала, найдите собственные слова, чтобы объяснить рассматриваемую тему, а затем берите новую проблему и решайте её, и погружайтесь еще больше в тему, пока не поймете каждую деталь. Затем научите этому других посредством этой статьи. Вы увидите, как заполнились знания в ваших пробелах и как написание обучающих статей открывает двери для вашей карьеры в долгосрочной перспективе.
- Если вы активны в социальных сетях, то поделитесь со своими друзьями тем, что узнали во время чтения книги. Например, вы можете написать в Твиттере о своем последнем уроке из книги, который может быть интересен и для других. Просто сделайте скриншот отрывка из книги или даже лучше — напишите об этом своими словами. Вот как вы можете начать обучать других, не вкладывая много времени.
- Если вы чувствуете себя уверенно для записи вашего процесса обучения во время чтения, поделитесь этим с другими, используя Facebook Live, YouTube Live или Twitch. Это поможет вам оставаться сосредоточенным и прокладывать себе путь через книгу. Даже если, что у вас не так много людей во время прямой трансляции, вы всегда можете позже загрузить запись на YouTube. Кроме того, это отличный способ рассказать о своих проблемах и как вы собираетесь их решить.

Мне бы очень хотелось, чтобы люди занимались последним пунктом: записывали себя во время чтения этой книги, во время разработки своих приложений и выполнения упражнений и выкладывали окончательную версию видеоролика на YouTube. Если части между записью занимают больше времени, просто вырежьте видео или используйте эффект «таймлапс». Если вы застряли и вам нужно исправить ошибку, не останавливайте запись, потому что такие отрывки видео могут быть ценные для вашей аудитории, которая может столкнуться с теми же проблемами. Я считаю, что важно сохранить подобные части в вашем видео. Несколько советов по видео:

- Записывайте на своем родном языке, но, возможно, и на английском, если вам это удобно.

- Как можно подробнее и яснее объясняйте свои мысли — то, что вы делаете или проблемы, с которыми вы сталкиваетесь. Наличие видео — это только одна часть задачи, другая — сам рассказ о реализации. Повествование не обязательно должно быть идеальным, вместе этого он должен чувствовать естественным и не таким безупречным, как и все другие видеокурсы в интернете, где никто не сталкивается с проблемами, а показывается только само решение без ясных объяснений.
- Если вы столкнетесь с багами, попытайтесь их исправить. Делайте это самостоятельно, не сдавайтесь и расскажите о проблеме и о том, как вы пытаетесь ее решить. Это поможет другим отслеживать ваш мыслительный процесс. Как я уже говорил, нет смысла подражать отточенным другим видеокурсам, где преподаватель никогда не сталкивается с проблемами. Самое ценная часть — это видеть, как кто-то еще исправляет ошибку в исходном коде, и заодно объясняет в чём была её причина.
- Несколько слов о технической стороне записи: проверьте аудиотехнику перед записью длинного видеоролика. Громкость и качество должны быть в порядке. Что касается вашего редактора/IDE/терминала — не забудьте увеличить размер шрифта. Возможно, стоит разместить редактор с кодом и браузер рядом, чтобы их вместе было видно. Если не получается или это вам не подходит, сделайте их полноэкранными и переключайтесь между ними (например, в MacOS это комбинация клавиш CMD и Tab).
- Отредактируйте видео самостоятельно, прежде чем загрузить его на YouTube. Оно не обязательно должно быть высокого качества, но вам следует стараться держать его кратким по длительности (например, опуская отрывки для чтения и скорее кратко формулировать шаги своими словами).

В конце концов, вы можете обратиться ко мне для продвижения видео. Если видео получится удачным, я бы также хотел включить его в эту книгу в качестве официального дополнительного материала. Просто свяжись со мной, как только вы закончите работу над ним. В конце концов, я надеюсь, что вы примете этот вызов, чтобы улучшить свой опыт обучения во время чтения книги. Я желаю вам всего наилучшего!

Об этом переводе

Многие люди приложили руку к написанию и улучшению книги *Путь к изучению React (The Road to learn React)* за последнее время. В настоящее время это одна из самых скачиваемых книг по изучению React.js. Первоначально книга была написана немецким инженером-программистом [Робином Вируком \(Robin Wieruch\)³⁵](#). Но все переводы книги были бы невозможны без помощи других людей.

³⁵<https://www.robinwieruch.de/>

Небольшая предыстория

Перевод этой книги первоначально затягивался фронтенд-разработчиком Азат С.³⁶ (к сожалению, фамилию так и не удалось выяснить) осенью 2017 года, но, добавив только файлы для перевода, ничего не перевёл и больше не проявил никакое участие в переводе. Лишь только спустя пять месяцев, в конце февраля 2018 года, в репозитории перевода появилось движение — были актуализированы файлы перевода и переведён файл [README.md³⁷](#), а дальше снова долгое затишье до июня 2018. Но не стоит думать, что за эти несколько месяцев ничего не переводилось, вовсе нет, но очень нехотя и медленно. И наступило жаркое южное лето, и вот с этого момента работа над переводом пошла в полную силу...

Небольшое важное примечание: это мой первый крупный перевод с английского, до этого я мало работал с React, поэтому перевести книгу было вдвойне интересно — изучить React и подучить английский язык. Я решил, что летом обязательно выпущу перевод книги, в конце концов это не может продолжаться вечно, и вот 22 июля 2018 книга *The Road to learn React* вышла в русском переводе. Наслаждайтесь! И пожалуйста, если найдёте неточность, опечатку или какую-либо ошибку, напишите мне любым предпочтительным для вас способом.

³⁶<https://github.com/azat-io>

³⁷<https://github.com/the-road-to-learn-react/the-road-to-learn-react-russian/blob/master/README.md>

Немного о переводчике

Раздел выше написан от моего имени, а всё остальное — переведено мною. Итак, пора представиться. Меня зовут Алексей Пыльцын, я веб-разработчик, в основном использую PHP и JavaScript, большим опытом похвастаться не могу, поэтому перечислю списком то, чем я занимаюсь в настоящее время:

- поддерживаю официальную документацию по PHP на русском языке³⁸;
- время от времени перевожу³⁹ статьи по веб-разработке для devSchacht⁴⁰
- участвую по мере своих возможностей в разного рода опенсорс-проектах на GitHub⁴¹
- перевожу книги (да-да, это точно не последний перевод книги, в котором я принимаю участие, несмотря на то, как плохо или хорошо получился этот!)

³⁸<http://docs.php.net/manual/ru/>

³⁹<https://medium.com/@lex111/latest>

⁴⁰<https://medium.com/devschacht>

⁴¹<https://github.com/lex111/>

Немного о редакторе

Трудно переоценить работу редактора в любой книге, даже пусть такой небольшой как эта, — это очень дорогое стоит! Я бесконечно благодарен Екатерине Назаровой, фронтенд-разработчику и просто отличному человеку, за вычитку перевода, за все те многочисленные найденные опечатки и неточности. Поверьте, без неё перевод книги точно был бы гораздо хуже, чем он сейчас есть! Екатерина — автор проекта [GetInstance⁴²](https://getinstance.info), интернет-журнала для фронтенд-разработчиков и автор одноимённого [YouTube-канала⁴³](https://www.youtube.com/channel/UCEBHIIT_L1ME6e9ixaRPp0wg), подписывайтесь на её канал, это определённо стоит того!

⁴²<https://getinstance.info>

⁴³https://www.youtube.com/channel/UCEBHIIT_L1ME6e9ixaRPp0wg

Соглашения, принятые в переводе

В тексте данного перевода книги используется буква «ё» и французские кавычки, а также терминология из замечательного [словаря «Веб-стандартов»⁴⁴](#).

И последнее, но не менее важное: ссылки на документацию React ведут на [недавно созданный сайт с переводом документации React⁴⁵](#). Кроме перевода книги я решил также создать перевод документации по React, на момент первоначального выпуска книги переведены только те страницы, на которые ссылается книга, но я призываю всех принять участие в дальнейшем переводе документации в [репозитории react-ru⁴⁶](#).

Давайте вместе переведём и будем поддерживать документацию по React на русском языке! Этот проект специально направлен на сообщество, чтобы оно непосредственно принимало в нём участие. Точнее, я имею в виду, что в рунете много вариантов с переводом документации React, но все они различаются и не совсем актуальны. И главное, все они не похожи на оригинальный сайт, хотя лицензия не запрещает клонирование оригинального сайта, поэтому почему бы не объединиться и не создать единый перевод документации? В любом случае, если вам это интересно — присоединяйтесь!

⁴⁴<https://github.com/web-standards-ru/dictionary>

⁴⁵<https://ru.react.js.org/>

⁴⁶<https://github.com/js-rus/react-ru>

В добрый путь

Спасибо, что скачали эту книгу, а если всю её прочли — большие молодцы, я надеюсь, это было не бесполезное, а интересное и познавательное чтение! Отдельно хочется поблагодарить всех тех, кто находил разного рода ошибки в переводе и сообщал о них. Лучше всего это сделать — создать ишью в [репозитории перевода⁴⁷](#) (буду признателен, если вы ещё поставите лайк-звёздочку этому репозиторию!).

⁴⁷<https://github.com/the-road-to-learn-react/the-road-to-learn-react-russian>

Введение в React

В этой главе даётся введение в React. Вы можете спросить себя: почему я должен изучить React в первую очередь? Эта глава пытается ответить на данный вопрос. После этого вы погрузитесь в экосистему, создав ваше первое React-приложение без какой-либо конфигурации. По ходу дела вы познакомитесь с JSX и ReactDOM. Поэтому будьте готовы к вашим первым компонентам React.

Привет, меня зовут React.

Зачем вам нужно изучать React? В последние годы стали популярны односторонние приложения ([single-page application, SPA⁴⁸](#)). Фреймворки, такие как Angular, Ember и Backbone, помогали разработчикам JavaScript создавать современные веб-приложения за пределами использования чистого (ванильного) JavaScript и jQuery. Список этих популярных решений далеко не полный. Существует широкий круг фреймворков для создания SPA. Если посмотреть на даты релизов, то большинство из них относятся к первому поколению SPA: Angular 2010, Backbone 2010 и Ember 2011.

React был изначально выпущен Facebook в 2013 году. React — это не SPA-фреймворк, а библиотека для разработки пользовательских интерфейсов (UI). Это только представление, буква V в аббревиатуре [MVC⁴⁹](#) (Model View Controller). Она позволяет вам отрисовывать (render) компоненты в качестве видимых элементов в браузере. Однако целая экосистема вокруг React позволяет создавать односторонние приложения.

Но почему вы должны рассмотреть использование React, а не первое поколение SPA-фреймворков? В то время как первое поколение фреймворков пыталось решить сразу много всего, React используется только для создания слоя представления. Это библиотека, а не фреймворк. Идея React заключается в том, что ваше представление представляет собой иерархию составных компонентов.

В React вы можете сосредоточиться на слое представления, перед тем как внедрять остальные концепции в приложение. Каждый новый аспект — это ещё один строительный блок вашего SPA-приложения. Эти строительные блоки необходимы для создания зрелого приложения, и у них есть два преимущества.

Во-первых, вы можете изучать строительные блоки по одному, не понимая их вообще. Напротив, SPA даёт вам каждый строительный блок с самого начала. В этой книге основное внимание уделяется React как первому строительному блоку. В дальнейшем последует всё больше строительных блоков.

Во-вторых, все строительные блоки взаимозаменяемы, что делает экосистему React очень инновационной. Несколько решений конкурируют друг с другом, и вы можете выбрать наиболее привлекательное решение для вас и вашего варианта использования.

Первое поколение SPA-фреймворков достигло промышленного уровня; такие фреймворки менее гибкие. React остаётся инновационным и используется многими технологическими компаниями-лидерами, такими как [Airbnb](#), [Netflix](#) и, конечно же, [Facebook⁵⁰](#). Все они инвестируют в будущее React и довольствуются React и его экосистемой.

React — один из лучших выборов для создания современных веб-приложений в настоящее время. Он обеспечивает только уровень представления, [но экосистема React представляет](#)

⁴⁸https://ru.wikipedia.org/wiki/%D0%9E%D0%B4%D0%BD%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5

⁴⁹<https://ru.wikipedia.org/wiki/Model-View-Controller>

⁵⁰<https://github.com/facebook/react/wiki/Sites-Using-React>

собой гибкий и взаимозаменяемый фреймворк⁵¹. React имеет небольшой API, удивительную экосистему и отличное сообщество. Вы можете прочитать о моём опыте, [почему я перешёл с Angular на React](#)⁵². Я настоятельно рекомендую понять, почему вы выбрали React, а не другой фреймворк или библиотеку. В конце концов, каждый стремится узнать, куда приведёт нас React в ближайшие несколько лет.

Упражнения

- прочтайте о том, [почему я перешёл с Angular на React](#)⁵³
- прочтайте о [гибкой экосистеме React](#)⁵⁴
- прочтайте о том, [как изучать фреймворк](#)⁵⁵

⁵¹<https://www.robinwieruch.de/essential-react-libraries-framework/>

⁵²<https://www.robinwieruch.de/reasons-why-i-moved-from-angular-to-react/>

⁵³<https://www.robinwieruch.de/reasons-why-i-moved-from-angular-to-react/>

⁵⁴<https://www.robinwieruch.de/essential-react-libraries-framework/>

⁵⁵<https://www.robinwieruch.de/how-to-learn-framework/>

Требования

Какие требования для чтения этой книги? Прежде всего, вы должны быть знакомы с основами веб-разработки. Вы должны знать, как использовать HTML, CSS и JavaScript. Возможно, имеет смысл также знать, что такое означает термин [API⁵⁶](#), потому что вы будете использовать API в данной книге. Кроме того, я призываю вас вступить в официальную группу в [Slack⁵⁷](#) этой книги для получения помощи или чтобы помочь другим.

Редактор и терминал

Как насчёт среды разработки? Вам понадобится работающий редактор или IDE, а также терминал (инструмент командной строки). Вы можете [последовать моему руководству по настройке⁵⁸](#). Он предназначен для пользователей MacOS, но вы также можете найти руководство по настройке для Windows. В целом, есть много статей, которые покажут вам, как наилучшим образом настроить окружение веб-разработки более изысканным способом для используемой вами ОС.

При желании вы можете использовать git для хранения своих проектов и отслеживания прогресса изучения в репозиториях на GitHub, выполняя упражнения этой в книге. Существует [небольшое руководство⁵⁹](#) по использованию этих инструментов. Но опять же, это не обязательно для книги и может стать сдерживающим фактором при изучении всего этого с нуля. Вы можете пропустить эту часть, если вы новичок в веб-разработке, чтобы сосредоточиться на основных частях, описанных в книге.

Node и NPM

И последнее, но не менее важное: вам потребуется установка [node](#) и [npm⁶⁰](#). Оба они используются для управления библиотеками, которые нам понадобятся по ходу дела. В этой книге вы будете устанавливать внешние node-пакеты через npm (node package manager). Эти node-пакеты могут быть библиотеками или целыми фреймворками.

Проверить версии node и npm можно в командной строке. Если вы не видите какой-либо вывод в терминале, вам сначала нужно установить node и npm. Ниже показаны только мои версии, которые я использовал во время написания книги:

⁵⁶<https://www.robinwieruch.de/what-is-an-api-javascript/>

⁵⁷<https://slack-the-road-to-learn-react.wieruch.com/>

⁵⁸<https://www.robinwieruch.de/developer-setup/>

⁵⁹<https://www.robinwieruch.de/git-essential-commands/>

⁶⁰<https://nodejs.org/en/>

Командная строка

```
node --version
```

```
*v8.9.4
```

```
npm --version
```

```
*v5.6.0
```

node и npm

В этой главе приведён небольшой обзор node и npm. Он не исчерпывающий, но даст вам все необходимые инструменты. Если вы знакомы с обоими из них, то вы можете пропустить этот раздел.

Менеджер пакетов **node** (node package manager, npm) позволяет устанавливать **node-пакеты** (**node packages**) из командной строки. Эти пакеты могут быть набором утилитарных функций, библиотеками или целыми фреймворками. Все они являются зависимостями для вашего приложения. Вы можете установить все эти зависимости в папку с глобальными или локальными node-пакетами.

Глобальные node-пакеты доступны из любого места в терминале, и их необходимо установить только один раз в глобальный каталог. Вы можете установить глобальный пакет, введя в терминал:

Командная строка

```
npm install -g <package>
```

Флаг `-g` сообщает npm, что пакет нужно установить глобально. Локальные пакеты используются в вашем приложении. Например, React как библиотека будет локальным пакетом, который требуется для работы вашего приложения. Вы можете установить его через терминал, набрав:

Командная строка

```
npm install <package>
```

Команда установки React будет выглядеть следующим образом:

Командная строка

```
npm install react
```

Установленный пакет автоматически появится в папке `node_modules/` и будет перечислен в файле `package.json` вместе с другими зависимостями.

Но как инициализировать папку `node_modules/` и файл `package.json` для проекта в первую очередь? Для этого у нас есть команда npm, инициализирующая проект npm и, следовательно, файл `package.json`. Когда у вас есть этот файл, вы можете установить новые пакеты, используя npm.

Командная строка

```
npm init -y
```

Флаг `-y` — ярлык для инициализации всех значений по умолчанию в `package.json`. Без использования этого флага, вам нужно самому решить, как сконфигурировать этот файл. После инициализации вашего прт-проекта вы готовы к установке новых пакетов через команду `npm install <package>`.

Ещё пару слов об `package.json`. Данный файл позволяет вам поделиться вашим проектом с другими разработчиками без передачи всех node-пакетов. Этот файл содержит все ссылки на пакеты node, используемые в вашем проекте. Эти пакеты называются зависимостями. Каждый может скопировать ваш проект без этих зависимостей. Зависимости — это ссылки в `package.json`. Кто-то, кто копирует ваш проект, может просто установить все пакеты, используя `npm install` в командной строке. Команда `npm install` возьмёт все зависимости, перечисленные в файле `package.json` и установит их в папку `node_modules/`.

Я хочу рассмотреть ещё одну прт-команду:

Командная строка

```
npm install --save-dev <package>
```

Флаг `--save-dev` указывает, что node-пакет используется только в окружении разработки. Он не будет использоваться в продакшене при развёртывании вашего приложения на сервер. Какие node-пакеты должны устанавливаться с помощью этого флага? Представьте, что вы хотите протестировать приложение с помощью node-пакета. Вам нужно установить этот пакет через прт, но вы хотите исключить его из рабочего окружения. Тестирование должно происходить только в процессе разработки, а не тогда, когда приложение уже работает в продакшене. Там вам больше не нужно тестировать приложение. Приложение должно уже быть протестировано и работать из коробки для ваших пользователей. Это как раз тот случай, когда вы захотите использовать флаг `--save-dev`.

Вы столкнётесь с большим количеством прт-команд по ходу чтения, но этого пока будет достаточно.

Упражнения:

- создайте прт-проект
 - создайте каталог с помощью `mkdir <folder_name>`
 - перейдите в каталог с помощью `cd <folder_name>`
 - выполните `npm init -y` или `npm init`
 - установите локальный пакет React с помощью `npm install react`
 - убедитесь, что существует файл `package.json` и каталог `node_modules/`
 - выясните самостоятельно, как удалить node-пакет `react`
- узнайте больше о [npm⁶¹](#)

⁶¹<https://docs.npmjs.com/>

Установка

Существует несколько способов начать работу с приложением React.

Первый из них — использовать CDN. Это может звучать сложнее, чем есть на самом деле. А CDN — [сеть доставки содержимого](#)⁶². У нескольких компаний есть CDN, которые публично размещают файлы, чтобы люди могли использовать их. Этими файлами могут быть библиотеки, такие как React, поскольку собранная (bundled) библиотека React — это обычный JavaScript-файл *react.js*. Он может быть размещён где-то, и вы можете использовать его в своём приложении.

Как использовать CDN для начала работы с React? Вы можете встроить его в HTML-разметку с помощью тега `<script>`, со ссылкой, которая будет указывать на URL-адрес CDN. Для начала работы с React вам нужны два файла (библиотеки): *react* и *react-dom*.

Код

```
<script crossorigin src="https://unpkg.com/react@16/umd/react.development.js"></script>
<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
```

Но почему мы должны использовать CDN, когда есть прм для установки node-пакетов, таких как React?

Когда в приложении есть файл *package.json*, вы можете установить *react* и *react-dom* из командной строки. Однако папка должна быть инициализирована как прм-проект с помощью прм `init -y` с файлом *package.json*. Вы можете установить несколько node-пакетов в одну строку через прм.

Командная строка

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

Этот подход часто используется для добавления React в существующее приложение, управляемое с использованием прм.

К сожалению, это ещё не всё. Вам придётся столкнуться с [Babel](#)⁶³ для того, чтобы приложение могло использовать JSX (синтаксис React) и JavaScript ES6. Babel транспилирует (transpiles) ваш код так, чтобы браузеры могли интерпретировать код JavaScript ES6 и JSX, поскольку не все браузеры способны интерпретировать этот синтаксис. Эта установка включает в себя много настроек и инструментов, и для новичков в React это может быть слишком трудным, чтобы возиться со всей этой конфигурацией самостоятельно.

⁶²https://ru.wikipedia.org/wiki/Content_Delivery_Network

⁶³<http://babeljs.io/>

По этой причине Facebook представил *create-react-app* в качестве решения для быстрого запуска создания React-приложений без конфигурации (или как пишут — zero-configuration). В следующей главе будет показано, как настроить приложение, используя этот инструмент для инициализации приложения.

Упражнения:

- узнайте больше про [установку React⁶⁴](https://ru.react.js.org/docs/getting-started.html)

⁶⁴<https://ru.react.js.org/docs/getting-started.html>

Установка без конфигурации

В нашей книге вы будете использовать [create-react-app⁶⁵](#) для начальной инициализации вашего приложения. Это предварительно настроенный без необходимости в ручной конфигурации стартовый набор для React-приложений, представленный Facebook в 2016 году, и согласно опросу в Twitter [рекомендуется 96% начинающим разработчикам React⁶⁶](#). В *create-react-app* инструменты и конфигурация отходит на задний план, тогда как основное внимание уделяется реализации приложения.

Чтобы начать работу, вам требуется установить пакет в каталог глобальных node-пакетов. После этого вы всегда будете иметь возможность из командной строки инициализировать новое React-приложение.

Командная строка

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

Вы можете проверить версию *create-react-app*, чтобы убедиться в успешной установке из командной строки:

Командная строка

```
create-react-app --version  
*v1.5.1
```

Теперь вы можете инициализировать своё первое React-приложение. Мы назовём его *hackernews*, но вы можете выбрать другое имя. Весь процесс настройки займёт пару секунд. После этого перейдите в папку:

Командная строка

```
create-react-app hackernews  
cd hackernews
```

Теперь вы можете открыть приложение в своём редакторе. Будет представлена следующая структура папок или её вариация, в зависимости от версии *create-react-app*:

⁶⁵<https://github.com/facebookincubator/create-react-app>

⁶⁶https://twitter.com/dan_abramov/status/806985854099062785

Структура каталогов

```
hackernews/
  README.md
  node_modules/
  package.json
  .gitignore
  public/
    favicon.ico
    index.html
    manifest.json
  src/
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
    registerServiceWorker.js
```

Ниже представлен краткий список каталогов и файлов. Это нормально, если вы не понимаете их все в самом начале.

- **README.md:** Расширение `.md` указывает, что это текстовый файл в формате Markdown. Markdown используется как лёгкий язык разметки с синтаксисом для форматирования текста. Во многих проектах с открытым исходным кодом есть файл `README.md`, чтобы дать пользователям первоначальные инструкции по проекту. Когда вы размещаете свой проект на такой платформе, как GitHub, при открытии репозитория вы увидите визуальное отображение этого файла `README.md`. Поскольку вы использовали `create-react-app`, ваш `README.md` будет таким же, как в [GitHub-репозитории create-react-app](#)⁶⁷.
- **node_modules/:** В этой папке находятся все node-пакеты, которые были и будут установлены через прм. Так как вы использовали `create-react-app`, здесь должно уже быть пару установленных node-модулей. Обычно вы никогда не будете иметь дело с этим каталогом, поскольку установка и удаление node-пакетов происходит с помощью пакетных менеджеров (например, прм) из командной строки.
- **package.json:** Данный файл показывает список зависимостей node-пакетов и прочую конфигурационную информацию проекта.
- **.gitignore:** В этом файле указываются все файлы и каталоги, которые не должны быть добавлены в ваш git-репозиторий при использовании git; такие файлы будут находиться только в локальном проекте. Каталог `node_modules/` как раз является таким каталогом, который должен игнорироваться git. Достаточно, чтобы файл `package.json`

⁶⁷<https://github.com/facebookincubator/create-react-app>

находился под git для возможности совместного использования с вашими коллегами, чтобы они смогли самостоятельно установить все зависимости без разделения с ними папки с зависимостями.

- **public/**: Этот каталог содержит корневые файлы разработки, такие как *public/index.html*. Это индексный файл, который отображается при переходе на localhost:3000 при разработке приложения. Стандартная заготовка (boilerplate) учитывает этот файл, чтобы связать его со всеми скриптами в *src/*.
- **build/** Этот каталог будет создан при сборке проекта для продакшена. Он содержит все готовые файлы при сборке приложения для продакшен-окружения. Весь ваш код, написанный в каталогах *src/* и *public/*, будет собран (bundled) в пару файлов при выполнении сборки проекта и будут размещены в каталоге *build*.
- **manifest.json** and **registerServiceWorker.js**: не обращайте внимания на эти файлы на данном этапе, нам они не понадобятся в этом проекте.

Вам не нужно трогать указанные файлы и каталоги. В самом начале всё, что вам нужно, находится в каталоге *src/*. Основное внимание уделяется файлу *src/App.js* для реализации React-компонентов. Он будет использоваться для реализации приложения, но позже вы можете разделить свои компоненты на несколько файлов, тогда каждый файл представляет собой один или несколько компонентов.

Кроме того, вы найдёте файл *src/App.test.js* для своих тестов и *src/index.js* как точку входа (entry point) в мир React. Об этих двух файлах вы узнаете в следующей главе. В добавок есть файлы *src/index.css* и *src/App.css* для стилизации общего приложения и ваших компонентов. У всех у них есть стили по умолчанию, если вы их откроете.

Приложение *create-react-app* — проект npm. Вы будете использовать npm для установки и удаления node-пакетов. Кроме того, вместе с ним идут npm-скрипты для выполнения в командной строке:

Командная строка

```
# Запускает приложение по адресу http://localhost:3000
npm start

# Запускает выполнение тестов
npm test

# Запускает сборку приложения для продакшена
npm run build
```

Эти скрипты определены в вашем *package.json*. Теперь заготовка для React-приложения готова к работе. Следующие упражнения позволят вам, наконец, запустить созданное приложение в браузере.

Упражнения:

- выполните команду `npm start` и перейдите к просмотру приложения в вашем браузере (вы можете выйти из команды, завершить её, нажав на Control + C)
- запустите интерактивный скрипт `npm test`
- запустите скрипт `npm run build` и убедитесь, что в проекте создался каталог `build/` (вы можете удалить его потом; обратите внимание, что каталог сборки может использоваться позже для [развертывания приложения](#)⁶⁸)
- ознакомьтесь со структурой каталогов
- ознакомьтесь с содержимым файлов
- узнайте подробнее про [npm-скрипты в пакете create-react-app](#)⁶⁹

⁶⁸<https://www.robinwieruch.de/deploy-applications-digital-ocean/>

⁶⁹<https://github.com/facebookincubator/create-react-app>

Введение в JSX

Теперь вы узнаете о JSX — синтаксисе React. Как упоминалось ранее, *create-react-app* уже подготовил заготовку приложения для вас. Каждый файл имеет реализацию по умолчанию. Давайте погрузимся в исходный код. Единственным файлом, с которым вы в первую очередь будете работать — *src/App.js*.

src/App.js

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">Добро пожаловать в React</h1>
        </header>
        <p className="App-intro">
          Для начала отредактируйте <code>src/App.js</code> и сохраните его для пере\загрузки.
        </p>
      </div>
    );
  }
}

export default App;
```

Не позволяйте себе запутаться в выражениях `import/export` и объявлении класса. Эти возможности уже JavaScript ES6. Мы рассмотрим их в следующей главе.

В файле есть **React-компонент, определённый через класс ES6** с именем `App`. Это объявление компонента. В основном после того, как вы объявили компонент, вы можете использовать его в качестве элемента повсюду в своём приложении. Он будет создавать экземпляр вашего компонента или другими словами: компонент создаёт экземпляр (инстанцируется).

Возвращаемый элемент указывается в методе `render()`. Элементы — это то, из чего состоят компоненты. Важно понимать различия между компонентом, экземпляром и элементом.

Довольно скоро вы увидите, где создаётся экземпляр компонента `App`. В противном случае вы не увидите отрисованный вывод, не так ли? Компонент `App` — это только объявление, но

не его использование. Вы должны инстанцировать компонент где-то в своём JSX с помощью `<App />`.

Содержимое в блоке `render` выглядит довольно похожим на HTML, но это JSX. JSX позволяет смешивать HTML и JavaScript. Он мощный, но сбивает с толку, когда вы используете его для разделения HTML и JavaScript. Вот почему хорошей отправной точкой считается использовать обычный HTML в JSX. Для начала откройте файл `App.js` и замените HTML-код на тот, который показан ниже.

src/App.js

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <h2>Добро пожаловать в Путь к изучению React</h2>
      </div>
    );
  }
}

export default App;
```

Теперь вы возвращаете только HTML из метода `render()` без всякого JavaScript. Давайте определим “Добро пожаловать в Путь к изучению React” в качестве переменной. Переменная может использоваться в JSX с использованием фигурных скобок.

src/App.js

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
    var helloWorld = 'Добро пожаловать в Путь к изучению React';
    return (
      <div className="App">
        <h2>{helloWorld}</h2>
      </div>
    );
  }
}
```

```
export default App;
```

Он должен заработать, когда вы запустите своё приложение в командной строке с помощью команды `npm start` снова.

Кроме того, вы могли заметить атрибут `className`. Он представляет стандартный атрибут `class` в HTML. По техническим причинам в JSX пришлось заменить несколько встроенных HTML-атрибутов. Вы можете найти все [поддерживаемые HTML-атрибуты в документации к React⁷⁰](#). Все они следуют соглашению написания в camelCase. На вашем пути к изучению React, вы столкнётесь с некоторыми специфическими атрибутами JSX.

Упражнения:

- определите больше переменных и отрисуйте их в вашем JSX
 - используйте сложный объект для представления пользователя с именем и фамилией
 - отобразите свойства пользователя в своём JSX
- ознакомьтесь подробнее с синтаксисом [JSX⁷¹](#)
- узнайте больше про [компоненты, элементы и экземпляры в React⁷²](#)

⁷⁰<https://ru.react.js.org/docs/dom-elements.html#all-supported-html-attributes>

⁷¹<https://ru.react.js.org/docs/introducing-jsx.html>

⁷²<https://reactjs.org/blog/2015/12/18/react-components-elements-and-instances.html>

const и let в ES6

Я думаю, вы заметили, что мы объявили переменную `helloWorld`, используя выражение `var`. В JavaScript ES6 появилось два варианта для определения переменных: `const` и `let`. В JavaScript ES6 вы редко теперь найдёте использование `var` для определения переменных.

Переменная, объявленная с помощью `const`, не может быть повторно объявлена или изменена (она неизменяемая). Как только структура данных определена, вы не сможете её изменить.

Код

```
// изменение не разрешено
const helloWorld = 'Добро пожаловать в Путь к изучению React';
helloWorld = 'Пока-пока, React';
```

Переменная, объявленная с помощью `let`, может быть изменена.

Код

```
// изменение разрешено
let helloWorld = 'Добро пожаловать в Путь к изучению React';
helloWorld = 'Пока-пока, React';
```

Вам стоит объявлять переменные через `let`, если потребуется позже повторно переназначить переменную.

Однако нужно быть аккуратнее с `const`. Переменная, объявленная с использованием `const` не может быть изменена. Но в случае, если эта переменная — массив или объект, значение изменится как обычно. Подобное значение не является неизменяемым.

Код

```
// изменение разрешено
const helloWorld = {
  text: 'Добро пожаловать в Путь к изучению React'
};
helloWorld.text = 'Пока-пока, React';
```

Но в каких случаях следует использовать тот или иной способ определения переменной? Существуют разные мнения на этот счёт. Я предлагаю использовать `const` каждый раз при определении переменной. Это будет означать, что вы хотите иметь неизменяемую структуру данных, даже несмотря на то, что значения в объектах и массивах могут изменяться. Если переменная будет изменяемой, то вы можете использовать `let`.

Неизменяемость охватывает React и его экосистему. Вот почему `const` должен быть вашим выбором по умолчанию при определении переменной. Тем не менее, в сложных объектах значения внутри могут быть изменены. Будьте осторожны с этим поведением.

В вашем приложении используйте `const` вместо `var`.

src/App.js

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {
  render() {
    const helloWorld = 'Добро пожаловать в Путь к изучению React';
    return (
      <div className="App">
        <h2>{helloWorld}</h2>
      </div>
    );
  }
}

export default App;
```

Упражнения:

- узнайте больше о [выражении const в ES6](#)⁷³
- узнайте больше о [выражении let в ES6](#)⁷⁴
- узнайте подробнее про неизменяемые структуры данных
 - почему они вообще имеют смысл в программировании
 - почему они используются в React и его экосистеме

⁷³<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/const>

⁷⁴<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/let>

ReactDOM

Прежде чем продолжить с компонентом App, возможно, вы захотите посмотреть, где он используется. Он используется в вашей точке входа в мир React — в файле `src/index.js`.

`src/index.js`

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import './index.css';

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

В основном `ReactDOM.render()` использует DOM-узел в вашей HTML-разметке для замены его вашим JSX. Вот так легко вы можете интегрировать React в любое стороннее приложение. Не запрещается использовать `ReactDOM.render()` несколько раз в приложении. Вы можете использовать его в нескольких местах для начальной загрузки простого JSX-синтаксиса, React-компонента, нескольких React-компонентов или всего приложения. Но в простом React-приложении вы будете использовать его только один раз для инициализации всего дерева компонентов.

`ReactDOM.render()` ожидает два аргумента. Первый аргумент — JSX, который будет отрисовываться. Второй аргумент указывает место, где React-приложение привязывается к вашему HTML. Он ожидает элемент с `id='root'`. Вы можете открыть файл `public/index.html`, чтобы найти этот атрибут.

В текущей реализации `ReactDOM.render()` уже принимает компонент App. Тем не менее, было бы неплохо передать более простой JSX. Это необязательно должно быть инстанцирование компонента.

Код

```
ReactDOM.render(
  <h1>Привет, мир React</h1>,
  document.getElementById('root')
);
```

Упражнения:

- откройте файл `public/index.html`, чтобы посмотреть, где React-приложение монтируется в HTML

- узнайте больше про [отрисовку элементов в React⁷⁵](https://ru.react.js.org/docs/rendering-elements.html)

⁷⁵<https://ru.react.js.org/docs/rendering-elements.html>

Горячая перезагрузка

Горячая перезагрузка модулей (Hot Module Replacement, HMR) — это то, что вы в качестве разработчика можете сделать в файле `src/index.js` для улучшения процесса разработки.

По умолчанию `create-react-app` заставит обновлять страницу в браузере при изменении исходного кода. Попробуйте сами, изменив переменную `helloWorld` в файле `src/App.js`. Браузер должен обновить содержимое страницы. Но есть лучший способ сделать это.

Горячая перезагрузка модулей или замена модулей без полной перезагрузки страницы — это инструмент для перезагрузки приложения в браузере. Браузер не выполняет обновление страницы. Вы можете легко активировать его в `create-react-app`. В `src/index.js`, точке входа React, вы можете добавить следующую настройку.

`src/index.js`

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import './index.css';

ReactDOM.render(
  <App />,
  document.getElementById('root')
);

if (module.hot) {
  module.hot.accept();
}
```

Вот и всё. Повторите попытку изменить переменную `helloWorld` в файле `src/App.js`. Браузер не должен обновить страницу, но приложение перезагрузится и покажет корректный (актуальный) вывод. У HMR есть несколько преимуществ:

Представьте, что делаете отладку кода с использованием выражений `console.log()`. Эти выражения останутся в консоли разработчика, даже если вы изменили код, потому что браузер больше не обновляет страницу. Это может быть удобно для целей отладки.

В развивающемся приложении обновление страницы задерживает вашу продуктивность. Вы должны подождать, пока страница загрузится. Перезагрузка страницы может занять несколько секунд в большом приложении. HMR устраняет этот недостаток.

Наконец, самое большое преимущество HMR в том, что вы можете сохранить состояние после перезагрузки приложения. Представьте, что у вас есть диалоговое окно в вашем приложении с несколькими шагами, и вы находитесь на шаге 3. В целом, это напоминает мастер настройки. Без HMR вы измените исходный код, и браузер обновит страницу. Вам

нужно снова открыть диалоговое окно и перейти с шага 1 на шаг 3. С использованием HMR диалоговое окно остаётся открытым на шаге 3. Он сохраняет состояние приложения, даже если исходный код изменяется. Перезагружается только само приложение, а не страница.

Упражнения:

- измените исходный код *src/App.js* несколько раз, чтобы увидеть работу HMR в действии
- посмотрите первые 10 минут видеоролика (на английском) Дэна Абрамова (Dan Abramov) [Live React: Hot Reloading with Time Travel⁷⁶](#)

⁷⁶<https://www.youtube.com/watch?v=xsSnOQynTHs>

Комплексный JavaScript в JSX

Вернёмся к нашему компоненту App. До сих пор вы отрисовывали некоторые примитивные переменные в вашем JSX. Теперь вы начнёте отрисовывать список элементов. В начале список будет состоять из демонстрационных данных, но позже данные вы будете получать из внешнего API⁷⁷, что будет намного интереснее.

Сначала определим список элементов.

src/App.js

```
import React, { Component } from 'react';
import './App.css';

const list = [
  {
    title: 'React',
    url: 'https://reactjs.org/',
    author: 'Jordan Walke',
    num_comments: 3,
    points: 4,
    objectID: 0,
  },
  {
    title: 'Redux',
    url: 'https://redux.js.org/',
    author: 'Dan Abramov, Andrew Clark',
    num_comments: 2,
    points: 5,
    objectID: 1,
  },
];
class App extends Component {
  ...
}
```

Демо-данные представляют данные, которые будут получены позже из API. У элемента в списке есть заголовок, ссылка и автор. Кроме того, он содержит идентификатор, баллы (которые указывают, насколько популярна статья) и количество комментариев.

Теперь вы можете использовать встроенную функцию JavaScript `map` в JSX. Она позволяет перебирать список элементов для их отображения. Снова вы будете использовать фигурные скобки для вставки (инкапсуляции) JavaScript-выражений в вашем JSX.

⁷⁷<https://www.robinwieruch.de/what-is-an-api-javascript/>

src/App.js

```
class App extends Component {  
  render() {  
    return (  
      <div className="App">  
        {list.map(function (item) {  
          return <div>{item.title}</div>;  
        })}  
      </div>  
    );  
  }  
}  
  
export default App;
```

Использование JavaScript вместе с HTML очень эффективно в JSX. Вы могли использовать `map` для преобразования одного списка элементов в другой список элементов. Но на этот раз вы можете использовать `map` для преобразования списка элементов в HTML-элементы.

Пока что для каждого элемента отображается `title`. Давайте отобразим ещё больше свойств элементов.

src/App.js

```
class App extends Component {  
  render() {  
    return (  
      <div className="App">  
        {list.map(function (item) {  
          return (  
            <div>  
              <span>  
                <a href={item.url}>{item.title}</a>  
              </span>  
              <span>{item.author}</span>  
              <span>{item.num_comments}</span>  
              <span>{item.points}</span>  
            </div>  
          );  
        })}  
      </div>  
    );  
  }  
}
```

```
export default App;
```

Вы видите, что функция `map` просто встроена в ваш JSX. Каждое свойство элемента отображается в теге ``. Кроме того, свойство `url` элемента используется в атрибуте `href` тега `<a>`.

React выполнит всю работу за вас и отобразит каждый элемент, но вы должны добавить одного помощника для React, чтобы полностью использовать его потенциал и улучшить его производительность. Вы должны назначить атрибут `key` для каждого элемента списка. Таким образом, React может идентифицировать добавленные, изменённые и удалённые элементы при изменении списка. У элементов списка демо-данных уже есть идентификатор.

`src/App.js`

```
{list.map(function (item) {
  return (
    <div key={item.objectID}>
      <span>
        <a href={item.url}>{item.title}</a>
      </span>
      <span>{item.author}</span>
      <span>{item.num_comments}</span>
      <span>{item.points}</span>
    </div>
  );
})}
```

Вы должны убедиться, что атрибут `key` — уникальный идентификатор. Не допускайте ошибку, использовав индекс элемента в массиве. Индекс массива вообще непостоянный. Например, когда список изменяет свой порядок, React будет трудно идентифицировать элементы правильно.

`src/App.js`

```
// не делайте так
{list.map(function (item, key) {
  return (
    <div key={key}>
      ...
    </div>
  );
})}
```

Теперь вы отображаете оба списка. Вы можете запустить приложение, открыть браузер и увидеть оба элемента списка.

Упражнения:

- узнайте подробнее про [списки и ключи React⁷⁸](https://ru.react.js.org/docs/lists-and-keys.html)
- повторите [стандартные встроенные функции массива в JavaScript⁷⁹](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/)
- используйте больше JavaScript-выражений в JSX

⁷⁸<https://ru.react.js.org/docs/lists-and-keys.html>

⁷⁹https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/

Стрелочные функции ES6

В стандарте ES6 появились стрелочные функции (arrow functions). Выражение стрелочной функции короче функциональных выражений.

Код

```
// объявление функции
function () { ... }

// объявление стрелочной функции
() => { ... }
```

Вам необходимо знать о функциональности стрелочных функций. Одна из их особенностей — это другое поведение с объектом `this`. Функция всегда определяет свой собственный объект `this`. Стрелочная функция получает значение `this` из окружающего контекста.

Есть ещё один важный факт о стрелочных функциях, касающийся круглых скобок. Вы можете удалить круглые скобки, когда функция принимает только один аргумент, но их нужно оставить в случае, если стрелочная функция принимает несколько аргументов (или ни одного — прим. пер.).

Код

```
// разрешено
item => { ... }

// разрешено
(item) => { ... }

// не разрешено
item, key => { ... }

// разрешено
(item, key) => { ... }
```

Давайте посмотрим на функцию `map`. Вы можете написать её более кратко с помощью стрелочных функций из ES6.

src/App.js

```
{list.map(item => {
  return (
    <div key={item.objectID}>
      <span>
        <a href={item.url}>{item.title}</a>
      </span>
      <span>{item.author}</span>
      <span>{item.num_comments}</span>
      <span>{item.points}</span>
    </div>
  );
})}
```

Кроме того, вы можете удалить *тело блока*, то есть фигурные скобки стрелочной функции ES6. В *сокращённом теле блока* подразумевается неявный возврат. Таким образом, вы можете удалить выражение `return`. Такая форма стрелочной функции в книге будет использоваться чаще, поэтому убедитесь, что понимаете разницу между телом блока и сокращённым телом блока при использовании стрелочных функций.

src/App.js

```
{list.map(item =>
  <div key={item.objectID}>
    <span>
      <a href={item.url}>{item.title}</a>
    </span>
    <span>{item.author}</span>
    <span>{item.num_comments}</span>
    <span>{item.points}</span>
  </div>
)}
```

Теперь ваш JSX выглядит более кратким и читаемым. В нём нет выражения `function`, фигурных скобок и выражения `return`. Вместо этого разработчик может сосредоточиться на деталях реализации.

Упражнения:

- изучите подробнее стрелочные функции ES6⁸⁰

⁸⁰https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Классы ES6

В стандарте ES6 появился синтаксис классов. Класс обычно используется в объектно-ориентированных языках программирования. JavaScript был и есть очень гибкий в своих парадигмах программирования. Вы можете использовать функциональное программирование и объектно-ориентированное программирование бок о бок, в зависимости от конкретных случаев использования.

Несмотря на то, что React охватывает функциональное программирование, например, с неизменяемыми структурами данных, классы используются для объявления компонентов. Они называются компонентами класса ES6 или классовыми компонентами ES6. React смешивает хорошие части этих парадигм программирования.

Давайте рассмотрим следующий класс Developer для изучения классов JavaScript ES6, не думая о компоненте.

Код

```
class Developer {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }

  getName() {
    return this.firstname + ' ' + this.lastname;
  }
}
```

У класса есть конструктор, чтобы сделать его инстанцируемым (иметь возможность создавать из него объекты — прим. пер.). Конструктор может принимать аргументы, чтобы их можно было в качестве свойств назначить экземпляру класса. Кроме того, в классе можно определять функции. Поскольку функция связана с классом, она называется методом. Часто он упоминается как метод класса.

Класс Developer — это только объявление класса. Вы можете создать несколько экземпляров класса путём его вызова. Он похож на компонент класса ES6, который имеет объявление, но вы должны использовать его в другом месте для создания экземпляра.

Давайте посмотрим, как вы можете создать экземпляр класса и как использовать его методы.

Код

```
const robin = new Developer('Robin', 'Wieruch');
console.log(robin.getName());
// выведет: Robin Wieruch
```

React использует классы JavaScript ES6 для классов-компонентов. Вы уже использовали один ES6-класс компонента.

src/App.js

```
import React, { Component } from 'react';

...
class App extends Component {
  render() {
    ...
  }
}
```

Класс App наследует Component. В основном, когда вы объявляете компонент App, он наследуется от другого компонента. Что значит “наследуется”? В объектно-ориентированном программировании у вас есть принцип наследования, который означает, что функциональные возможности могут передаваться из одного класса в другой.

Класс App наследует функциональность из класса Component. Более конкретно — он наследует функциональность из класса Component. Класс Component используется для наследования базового класса ES6 в класс компонента ES6. Он имеет всю функциональность, которую имеет компонент в React. Метод render — одна из тех функциональностей, которые вы уже использовали. В дальнейшем вы узнаете о других методах класса компонента.

Класс Component инкапсулирует все детали реализации компонента React. Это позволяет разработчикам использовать классы как компоненты в React.

Методы, предоставляемые компонентом Component, являются открытым интерфейсом. Один из этих методов должен быть переопределён, другие не должны быть переопределены. Вы узнаете о последних методах, когда позже в этой книге будут рассматриваться методы жизненного цикла. Метод render() должен быть переопределён, поскольку он определяет вывод компонента React Component.

В данный момент вы узнали об основах классов JavaScript ES6 и о том, как они используются в React для создания компонентов. Вы узнаете подробнее о методах Component, когда в книге будут описаны методы жизненного цикла React.

Упражнения:

- узнайте получше [классы в ES6⁸¹](#)
- убедитесь, что знаете хорошо основы JavaScript перед тем как перейти к изучению [React⁸²](#)

⁸¹<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Classes>

⁸²<https://www.robinwieruch.de/javascript-fundamentals-react-requirements/>

Вы научились создавать своё собственное React-приложение! Давайте вспомним последние темы:

- React
 - `create-react-app` для создания заготовки приложения React
 - JSX смешивает HTML и JavaScript для определения вывода React-компонентов в своих методах `render`
 - компоненты, экземпляры и элементы — разные понятия в React
 - `ReactDOM.render()` — это точка входа для React-приложения, которая привязывает React к DOM
 - встроенная функциональность JavaScript может использоваться в JSX
 - * функцию `map` можно использовать для отрисовки списка элементов как HTML-элементов
- ES6
 - объявления переменных с помощью `const` и `let` могут использоваться в зависимости от конкретных случаев
 - * использование `const` вместо `let` в React-приложениях
 - стрелочные функции используются для краткого написания ваших функций
 - классы используются для определения компонентов в React путём их наследования

На данном этапе имеет смысл сделать перерыв. Усвоить полученные знания и применить их на практике самостоятельно. Вы можете поэкспериментировать с исходным кодом, написанным в рамках этой главы. Его можно найти в [официальном репозитории](#)⁸³.

⁸³<https://github.com/the-road-to-learn-react/hackernews-client/tree/5.1>

Основы React

В этой главе вы познакомитесь с основами React. Она охватывает состояние и взаимодействие в компонентах, потому что статические компоненты немного скучны, не так ли? Кроме того, вы узнаете о различных способах объявления компонента и о том, как сохранять компоненты компонуемыми и повторно используемыми.

Внутреннее состояние компонента

Внутреннее состояние компонента, также известное как локальное состояние, позволяет сохранять, изменять и удалять свойства, хранящиеся в вашем компоненте. ES6-класс компонента может использовать конструктор для инициализации внутреннего состояния компонента позже. Конструктор вызывается только один раз, когда компонент инициализируется.

Давайте покажем конструктор класса.

src/App.js

```
class App extends Component {  
  
  constructor(props) {  
    super(props);  
  }  
  
  ...  
}
```

Компонент App наследуется от класса Component: об этом говорит ключевое слово `extends Component` в объявлении компонента App.

Обязательно вызвать `super(props)`: он устанавливает `this.props` в конструкторе на случай, если вы хотите получить доступ к ним оттуда. В противном случае при доступе к свойствам компонента через `this.props` из конструктора они будут иметь значение `undefined`. В дальнейшем вы узнаете больше о свойствах React.

Теперь в вашем случае начальным состоянием компонента будет список элементов с демо-данными.

src/App.js

```
const list = [  
  {  
    title: 'React',  
    url: 'https://reactjs.org/',  
    author: 'Jordan Walke',  
    num_comments: 3,  
    points: 4,  
    objectID: 0,  
  },  
  ...  
];
```

```
class App extends Component {  
  
  constructor(props) {  
    super(props);  
  
    this.state = {  
      list: list,  
    };  
  }  
  
  ...  
}

---


```

Состояние связано с классом с помощью объекта `this`. Таким образом, вы можете получить доступ к локальному состоянию во всём компоненте. Например, его можно использовать в методе `render()`. Ранее мы использовали функцию `map` в методе `render()` со статическим списком, который был определён вне вашего компонента. Теперь мы собираемся использовать этот список из локального состояния компонента.

src/App.js

```
class App extends Component {  
  
  ...  
  
  render() {  
    return (  
      <div className="App">  
        {this.state.list.map(item =>  
          <div key={item.objectID}>  
            <span>  
              <a href={item.url}>{item.title}</a>  
            </span>  
            <span>{item.author}</span>  
            <span>{item.num_comments}</span>  
            <span>{item.points}</span>  
          </div>  
        )}  
      </div>  
    );  
  }  
}
```

Список теперь является частью компонента. Он находится во внутреннем состоянии компонента. Вы можете добавлять, изменять или удалять элементы в вашем списке. Каждый раз, когда вы будете изменять состояние компонента, будет вызываться метод `render()`. Вот как вы просто можете изменить состояние и убедиться, что компонент повторно отрисовывается и отображает корректные данные из локального состояния.

Но будьте осторожны. Не изменяйте состояние напрямую. Вы должны использовать метод `setState()` для изменения состояния. Вы узнаете об этом в следующей главе.

Упражнения:

- поэкспериментируйте с локальным состоянием
 - определите больше данных в начальном состоянии в вашем конструкторе
 - используйте состояние в методе `render()`
- узнайте подробнее [конструктор класса в ES6](#)⁸⁴

⁸⁴<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Classes#Constructor>

Инициализация объектов ES6

В JavaScript ES6 вы можете использовать сокращённый синтаксис для более короткой инициализации свойств объектов. Представьте себе следующую инициализацию объекта:

Код

```
const name = 'Robin';

const user = {
  name: name,
};
```

Когда имя свойства в объекте совпадает с именем переменной, вы можете использовать следующее:

Код

```
const name = 'Robin';

const user = {
  name,
};
```

В вашем приложении вы можете сделать то же самое. Имя переменной списка и имя свойства состояния используют одинаковое имя.

Код

```
// ES5
this.state = {
  list: list,
};

// ES6
this.state = {
  list,
};
```

Другим классным помощником являются сокращённые имена методов. В JavaScript ES6 вы можете определять методы в объекте в более лаконичной форме.

Код

```
// ES5
var userService = {
  getUserName: function (user) {
    return user.firstname + ' ' + user.lastname;
  },
};

// ES6
const userService = {
  getUserName(user) {
    return user.firstname + ' ' + user.lastname;
  },
};
```

И последнее, но не менее важное: вы можете использовать вычисляемые имена свойств в JavaScript ES6.

Код

```
// ES5
var user = {
  name: 'Robin',
};

// ES6
const key = 'name';
const user = {
  [key]: 'Robin',
};
```

Возможно, вычисляемые имена свойств пока не имеют для вас никакого смысла. Зачем они нужны? В следующей главе вы перейдёте к этапу, где вы можете использовать их для перераспределения значений по ключу динамическим способом в объекте. Это здорово для генерации таблиц поиска в JavaScript.

Упражнения:

- поэкспериментируйте с инициализацией объекта в ES6
- узнайте подробнее, как происходит [инициализация объектов в ES6](#)⁸⁵

⁸⁵https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Object_initializer

Однонаправленный поток данных

Теперь у нас есть внутреннее состояние в компоненте App. Однако вы ещё не манипулировали локальным состоянием. Состояние является статическим, а следовательно, и сам компонент. Хороший способ познакомиться с манипуляциями с состоянием — реализовать некоторое взаимодействие с компонентом.

Давайте добавим кнопку для каждого элемента в отображаемом списке. Назовём кнопку «Отбросить», она будет удалять элемент из списка. Это может быть полезно в конце концов, когда у вас есть список непрочитанных элементов и вы хотите отклонить те элементы, которые вас не интересуют.

src/App.js

```
class App extends Component {  
  ...  
  render() {  
    return (  
      <div className="App">  
        {this.state.list.map(item =>  
          <div key={item.objectID}>  
            <span>  
              <a href={item.url}>{item.title}</a>  
            </span>  
            <span>{item.author}</span>  
            <span>{item.num_comments}</span>  
            <span>{item.points}</span>  
            <span>  
              <button  
                onClick={() => this.onDismiss(item.objectID)}  
                type="button"  
              >  
                Отбросить  
              </button>  
            </span>  
          </div>  
        )}  
      </div>  
    );  
  }  
}
```

Метод класса `onDismiss()` ещё не определён. Мы сделаем это чуть позже, а пока сфокусируемся на обработчике `onClick` элемента кнопки. Как вы можете видеть, метод `onDismiss()` в обработчике `onClick` заключён в другую функцию. Это стрелочная функция. Таким образом, вам доступно свойство `objectID` объекта `item` для идентификации элемента, который будет отброшен. Альтернативным способом было бы определить функцию вне обработчика `onClick` и передавать только определённую функцию обработчику. В более поздней главе будет дано подробное объяснение темы обработчиков элементов.

Вы заметили, что я использовал несколько строк для элемента кнопки? Обратите внимание: элементы с несколькими атрибутами в один момент становятся сложными для чтения, если записывать их в одну строку. Вот почему элемент кнопки записан в нескольких строках с добавлением отступов для сохранения читаемости кода. Но это не обязательно. Это только моя настоятельная рекомендация по стилю кода.

Теперь нам нужно реализовать функциональность метода `onDismiss()`. Он принимает идентификатор для отклонения элемента. Функция привязывается к классу `i`, таким образом, становится методом класса. Вот почему вы получаете доступ к ней с помощью `this.onDismiss()`, а не `onDismiss().this` — ваш экземпляр класса. Чтобы определить метод класса `onDismiss()`, вы должны связать его в конструкторе. Привязки (bindings) будут объяснены в другой главе позже.

src/App.js

```
class App extends Component {  
  
  constructor(props) {  
    super(props);  
  
    this.state = {  
      list,  
    };  
  
    this.onDismiss = this.onDismiss.bind(this);  
  }  
  
  render() {  
    ...  
  }  
}
```

Следующим шагом нам нужно определить функциональность, бизнес-логику, в вашем классе. Методы класса определяются следующим образом.

src/App.js

```
class App extends Component {  
  
  constructor(props) {  
    super(props);  
  
    this.state = {  
      list,  
    };  
  
    this.onDismiss = this.onDismiss.bind(this);  
  }  
  
  onDismiss(id) {  
    ...  
  }  
  
  render() {  
    ...  
  }  
}
```

Теперь вы определите, что происходит внутри метода класса. В основном вы хотите удалить элемент с определённым идентификатором из списка и сохранить обновлённый список в вашем локальном состоянии. После этого обновлённый список запустит повторный вызов метода `render()` для его отображения. Удалённый элемент больше не должен появляться.

Вы можете удалить элемент из списка с помощью встроенной JavaScript-функции `filter` в JavaScript. Эта функция принимает функцию в качестве входного параметра. Эта функция имеет доступ к каждому значению в списке, потому что выполняет итерацию по списку. Таким образом, вы можете проверить каждый элемент списка, основываясь на условии фильтрации. Если проверка вычисляется как `true`, то элемент остаётся в списке. В противном случае он будет исключён из результата. Кроме того, хорошо знать, что эта функция возвращает новый список и не изменяет старый. Она поддерживает соглашение React о наличии неизменяемых структур данных.

src/App.js

```
onDismiss(id) {  
  const updatedList = this.state.list.filter(function isNotId(item) {  
    return item.objectID !== id;  
  });  
}
```

На следующем шаге вы можете извлечь функцию и передать её функции filter.

src/App.js

```
onDismiss(id) {  
  function isNotId(item) {  
    return item.objectID !== id;  
  }  
  
  const updatedList = this.state.list.filter(isNotId);  
}
```

Кроме того, вы можете сделать это более кратко, снова используя стрелочные функции из JavaScript ES6.

src/App.js

```
onDismiss(id) {  
  const isNotId = item => item.objectID !== id;  
  const updatedList = this.state.list.filter(isNotId);  
}
```

Вы могли бы даже снова встроить выражение, по аналогии с обработчиком onClick кнопки, но это может стать менее читабельно.

src/App.js

```
onDismiss(id) {  
  const updatedList = this.state.list.filter(item => item.objectID !== id);  
}
```

Сейчас нажатие на кнопку удаляет содержащий её элемент. Однако состояние ещё не обновлено. Поэтому вы можете, наконец, использовать метод класс setState() для обновления списка во внутреннем состоянии компонента.

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedList = this.state.list.filter(isNotId);
  this.setState({ list: updatedList });
}
```

Теперь снова запустите своё приложение и попробуйте нажать на кнопку «Отбросить». Это должно работать. Теперь вы получили представление, что такое *однонаправленный поток данных* в React. Вы активируете действие в своём представлении с помощью `onClick()`, функция или метод класса изменяет внутреннее состояние компонента и метод `render()` компонента снова вызывается для обновления представления.

Упражнения:

- узнайте больше про [состояние и жизненный цикл в React](#)⁸⁶

⁸⁶<https://ru.react.js.org/docs/state-and-lifecycle.html>

Привязки

При использовании компонентов класса React ES6 важно изучить привязки в JavaScript-классах. В предыдущей главе вы связали метод класса `onDismiss()` в конструкторе.

src/App.js

```
class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      list,
    };

    this.onDismiss = this.onDismiss.bind(this);
  }

  ...
}
```

Зачем вам это делать в первую очередь? Этап привязки необходим, потому что методы класса автоматически не привязывают `this` к экземпляру класса. Давайте продемонстрируем это с помощью следующего ES6-класса компонента.

Код

```
class ExplainBindingsComponent extends Component {
  onClickMe() {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Нажми на меня
      </button>
    );
  }
}
```

Компонент отрисовывается просто отлично, но при нажатии на кнопку, вы получите `undefined` в консоли разработчика. Это основной источник багов при использовании React. Если вы хотите получить доступ к `this.state` в своём методе класса, его нельзя получить, поскольку `this` не определён. Поэтому для того, чтобы сделать `this` доступным в методах класса, вам нужно привязать их к `this`.

В следующем классе компонента метод класса правильно привязан в конструкторе класса.

Код

```
class ExplainBindingsComponent extends Component {
  constructor() {
    super();

    this.onClickMe = this.onClickMe.bind(this);
  }

  onClickMe() {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Нажми на меня
      </button>
    );
  }
}
```

Если попробовать нажать на кнопку ещё один раз, объект `this` должен быть определён конкретным экземпляром класса, и теперь вы сможете получить доступ к `this.state` или `this.props`, о котором вы узнаете позже.

Привязка метода класса может также произойти в другом месте. Например, это может быть в методе `render()`.

Код

```
class ExplainBindingsComponent extends Component {  
  onClickMe() {  
    console.log(this);  
  }  
  
  render() {  
    return (  
      <button  
        onClick={this.onClickMe.bind(this)}  
        type="button"  
      >  
        Нажми на меня  
      </button>  
    );  
  }  
}
```

Но вам не следует так делать, потому что при таком подходе метод будет привязываться каждый раз при вызове `render()`. По сути, он выполняется каждый раз при обновлении компонентов, что влияет на производительность. При привязке метода класса в конструкторе, вы привязываете метод только один раз в самом начале создания компонента. Это лучший подход для привязки.

И ещё кое-что, что люди иногда придумывают — определение бизнес-логики методов класса в конструкторе.

Код

```
class ExplainBindingsComponent extends Component {  
  constructor() {  
    super();  
  
    this.onClickMe = () => {  
      console.log(this);  
    }  
  }  
  
  render() {  
    return (  
      <button  
        onClick={this.onClickMe}  
        type="button"  
      >  
    );  
  }  
}
```

```
    Нажми на меня
    </button>
);
}
}
```

Вы также не должны делать этого, потому что со временем эти методы будут загромождать конструктор. Конструктор должен служить для создания экземпляра класса со всеми его свойствами. Вот именно по этой причине методы следует определять вне конструктора.

Код

```
class ExplainBindingsComponent extends Component {
  constructor() {
    super();

    this.doSomething = this.doSomething.bind(this);
    this.doSomethingElse = this.doSomethingElse.bind(this);
  }

  doSomething() {
    // сделать что-то
  }

  doSomethingElse() {
    // сделать что-то ещё
  }

  ...
}
```

И последнее, но немаловажное, о чём стоит упомянуть — методы класса могут автоматически привязываться, без явной привязки с использованием стрелочных функций в JavaScript ES6.

Код

```
class ExplainBindingsComponent extends Component {
  onClickMe = () => {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Нажми на меня
      </button>
    );
  }
}
```

Если повторное связывание в конструкторе вас раздражает, вы можете использовать этот способ. Поскольку официальная документация React придерживается использования привязок метода в конструкторе, мы также последуем этому в книге.

Упражнения:

- попробуйте различные подходы к привязкам и выведите на консоль объект `this`, используя `console.log`.

Обработчик событий

Данная глава должна дать вам более глубокое понимание обработчиков событий в элементах. В вашем приложении вы используете следующий элемент кнопки для удаления элемента из списка.

src/App.js

```
...
<button
  onClick={() => this.onDismiss(item.objectID)}
  type="button"
>
  Отбросить
</button>
```

Это уже сложный вариант использования, потому что вам нужно передать значение методу класса, и, таким образом, вам нужно обернуть его в другую (стрелочную) функцию. Поэтому в основном это должна быть функция, которая передаётся обработчику события. Следующий код не будет работать, поскольку метод выполняется сразу же при открытии приложения в браузере.

src/App.js

```
...
<button
  onClick={this.onDismiss(item.objectID)}
  type="button"
>
  Отбросить
</button>
```

При использовании `onClick={doSomething()}` функция `doSomething()` будет выполняться немедленно при открытии приложения в вашем браузере. Выполнится выражение в обработчике и поскольку возвращаемое значение больше не будет функцией, ничего не произойдёт при повторном нажатии кнопки в дальнейшем. Но при использовании `onClick={doSomething}` поскольку `doSomething` — функция, она будет выполняться только при нажатии на кнопку. Те же правила применяются к используемому в вашем приложении методу `onDismiss()`.

Однако, просто использовать `onClick={this.onDismiss}` недостаточно, потому что каким-то образом нужно передать методу класса свойство `item.objectID` для идентификации элемента, который будет отброшен. Вот почему он может быть завернут в другую функцию, чтобы проникнуть в свойство. Данная концепция называется функциями высшего порядка в JavaScript и будет объяснена вкратце позже.

src/App.js

```
...
<button
  onClick={() => this.onDismiss(item.objectID)}
  type="button"
>
  Отбросить
</button>
```

Обходной путь — определить функцию-обёртку где-то снаружи и передать только определённую функцию обработчику. Поскольку для этого требуется доступ к отдельному элементу, она должна находиться внутри блока функции тар.

src/App.js

```
class App extends Component {
  ...
  render() {
    return (
      <div className="App">
        {this.state.list.map(item => {
          const onHandleDismiss = () =>
            this.onDismiss(item.objectID);

          return (
            <div key={item.objectID}>
              <span>
                <a href={item.url}>{item.title}</a>
              </span>
              <span>{item.author}</span>
              <span>{item.num_comments}</span>
              <span>{item.points}</span>
              <span>
```

```
        <button
            onClick={onHandleDismiss}
            type="button"
            >
                Отбросить
            </button>
        </span>
    </div>
);
}
)
}
</div>
);
}
}
```

В конце концов, это должна быть функция, которая передаётся обработчику элемента. В качестве примера попробуйте использовать следующий код:

src/App.js

```
class App extends Component {  
  ...  
  render() {  
    return (  
      <div className="App">  
        {this.state.list.map(item =>  
          ...  
          <span>  
            <button  
              onClick={console.log(item.objectID)}  
              type="button"  
            >  
              Отбросить  
            </button>  
          </span>  
        </div>  
      )}  
    </div>  
  );  
}  
}
```

Она будет выполнена при открытии приложения в браузере, но не при нажатии кнопки. В то время как следующий код запускается только при нажатии кнопки. Эта функция выполняется при запуске обработчика.

src/App.js

...

```
<button
  onClick={function () {
    console.log(item.objectID)
 }}
  type="button"
>
  Отбросить
</button>
```

...

Чтобы сделать этот обработчик короче, вы можете снова преобразовать его в стрелочную функцию из JavaScript ES6. Это то, что мы также сделали с методом onDismiss().

src/App.js

...

```
<button
  onClick={() => console.log(item.objectID)}
  type="button"
>
  Отбросить
</button>
```

...

Часто новички в React испытывают трудности с темой использования функций в обработчиках событий. Вот почему я попытался объяснить это более подробно. В итоге вы получите следующий код для вашей кнопки, который состоит из краткой односторонней стрелочной функции JavaScript ES6 и имеет доступ к свойству objectID объекта item.

src/App.js

```
class App extends Component {  
  ...  
  
  render() {  
    return (  
      <div className="App">  
        {this.state.list.map(item =>  
          <div key={item.objectID}>  
            ...  
            <span>  
              <button  
                onClick={() => this.onDismiss(item.objectID)}  
                type="button"  
              >  
              Отбросить  
            </button>  
            </span>  
          </div>  
        )}  
      </div>  
    );  
  }  
}
```

Другая довольно актуальная тема, связанная с производительностью — это последствия использования стрелочных функций в обработчиках событий. Например, обработчик `onClick` для метода `onDismiss()` оборачивает метод в другую стрелочную функции для передачи идентификатора элемента. Поэтому каждый раз при выполнении метода `render()`, обработчик создаёт экземпляр стрелочной функции высшего порядка и запускает его. Это *может* влиять на производительность приложения, но в большинстве случаев вы не заметите разницы. Представьте, что у вас есть огромная таблица с 1000 элементами, и у каждой строки или столбца есть такая стрелочная функция как обработчик события. Здесь стоит думать о влиянии на производительность, и поэтому вы можете реализовать отдельный компонент `Button` для привязки метода в конструкторе. Но прежде чем это произойдёт, это будет преждевременная оптимизация. Целесообразнее сосредоточиться на изучении React.

Упражнения:

- попробуйте различные подходы к использованию функций в обработчике `onClick` вашей кнопки

Взаимодействия с формами и событиями

Давайте добавим ещё одно взаимодействие в приложение для получения опыта работы с формами и событиями в React. Следующее взаимодействие — это функциональность поиска. Поле для поиска будет использоваться для временной фильтрации списка на основе свойства title в элементе.

На первом этапе нужно определить форму с полем для ввода в JSX.

src/App.js

```
class App extends Component {  
  ...  
  render() {  
    return (  
      <div className="App">  
        <form>  
          <input type="text" />  
        </form>  
        {this.state.list.map(item =>  
          ...  
        )}  
      </div>  
    );  
  }  
}
```

В следующем сценарии вы вводите текст в поле ввода и список временно фильтруется по введённой строке. Для возможности фильтровать список на основе значения из поля ввода, нужно сохранить значение в локальном состоянии. Но как получить доступ к значению? Вы можете использовать *синтетические события* (*synthetic events*) в React для доступа к данным (payload) события.

Давайте определим обработчик onChange для поля ввода.

src/App.js

```
class App extends Component {  
  ...  
  
  render() {  
    return (  
      <div className="App">  
        <form>  
          <input  
            type="text"  
            onChange={this.onSearchChange}  
          />  
        </form>  
        ...  
      </div>  
    );  
  }  
}
```

Функция снова привязана к компоненту и, следовательно, к методу класса. Теперь вам нужно определить и привязать этот метод.

src/App.js

```
class App extends Component {  
  
  constructor(props) {  
    super(props);  
  
    this.state = {  
      list,  
    };  
  
    this.onSearchChange = this.onSearchChange.bind(this);  
    this.onDismiss = this.onDismiss.bind(this);  
  }  
  
  onSearchChange() {  
    ...  
  }  
}
```

```
...  
}
```

При использовании обработчика в элементе вы получаете доступ к синтетическому событию React в объявлении колбэка.

src/App.js

```
class App extends Component {  
  ...  
  onSearchChange(event) {  
    ...  
  }  
  ...  
}
```

У события есть значение поля ввода в его объекте `target`. Следовательно, вы можете обновить локальное состояние с помощью поисковой строки, снова используя `this.setState()`.

src/App.js

```
class App extends Component {  
  ...  
  onSearchChange(event) {  
    this.setState({ searchTerm: event.target.value });  
  }  
  ...  
}
```

Кроме того, важно не забыть определить начальное состояние для свойства `searchTerm` в конструкторе. Поле ввода должно быть пустым в начале, и поэтому значение будет пустой строкой.

src/App.js

```
class App extends Component {  
  
  constructor(props) {  
    super(props);  
  
    this.state = {  
      list,  
      searchTerm: '',  
    };  
  
    this.onSearchChange = this.onSearchChange.bind(this);  
    this.onDismiss = this.onDismiss.bind(this);  
  }  
  
  ...  
}
```

Теперь вы сохраняете входное значение во внутреннем состоянии компонента каждый раз при изменении поля ввода.

Краткая заметка об обновлении локального состояния в компоненте React. Было бы справедливо предположить, что при обновлении `searchTerm` с использованием `this.setState()` необходимо также передать список для сохранения его. Но это не совсем так. `this.setState()` в React выполняет поверхностное (неглубокое) слияние объектов (`shallow merge`). Он сохраняет свойства на одном уровне в объекте состояния при обновлении одного-единственного свойства в нём. Таким образом, состояние списка, даже если вы отбросили элемент из него, останется таким же при обновлении свойства `searchTerm`.

Давайте вернёмся к нашему приложению. Список ещё не фильтруется на основе значения из поля ввода, хранящегося в локальном состоянии. По сути вам нужно временно фильтровать список, основываясь на значении `searchTerm`. У вас есть всё необходимое для этого. Итак, а как временно фильтровать список? В методе `render()` перед использованием функции `map` на списке можно применить к нему фильтр. Фильтрация будет проходить только в случае, если `searchTerm` соответствует свойству `title` элемента. Ранее вы использовали встроенную в JavaScript функцию `filter`, поэтому давайте применим её ещё раз. Можно использовать функцию фильтрации перед функцией `map`, потому что она возвращает новый массив, и тем самым функция `map` может использоваться таким удобным способом.

src/App.js

```
class App extends Component {  
  ...  
  
  render() {  
    return (  
      <div className="App">  
        <form>  
          <input  
            type="text"  
            onChange={this.onSearchChange}  
          />  
        </form>  
        {this.state.list.filter(...).map(item =>  
          ...  
        )}  
      </div>  
    );  
  }  
}
```

Давайте на этот раз применим функцию фильтрации по-другому. Мы хотим определить вне класса компонента аргумент фильтра и функцию, которая передаётся функции filter. Вне компонента у нас нет доступа к его состоянию и поэтому у нас нет доступа к свойству searchTerm для выполнения условия фильтра. Мы должны передать searchTerm в функцию фильтра и должны вернуть новую функцию для проверки условия. Это называется функцией высшего порядка (higher-order function).

Обычно я бы не упомянул функции высшего порядка, но в книге про React в этом определённо есть смысл, потому что React связан с концепцией, называемой компонентами высшего порядка. Вы познакомитесь с этой концепцией позже в этой книге. А пока давайте сосредоточимся на функциональности фильтра.

Во-первых, определим функцию высшего порядка вне компонента App.

src/App.js

```
function isSearched(searchTerm) {
  return function(item) {
    // условие, возвращающее true или false
  }
}

class App extends Component {
  ...
}
```

Функция принимает `searchTerm` и возвращает другую функцию, потому что в конце концов функция `filter` принимает функцию в качестве входного параметра. Возвращаемая функция имеет доступ к объекту элемента, так как она была передана функции `filter`. Кроме того, возвращаемая функция будет использоваться для фильтрации списка на основе условия, определённого в функции. Давайте теперь определим это условие.

src/App.js

```
function isSearched(searchTerm) {
  return function(item) {
    return item.title.toLowerCase().includes(searchTerm.toLowerCase());
  }
}

class App extends Component {
  ...
}
```

В условии утверждается, что сопоставляется входящий шаблон (строка) `searchTerm` со свойством `title` элемента списка. Вы можете сделать это с помощью встроенной функции JavaScript `includes`. Только когда шаблон совпадает, возвращается `true` и элемент остаётся в списке, иначе элемент удаляется из списка. Но будьте осторожны с сопоставлением по шаблону: нужно помнить про нижний регистр обеих строк. В противном случае между строкой поиска ‘redux’ и элементом с заголовком ‘Redux’ не будет соответствия. Поскольку мы работаем над неизменяемым списком и возвращаем новый список с помощью фильтрации, исходный список в локальном состоянии не изменяется вовсе.

Осталось только упомянуть: мы немного сжульничали, использовав встроенную функцию JavaScript `includes`. Это функция из ES6. Как она будет выглядеть в JavaScript ES5? Вы можете

использовать функцию `indexOf()` для получения индекса элемента в списке. Если элемент находится в списке, `indexOf()` вернёт его индекс в массиве.

Код

```
// ES5
string.indexOf(pattern) !== -1

// ES6
string.includes(pattern)
```

Ещё один элегантный рефакторинг можно сделать с помощью стрелочных функций ES6. Это сделает функцию более лаконичной:

Код

```
// ES5
function isSearched(searchTerm) {
  return function(item) {
    return item.title.toLowerCase().indexOf(searchTerm.toLowerCase()) !== -1;
  }
}

// ES6
const isSearched = searchTerm => item =>
  item.title.toLowerCase().includes(searchTerm.toLowerCase());
```

Можно поспорить, какая из функций более удобочитаемая. Лично я предпочитаю вторую. В экосистеме React используются множество концепций функционального программирования. Вы часто будете использовать функцию, возвращающую другую функцию (функции высшего порядка). В JavaScript ES6 вы можете написать это более кратко с помощью стрелочных функций.

И последнее, но не менее важное: вам нужно использовать определённую функцию `isSearched()` для фильтрации списка. Вы передаёте ей свойство `searchTerm` из локального состояния, она возвращает функцию фильтрации входных данных и фильтрует список на основе условия фильтра. После этого функция `map`, применённая на отфильтрованном списке, отображает элемент для каждого элемента списка.

src/App.js

```
class App extends Component {  
  ...  
  
  render() {  
    return (  
      <div className="App">  
        <form>  
          <input  
            type="text"  
            onChange={this.onSearchChange}  
          />  
        </form>  
        {this.state.list.filter(isSearched(this.state.searchTerm)).map(item =>  
          ...  
        )}  
      </div>  
    );  
  }  
}
```

Теперь функциональность поиска должна работать. Попробуйте сами в браузере.

Упражнения:

- узнайте больше про [события React](#)⁸⁷
- узнайте подробнее [функции высшего порядка](#)⁸⁸

⁸⁷<https://ru.react.js.org/docs/handling-events.html>

⁸⁸https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_%D0%B2%D1%8B%D1%81%D1%88%D0%B5%D0%B3%D0%BE_%D0%BF%D0%BE%D1%80%D1%8F%D0%B4%D0%BA%D0%B0

Деструктуризация ES6

В JavaScript ES6 есть способ упросить доступ к свойствам объектов и массивов. Он называется деструктуризацией (destructuring). Сравните следующий фрагмент код в JavaScript ES5 и ES6.

Код

```
const user = {
  firstname: 'Robin',
  lastname: 'Wieruch',
};

// ES5
var firstname = user.firstname;
var lastname = user.lastname;

console.log(firstname + ' ' + lastname);
// выведет: Robin Wieruch

// ES6
const { firstname, lastname } = user;

console.log(firstname + ' ' + lastname);
// выведет: Robin Wieruch
```

Тогда как в JavaScript ES5 каждый раз требуется новая строка для получения значения свойства объекта, в JavaScript ES6 это можно сделать в одну строку. Передовая практика для читабельности — использование нескольких строк при деструктуризации нескольких свойств объекта.

Код

```
const {
  firstname,
  lastname
} = user;
```

То же самое относится и к массивами: их то же можно деструктуризировать. Опять же, благодаря использованию нескольких строк код останется читабельным и лёгким для понимания.

Код

```
const users = ['Robin', 'Andrew', 'Dan'];
const [
  userOne,
  userTwo,
  userThree
] = users;

console.log(userOne, userTwo, userThree);
// выведет: Robin Andrew Dan
```

Возможно, вы заметили, что таким же образом можно применить деструктуризацию к объекту локального состояния компонента App. Это сократит длину строки кода в функциях `filter` и `map`.

src/App.js

```
render() {
  const { searchTerm, list } = this.state;
  return (
    <div className="App">
      ...
      {list.filter(isSearched(searchTerm)).map(item =>
        ...
      )}
    </div>
  );
}
```

Далее показаны различия использования ES5 и ES6 на одном и том же примере:

Код

```
// ES5
var searchTerm = this.state.searchTerm;
var list = this.state.list;

// ES6
const { searchTerm, list } = this.state;
```

Данная книга будет использовать в основном JavaScript ES6, поэтому вам стоит придерживаться этого выбора.

Упражнения:

- изучите подробнее [деструктуризацию в ES6⁸⁹](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

⁸⁹https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Контролируемые компоненты

Вы уже узнали об одностороннем потоке данных в React. Этот же принцип применяется к полю ввода, которое обновляет локальное состояние в зависимости от значения `searchTerm` для фильтрации списка. При изменении состояния, метод `render()` выполняется снова и используется последнее значение `searchTerm` из локального состояния для проверки условия фильтра.

Но разве мы не забыли что-то в элементе поля ввода? У тега HTML `input` есть атрибут `value`. Значение атрибута, как правило, эквивалентно значению, которое отображается в самом поле ввода. В нашем случае этим значением будет `searchTerm`. Однако, похоже, нам не требуется подобное в React.

Это неправильно. Элементы формы, такие как `<input>`, `<textarea>` и `<select>` хранят своё состояние в обычном HTML. Они изменяют значение внутри, когда кто-то изменяет его извне. В React это называется **неконтролируемым компонентом (uncontrolled component)**, потому что он (сам) обрабатывает своё собственное состояние. В React вы должны убедиться, что эти элементы **контролируются компонентами**.

Как мы можем этого добиться? Нужно только установить атрибут со значением поля ввода. Значение уже сохранено в свойстве локального состояния `searchTerm`. Так почему бы нам не использовать его?

src/App.js

```
class App extends Component {  
  ...  
  render() {  
    const { searchTerm, list } = this.state;  
    return (  
      <div className="App">  
        <form>  
          <input  
            type="text"  
            value={searchTerm}  
            onChange={this.onSearchChange}  
          />  
        </form>  
        ...  
      </div>  
    );  
  }  
}
```

Вот и всё. Теперь цикл одностороннего потока данных для поля ввода — замкнутый. Внутреннее состояние компонента — единственный источник информации для поля ввода.

Всё это управление внутренним состоянием и односторонним источником данных, возможно, в новинку для вас. Но как только вы привыкнете к этому всему, это станет естественным потоком данных при реализации приложений на React. В целом, React с односторонним потоком данных принёс новый шаблон в мир одностраничных приложений. В настоящее время эта концепция принята некоторыми фреймворками и библиотеками.

Упражнения:

- узнайте больше о [формах в React](#)⁹⁰
- изучите подробнее [различные контролируемые компоненты](#)⁹¹

⁹⁰<https://ru.react.js.org/docs/forms.html>

⁹¹<https://github.com/the-road-to-learn-react/react-controlled-components-examples>

Разделение компонентов

Сейчас у вас один огромный компонент App, который продолжает расти и в один момент в нём можно будет запутаться. Давайте начнём разделять его на небольшие компоненты.

src/App.js

```
class App extends Component {  
  ...  
  
  render() {  
    const { searchTerm, list } = this.state;  
    return (  
      <div className="App">  
        <Search />  
        <Table />  
      </div>  
    );  
  }  
}
```

Вы можете передавать только те свойства компонентов, которые они сами используют. В случае компонента App, ему необходимо передать свойства, управляемые локальным состоянием и его методами класса.

src/App.js

```
class App extends Component {  
  ...  
  
  render() {  
    const { searchTerm, list } = this.state;  
    return (  
      <div className="App">  
        <Search  
          value={searchTerm}  
          onChange={this.onSearchChange}  
        />  
        <Table  
          list={list}  
          pattern={searchTerm}  
          onDismiss={this.onDismiss}  
        </Table>  
      </div>  
    );  
  }  
}
```

```
    />
  </div>
);
}
}
```

Теперь вы можете определить компоненты рядом с вашим компонентом App. Эти компоненты также будут классовыми компонентами. Они отрисовывают те же самые элементы, как и раньше.

Первый из них — компонент Search.

src/App.js

```
class App extends Component {
  ...
}

class Search extends Component {
  render() {
    const { value, onChange } = this.props;
    return (
      <form>
        <input
          type="text"
          value={value}
          onChange={onChange}
        />
      </form>
    );
  }
}
```

Второй — компонент Table.

src/App.js

```
...
class Table extends Component {
  render() {
    const { list, pattern, onDismiss } = this.props;
    return (
      <div>
        {list.filter(isSearched(pattern)).map(item =>
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
            <span>
              <button
                onClick={() => onDismiss(item.objectID)}
                type="button"
              >
                Отбросить
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

Итак, теперь у вас есть три класса компонентов. Возможно, вы обратили внимание на объект `props`, доступный через экземпляр класса, используя `this.props` — это сокращённая форма для свойств (`properties`), они хранят все значения, которые вы передали компонентам, когда использовали их в компоненте `App`. Таким образом, компоненты могут передавать свойства через дерево компонентов.

Путём выделения этих компонентов из компонента `App`, они стали повторно используемыми где-нибудь ещё. Поскольку эти компоненты получают свои значения с помощью объекта `props`, вы можете передавать различные свойства компонентам каждый раз, когда используете их где-то ещё.

Упражнения:

- подумайте, какие компоненты вы ещё могли бы разделить, как это было сделано с компонентами Search и Table
 - но не делайте это сейчас, иначе вы столкнётесь с конфликтами в следующих главах

Компонуемые компоненты

Существует одно небольшое свойство, доступное в объекте `props` — `children`. Вы можете использовать его для передачи элементов вашим компонентам сверху, которые неизвестны самому компоненту, что позволяет образовывать компоненты друг из друга. Давайте посмотрим, как это выглядит при передаче строки текста в качестве дочернего элемента в компонент `Search`.

src/App.js

```
class App extends Component {  
  ...  
  
  render() {  
    const { searchTerm, list } = this.state;  
    return (  
      <div className="App">  
        <Search  
          value={searchTerm}  
          onChange={this.onSearchChange}>  
        </Search>  
        Поиск  
        </div>  
        <Table  
          list={list}  
          pattern={searchTerm}  
          onDismiss={this.onDismiss}>  
        </Table>  
      </div>  
    );  
  }  
}
```

Теперь компонент `Search` может деструктурировать свойство `children` из объекта `props`, а затем указать, где он должен отображаться.

src/App.js

```
class Search extends Component {  
  render() {  
    const { value, onChange, children } = this.props;  
    return (  
      <form>  
        {children} <input  
          type="text"  
          value={value}  
          onChange={onChange}  
        />  
      </form>  
    );  
  }  
}
```

Теперь текст “Поиск” будет показываться рядом с полем ввода. И в случае, если вы используете компонент Search где-нибудь ещё, вы можете задать другой текст, если хотите. В конце концов, необязательно в качестве children должен быть простой текст. Вы можете передать элемент или дерево элементов (которые снова могут быть инкапсулированы компонентами) в свойство children. Свойство children также позволяет вкладывать (weave) компоненты друг в друга.

Упражнения:

- прочитайте подробнее про [модель композиции React](#)⁹²

⁹²<https://ru.react.js.org/docs/composition-vs-inheritance.html>

Повторно используемые компоненты

Повторно используемые и составные компоненты предоставляют возможность создавать иерархии компонентов. Они — основа слоя представления в React. В последних главах говорилось о повторном использовании. Теперь вы можете повторно использовать компоненты Table и Search. Даже компонент App можно повторно применять, создав экземпляр его где-нибудь в другом месте.

Давайте определим ещё один повторно используемый компонент — Button, который, в конечном итоге, будет повторно использоваться почаще.

src/App.js

```
class Button extends Component {
  render() {
    const {
      onClick,
      className,
      children,
    } = this.props;

    return (
      <button
        onClick={onClick}
        className={className}
        type="button"
      >
        {children}
      </button>
    );
  }
}
```

Может показаться лишним объявлять подобный компонент. Вы будете использовать компонент Button вместо элемента button. Он сохранит только type="button". За исключением типа атрибута, вам потребуется определить всё остальное при использовании компонента Button. Но вы должны подумать здесь о долгосрочных инвестициях. Представьте, что у вас есть несколько кнопок в вашем приложении, но вам нужно изменить атрибут, стиль или поведение для кнопки. Без компонента вам придётся рефакторить каждую кнопку. Вместо этого компонент Button гарантирует наличие только одного источника истины. Один компонент Button для рефакторинга всех кнопок одновременно. Один компонент Button — это как одна кнопка для управления всем необходимым ей.

Поскольку у вас уже есть элемент кнопки, вы можете использовать компонент Button вместо него. При использовании этого компонента нам не нужно задавать тип атрибута, поскольку в компоненте уже он указан.

src/App.js

```
class Table extends Component {
  render() {
    const { list, pattern, onDismiss } = this.props;
    return (
      <div>
        {list.filter(isSearched(pattern)).map(item =>
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
            <span>
              <Button onClick={() => onDismiss(item.objectID)}>
                Отбросить
              </Button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

Компонент Button ожидает свойство `className` в `props`. Атрибут `className` представляет соответствующий HTML атрибут для класса. Но мы не передали имя класса в `className` при использовании компонента Button. В коде компонента Button должно быть явно показано, что `className` необязательный, поэтому при деструктуризации указывается пустая строка в качестве значения по умолчанию.

src/App.js

```
class Button extends Component {
  render() {
    const {
      onClick,
      className = '',
      children,
    } = this.props;

    ...
  }
}
```

Теперь каждый раз, когда нет свойства `className` при использовании компонента `Button`, значение будет пустой строкой вместо `undefined`.

Объявления компонентов

На текущий момент у вас есть четыре класса компонентов. Но их можно объявить по-другому. Позвольте мне представить функциональные компоненты без состояния в качестве альтернативы классовым компонентам. Прежде чем заняться рефакторингом ваших компонентов, давайте рассмотрим различные типы компонентов в React.

- **Функциональные компоненты без состояния (Functional Stateless Components):** Это компоненты, представленные в виде функций, которые получают входные данные и возвращают вывод. Входные данные — это `props`, а вывод — экземпляр компонента, то есть простой JSX. До сих пор он очень похож на классовый компонент ES6. Однако, функциональные компоненты без состояния — это функции (функциональный), у которых нет локального состояния (поэтому и в названии — без состояния). У вас нет доступа к состоянию и вы не можете обновить состояние, используя `this.state` или `this.setState()`, соответственно, поскольку нет объекта `this`. Кроме того, у функциональных компонентов нет методов жизненного цикла. Мы ещё не рассматривали методы жизненного цикла, но вы уже использовали два из них: `constructor()` и `render()`. Тогда как конструктор выполняется только один раз в течение жизни компонента, метод класса `render()` запускается один раз в начале и каждый раз при обновлениях компонента. Имейте в виду, что у функциональных компонентов без состояния нет методов жизненного цикла, о которых будет рассказано в последующих главах.
- **Классовые компоненты или классы компонентов (ES6 Class Components):** Вы уже использовали этот тип объявления в ваших компонентах. В определении класса они наследуются (расширяются, `extend`) от компонента React. Ключевое слово `extend` привязывает компоненту все методы жизненного цикла, доступные API компонента React. Именно поэтому вы можете использовать метод `render()`. Кроме того, вы можете хранить и управлять состоянием в классах компонентов через использование `this.state` и `this.setState()`.
- **React.createClass:** Подобное объявление компонента использовалось в старых версиях React и всё ещё используется в React-приложениях на JavaScript ES5. Но Facebook [объявил их устаревшими⁹³](#) в пользу JavaScript ES6. Они даже добавили [соответствующее предупреждение в версии 15.5⁹⁴](#). Вы не будете использовать их в этой книге.

Таким образом, в основном осталось только два объявления компонента. Но когда следует предпочесть функциональные компоненты без состояния классовым компонентам? Эмпирическое правило — использовать функциональные компоненты без состояния, если вам не нужны локальное состояние или методы жизненного цикла. Как правило, вы начинаете реализовать свои компоненты именно как функциональные компоненты без состояния. Как только вам нужен доступ к состоянию или методам жизненного цикла — преобразуйте

⁹³<https://reactjs.org/blog/2015/03/10/react-v0.13.html>

⁹⁴<https://reactjs.org/blog/2017/04/07/react-v15.5.0.html>

компонент в классовый компонент. В нашем приложении мы поступили как раз наоборот, но всё ради изучения React.

Вернёмся к нашему приложению. Компонент App использует внутреннее состояние. Именно поэтому он должен оставаться нетронутым, то есть классом компонента. Но три других классовых компонента не используют состояние, то есть им не нужен доступ к `this.state` или `this.setState()`. Более того, у них нет методов жизненного цикла. Поэтому давайте преобразуем компонент Search в функциональный. Преобразование компонентов Table и Button вы сделаете самостоятельно, в качестве упражнения.

src/App.js

```
function Search(props) {
  const { value, onChange, children } = props;
  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}
      />
    </form>
  );
}
```

Вот в принципе и всё. `props` доступны в объявлении функции, функция возвращает JSX. Однако кое-что ещё можно улучшить. Так как вы уже знаете деструктуризацию, и поэтому довольно эффективным методом является её использование в объявлении функции.

src/App.js

```
function Search({ value, onChange, children }) {
  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}
      />
    </form>
  );
}
```

Стало лучше, но нет предела совершенству. Вы уже знаете, что стрелочные функции из ES6 позволяют писать функции короче. Вы можете удалить тело блока функции. В стрелочных

функциях подразумевается неявный возврат, поэтому вы можете удалить выражение `return`. Поскольку ваш функциональный компонент без состояния — это функция, вы можете сделать текущий код функции более кратким.

src/App.js

```
const Search = ({ value, onChange, children }) =>
  <form>
    {children} <input
      type="text"
      value={value}
      onChange={onChange}>
    />
  </form>
```

Последний шаг был особенно полезен для обеспечения того, что только свойства передаются как входные данные, а JSX возвращается в качестве вывода. Между ними нет ничего. Тем не менее, вы можете *что-то сделать* между ними, используя тело блока в вашей стрелочной функции.

Код

```
const Search = ({ value, onChange, children }) => {
  // что-нибудь делать

  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}>
      />
    </form>
  );
}
```

Но пока нам это не нужно. Поэтому можно сохранить предыдущую версию без тела блока. При использовании блоков с телом кода, люди часто склонны делать много лишнего в функции. Удалив блок с кодом, вы можете сосредоточиться на входных данных и выводе вашей функции.

К этому времени у вас есть один лёгкий функциональный компонент без состояния. Как только вам понадобится внутреннее состояние или методы жизненного цикла, вы всегда можете обратиться к классовым компонентам. Кроме того, вы видели, как JavaScript ES6 может использоваться в компонентах React для того, чтобы сделать их краткими и элегантными.

Упражнения:

- преобразуйте компоненты Table и Button в функциональные компоненты без состояния
- ознакомьтесь подробнее с [классовыми компонентами и функциональными компонентами без состояния](#)⁹⁵

⁹⁵<https://ru.react.js.org/docs/components-and-props.html>

СТИЛИЗАЦИЯ КОМПОНЕНТОВ

Давайте добавим базовые стили к вашему приложению и компонентам. Вы можете повторно использовать файлы `src/App.css` и `src/index.css`. Эти файлы уже должны быть в вашем проекте, потому что вы начали проект с помощью `create-react-app`. Они также должны быть импортированы в файлах `src/App.js` и `src/index.js`. Я подготовил CSS-код, который вы можете просто скопировать и вставить в эти файлы, но не стесняйтесь использовать собственные стили.

Для начала добавим стили для всего приложения.

`src/index.css`

```
body {  
  color: #222;  
  background: #f4f4f4;  
  font: 400 14px CoreSans, Arial, sans-serif;  
}  
  
a {  
  color: #222;  
}  
  
a:hover {  
  text-decoration: underline;  
}  
  
ul, li {  
  list-style: none;  
  padding: 0;  
  margin: 0;  
}  
  
input {  
  padding: 10px;  
  border-radius: 5px;  
  outline: none;  
  margin-right: 10px;  
  border: 1px solid #dddddd;  
}  
  
button {  
  padding: 10px;  
  border-radius: 5px;  
  border: 1px solid #dddddd;
```

```
background: transparent;
color: #808080;
cursor: pointer;
}

button:hover {
  color: #222;
}

*:focus {
  outline: none;
}
```

Теперь перейдём к стилизации в файле App.

src/App.css

```
.page {
  margin: 20px;
}

.interactions {
  text-align: center;
}

.table {
  margin: 20px 0;
}

.table-header {
  display: flex;
  line-height: 24px;
  font-size: 16px;
  padding: 0 10px;
  justify-content: space-between;
}

.table-empty {
  margin: 200px;
  text-align: center;
  font-size: 16px;
}

.table-row {
```

```
display: flex;
line-height: 24px;
white-space: nowrap;
margin: 10px 0;
padding: 10px;
background: #ffffff;
border: 1px solid #e3e3e3;
}

.table-header > span {
  overflow: hidden;
  text-overflow: ellipsis;
  padding: 0 5px;
}

.table-row > span {
  overflow: hidden;
  text-overflow: ellipsis;
  padding: 0 5px;
}

.button-inline {
  border-width: 0;
  background: transparent;
  color: inherit;
  text-align: inherit;
  -webkit-font-smoothing: inherit;
  padding: 0;
  font-size: inherit;
  cursor: pointer;
}

.button-active {
  border-radius: 0;
  border-bottom: 1px solid #38BB6C;
}
```

Теперь у нас всё готово для стилизации компонентов. Не забудьте использовать `className` вместо `class` в качестве HTML-атрибута.

Во-первых, добавим классы в классе-компоненте `App`.

src/App.js

```
class App extends Component {  
  ...  
  
  render() {  
    const { searchTerm, list } = this.state;  
    return (  
      <div className="page">  
        <div className="interactions">  
          <Search  
            value={searchTerm}  
            onChange={this.onSearchChange}  
          >  
          Поиск  
        </Search>  
      </div>  
      <Table  
        list={list}  
        pattern={searchTerm}  
        onDismiss={this.onDismiss}  
      />  
    </div>  
  );  
}  
}
```

Во-вторых, сделаем аналогичное для функционального компонента без состояния Table.

src/App.js

```
const Table = ({ list, pattern, onDismiss }) =>  
  <div className="table">  
    {list.filter(isSearched(pattern)).map(item =>  
      <div key={item.objectID} className="table-row">  
        <span>  
          <a href={item.url}>{item.title}</a>  
        </span>  
        <span>{item.author}</span>  
        <span>{item.num_comments}</span>  
        <span>{item.points}</span>  
        <span>  
          <Button  
            ...>  
        </span>  
      </div>  
    )}  
  </div>
```

```
        onClick={() => onDismiss(item.objectID)}
        className="button-inline"
      >
    Отбросить
  </Button>
</span>
</div>
)
</div>
```

На данный момент вы стилизовали приложение и компоненты с помощью простого CSS. Всё это вместе должно выглядеть довольно прилично. Как вы знаете, JSX смешивает HTML и JavaScript. Конечно, можно предположить добавление CSS к этому сочетанию. Это называется встроенными стилями. Вы можете определить JavaScript-объекты и передать их в атрибуте `style` элемента.

Давайте сделаем фиксированную ширину столбцов в компоненте `Table`, используя встроенные стили.

src/App.js

```
const Table = ({ list, pattern, onDismiss }) =>
  <div className="table">
    {list.filter(isSearched(pattern)).map(item =>
      <div key={item.objectID} className="table-row">
        <span style={{ width: '40%' }}>
          <a href={item.url}>{item.title}</a>
        </span>
        <span style={{ width: '30%' }}>
          {item.author}
        </span>
        <span style={{ width: '10%' }}>
          {item.num_comments}
        </span>
        <span style={{ width: '10%' }}>
          {item.points}
        </span>
        <span style={{ width: '10%' }}>
          <Button
            onClick={() => onDismiss(item.objectID)}
            className="button-inline"
          >
            Отбросить
          </Button>
        </span>
      </div>
    )
  </div>
```

```
    </span>
  </div>
)
</div>
```

Так, стиль добавлен. Вы можете определить объекты со стилем вне элементов для большей чистоты кода.

Код

```
const largeColumn = {
  width: '40%',
};

const midColumn = {
  width: '30%',
};

const smallColumn = {
  width: '10%',
};
```

Поступив подобным образом, вы можете использовать их в столбцах так: ``.

В общем, вы увидите различные мнения и решения для стилизации в React. Сейчас вы использовали чистый CSS и встроенные стили. Это достаточно удобно для старта.

Я не хочу быть слишком самоуверенным, но хочу оставить вам ещё несколько альтернативных вариантов. Вы можете прочитать о них и применить самостоятельно. Но если вы новичок в React, я бы порекомендовал придерживаться чистого CSS и встроенных стилей.

- [styled-components⁹⁶](#)
- [CSS Modules⁹⁷](#)

⁹⁶<https://github.com/styled-components/styled-components>

⁹⁷<https://github.com/css-modules/css-modules>

Вы изучили основы написания собственного React-приложения! Давайте повторим последние темы:

- React
 - использование `this.state` и `setState()` для управления внутренним состоянием компонента
 - передача функций или методов класса элементу обработчика
 - использование форм и событий в React для добавления взаимодействий
 - односторонний поток данных — важная концепция в React
 - постигли контролируемые компоненты
 - составлять компоненты с дочерними компонентами и повторно используемыми компонентами
 - использование и реализация классовых компонентов и функциональных компонентов без состояния
 - подходы к стилизации компонентов
- ES6
 - функции, связанные с классом — это методы класса
 - деструктуризация объектов и массивов
 - параметры по умолчанию
- Общее
 - функции высшего порядка

Опять же имеет смысл сделать перерыв. Усвоить полученные значения и применить их самостоятельно на практике. Вы можете поэкспериментировать с исходным кодом, написанным в течение этой главы, его можно в [официальном репозитории](#)⁹⁸.

⁹⁸<https://github.com/the-road-to-learn-react/hackernews-client/tree/5.2>

Получение реальных данных с API

Теперь пришло время получать реальные данные через API, поскольку может быть скучно работать с демонстрационными данными.

Если вы не знакомы с API, я рекомендую вам прочитать [моё путешествие по знакомству с API⁹⁹](#).

Вы знаете платформу [Hacker News¹⁰⁰](#)? Это отличный новостной агрегатор на технические темы. В этой книге мы будем использовать API Hacker News для получения популярных (трендовых) историй. Для того, чтобы это сделать есть [базовый¹⁰¹](#) и [поисковый¹⁰²](#) API для получения данных с этой платформы. Последний предназначен для нашего разрабатываемого приложения для поиска историй на Hacker News. Вы можете открыть спецификацию API для понимания структуры данных.

⁹⁹<https://www.robinwieruch.de/what-is-an-api-javascript/>

¹⁰⁰<https://news.ycombinator.com/>

¹⁰¹<https://github.com/HackerNews/API>

¹⁰²<https://hn.algolia.com/api>

Методы жизненного цикла

Вам нужно будет узнать о методах жизненного цикла React до того, как вы начнёте получать данные в ваших компонентах с использованием API. Эти методы — хуки в жизненном цикле React-компонента. Их можно использовать в ES6-классе компонента, но не в функциональных компонентах без состояния.

Помните, в предыдущей главе вы изучали классы в JavaScript ES6 и их использование в React? Помимо метода `render()` существует несколько методов, которые переопределются в классовых компонентах React. Всё это методы жизненного цикла, давайте погрузимся в них.

Вам уже знакомы два метода жизненного цикла, которые можно использовать в классе компонента: `constructor()` и `render()`.

Конструктор вызывается только тогда, когда экземпляр компонента создан и добавлен в DOM, т.е. компонент проинициализирован. Этот процесс называется монтированием (установкой) компонента.

Метод `render()` также вызывается во время процесса монтирования, но ещё и при обновлении компонента. Каждый раз при изменении состояния или свойств компонента, вызывается метод `render()`.

Теперь вы узнали больше об этих двух методах жизненного цикла и о том, в каких случаях они вызываются. Вы их уже использовали, но есть и другие методы, которые мы сейчас рассмотрим.

У монтирования компонента есть два метода жизненного цикла: `getDerivedStateFromProps()` и `componentDidMount()`. Сначала вызывается конструктор, метод `getDerivedStateFromProps()` вызывается до метода `render()` и `componentDidMount()` вызывается после метода `render()`.

В целом процесс монтирования имеет 4 метода жизненного цикла. Они вызываются в следующем порядке:

- `constructor()`
- `getDerivedStateFromProps()`
- `render()`
- `componentDidMount()`

Но как насчёт обновления жизненного цикла компонента, которое происходит при изменении состояния или свойств?

- `getDerivedStateFromProps()`
- `shouldComponentUpdate()`
- `componentWillUpdate()`
- `render()`

- `getSnapshotBeforeUpdate()`
- `componentDidUpdate()`

И последнее, но не менее важное — жизненный цикл размонтирования. Для этого процесса доступен только один метод — `componentWillUnmount()`.

В конце концов, вам не нужно знать все эти методы жизненного цикла с самого начала. Это может быть пугающим, тем более не все вы будете использовать. Даже в крупном React-приложении вы будете использовать только некоторые из них, помимо методов `constructor()` и `render()`. Тем не менее, хорошо знать, какой метод жизненного цикла в каких случаях может использоваться:

- **`constructor(props)`** — вызывается, когда компонент инициализируется. Вы можете установить начальное состояние компонента и связать методы класса в этом методе жизненного цикла.
- **`static getDerivedStateFromProps(props, state)`** — вызывается перед методом жизненного цикла `render()` как при начальном монтировании, так и при последующих обновлениях. Этот метод должен возвратить объект для обновления состояния или `null`, если ничего не нужно обновлять. Он существует для **редких** случаев использования, когда состояние зависит от изменений в свойствах со временем. Важно знать, что это статический метод, и поэтому у него нет доступа к экземпляру компонента.
- **`render()`** — требуется обязательно и возвращает элементы в качестве вывода компонента. Метод должен быть чистым, т.е. не изменять состояние компонента. Он получает входные данные в виде свойств и состояния и возвращает элемент.
- **`componentDidMount()`** — вызывается только один раз при монтировании компонента. Это идеальный момент для выполнения асинхронных запросов на получение данных из API. Полученные данные будут сохранены во внутреннем состоянии компонента для его отображения в методе жизненного цикла `render()`.
- **`shouldComponentUpdate(nextProps, nextState)`** — всегда вызывается, когда компонент изменяется во время изменения состояния или свойств. Вы будете использовать его в зрелых React-приложениях для оптимизации производительности. В зависимости от возвращаемого из этого метода логического значения компонент и все его дочерние элементы либо будут перерисовываться, либо нет. Этот метод может предотвратить отрисовку компонента.
- **`getSnapshotBeforeUpdate(prevProps, prevState)`** — вызывается непосредственно перед тем, как последний отрисованный вывод будет зафиксирован в DOM. В редких вариантах использования компонент должен захватить информацию из DOM прежде чем он в принципе будет изменён. Этот метод жизненного цикла позволяет компоненту это сделать. Другой метод (`componentDidUpdate()`) получит любое значение, возвращаемое `getSnapshotBeforeUpdate()` в качестве параметра.
- **`componentDidUpdate(prevProps, prevState)`** — вызывается сразу после обновления, но не при первоначальной отрисовке метода `render()`. Вы можете использовать его как возможность выполнять операции с DOM или дальнейшие асинхронные запросы. Если

ваш компонент реализует метод `getSnapshotBeforeUpdate()`, возвращаемое им значение будет приниматься в качестве параметра `snapshot`.

- `componentWillUnmount()` — вызывается до того, как компонент будет уничтожен (удалён из DOM). Вы можете использовать этот метод для выполнения любых задач очистки.

Вы уже использовали методы `constructor()` и `render()`. Это часто используемые методы жизненного цикла в классовых компонентах. На самом деле только метод `render()` является обязательным, в противном случае вы не возвратите экземпляр компонента.

Существует ещё один метод жизненного цикла — `componentDidCatch(error, info)`. Он представлен в [React 16¹⁰³](#) и используется для обработки ошибок в компонентах. К примеру, список хорошо отображается в приложении. Но что, если список в локальном состоянии случайно установлен в `null` (например, при получении данных из внешнего API запрос завершился неудачно, поэтому вы установили локальное состояние для списка в значение `null`). Впоследствии было невозможно фильтровать и отображать список, поскольку вместо элементов списка — значение `null`. Компонент будет сломан, и приложение полностью потерпит неудачу. Теперь, используя `componentDidCatch()`, вы можете перехватить ошибку, сохранить её в локальном состоянии и необязательно показать сообщение в приложении для уведомления пользователя об ошибке.

Упражнения:

- узнайте подробнее [методы жизненного цикла в React¹⁰⁴](#)
- узнайте подробнее о [состоянии, связанном с методами жизненного цикла в React¹⁰⁵](#)
- узнайте подробнее про [обработку ошибок в компонентах¹⁰⁶](#)

¹⁰³<https://www.robinwieruch.de/what-is-new-in-react-16/>

¹⁰⁴<https://ru.react.js.org/docs/react-component.html>

¹⁰⁵<https://ru.react.js.org/docs/state-and-lifecycle.html>

¹⁰⁶<https://reactjs.org/blog/2017/07/26/error-handling-in-react-16.html>

Получение данных

Теперь вы готовы извлечь данные из API Hacker News. Для этого можно воспользоваться ранее упомянутым методом жизненного цикла — `componentDidMount()`. Для выполнения запроса мы будем использовать нативный в JavaScript `fetch`.

Прежде чем мы будем его использовать, давайте настроим константы для URL-адреса и параметры по умолчанию, чтобы разбить запрос к API на куски.

`src/App.js`

```
import React, { Component } from 'react';
import './App.css';

const DEFAULT_QUERY = 'redux';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';

// ...
```

В JavaScript ES6 вы можете использовать [шаблонные строки](#)¹⁰⁷ для конкатенации строк. В вашем случае вы будете их использовать для конкатенации URL-адреса к конечной точке (endpoint) API.

Код

```
// ES6
const url = `${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${DEFAULT_QUERY}`;

// ES5
var url = PATH_BASE + PATH_SEARCH + '?' + PARAM_SEARCH + DEFAULT_QUERY;

console.log(url);
// вывод: https://hn.algolia.com/api/v1/search?query=redux
```

Это позволит сохранить гибкость структуры URL-адреса в будущем.

Но давайте перейдём к API-запросу, в котором будет использоваться URL-адрес. Весь процесс получения данных приводится сразу, но каждый шаг будет объяснён позже.

¹⁰⁷https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals

src/App.js

...

```
class App extends Component {  
  
  constructor(props) {  
    super(props);  
  
    this.state = {  
      result: null,  
      searchTerm: DEFAULT_QUERY,  
    };  
  
    this.setSearchTopStories = this.setSearchTopStories.bind(this);  
    this.onSearchChange = this.onSearchChange.bind(this);  
    this.onDismiss = this.onDismiss.bind(this);  
  }  
  
  setSearchTopStories(result) {  
    this.setState({ result });  
  }  
  
  componentDidMount() {  
    const { searchTerm } = this.state;  
  
    fetch(`#${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}`)  
      .then(response => response.json())  
      .then(result => this.setSearchTopStories(result))  
      .catch(error => error);  
  }  
  
  ...  
}
```

В коде много чего происходит. Я думал о том, чтобы разбить его на небольшие части. Тогда снова было бы трудно понять связь каждой части друг с другом. Позвольте мне подробно объяснить каждый шаг.

Во-первых, вы можете удалить демонстрационный список элементов, потому что вы возвращаете реальный список из API Hacker News. Демонстрационные данные больше не используются. Исходное состояние вашего компонента теперь имеет пустой результат и поисковый запрос по умолчанию. Тот же самый поисковый запрос по умолчанию используется в поле ввода компонента Search и в вашем первом запросе.

Во-вторых, вы используете метод жизненного цикла `componentDidMount()` для получения данных после монтирования компонента. При самом первом получении используется поисковый запрос по умолчанию из локального состояния. Он будет получать истории, связанные с “redux”, поскольку это параметр по умолчанию.

В-третьих, используется нативный API. Шаблонные строки в JavaScript ES6 позволяют ему составлять URL-адрес с `searchTerm`. URL-адрес — аргумент для нативной функции API `fetch`. Ответ требуется преобразовать в структуру данных JSON, что является обязательным шагом в нативной функции `fetch` при работе с JSON-данными, и, наконец, они могут быть сохранены в качестве результата во внутреннем состоянии компонента. Кроме того, блок `catch` используете в случае ошибки. Если во время выполнения запроса произошла ошибка, выполнение функции перейдёт из блока `then` в блок `catch`. В следующей главе книги будет включена обработка ошибок.

И последнее, но не менее важно: не забудьте связать новый метод компонента в конструкторе.

Теперь вы можете использовать полученные данные вместо демонстрационного списка элементов. Однако нужно быть осторожным. Результат содержит не только список данных. Это сложный объект с метаинформацией и свойством `hits`, который содержит нужные нам истории¹⁰⁸. Вы можете вывести внутреннее состояние через `console.log(this.state)`; в методе для `render()` для наглядного отображения.

Следующим шагом будет использование результата для его отрисовки. Мы предотвратим отрисовку, вернув значение `null`, если данных с API нет в первый раз. После того, как запрос к API выполнен успешно, результат будет сохранён в состоянии и компонент App повторно отрисуется с обновлённым состоянием.

src/App.js

```
class App extends Component {  
  
  ...  
  
  render() {  
    const { searchTerm, result } = this.state;  
  
    if (!result) { return null; }  
  
    return (  
      <div className="page">  
        ...  
        <Table  
          list={result.hits}  
          pattern={searchTerm}>  
    );  
  }  
}
```

¹⁰⁸<https://hn.algolia.com/api>

```

        onDismiss={this.onDismiss}
    />
</div>
);
}
}

```

Давайте вспомним, что происходит во время жизненного цикла компонента. Компонент инициализируется через конструктор. После этого он отрисовывается в первый раз. Но вы препятствуете отрисовке чего-либо, поскольку результат в локальном состоянии равен `null`. Разрешено возвращать `null` из компонента, если нет данных для отображения. Затем выполняется метод жизненного цикла `componentDidMount()`. В этом методе вы асинхронно получаете данные из API Hacker News. Как только данные придут, изменится внутреннее состояние компонента в `setSearchTopStories()`. Поскольку локальное состояние обновлено, вступает в действие обновление жизненного цикла. Компонент снова выполняет метод `render()`, но на этот раз с заполненными данными результата во внутреннем состоянии компонента. Компонент, и таким образом, компонент `Table` с его содержимым будут отрисовываться.

Вы использовали нативный API `fetch`, поддерживаемый большинством браузеров для выполнения асинхронных запросов. Конфигурация `create-react-app` гарантирует его поддержку в каждом браузере. Существуют также сторонние пакеты, которыми можно заменить нативный API `fetch`: [superagent¹⁰⁹](#) и [axios¹¹⁰](#).

Имейте в виду, что в книге используется сокращённая нотация JavaScript для проверки истинности. В предыдущем примере `if (!result)` вместо `if (result === null)`. То же самое относится и к другим частям примеров на протяжении всей книги. Например, используете `if (!list.length)` вместо `if (list.length === 0)` или `if (someString)` вместо `if (someString !== '')`. Почитайте на эту тему, если вы не слишком знакомы с этим.

Вернёмся к нашему приложению: показывается список историй. Однако сейчас в приложении есть два регрессионных бага. Во-первых, кнопка “Отбросить” не работает. Она не знает о сложном объекте результата и по-прежнему работает на обычном списке из локального состояния при отклонении элемента. Во-вторых, при попытке поиска в отображённом списке, он фильтруется на стороне клиента, даже если изначальное получение данных было выполнено посредством поиска историй на стороне сервера. Идеально было бы при использовании компонента `Search` получать другой объект результата из API. Обе ошибки регрессии будут исправлены в следующих главах.

Упражнения:

- узнайте подробнее [строковые шаблоны ES6¹¹¹](#)

¹⁰⁹<https://github.com/visionmedia/superagent>

¹¹⁰<https://github.com/mzabriskie/axios>

¹¹¹https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals

- узнайте подробнее [нативный API fetch¹¹²](https://developer.mozilla.org/ru/docs/Web/API/Fetch_API)
- узнайте подробнее [получение данных в React¹¹³](https://www.robinwieruch.de/react-fetching-data/)

¹¹²https://developer.mozilla.org/ru/docs/Web/API/Fetch_API

¹¹³<https://www.robinwieruch.de/react-fetching-data/>

Оператор расширения ES6

Кнопка “Отбросить” не работает, потому что метод `onDismiss()` не знает о сложном результирующем объекте. Она знает только про простой список в локальном состоянии. Но это уже не простой список. Давайте изменим его для работы с результирующим объектом вместо самого списка.

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
    ...
  });
}
```

Но что теперь происходит в `setState()`? К сожалению, результат — сложный объект. Список хитов — это только одно из нескольких свойств объекта. Тем не менее, только список обновляется, когда элемент удаляется в результирующем объекте, тогда как остальные свойства остаются такими же.

Один из подходов может заключаться в том, чтобы изменять свойство `hits` в результирующем объекте. Я продемонстрирую это, но мы не будем так делать.

Код

```
// не делайте этого
this.state.result.hits = updatedHits;
```

React охватывает неизменяемые структуры данных. Таким образом, вы не должны изменять объект (или напрямую изменить состояние). Лучший подход — создать новый объект на основе имеющейся информации. Таким образом, ни один из объектов не будет изменён. Вы сохраните неизменяемые структуры данных. Вы всегда будете возвращать новый объект, а не изменять объект.

Поэтому вы можете использовать `Object.assign()` в JavaScript ES6. Он принимает в качестве первого аргумента целевой объект. Все следующие аргументы — исходные объекты. Эти объекты объединяются в целевой объект. Целевой объект может быть пустым. Метод `Object.assign()` удовлетворяет принципу неизменяемости, так что ни один из исходных объектов не изменяется. В итоге код будет выглядеть следующим образом:

Код

```
const updatedHits = { hits: updatedHits };
const updatedResult = Object.assign({}, this.state.result, updatedHits);
```

Последующие объекты будут переопределять прежние объединённые объекты, если у них одинаковые имена свойств. Теперь давайте применим его в методе onDismiss():

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
    result: Object.assign({}, this.state.result, { hits: updatedHits })
  });
}
```

Мы эту функцию уже использовали. Но в JavaScript ES6 и будущих релизах JavaScript есть более простой способ. Могу ли я представить вам оператор расширения? Он состоит из трёх точек: При его использовании каждое значение из массива или объекта копируется в другой массив или объект.

Давайте изучим оператор расширения **массива** ES6, даже если он вам ещё не нужен.

Код

```
const userList = ['Robin', 'Andrew', 'Dan'];
const additionalUser = 'Jordan';
const allUsers = [ ...userList, additionalUser ];

console.log(allUsers);
// выведет ['Robin', 'Andrew', 'Dan', 'Jordan']
```

Переменная allUsers — это полностью новый массив. Остальные переменные userList и additionalUser остаются такими же. Вы можете объединить два массива таким образом в новый массив.

Код

```
const oldUsers = ['Robin', 'Andrew'];
const newUsers = ['Dan', 'Jordan'];
const allUsers = [ ...oldUsers, ...newUsers ];

console.log(allUsers);
// выведет: ['Robin', 'Andrew', 'Dan', 'Jordan']
```

Теперь давайте посмотрим на оператор расширения объекта. Это не JavaScript ES6. Это предложение для следующей версии JavaScript¹¹⁴, которое уже используется сообществом React. Именно поэтому *create-react-app* включило эту возможность в свою конфигурацию.

В основном это то же самое, что и оператор расширения массива в JavaScript ES6, но с объектами. Он копирует каждую пару ключ-значение в новый объект.

Код

```
const userNames = { firstname: 'Robin', lastname: 'Wieruch' };
const age = 28;
const user = { ...userNames, age };

console.log(user);
// выведет: { firstname: 'Robin', lastname: 'Wieruch', age: 28 }
```

Может использоваться несколько объектов, как в аналогичном примере с массивом.

Код

```
const userNames = { firstname: 'Robin', lastname: 'Wieruch' };
const userAge = { age: 28 };
const user = { ...userNames, ...userAge };

console.log(user);
// выведет: { firstname: 'Robin', lastname: 'Wieruch', age: 28 }
```

В конце концов его можно использовать для замены `Object.assign()`.

¹¹⁴<https://github.com/sebmarkbage/ecmascript-rest-spread>

src/App.js

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
    result: { ...this.state.result, hits: updatedHits }
  });
}
```

Теперь кнопка “Отбросить” должна снова работать, потому что метод `onDismiss()` знает о сложном результирующем объекте и о том, как его обновлять после отклонения элемента из списка.

Упражнения:

- узнайте получше `Object.assign()` из ES6¹¹⁵
- узнайте подробнее `оператор расширения массива ES6`¹¹⁶
 - оператор расширения объектов кратко упоминался уже

¹¹⁵https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/assign

¹¹⁶https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Spread_operator

Отрисовка по условию

Условная отрисовка применяется довольно рано в React-приложении. Но не в этой книге, потому что не было такого случая использования. Условная отрисовка происходит, когда вы хотите решить — отрисовать тот или иной элемент, или нет. Иногда это означает отрисовать либо элемент, либо вообще ничего. В конце концов, упрощённое использование условной отрисовки может быть сформулировано выражением if-else в JSX.

Объект `result` во внутреннем состоянии объекта равен значению `null` в самом начале. До сих пор компонент `App` не возвращал элементов, когда `result` не был заполнен с API. Это уже условная отрисовка, потому что из метода жизненного цикла `render()` заранее возвращается что-то в зависимости от определённого условия. Компонент `App` либо отрисовывает свои элементы, либо ничего не отрисовывает.

Но давайте пойдём дальше. Имеет смысл обернуть компонент `Table`, единственный компонент, который зависит от `result`, в независимую условную отрисовку. Всё остальное должно отображаться, даже если пока `result` ещё нет. Вы можете просто использовать тернарный оператор в JSX.

src/App.js

```
class App extends Component {  
  ...  
  
  render() {  
    const { searchTerm, result } = this.state;  
    return (  
      <div className="page">  
        <div className="interactions">  
          <Search  
            value={searchTerm}  
            onChange={this.onSearchChange} />  
          Поиск  
        </Search>  
      </div>  
      { result ? <Table  
        list={result.hits}  
        pattern={searchTerm}  
        onDismiss={this.onDismiss} />  
        : null  
    )  
  }  
}
```

```
        </div>
    );
}
}
```

Это второй вариант выражения условной отрисовки. Третий вариант заключается в логическом операторе `&&`. В JavaScript выражение `true && 'Привет, мир'` всегда выполняется как '`Привет, мир`'. А выражение `false && 'Привет, мир'` всегда будет `false`.

Код

```
const result = true && 'Привет, мир';
console.log(result);
// выведет: Привет, мир

const result = false && 'Привет, мир';
console.log(result);
// выведет: false
```

В React вы можете использовать подобное поведение. Если условие равняется `true`, выражение после оператора `&&` будет выведено. Если условие равняется `false`, React проигнорирует и пропустит выражение. Это подходит в случае условной отрисовки компонента `Table`, поскольку он должен возвращать `Table`, а может его не вернуть.

src/App.js

```
{ result &&
  <Table
    list={result.hits}
    pattern={searchTerm}
    onDismiss={this.onDismiss}
  />
}
```

Сейчас были показаны несколько подходов использования условной отрисовки в React. Вы можете узнать [больше о других альтернативах в исчерпывающем списке примеров условной отрисовки¹¹⁷](#). Кроме того, вы узнаете об их различных вариантах использования и применения их.

В конце концов, вы должны иметь возможность видеть полученные данные в приложении. Когда данные находятся только в процессе получения, отображается всё, кроме таблицы. После того как результат будет получен из запроса и сохранён в локальном состоянии, будет отображена таблица, потому что метод `render()` запускается снова, и условие разрешается в пользу отображения компонента таблицы.

¹¹⁷<https://www.robinwieruch.de/conditional-rendering-react/>

Упражнения:

- ознакомьтесь подробнее с различными способами условной отрисовки¹¹⁸
- узнайте больше про отрисовку по условию в React¹¹⁹

¹¹⁸<https://www.robinwieruch.de/conditional-rendering-react/>

¹¹⁹<https://ru.react.js.org/docs/conditional-rendering.html>

Поиск на стороне клиента и на стороне сервера

Теперь, когда вы используете компонент поиска со своим полем ввода, пришло время фильтровать список. Сейчас это происходит на стороне клиента, а теперь вы будете использовать фильтрацию на стороне сервера. В противном случае вы будете иметь дело только с первым запросом, который вы сделали в методе `componentDidMount()` с параметром поиска по умолчанию.

Вы можете определить метод `onSearchSubmit()` в компонента App, который получает результаты из Hacker News API при выполнении поиска в компоненте Search.

src/App.js

```
class App extends Component {  
  
  constructor(props) {  
    super(props);  
  
    this.state = {  
      result: null,  
      searchTerm: DEFAULT_QUERY,  
    };  
  
    this.setSearchTopStories = this.setSearchTopStories.bind(this);  
    this.onSearchChange = this.onSearchChange.bind(this);  
    this.onSearchSubmit = this.onSearchSubmit.bind(this);  
    this.onDismiss = this.onDismiss.bind(this);  
  }  
  
  ...  
  
  onSearchSubmit() {  
    const { searchTerm } = this.state;  
  }  
  
  ...  
}
```

Метод `onSearchSubmit()` должен использовать ту же функциональность, что и метод жизненного цикла `componentDidMount()`, но на этот раз с изменённым поисковым запросом из локального состояния, а не с начальным запросом поиска по умолчанию. Таким образом, вы можете извлечь функциональность как метод класса для повторного использования.

src/App.js

```
class App extends Component {  
  
  constructor(props) {  
    super(props);  
  
    this.state = {  
      result: null,  
      searchTerm: DEFAULT_QUERY,  
    };  
  
    this.setSearchTopStories = this.setSearchTopStories.bind(this);  
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);  
    this.onSearchChange = this.onSearchChange.bind(this);  
    this.onSearchSubmit = this.onSearchSubmit.bind(this);  
    this.onDismiss = this.onDismiss.bind(this);  
  }  
  
  ...  
  
  fetchSearchTopStories(searchTerm) {  
    fetch(`#${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}`)  
      .then(response => response.json())  
      .then(result => this.setSearchTopStories(result))  
      .catch(error => error);  
  }  
  
  componentDidMount() {  
    const { searchTerm } = this.state;  
    this.fetchSearchTopStories(searchTerm);  
  }  
  
  ...  
  
  onSearchSubmit() {  
    const { searchTerm } = this.state;  
    this.fetchSearchTopStories(searchTerm);  
  }  
  
  ...  
}
```

Теперь в компоненте Search нужно добавить дополнительную кнопку. Кнопка должна явно вызывать запрос на поиск. В противном случае вы будете получать данные с API Hacker News каждый раз, когда изменяется поле ввода. Скорее всего вы захотите это сделать в обработчике onClick().

В качестве альтернативы вы можете отложить (задержать выполнение) функцию onChange() и обойтись без этой кнопки. Но это добавит больше сложности и, возможно, не будет иметь желаемого эффекта. Давайте оставим простой вариант без задержки.

Сначала передадим метод onSearchSubmit() в компонент Search.

src/App.js

```
class App extends Component {  
  
  ...  
  
  render() {  
    const { searchTerm, result } = this.state;  
    return (  
      <div className="page">  
        <div className="interactions">  
          <Search  
            value={searchTerm}  
            onChange={this.onSearchChange}  
            onsubmit={this.onSearchSubmit}  
          >  
            Поиск  
          </Search>  
        </div>  
        { result &&  
          <Table  
            list={result.hits}  
            pattern={searchTerm}  
            onDismiss={this.onDismiss}  
          />  
        }  
      </div>  
    );  
  }  
}
```

Во-вторых, создадим кнопку в компоненте Search. У кнопки будет атрибут type="submit", а у формы будет свой атрибут onSubmit для передачи метода onSearchSubmit(). Вы можете повторно использовать свойство children, но на этот раз в нём будет содержаться текст кнопки.

src/App.js

```
const Search = ({  
  value,  
  onChange,  
  onSubmit,  
  children  
) =>  
  <form onSubmit={onSubmit}>  
    <input  
      type="text"  
      value={value}  
      onChange={onChange}  
    />  
    <button type="submit">  
      {children}  
    </button>  
  </form>
```

В компоненте Table можно удалить функциональность фильтра, потому что больше не будет фильтра (поиска) на стороне клиента. Не забудьте также удалить функцию isSearched(): она больше не будет использоваться. Результат приходит непосредственно из API Hacker News после того, как была нажата кнопка “Поиск”.

src/App.js

```
class App extends Component {  
  
  ...  
  
  render() {  
    const { searchTerm, result } = this.state;  
    return (  
      <div className="page">  
        ...  
        { result &&  
          <Table  
            list={result.hits}  
            onDismiss={this.onDismiss}  
          />  
        }  
      </div>  
    );  
  }  
}
```

```
}

...

const Table = ({ list, onDismiss }) =>
  <div className="table">
    {list.map(item =>
      ...
    )}
  </div>
```

Если вы попытаетесь выполнить данный код сейчас, вы заметите, что браузер перезагружается. Это стандартное поведение браузера для колбэка при отправки формы HTML. В React вы часто столкнётесь с методом события `preventDefault()` для предотвращения нативного поведения браузера.

src/App.js

```
onSearchSubmit(event) {
  const { searchTerm } = this.state;
  this.fetchSearchTopStories(searchTerm);
  event.preventDefault();
}
```

Теперь вы сможете искать различные истории по Hacker News. Прекрасно, сейчас вы взаимодействуете с реальным API. Больше не должно быть поиска на стороне клиента.

Упражнения:

- узнайте больше [синтетические события в React](#)¹²⁰
- поэкспериментируйте с [API Hacker News](#)¹²¹

¹²⁰<https://ru.react.js.org/docs/events.html>

¹²¹<https://hn.algolia.com/api>

Получение данных с разбивкой на страницы

Вы внимательно изучили структуру возвращаемых данных до сих пор? [API Hacker News¹²²](#) возвращает больше данных, чем просто список историй (hits). Как раз этот API возвращает список историй с разбивкой на страницы. Свойство page, равное 0 в первом ответе, может быть использовано для получения большего количества с делением на страницы в качестве результата. Вам нужно передать следующую страницу с тем же самым поисковым запросом к API.

Давайте расширим вспомогательные константы для API, чтобы мы могли работать с данными, разбитыми на страницы.

src/App.js

```
const DEFAULT_QUERY = 'redux';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';
const PARAM_PAGE = 'page=';
```

Теперь можно использовать новую константу для добавления параметра страницы к запросу API.

Код

```
const url = `${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${DEFAULT_QUERY}&${PARAM_PAGE}\`;  
console.log(url);  
// выведет: https://hn.algolia.com/api/v1/search?query=redux&page=
```

Метод `fetchSearchTopStories()` принимает страницу в качестве второго аргумента. Если второй аргумент не предоставлен, будет использоваться страница 0 для первоначального запроса. Таким образом, методы `componentDidMount()` и `onSearchSubmit()` извлекают первую страницу по первому запросу. Каждая дополнительная выборка будет отображать следующую страницу, передавая её вторым аргументом.

¹²²<https://hn.algolia.com/api>

src/App.js

```
class App extends Component {  
  
  ...  
  
  fetchSearchTopStories(searchTerm, page = 0) {  
    fetch(`#${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${page}`)  
      .then(response => response.json())  
      .then(result => this.setSearchTopStories(result))  
      .catch(error => error);  
  }  
  
  ...  
  
}
```

Аргумент страницы использует значения параметров по умолчанию из JavaScript ES6, чтобы вернуться к странице 0 в случае, если функции не предоставлен аргумент страницы.

Теперь мы знаем, как получить текущую страницу из ответа API в `fetchSearchTopStories()`. Вы можете использовать этот метод при нажатии на кнопку, чтобы получить ещё больше историй в обработчике кнопки `onClick`. Давайте воспользуемся компонентом `Button` для выборки большего количества данных с разбивкой на страницы из API Hacker News. Вам нужно только определить обработчик `onClick()`, который принимает текущий поисковый запрос пользователя и следующую страницу (текущая страница + 1).

src/App.js

```
class App extends Component {  
  
  ...  
  
  render() {  
    const { searchTerm, result } = this.state;  
    const page = (result && result.page) || 0;  
    return (  
      <div className="page">  
        <div className="interactions">  
          ...  
          { result &&  
            <Table  
              list={result.hits}  
              onDismiss={this.onDismiss}>
```

```
        />
    }
    <div className="interactions">
      <Button onClick={() => this.fetchSearchTopStories(searchTerm, page + 1)}>
        Больше историй
      </Button>
    </div>
  </div>
);
}
}
```

Кроме того, в методе `render()` вам надо убедиться, что по умолчанию страница 0, когда результата пока нет. Помните, что метод `render()` вызывается до того, как данные будут извлечены асинхронно в методе жизненного цикла `componentDidMount()`.

Но мы пропустили ещё один шаг. Вы получаете следующую страницу данных, но она не переопределяет предыдущую страницу данных. Было бы идеально объединять старый и новый список историй из локального состояния и нового объекта результата. Давайте скорректируем функциональность добавления новых данных, вместо того чтобы переопределять их.

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;

  const oldHits = page !== 0
    ? this.state.result.hits
    : [];

  const updatedHits = [
    ...oldHits,
    ...hits
  ];

  this.setState({
    result: { hits: updatedHits, page }
  });
}
```

Несколько моментов в методе `setSearchTopStories()` нужно отметить. Во-первых, вы получаете истории и текущую страницу из результата.

Во-вторых, вам нужно проверить, есть ли уже старые истории. Если страница равна 0, значит это новый поисковый запрос от `componentDidMount()` или `onSearchSubmit()`. Истории пустые. Но когда вы нажимаете на кнопку «Больше историй», чтобы получить больше данных с разбивкой по страницам, текущая страница больше не будет равняться 0. Это следующая страница. Старые истории уже хранятся в состоянии компонента и поэтому могут быть использованы.

В-третьих, вы не хотите переопределять старые истории. Вы можете объединить старые и новые истории из последнего запроса API. Объединение обоих списков может быть выполнено с помощью оператора расширения массива из JavaScript ES6.

В-четвёртых, вы сохранили объединённые истории и страницу в состоянии локального компонента.

Мы можем сделать последнюю правку. Когда вы пытаетесь нажать на кнопку «Больше историй», будут получены только несколько элементов списка. URL-адрес API может быть расширен для получения большего количества элементов списка с каждым запросом. Опять же, вы можете добавить дополнительные пути для запроса в виде констант.

src/App.js

```
const DEFAULT_QUERY = 'redux';
const DEFAULT_HPP = '100';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';
const PARAM_PAGE = 'page=';
const PARAM_HPP = 'hitsPerPage=';
```

Теперь вы можете использовать константы для расширения URL-адреса к API-запросу.

src/App.js

```
fetchSearchTopStories(searchTerm, page = 0) {
  fetch(`.${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${page}\n&${PARAM_HPP}${DEFAULT_HPP}`)
    .then(response => response.json())
    .then(result => this.setSearchTopStories(result))
    .catch(error => error);
}
```

После этого запрос к API Hacker News получает больше элементов списка за один запрос, чем раньше. Как вы можете видеть, мощный API, такой как API Hacker News, даёт вам множество способов экспериментировать с данными из реального мира. Вы должны это использовать, чтобы приложить усилия, когда узнаете что-нибудь новое, более захватывающее. Вот [как](#)

я узнал о расширении возможностей, предоставляемых API¹²³ при изучении нового языка программирования или библиотеки.

Упражнения:

- узнайте больше параметры по умолчанию из ES6¹²⁴
- поэкспериментируйте с параметрами API Hacker News¹²⁵

¹²³<https://www.robinwieruch.de/what-is-an-api-javascript/>

¹²⁴https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/Default_parameters

¹²⁵<https://hn.algolia.com/api>

Кеш клиента

При каждой отправке формы поиска отправляется запрос к API Hacker News. Вы можете поискать «redux», а затем «react», и в конце концов снова «redux». Таким образом, получаются 3 запроса. Но вы дважды искали «redux», и оба раза для получения данных потребовалось полное асинхронное получение данных в обе стороны. В кеше на стороне клиента вы сохраните каждый результат. Когда выполняется запрос к API, проверяется, есть ли уже результат. Если он есть, используется кеш. В противном случае для получения данных выполняется запрос к API.

Для обеспечения существования клиентского кеша для каждого результата, вам нужно сохранить несколько результатов (`results`), а не один результат (`result`) во внутреннем состоянии компонента. Объектом результатов будет объект с поисковой строкой в качестве ключа и результатом запроса в качестве значения. Каждый результат, полученный от API, будет сохранён с соответствующей поисковой строкой в качестве ключа.

На данный момент ваш результат в локальном состоянии выглядит примерно так:

Код

```
result: {  
  hits: [ ... ],  
  page: 2,  
}
```

Представьте, что вы сделали два запроса API. Один для поисковой строки «redux» и ещё один для «react». Объект результатов будет выглядеть следующим образом:

Код

```
results: {  
  redux: {  
    hits: [ ... ],  
    page: 2,  
  },  
  react: {  
    hits: [ ... ],  
    page: 1,  
  },  
  ...  
}
```

Давайте реализуем кеширование на стороне клиента с помощью `setState()`. Во-первых, переименуйте объект `result` в `results` в начальном состоянии компонента. Во-вторых, определите временный `searchKey`, который используется для хранения каждого результата (`result`).

src/App.js

```
class App extends Component {  
  
  constructor(props) {  
    super(props);  
  
    this.state = {  
      results: null,  
      searchKey: '',  
      searchTerm: DEFAULT_QUERY,  
    };  
  
    ...  
  
  }  
  
  ...  
  
}
```

Ключ `searchKey` должен быть установлен перед каждым запросом. Он отражает `searchTerm`. Вы могли бы задаться вопросом: почему мы не используем `searchTerm` в первую очередь? Это важно понять, прежде чем продолжать реализацию. `searchTerm` — это неустойчивая (временная) переменная, поскольку она изменяется каждый раз при вводе в поле поиска. Однако в конце вам понадобится постоянная переменная. Она определяет недавно отправленный поисковый запрос к API и может использоваться для получения правильного результата из объекта с результатами. Это указатель на ваш текущий результат в кеше и, следовательно, её можно использовать для отображения текущего результата в методе `render()`.

src/App.js

```
componentDidMount() {  
  const { searchTerm } = this.state;  
  this.setState({ searchKey: searchTerm });  
  this.fetchSearchTopStories(searchTerm);  
}  
  
onSearchSubmit(event) {  
  const { searchTerm } = this.state;  
  this.setState({ searchKey: searchTerm });  
  this.fetchSearchTopStories(searchTerm);  
  event.preventDefault();  
}
```

Теперь вам нужно подправить код для сохранения результата во внутреннее состояние компонента. Он должен хранить каждый результат с использованием ключа searchKey.

src/App.js

```
class App extends Component {  
  
  ...  
  
  setSearchTopStories(result) {  
    const { hits, page } = result;  
    const { searchKey, results } = this.state;  
  
    const oldHits = results && results[searchKey]  
      ? results[searchKey].hits  
      : [];  
  
    const updatedHits = [  
      ...oldHits,  
      ...hits  
    ];  
  
    this.setState({  
      results: {  
        ...results,  
        [searchKey]: { hits: updatedHits, page }  
      }  
    });  
  }  
  
  ...  
}
```

searchKey будет использоваться в качестве ключа для сохранения обновлённых историй и страницы в объекте results.

Во-первых, вы должны получить searchKey из состояния компонента. Помните, что searchKey устанавливается в componentDidMount() и onSearchSubmit().

Во-вторых, старые истории должны быть объединены с новыми историями, как и раньше. Но на этот раз старые истории получаются из объекта results с помощью searchKey в качестве ключа.

В-третьих, новый результат может быть сохранён в объекте results состояния. Рассмотрим объект results в setState().

src/App.js

```
results: {
  ...results,
  [searchKey]: { hits: updatedHits, page }
}
```

Нижняя часть гарантирует сохранение обновлённого результата с ключом `searchKey` в объекте результатов. Значение — это объект со свойствами `hits` и `page`. `searchKey` — это поисковая строка. Вы уже изучили синтаксис `[searchKey]: ...`. Это вычисляемое имя свойства в ES6. Оно помогает динамически распределять значения в объекте.

Верхняя часть должна добавить все остальные результаты по `searchKey` в состоянии с помощью оператора расширения объектов. В противном случае вы потеряете все результаты, которые вы сохранили ранее.

Теперь вы сохраняете все результаты по поисковой строке. Это первый шаг для реализации кеша. На следующем шаге вы можете получить результат в зависимости от не временного значения в `searchKey` в объекте результатов. Вот почему вам сначала нужно было ввести `searchKey` как постоянную переменную. В противном случае процесс поиска будет нарушен, если вы будете использовать временный `searchTerm` для получения текущего результата, потому что это значение может измениться, если вы будете использовать компонент `Search`.

src/App.js

```
class App extends Component {
```

```
  ...
  render() {
    const {
      searchTerm,
      results,
      searchKey
    } = this.state;

    const page = (
      results &&
      results[searchKey] &&
      results[searchKey].page
    ) || 0;

    const list = (
      results &&
      results[searchKey] &&
```

```
    results[searchKey].hits
) || [];

return (
  <div className="page">
    <div className="interactions">
      ...
    </div>
    <Table
      list={list}
      onDismiss={this.onDismiss}
    />
    <div className="interactions">
      <Button onClick={() => this.fetchSearchTopStories(searchKey, page + 1)}>
        Больше историй
      </Button>
    </div>
  </div>
);
}

}
```

Поскольку по умолчанию у нас пустой список, потому что нет результата по `searchKey`, мы можем сейчас сэкономить условную отрисовку для компонента `Table`. Кроме того, вам нужно будет передать `searchKey`, а не `searchTerm` в кнопку «Больше историй». В противном случае постраничное получение данных зависит от значения `searchTerm`, которое является изменчивым. Кроме того, не забудьте сохранить свойство `searchTerm` для поля ввода в компоненте `Search`.

Функциональность поиска должна снова заработать. Она сохраняет все результаты от API Hacker News.

Кроме того, метод `onDismiss()` должен быть улучшен. Он по-прежнему работает с объектом `result`. Теперь он должен иметь дело с несколькими результатами (`results`).

src/App.js

```
onDismiss(id) {
  const { searchKey, results } = this.state;
  const { hits, page } = results[searchKey];

  const isNotId = item => item.objectID !== id;
  const updatedHits = hits.filter(isNotId);

  this.setState({
    results: {
      ...results,
      [searchKey]: { hits: updatedHits, page }
    }
  });
}
```

Кнопка «Отбросить» должна снова работать.

Тем не менее, ничто не мешает приложению отсылать запрос к API при каждом запросе на поиск. Несмотря на то, что результат уже может быть, нет проверки, которая предотвращает запрос. Таким образом, функциональность кеша ещё не завершена. Она кеширует результаты, но не использует их. Последним шагом было бы предотвратить запрос к API, когда результат имеется в кеше.

src/App.js

```
class App extends Component {

  constructor(props) {
    ...
    this.needsToSearchTopStories = this.needsToSearchTopStories.bind(this);
    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  needsToSearchTopStories(searchTerm) {
    return !this.state.results[searchTerm];
  }
}
```

```
...  
  
onSearchSubmit(event) {  
  const { searchTerm } = this.state;  
  this.setState({ searchKey: searchTerm });  
  
  if (this.needsToSearchTopStories(searchTerm)) {  
    this.fetchSearchTopStories(searchTerm);  
  }  
  
  event.preventDefault();  
}  
  
...  
  
}
```

Теперь на клиенте делается запрос к API только один раз, хотя вы дважды ищете поисковую строку. Даже данные с разбивкой на страницы с несколькими страницами кешируются таким образом, потому что вы всегда сохраняете последнюю страницу для каждого результата в объекте `results`. Разве это не мощный подход для внедрения кеширования в ваше приложение? API Hacker News предоставляет всё необходимое для того, чтобы эффективно кешировать данные с разбивкой на страницы.

Обработка ошибок

Всё готово для работы с API Hacker News. Мы даже попробовали реализовать элегантный способ кеширования результатов из API и использовать возможность API для постраничного получения бесконечного списка историй. Но есть недостающая часть. К сожалению, сегодня она часто пропускается при разработке приложений — обработка ошибок. Слишком просто реализовать работающий сценарий функционирования приложения, не задумываясь о тех ошибках, которые могут произойти во время его работы.

В этом разделе главы вы познакомитесь с эффективным решением для добавления обработки ошибок для вашего приложения в случае ошибочного запроса API. Вы уже узнали о необходимых строительных блоках в React для реализации обработки ошибок: локальное состояние и отрисовка по условию. В основном, ошибка — это всего лишь другое состояние в React. При возникновении ошибки вы сохраните её в локальном состоянии и отобразите её, используя условную отрисовку в компоненте. Вот и всё. Давайте реализуем обработку ошибок в компоненте App, потому что прежде всего это компонент, который используется для получения данных из API Hacker News. Во-первых, нам нужно свойство для ошибки в локальном состоянии. Оно инициализируется значением null, но ему будет присвоен объект ошибки в случае неудачного запроса.

src/App.js

```
class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
      error: null,
    };

    ...
  }

  ...
}
```

Во-вторых, вы можете использовать блок catch в нативном fetch для сохранения объекта ошибки в локальном состоянии с помощью setState(). Каждый раз, когда запрос к API потерпит неудачу, будет выполнен код в блоке catch.

src/App.js

```
class App extends Component {  
  ...  
  
  fetchSearchTopStories(searchTerm, page = 0) {  
    fetch(`#${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${page}&${PARAM_HPP}${DEFAULT_HPP}`)  
      .then(response => response.json())  
      .then(result => this.setSearchTopStories(result))  
      .catch(error => this.setState({ error }));  
  }  
  
  ...  
}
```

В-третьих, вы можете получить объект ошибки из своего локального состояния в методе `render()` и отобразить сообщение об ошибке с помощью условной отрисовки в React.

src/App.js

```
class App extends Component {  
  ...  
  
  render() {  
    const {  
      searchTerm,  
      results,  
      searchKey,  
      error  
    } = this.state;  
  
    ...  
  
    if (error) {  
      return <p>Что-то произошло не так.</p>;  
    }  
  
    return (  
      <div className="page">  
        ...  
    );  
  }  
}
```

```
        </div>
    );
}
}
```

Вот так. Если вы хотите проверить, работает ли обработка ошибок, вы можете изменить URL-адрес API на любой несуществующий.

src/App.js

```
const PATH_BASE = 'https://hn.foo.bar.com/api/v1';
```

При использовании такого значения для константы пути к API вы должны получить сообщение об ошибке вместо отображения приложения. Это зависит от вас, где вы хотите поместить условную отрисовку для сообщения об ошибке. В данном случае всё приложение не будет отображаться. Это не самый лучший опыт для пользовательского взаимодействия. А что насчёт отображения компонента Table или сообщения об ошибке? Остальная часть приложения даже в случае ошибки всё равно будет отображаться.

src/App.js

```
class App extends Component {
```

```
...
```

```
    render() {
        const {
            searchTerm,
            results,
            searchKey,
            error
        } = this.state;

        const page = (
            results &&
            results[searchKey] &&
            results[searchKey].page
        ) || 0;

        const list = (
            results &&
            results[searchKey] &&
            results[searchKey].hits
        ) || [];
    }
}
```

```
return (
  <div className="page">
    <div className="interactions">
      ...
    </div>
    { error
      ? <div className="interactions">
          <p>Something went wrong.</p>
        </div>
      : <Table
          list={list}
          onDismiss={this.onDismiss}
        />
    }
    ...
  </div>
);
}
```

Напоследок не забудьте вернуть прежний URL-адрес API.

src/App.js

```
const PATH_BASE = 'https://hn.algolia.com/api/v1';
```

Ваше приложение должно по-прежнему работать, но на этот раз с обработкой ошибок в случае неудачного запроса к API.

Упражнения:

- узнайте подробнее про [обработку ошибок в компонентах React](#)¹²⁶

¹²⁶<https://reactjs.org/blog/2017/07/26/error-handling-in-react-16.html>

Использование Axios вместо Fetch

В одном из предыдущих разделов вы внедрили нативный API fetch для выполнения запроса на платформу Hacker News. Браузер позволяет использовать этот нативный API fetch. Однако не все браузеры, особенно старые, поддерживают его. Кроме того, как только вы начнёте тестировать своё приложение в окружении браузера без пользовательского интерфейса (без браузера, а это только имитация), могут возникнуть проблемы с API fetch. Такое окружение браузера из консоли может быть при написании и выполнении тестов для вашего приложения, которые не запускаются в реальном браузере. Есть несколько способов сделать работу fetch в старых браузерах (через полифили) и в тестах ([isomorphic-fetch¹²⁷](#)), но в этой книге мы не будем заниматься этим.

Альтернативным способом решения этой проблемы было бы заменить нативный API fetch стабильной библиотекой, такой как [axios¹²⁸](#). Axios — это библиотека, которая решает только одну проблему, но делает это качественно: выполнение асинхронных запросов к удалённым API. Вот почему вы будете использовать его в этой книге. Конкретно этот раздел главы продемонстрирует вам, как вы можете заменить библиотеку (которая является нативным API браузера в этом случае) другой библиотекой. На абстрактном уровне он должен показать вам, как вы всегда можете найти решение для причуд (например, старых браузеров, тестов браузеров без пользовательского интерфейса) в веб-разработке. Поэтому никогда не переставайте искать решения, если что-то мешает вам.

Давайте посмотрим, как нативный API fetch можно заменить на axios. На самом деле всё сказанное ранее звучит сложнее, чем есть на самом деле. Во-первых, вам нужно установить axios в командной строке:

Командная строка

```
npm install axios
```

Во-вторых, импортировать axios в файл компонента App:

src/App.js

```
import React, { Component } from 'react';
import axios from 'axios';
import './App.css';

...
```

И последнее, но не менее важное: вы можете использовать его вместо `fetch()`. Его использование почти идентично нативному API fetch. Он принимает URL в виде аргумента и

¹²⁷<https://github.com/matthew-andrews/isomorphic-fetch>

¹²⁸<https://github.com/axios/axios>

возвращает промис. Вам больше не нужно преобразовывать возвращённый ответ JSON. Axios делает это за вас и обёртывает результат в объект `data` в JavaScript. Таким образом, не забудьте адаптировать свой код к возвращаемой структуре данных.

src/App.js

```
class App extends Component {  
  ...  
  
  fetchSearchTopStories(searchTerm, page = 0) {  
    axios(`#${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${page}&${PARAM_HPP}${DEFAULT_HPP}`)  
      .then(result => this.setSearchTopStories(result.data))  
      .catch(error => this.setState({ error }));  
  }  
  
  ...  
}
```

Это всё, что нужно для замены `fetch` на `axios` в этой главе. В вашем коде вы вызываете `axios()`, который по умолчанию использует HTTP GET-запрос. Вы можете сделать запрос GET явным образом, вызвав `axios.get()`. Также вы можете использовать другой HTTP-метод, такой как HTTP POST с помощью `axios.post()`. Сейчас вы уже видите, что `axios` — это мощная библиотека для выполнения запросов к удалённым API. Я часто рекомендую использовать его вместо нативного API `fetch`, когда ваши запросы API усложняются или вам приходится сталкиваться со странностями веб-разработки, связанными с промисами. Кроме того, в следующей главе вы познакомитесь с тестированием своего приложения. Тогда вам больше не нужно будет беспокоиться о браузере или окружении консольного браузера.

Я хочу представить ещё одно улучшение запроса к Hacker News в компоненте `App`. Представьте, что ваш компонент монтируется, когда веб-страница отображается в браузере в первый раз. В `componentDidMount()` компонент делает запрос, но затем, поскольку ваше приложение произвело некую навигацию, вы переходите от этой страницы к другой странице. Ваш компонент `App` демонтируется, но все ещё остаётся ожидающий запрос из вашего жизненного цикла `componentDidMount()`. Он попытается использовать `this.setState()` в конечном итоге в блоке промиса `then()` или `catch()`. Возможно, в первый раз вы увидите следующее предупреждение в командной строке или в выводе консоли разработчика браузера:

Командная строка

Warning: Can only update a mounted or mounting component. This usually means you called setState, replaceState, or forceUpdate on an unmounted component. This is a no-op.

Вы можете решить эту проблему, прервав запрос, когда ваш компонент размонтируется или предотвратить вызов `this.setState()` в размонтированном компоненте. Это передовая практика в React, хотя ей следуют далеко не все разработчики, чтобы оставить чистое приложение без каких-либо раздражающих предупреждений. Однако в текущем API промисов не реализует прерывание запроса. Таким образом, вам нужно самостоятельно справиться с этой проблемой. Возможно также, это та причина, по которой не многие разработчики следуют этой лучшей практике. Следующая реализация выглядит скорее как обходное решение, чем поддерживаемая в дальнейшем реализация. Исходя из этого вы можете самостоятельно решить, хотите ли вы это реализовывать, чтобы обойти предупреждение из-за размонтированного компонента. Тем не менее, помните об этом предупреждении, если оно появится в следующей главе этой книги или в вашем собственном приложении. Теперь в подобных случаях вы знаете как справиться с этим.

Давайте перейдём к решению проблемы. Вы можете добавить поле класса, которое содержит состояние жизненного цикла вашего компонента. Он может быть инициализирован как `false`, когда компонент инициализируется, и изменяется на `true`, когда компонент установлен, а затем снова устанавливается на `false` при удалении компонента. Таким образом, вы можете отслеживать состояние жизненного цикла вашего компонента. Это поле не имеет ничего общего с локальным состоянием, которое хранится и модифицируется с помощью `this.state` и `this.setState()`, поскольку у вас должен быть к нему доступ непосредственно в экземпляре компонента, не полагаясь на управление локальным состоянием React. Более того, это не приводит к повторной отрисовке компонента, когда поле класса изменяется таким образом.

src/App.js

```
class App extends Component {
  _isMounted = false;

  constructor(props) {
    ...
  }

  ...

  componentDidMount() {
    this._isMounted = true;

    const { searchTerm } = this.state;
```

```
    this.setState({ searchKey: searchTerm });
    this.fetchSearchTopStories(searchTerm);
}

componentWillUnmount() {
  this._isMounted = false;
}

...
}
```

Наконец, вы можете использовать эти знания, чтобы не прервать сам запрос, а избежать вызов `this.setState()` в вашем экземпляре компонента, даже если компонент уже удалён. Это предотвратит упомянутое предупреждение.

src/App.js

```
class App extends Component {

  ...

  fetchSearchTopStories(searchTerm, page = 0) {
    axios(`.${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${page}&${PARAM_HPP}${DEFAULT_HPP}`)
      .then(result => this._isMounted && this.setSearchTopStories(result.data))
      .catch(error => this._isMounted && this.setState({ error }));
  }

  ...
}

}
```

В целом в главе показано, как вы можете заменить одну библиотеку другой библиотекой в React. Если у вас возникнут какие-либо проблемы, вы можете использовать обширную экосистему JavaScript-библиотек, чтобы помочь самому себе. Кроме того, вы видели способ, как можно избежать вызова `this.setState()` в React на размонтированном компоненте. Если вы получите изучите библиотеку axios, вы также найдёте способ отменить запрос. Вам предстоит больше узнать об этой теме.

Упражнения:

- прочтайте про то, почему выбор фреймворка имеет значение¹²⁹

¹²⁹<https://www.robinwieruch.de/why-frameworks-matter/>

- узнайте больше про [альтернативный синтаксис для определения компонента¹³⁰](#)

¹³⁰<https://github.com/the-road-to-learn-react/react-alternative-class-component-syntax>

Вы научились взаимодействовать с API в React! Давайте повторим последние темы:

- React
 - Методы жизненного цикла ES6-класса компонента для разных случаев использования
 - `componentDidMount()` для взаимодействий с API
 - отрисовка по условию
 - синтетические события на формах
 - обработка ошибок
 - отмена удалённого API-запроса
- ES6 и за его пределами
 - шаблонные строки для объединения строк
 - оператор расширения для неизменяемых структур данных
 - вычисляемые имена свойств
 - поля класса
- Общее
 - Работа с API Hacker News
 - API нативного `fetch` в браузере
 - Поиск на стороне клиента и сервера
 - Разбивка на страницы данных
 - Кэширование на стороне клиента
 - Использование `axios` в качестве альтернативы для нативного API `fetch`

Повторю снова, имеет смысл сделать перерыв. Усвоить полученные знания и применить их на практике самостоятельно. Вы можете поэкспериментировать с исходным кодом, который вы написали к настоящему времени. Исходный код можно найти в [официальном репозитории](#)¹³¹.

¹³¹<https://github.com/the-road-to-learn-react/hackernews-client/tree/5.3.1>

Организация и тестирование кода

В этой главе мы сосредоточим внимание на важных темах сохранения кода удобным для поддержки в масштабируемом приложении. Вы узнаете об организации кода, чтобы освоить лучшие способы структурирования ваших папок и файлов. Ещё один аспект, который вы узнаете, — это тестирование, которое важно для написания надёжного кода. Наконец, вы узнаете о полезном инструменте для отладки ваших React-приложений. Большая часть главы отойдёт в сторону от практического приложения и объяснит вам пару этих тем.

Модули ES6: импорт и экспорт

В JavaScript ES6 вы можете импортировать и экспортировать функциональность из модулей. Этой функциональностью могут быть функции, классы, компоненты, константы и т.д. В основном это всё то, что вы можете присвоить переменной. Модули могут быть отдельными файлами или целыми папками с одним индексным файлом (`index.js`) в качестве точки входа.

В начале книги после того, как вы инициализировали приложение с помощью `create-react-app`, вы уже сталкивались с несколькими выражениями `import` и `export` в ваших файлах исходного кода. Настало время объяснить, зачем всё это.

Выражения `import` и `export` помогают распространять код между несколькими файлами. Раньше в среде JavaScript было несколько решений для этой цели. Это был беспорядок, потому что вы, вероятно, хотели бы следовать стандартизированному способу, а не иметь несколько подходов к одному и тому же. Теперь это нативная возможность, начиная с JavaScript ES6.

Кроме того, эти выражения охватывают идею разделения кода. Вы разделяете свой код между несколькими файлами для возможности сделать его многоразовым и поддерживаемым. Первое верно, потому что вы можете импортировать часть кода в несколько файлов. Последнее верно, потому что у вас есть один источник, в котором вы поддерживаете часть кода.

И последнее, но не менее важное: это поможет вам подумать о инкапсуляции кода. Не каждая функциональность должна быть экспортована из файла. Некоторые из этих функциональных возможностей должны использоваться только в файле, где они были определены. Экспорт файла в основном означает общедоступный API для использования в другом файле. Доступны для повторного использования в другом месте кодовой базы только экспортованные функциональные возможности. Это следует передовой практике инкапсуляции.

Но давайте перейдём к практике. Как эти выражения `import` и `export` работают? В следующих примерах приводятся выражения, совместно использующие одну или несколько переменных в двух файлах. В конце концов, этот подход можно масштабировать на несколько файлов и обмениваться не только простыми переменными.

Вы можете экспортить одну или несколько переменных. Это называется именованным экспортом.

Код: `file1.js`

```
const firstname = 'Robin';
const lastname = 'Wieruch';

export { firstname, lastname };
```

И импортируйте их в другой файл с помощью относительного пути к первому файлу.

Код: file2.js

```
import { firstname, lastname } from './file1.js';

console.log(firstname);
// выведет: Robin
```

Вы также можете импортировать все экспортируемые данные из другого файла в виде одного объекта.

Код: file2.js

```
import * as person from './file1.js';

console.log(person.firstname);
// выведет: Robin
```

У выражений импорта могут быть псевдонимы. Может произойти такая ситуация, когда вы импортируете функциональность из нескольких файлов с одним и тем же именем при экспорте. По этой причине вы можете использовать псевдоним.

Код: file2.js

```
import { firstname as username } from './file1.js';

console.log(username);
// выведет: Robin
```

И последнее, но не менее важное: существует выражение по умолчанию — `default`. Его можно использовать в следующих случаях: `*` для экспорта и импорта единственной функциональной возможности `*` для выделения основных функциональных возможностей экспортированного API модуля `*` для того, чтобы иметь резервную (фолбэк) функциональность при импорте

Код: file1.js

```
const robin = {
  firstname: 'Robin',
  lastname: 'Wieruch',
};

export default robin;
```

Вы можете опустить фигурные скобки при импорте по умолчанию.

Код: file2.js

```
import developer from './file1.js';

console.log(developer);
// выведет: { firstname: 'Robin', lastname: 'Wieruch' }
```

Кроме того, название импорта можно различать от экспортированного названия по умолчанию. Вы также можете использовать это в сочетании с именованными выражениями экспорта и импорта.

Код: file1.js

```
const firstname = 'Robin';
const lastname = 'Wieruch';

const person = {
  firstname,
  lastname,
};

export {
  firstname,
  lastname,
};

export default person;
```

Затем импортируйте импорт по умолчанию, а также экспорт по имени в другой файл.

Код: file2.js

```
import developer, { firstname, lastname } from './file1.js';

console.log(developer);
// выведет: { firstname: 'Robin', lastname: 'Wieruch' }
console.log(firstname, lastname);
// выведет: Robin Wieruch
```

При использовании именованного экспорта вы можете сэкономить дополнительные строки и непосредственно экспортировать переменные.

Код: file1.js

```
export const firstname = 'Robin';
export const lastname = 'Wieruch';
```

Это основная функциональность модулей ES6. Они помогают вам организовать собственный код, поддерживать его и проектировать API-интерфейсы повторно используемых модулей. Вы также можете экспортировать и импортировать функциональность для их тестирования. Вы сделаете это в одной из следующих глав.

Упражнения:

- ознакомьтесь подробнее с [импортом в ES6](#)¹³²
- ознакомьтесь подробнее с [экспортом в ES6](#)¹³³

¹³²<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/import>

¹³³<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/export>

Организация кода с помощью модулей в ES6

Вы могли бы задаться вопросом: почему мы не последовали передовому опыту разделения кода для файла `src/App.js`? В этом файле у нас уже есть несколько компонентов, которые могут быть определены в собственных файлах/папках (модулях). В интересах изучения React, целесообразно хранить их в одном месте. Но как только ваше приложение React растёт, вам следует рассмотреть возможность разделения этих компонентов на несколько модулей. Только так ваше приложение масштабируется.

Ниже я предлагаю несколько модулей, которые вы *могли бы* применить. Я бы рекомендовал применить их в качестве упражнения в конце книги. Чтобы книга оставалась простой, я не буду выполнять разделение кода и продолжу следующие главы с файлом `src/App.js`.

Одной из возможных структур модулей могла быть такая:

Структура каталогов

```
src/
  index.js
  index.css
  App.js
  App.test.js
  App.css
  Button.js
  Button.test.js
  Button.css
  Table.js
  Table.test.js
  Table.css
  Search.js
  Search.test.js
  Search.css
```

Такой подход разделяет компоненты на отдельные файлы, но он не выглядит слишком перспективным. Вы можете видеть много дублирования названий, которые различаются только расширением файла. Совершенно другой структурой модулей может быть подобное разделение:

Структура каталогов

```
src/
  index.js
  index.css
  App/
    index.js
    test.js
    index.css
  Button/
    index.js
    test.js
    index.css
  Table/
    index.js
    test.js
    index.css
  Search/
    index.js
    test.js
    index.css
```

Это уже выглядит чище, чем предыдущий способ. Файлы с названием `index` описывает его как файл точки входа в папку (модуль). Это просто обычное соглашение (правило) об наименовании, но вы также можете использовать собственное соглашение. В этой структуре модулей компонент определяется объявлением компонента в файле JavaScript (`index.js`) , а также его стилем (`index.css`) и тестами для него (`test.js`).

Следующим шагом могло бы быть стать извлечение констант из компонента `App`. Эти константы использовались для формирования URL к API Hacker News.

Структура каталогов

```
src/
  index.js
  index.css
  constants/
    index.js
  components/
    App/
      index.js
      test.js
      index..css
    Button/
      index.js
```

```
test.js  
index.css  
...
```

Естественно, что модули бы разделились на *src/constants/* и *src/components/*. Теперь файл *src/constants/index.js* может выглядеть следующим образом:

Код: *src/constants/index.js*

```
export const DEFAULT_QUERY = 'redux';
export const DEFAULT_HPP = '100';
export const PATH_BASE = 'https://hn.algolia.com/api/v1';
export const PATH_SEARCH = '/search';
export const PARAM_SEARCH = 'query=';
export const PARAM_PAGE = 'page=';
export const PARAM_HPP = 'hitsPerPage=';
```

В файле *App/index.js* вы можете импортировать эти константы для использования.

Код: *src/components/App/index.js*

```
import {
  DEFAULT_QUERY,
  DEFAULT_HPP,
  PATH_BASE,
  PATH_SEARCH,
  PARAM_SEARCH,
  PARAM_PAGE,
  PARAM_HPP,
} from './constants/index.js';

...
```

Когда вы используете соглашение по именованию, включающее *index.js*, вы можете опустить название файла из относительно пути.

Код: `src/components/App/index.js`

```
import {  
  DEFAULT_QUERY,  
  DEFAULT_HPP,  
  PATH_BASE,  
  PATH_SEARCH,  
  PARAM_SEARCH,  
  PARAM_PAGE,  
  PARAM_HPP,  
} from '../constants';  
  
...
```

Но что стоит за именованием файлов `index.js`? Соглашение было введено в мире Node.js. Индексный файл является точкой входа в модуль. Он описывает общедоступный API для модуля. Внешним модулям разрешено использовать файл `index.js` для импорта разделяемого кода из модуля. Рассмотрим следующую составленную структуру модуля, чтобы продемонстрировать её:

Структура каталогов

```
src/  
  index.js  
  App/  
    index.js  
  Buttons/  
    index.js  
    SubmitButton.js  
    SaveButton.js  
    CancelButton.js
```

В каталоге `Buttons/` есть несколько компонентов, определённых в разных файлах. Каждый файл компонента может использовать `export default`, чтобы его можно было импортировать в файле `Buttons/index.js`. Файл `Buttons/index.js` импортирует все различные представления кнопок и экспортирует их в качестве общедоступного API модуля.

Код: src/Buttons/index.js

```
import SubmitButton from './SubmitButton';
import SaveButton from './SaveButton';
import CancelButton from './CancelButton';

export {
  SubmitButton,
  SaveButton,
  CancelButton,
};
```

Теперь из файла *src/App/index.js* можно импортировать кнопки из общедоступного API модуля, расположенного в файле *index.js*.

Код: src/App/index.js

```
import {
  SubmitButton,
  SaveButton,
  CancelButton
} from '../Buttons';
```

Рассматривая ограничения данного подхода, было бы неудачной практикой импортировать функциональность из других файловых путей, отличных от *index.js*, в модуле. Это нарушит правила инкапсуляции.

Код: src/App/index.js

```
// порочная практика, не делайте так
import SubmitButton from '../Buttons/SubmitButton';
```

Теперь вы знаете, как можно отрефакторить исходный код в модулях с ограничениями инкапсуляции. Как я уже сказал, ради сохранения простоты в книге я не буду использовать описанные выше изменения. Но вам стоит заняться рефакторингом после прочтения книги.

Упражнения:

- займитесь рефакторингом файла *src/App.js*, разделив его на несколько модулей-компонентов, когда закончите книгу

Тестирование снимками с помощью Jest

Данная книга не будет глубоко погружаться в тему тестирования, но это не значит, что её следует игнорировать. Тестирование кода в программировании имеет важное значение и должно рассматриваться в качестве обязательного требования, если вы хотите, чтобы качество вашего кода было высоким и всё работало как надо.

Возможно, вы слышали о пирамиде тестирования. Существуют сквозные (end-to-end), интеграционные (integration) и модульные (unit) тесты. Если вы не знакомы с ними, то в книге приводится быстрый и краткий обзор. Модульный тест служит для проверки изолированного и небольшого куска кода. Это может быть всего лишь одна функция, проверяемая модульным тестом. Тем не менее, иногда модульные тесты работают хорошо в изоляции, но не работают в сочетании с другими модульными тестами. Поэтому их нужно тестировать группой из модульных тестов. Вот где интеграционные тесты могут помочь, учитывая, что они неплохо работают вместе. И последнее, но не менее важное: сквозной тест — это имитация реального использования приложения пользователем. Это может быть автоматическая настройка в браузере, имитирующая процесс входа пользователя в приложение. Если модульные тесты быстро и легко писать и поддерживать, то к сквозным тестам это не относится.

Сколько тестов нужно для каждого типа тестирования? Скорее всего, вы хотите, чтобы было как можно больше модульных тестов для покрытия тестами функций в изоляции. После этого вы можете провести несколько интеграционных тестов, чтобы проверить, что наиболее важные функции работают в сочетании вместе, как ожидалось. И наконец, что немаловажно, вам может потребоваться только несколько сквозных тестов для симулирования критических сценариев использования приложения. На этом общая экскурсия в мир тестирования заканчивается.

Итак, как можно применить полученные знания для тестирования своего приложения на React? Основа тестирования в React — это тесты компонентов, которые можно тестировать модульными тестами, а часть из них — тестированием снимками (snapshot). Вы напишите модульные тесты для своих компонентов в следующей главе, используя библиотеку Enzyme. В этой главе вы остановитесь на другом типе тестов — тестов снимками. Для данного тестирования применяется библиотека под названием Jest.

Jest¹³⁴ — это платформа для тестирования JavaScript, используемая и разработанная в Facebook. В сообществе React она используется для тестирования React-компонентов. К счастью, *create-react-app* уже поставляется с Jest, поэтому вам не нужно дополнительно его настраивать.

Давайте начнём тестировать ваши первые компоненты. Прежде чем приступить к этому, вам нужно экспортить компоненты, которые вы собираетесь тестировать, из файла *src/App.js*. После этого вы можете протестировать их в другом файле. Об этом вы узнали в разделе про организацию кода.

¹³⁴<https://jestjs.io/>

src/App.js

```
...
class App extends Component {
  ...
}

...
export default App;

export {
  Button,
  Search,
  Table,
};
```

В файле *App.test.js* вы найдёте ваш первый тест, который появился при инициализации *create-react-app*. Он проверяет, что компонент App отображается без каких-либо ошибок.

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

it('отрисовывает без ошибки', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
  ReactDOM.unmountComponentAtNode(div);
});
```

Блок «it» представляет собой один тест. Этот блок принимает описание теста и собственно код теста, который может либо пройти успешно, либо потерпеть неудачу (провалиться). Кроме того, вы можете обернуть его в блок «describe», который определяет набор тестов. Набор тестов может включать в себя множество блоков «it» для одного конкретного компонента. Мы рассмотрим блоки «describe» чуть позже. Оба этих блока используются для разделения и организации ваших тестов.

Обратите внимание, что функция *it* считается в сообществе JavaScript функцией, в которой выполняется один тест. Однако в Jest часто встречается псевдоним *test* для функции тестирования.

Вы можете запускать свои тесты с помощью интерактивного скрипта тестирования приложения, предоставляемого *create-react-app* в командной строке. Вам будет показан вывод по каждому тесту в терминале.

Командная строка

```
npm test
```

Теперь Jest позволяет создавать снимки. Эти тесты делают снимок вашего отрисованного компонента и сравнивает этот снимок с будущими снимками. При изменении будущих снимков, будут отображаться соответствующие уведомления в тесте. Вы можете принять изменение снимка, потому что вы специально изменили реализацию компонента, либо отказать в изменении снимка, что означает, что была допущена ошибка, которую нужно исправить. Jest очень хорошо дополняет модульные тесты, потому что проверяются только различия в отрисованных компонентах в разные моменты времени. Кроме того, это не добавляет больших затрат на поддержку таких тестов, потому что вы можете просто принимать изменённые снимки, когда вы намеренно что-то изменили, что повлияло на отрисовку компонента.

Jest хранит снимки в каталоге `snapshots`. Только таким образом он может проверить различия с будущим снимком. Кроме того, снимками можно обмениваться между командами, если они находятся в одной папке.

Перед написанием первого теста снимка с помощью Jest вам необходимо установить следующую библиотеку.

Командная строка

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

Теперь вы можете расширить тест компонента App с помощью своего первого снимка. Во-первых, импортируйте новую функциональность из node-пакета и поместите свой тестируемый код компонента App в блок «`it`», а его в свою очередь в блок «`describe`». В данном случае тестовый набор предназначен только для компонента App.

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import App from './App';

describe('App', () => {
  it('отрисовывает без ошибки', () => {
    const div = document.createElement('div');
```

```
ReactDOM.render(<App />, div);
ReactDOM.unmountComponentAtNode(div);
});

});
```

Теперь вы можете написать свой первый тест снимком с помощью блока «test».

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import App from './App';

describe('App', () => {

  it('отрисовывает без ошибки', () => {
    const div = document.createElement('div');
    ReactDOM.render(<App />, div);
    ReactDOM.unmountComponentAtNode(div);
  });

  test('есть корректный снимок', () => {
    const component = renderer.create(
      <App />
    );
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
});
```

Запустите ваши тесты снова и посмотрите, какие из них были выполнены успешно или неудачно. Все они должны пройти. Если вы измените вывод блока отрисовки в компоненте App, тест снимками должен завершиться неудачно. Затем вы можете решить, принять обновление снимка или выяснить причину ошибки в компоненте App.

В основном функция `renderer.create()` создаёт снимок вашего компонента App. Она отрисовывает компонент виртуально и сохраняет полученный DOM в снимок. После этого снимок, как ожидается, будет соответствовать предыдущему снимку с момента последнего запуска проверки снимков. Таким образом, вы можете утверждать, что ваш DOM остался таким же и ничего не поменялось по ошибке.

Давайте добавим больше тестов для наших независимых компонентов. Во-первых, для компонента Search:

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import App, { Search } from './App';

...

describe('Search', () => {

  it('отрисовывает без ошибки', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Search>Поиск</Search>, div);
    ReactDOM.unmountComponentAtNode(div);
  });

  test('есть корректный снимок', () => {
    const component = renderer.create(
      <Search>Поиск</Search>
    );
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
});
```

У компонента Search два теста, аналогичных компоненту App. Первый тест просто отрисовывает компонент Search в DOM и проверяет, что во время процесса отрисовки не произошла ошибка. Если ошибка возникла, тест провалится, даже если в тестовом блоке нет никакого утверждения (например, `expect`, `match`, `equal`). Второй — тест снимком — используется для хранения снимка отрисованного компонента и сравнивает его с предыдущего снимком. Этот тест закончится неудачей, когда снимок изменится.

Во-вторых, вы можете протестировать компонент Button, так как применяются те же правила тестирования, что и в компоненте Search.

```
src/App.test.js
```

```
...
import App, { Search, Button } from './App';

...

describe('Button', () => {

  it('отрисовывает без ошибки', () => {
    const div = document.createElement('div');
    ReactDOM.render(<Button>Дай мне больше</Button>, div);
    ReactDOM.unmountComponentAtNode(div);
  });

  test('есть корректный снимок', () => {
    const component = renderer.create(
      <Button>Дай мне больше</Button>
    );
    const tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });
});

});
```

И последнее, но не менее важное: компоненту Table вы можете передать много первоначальных свойств, чтобы отрисовать с демонстрационном списком.

```
src/App.test.js
```

```
...
import App, { Search, Button, Table } from './App';

...

describe('Table', () => {

  const props = {
    list: [
      { title: '1', author: '1', num_comments: 1, points: 2, objectID: 'y' },
      { title: '2', author: '2', num_comments: 1, points: 2, objectID: 'z' },
    ],
  };
});
```

```
it('отрисовывает без ошибки', () => {
  const div = document.createElement('div');
  ReactDOM.render(<Table { ...props } />, div);
});

test('есть корректный снимок', () => {
  const component = renderer.create(
    <Table { ...props } />
  );
  const tree = component.toJSON();
  expect(tree).toMatchSnapshot();
});

});
```

Тесты снимком обычно остаются довольно простыми. Вам нужно лишь заметить, что компонент не изменил свой отрисованный вывод. После изменения вывода вы должны решить, принимаете ли вы эти изменения или нет. В противном случае вам нужно исправить компонент, если его вывод отрисовки не такой, как ожидался.

Упражнения:

- посмотрите, как тест снимком завершится неудачно после изменения возвращаемого значения компонента в методе `render()`
 - и либо примите, либо отклоните изменение снимка
- обновите свои снимки при обновлении компонентов в следующих главах
- узнайте подробнее про использование Jest в React¹³⁵

¹³⁵<https://jestjs.io/docs/en/tutorial-react>

Модульное тестирование с помощью Enzyme

Enzyme¹³⁶ — это утилита тестирования от Airbnb для проверки утверждений, манипулирования и навигации по компонентам React. Вы можете использовать её для проведения модульных тестов в дополнение к вашим снимкам в React.

Давайте посмотрим, как вы можете использовать Enzyme. Сначала нужно установить его, поскольку он не поставляется по умолчанию в *create-react-app*. У Enzyme также есть расширение для использования в React.

Командная строка

```
npm install --save-dev enzyme react-addons-test-utils enzyme-adapter-react-16
```

Во-вторых, вам нужно подключить Enzyme и инициализировать его адаптер для использования в React.

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import Enzyme from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
import App, { Search, Button, Table } from './App';

Enzyme.configure({ adapter: new Adapter() });
```

Теперь вы можете написать свой первый модульный тест для компонента Table в блоке «describe». Вы будете использовать `shallow()` для отрисовки вашего компонента и проверять, что у отрисованного компонента Table два элемента, так как ему было передано два элемента списка. Утверждение просто проверяет, есть у элемента таблицы два дочерних элемента с классом `table-row`.

¹³⁶<https://github.com/airbnb/enzyme>

src/App.test.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import renderer from 'react-test-renderer';
import Enzyme, { shallow } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
import App, { Search, Button, Table } from './App';

// ...

describe('Table', () => {

  const props = {
    list: [
      { title: '1', author: '1', num_comments: 1, points: 2, objectID: 'y' },
      { title: '2', author: '2', num_comments: 1, points: 2, objectID: 'z' },
    ],
  };

  ...

  it('shows two items in list', () => {
    const element = shallow(
      <Table { ...props } />
    );

    expect(element.find('.table-row').length).toBe(2);
  });
});
```

Функция `shallow` отрисовывает компонент без его дочерних компонентов. Таким образом, вы можете написать тест, проверяющий непосредственно сам компонент.

У `Enzyme` есть три общих механизма отрисовки, доступные в API. Вы уже знаете `shallow()`, но также существуют `mount()` и `render()`. Они оба создают экземпляры родительского компонента и всех его дочерних компонентов. Кроме того `mount()` предоставляет вам доступ к методам жизненного цикла компонента. Но какой механизм отрисовки использовать? Существует следующие проверенные на практике правила:

- Всегда начинайте с поверхностного (`shallow`) теста
- Если необходимо проверить `componentDidMount()` или `componentDidUpdate()`, используйте `mount()`

- Если хотите протестировать жизненный цикл компонентов и поведение дочерних элементов, используйте `mount()`
- Если хотите протестировать отрисовку дочерних элементов компонента с меньшими накладными расходами, чем `mount()`, и вам не интересны методы жизненного цикла, используйте `render()`

Вы можете продолжить тестирование своих компонентов. Но следите за тем, чтобы тесты были простыми и удобными для дальнейшей поддержки. В противном случае вам придётся рефакторить их, как только вы измените свои компоненты. Именно поэтому Facebook в первую очередь представил инструмент для тестирования снимками Jest.

Упражнения:

- напишите модульный тест с использованием Enzyme для компонента Button
- постоянно выполняйте и актуализируйте при необходимости свои модульные тесты в следующих главах
- узнайте больше про [Enzyme и о его API отрисовки](#)¹³⁷

¹³⁷<https://github.com/airbnb/enzyme>

Интерфейс компонента с помощью PropTypes

Возможно вы знаете TypeScript¹³⁸ или Flow¹³⁹ для того, чтобы ввести интерфейс типа к JavaScript. Типизированный язык менее подвержен ошибкам, поскольку код проверяется на основе его текста программы. Редакторы и другие утилиты могут поймать эти ошибки до запуска программы. Это делает вашу программу более надёжной.

В книге мы не будем использовать Flow или TypeScript, а вместо этого представим ещё один аккуратный способ проверить свои типы в компонентах. React поставляется со встроенным инструментом проверки типов для предотвращения ошибок. Вы можете использовать PropTypes для описания интерфейса вашего компонента. Все свойства, которые передаются от родительского компонента к дочернему компоненту, проверяются на основе интерфейса PropTypes, назначенному дочернему компоненту.

В разделе этой главы показывается, как вы можете сделать компоненты безопасно типизированными с помощью PropTypes. Я пропускаю данные изменения для следующих глав, потому что они добавляют ненужные улучшения кода. Но вы должны сохранить и обновить их, чтобы сохранить тип интерфейса вашего компонента безопасным.

Во-первых, вам нужно установить отдельный пакет для React.

Командная строка

```
npm install prop-types
```

Теперь вы можете импортировать PropTypes.

src/App.js

```
import React, { Component } from 'react';
import axios from 'axios';
import PropTypes from 'prop-types';
```

Давайте установим типы для свойств в компонентах:

¹³⁸<https://www.typescriptlang.org/>

¹³⁹<https://flowtype.org/>

src/App.js

```
const Button = ({  
  onClick,  
  className = '',  
  children,  
) =>  
  <button  
    onClick={onClick}  
    className={className}  
    type="button"  
>  
  {children}  
</button>  
  
Button.propTypes = {  
  onClick: PropTypes.func,  
  className: PropTypes.string,  
  children: PropTypes.node,  
};
```

В принципе, вот и всё. Вы выбираете каждый аргумент из определения функции и присваиваете ему PropType. Основные PropTypes для примитивных типов данных и сложных объектов перечислены ниже:

- PropTypes.array
- PropTypes.bool
- PropTypes.func
- PropTypes.number
- PropTypes.object
- PropTypes.string

Однако, у вас есть ещё два PropTypes для определения отрисованного фрагмента (узла), например, строка и элемент React:

- PropTypes.node
- PropTypes.element

Вы уже использовали PropType node для компонента Button. В целом есть больше определений PropType, про которые вы можете узнать в официальной документации React.

На данный момент все определённые PropTypes для Button — необязательные. Параметры могут быть `null` или `undefined`. Но для нескольких свойств вы можете определить их обязательными. Вы можете установить требование, чтобы эти свойства были переданы компоненту.

src/App.js

```
Button.propTypes = {
  onClick: PropTypes.func.isRequired,
  className: PropTypes.string,
  children: PropTypes.node.isRequired,
};
```

className не обязательен, поскольку по умолчанию может быть пустая строка. Далее вы определяете интерфейс PropType для компонента Table:

src/App.js

```
Table.propTypes = {
  list: PropTypes.array.isRequired,
  onDismiss: PropTypes.func.isRequired,
};
```

Вы можете определить содержимое массива PropType более явно:

src/App.js

```
Table.propTypes = {
  list: PropTypes.arrayOf(
    PropTypes.shape({
      objectID: PropTypes.string.isRequired,
      author: PropTypes.string,
      url: PropTypes.string,
      num_comments: PropTypes.number,
      points: PropTypes.number,
    })
  ).isRequired,
  onDismiss: PropTypes.func.isRequired,
};
```

Требуется только objectID, потому что вы знаете, что от этого зависит какой-то ваш код. Другие свойства только отображаются, поэтому они не обязательны. Кроме того, вы не можете быть уверены, что API Hacker News всегда имеет определённое свойство для каждого объекта в массиве.

Это для PropTypes. Но есть ещё один аспект. Вы можете определить свойства по умолчанию в своём компоненте. Давайте снова вернёмся к компоненту Button. У свойства className есть значение по умолчанию в объявлении компонента.

src/App.js

```
const Button = ({  
  onClick,  
  className = '',  
  children  
) =>  
  ...
```

Вы можете заменить его внутренним свойством по умолчанию в React:

src/App.js

```
const Button = ({  
  onClick,  
  className,  
  children  
) =>  
  <button  
    onClick={onClick}  
    className={className}  
    type="button"  
  >  
    {children}  
  </button>
```

```
Button.defaultProps = {  
  className: '',  
};
```

Так же, как и параметр по умолчанию в ES6, свойство по умолчанию гарантирует, что для свойства установлено значение по умолчанию, если родительский компонент не указал его. Проверка типа PropType происходит после вычисления свойства по умолчанию.

Если вы снова запустите свои тесты, вы увидите ошибки PropType по вашим компонентам в командной строке. Это может произойти, потому что вы не определили все свойства для своих компонентов в тестах, которые определены в соответствии с вашим определением PropType. Однако сами тесты проходят правильно. Вы можете передать все необходимые свойства компонентам в своих тестах, чтобы избежать этих ошибок.

Упражнения:

- определите интерфейс PropType для компонента Search

- добавьте и обновите интерфейсы PropType при добавлении и обновлении компонентов в последующих главах
- узнайте больше про PropTypes в React¹⁴⁰

¹⁴⁰<https://ru.react.js.org/docs/typechecking-with-proptypes.html>

Отладка с помощью инструментов разработчика React

В этом последнем разделе представлен полезный инструмент, обычно используемый для исследования и отладки React-приложений. [React Developer Tools](#)¹⁴¹ позволяет вам изучать иерархию, свойства и состояние React-компонентов. Он распространяется в виде расширения для браузера (для Chrome и Firefox) и как автономное приложение (которое работает с другими окружениями). После установки на сайтах, разработанных с помощью React, загорится значок расширения. На таких страницах вы увидите вкладку «React» в инструментах разработчика браузера.

Давайте попробуем это расширение в вашем приложении Hacker News. В большинстве браузеров быстрый способ загрузить его в *инструменты разработчика* — щёлкнуть правой кнопкой мыши на странице, а затем нажать «Inspect». Сделайте это, когда ваши приложения загружены, затем нажмите вкладку «React». Вы увидите его иерархию элементов с корневым элементом `<App>`. Если вы раскроете его, вы найдёте экземпляры ваших компонентов `<Search>`, `<Table>` и `<Button>`.

Расширение показывает на боковой панели состояние компонента и свойства для выбранного элемента. Например, если вы нажмёте на `<App>`, вы увидите, что у него нет свойств, но у него уже есть состояние. Очень простая методика отладки заключается в том, чтобы отслеживать изменение состояния приложения из-за взаимодействия с пользователем.

Во-первых, вы можете проверить параметр «Highlight Updates» (обычно над деревом элементов). Во-вторых, вы можете ввести разную строку поиска в поле ввода приложения. Как вы увидите, только `searchTerm` в состоянии компонента будет изменён. Вы уже знаете, что это произойдёт, но теперь вы можете увидеть, что он работает, как и планировалось.

Наконец, вы можете нажать кнопку «Search». Состояние `searchKey` будет немедленно изменено для того же значения, что и `searchTerm`, и после этого объект ответа будет добавлен в `results` через несколько секунд. Асинхронный характер вашего кода теперь виден вашим глазам.

И последнее, но не менее важное: если вы щёлкните правой кнопкой мыши по любому элементу, то выпадающее меню покажет вам несколько полезных опций. Например, вы можете скопировать свойство или имя элемента, найти соответствующий DOM-узел или перейти к исходному коду приложения в браузере. Этот последний параметр очень полезен для вставки точек останова и отладки ваших функций JavaScript.

Упражнения:

- установите расширение [React Developer Tools](#)¹⁴² в своём любимом браузере

¹⁴¹<https://github.com/facebook/react-devtools>

¹⁴²<https://github.com/facebook/react-devtools>

- запустите приложение-клон Hacker News и изучите его с помощью расширения
 - поэкспериментируйте с изменениями состояния и свойств
 - следите за тем, что происходит при запуске асинхронного запроса
 - выполните несколько запросов, в том числе повторяющихся. Понаблюдайте за работой механизма кеширования
- Узнайте больше о том, как отлаживать ваши функции JavaScript в браузере¹⁴³

¹⁴³<https://developers.google.com/web/tools/chrome-devtools/javascript/>

Вы узнали, как организовать свой код и как его протестировать! Давайте повторим последние темы:

- React
 - PropTypes позволяет вам определять проверки типов для компонентов
 - Jest позволяет писать снимки для ваших компонентов
 - Enzyme позволяет писать модульные тесты для ваших компонентов
 - React Developer Tools — полезный инструмент для отладки
- ES6
 - операции импорта и экспорта помогают организовать ваш код
- Общее
 - организация кода позволяет масштабировать ваше приложение с помощью лучших практик

Исходный код можно найти в [официальном репозитории](#)¹⁴⁴.

¹⁴⁴<https://github.com/the-road-to-learn-react/hackernews-client/tree/5.4>

Продвинутые React-компоненты

В этой главе основное внимание будет уделено реализации продвинутых компонентов React. Вы узнаете о компонентах высшего порядка (higher-order component) и о том, как их реализовать. Кроме того, вы погрузитесь в более сложные темы в React и реализуете сложные взаимодействия с ним.

Ссылка на DOM-элемент

Иногда вам нужно взаимодействовать с вашими DOM-узлами в React. Атрибут `ref` даёт вам доступ к узлу в ваших элементах. Обычно это антипаттерн в React, потому что вам следует использовать декларативный способ работы и односторонний поток данных. Вы узнали об этом, когда добавили своё первое поле ввода для поиска. Но есть определённые случаи, когда вам нужен доступ к узлу DOM. В официальной документации упоминаются три варианта использования:

- для использования API DOM (фокус, воспроизведение мультимедиа и т.д.)
- для вызова императивных анимаций DOM-узлов
- для интеграции со сторонней библиотекой, которой нужен DOM-узел (например, [D3.js¹⁴⁵](#))

Давайте сделаем это на примере с помощью компонента `Search`. Когда приложение отрисовывается в первый раз, поле ввода должно быть сфокусировано. Это один из вариантов, когда вам нужен доступ к API DOM. В этой главе вы узнаете, как это работает, но поскольку это не очень полезно для самого приложения, мы опустим изменения после главы. Однако вы можете сохранить эти изменения для своего приложения.

В общем, вы можете использовать атрибут `ref` как в функциональных компонентах без состояния, так и в ES6-классах компонентов. В варианте использования в примере с фокусом вам понадобится метод жизненного цикла. Вот почему подход сначала демонстрируется с использованием атрибута `ref` с классовым компонентом.

Первоначальный шаг — миграция с функционального компонента без состояния на классовый компонент.

src/App.js

```
class Search extends Component {
  render() {
    const {
      value,
      onChange,
      onSubmit,
      children
    } = this.props;

    return (
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
```

¹⁴⁵<https://d3js.org/>

```
        onChange={onChange}
    />
    <button type="submit">
      {children}
    </button>
  </form>
);
}
}
```

Объект `this` ES6-класса компонента помогает нам ссылаться на DOM-узел с атрибутом `ref`.

src/App.js

```
class Search extends Component {
  render() {
    const {
      value,
      onChange,
      onSubmit,
      children
    } = this.props;

    return (
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
          onChange={onChange}
          ref={(node) => { this.input = node; }}
        />
        <button type="submit">
          {children}
        </button>
      </form>
    );
  }
}
```

Теперь вы можете сфокусировать поле ввода, когда компонент монтируется с использованием объекта `this`, соответствующего метода жизненного цикла и API DOM.

src/App.js

```
class Search extends Component {
  componentDidMount() {
    if (this.input) {
      this.input.focus();
    }
  }

  render() {
    const {
      value,
      onChange,
      onSubmit,
      children
    } = this.props;

    return (
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
          onChange={onChange}
          ref={(node) => { this.input = node; }}
        />
        <button type="submit">
          {children}
        </button>
      </form>
    );
  }
}
```

Поле ввода должно быть сфокусировано при отрисовке приложения. В основном это для использования атрибута `ref`.

Но как бы вы получили доступ к `ref` в функциональном компоненте, не имеющем состояния, без использования объекта `this`? Следующий пример демонстрирует такой случай.

src/App.js

```
const Search = ({  
  value,  
  onChange,  
  onSubmit,  
  children  
) => {  
  let input;  
  return (  
    <form onSubmit={onSubmit}>  
      <input  
        type="text"  
        value={value}  
        onChange={onChange}  
        ref={(node) => input = node}  
      />  
      <button type="submit">  
        {children}  
      </button>  
    </form>  
  );  
}
```

Теперь вы сможете получить доступ к элементу ввода DOM. В примере использования с фокусировкой в поле ввода это не поможет вам, потому что у вас нет метода жизненного цикла в функциональном компоненте без состояния, чтобы установить фокус на элементе. Но в будущем вы можете столкнуться с другими вариантами использования, когда имеет смысл использовать функциональный компонент без состояния с атрибутом `ref`.

Упражнения

- узнайте подробнее про [использование атрибута ref в React¹⁴⁶](#)
- узнайте подробнее в целом про [атрибут ref в React¹⁴⁷](#)

¹⁴⁶<https://www.robinwieruch.de/react-ref-attribute-dom-node/>

¹⁴⁷<https://ru.react.js.org/docs/refs-and-the-dom.html>

Загрузка ...

Теперь вернёмся к приложению. Возможно, вам понадобится показать индикатор загрузки при отправке поискового запроса к API Hacker News. Поскольку запрос является асинхронным, вы, возможно, захотите дать своему пользователю обратную связь, что сейчас кое-что произойдёт. Давайте определим повторно используемый компонент Loading в файле `src/App.js`.

src/App.js

```
const Loading = () =>
  <div>Загрузка ...</div>
```

Теперь вам нужно свойство для хранения состояния загрузки. В зависимости от значения состояния загрузки вы можете позднее отобразить компонент Loading.

src/App.js

```
class App extends Component {
  _isMounted = false;

  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
      error: null,
      isLoading: false,
    };
  }

  ...
}

...
```

Начальное значение такого свойства `isLoading` устанавливается в значение `false`. Вы не загружаете ещё что-либо перед тем, как компонент App смонтирован.

Когда вы делаете запрос, то устанавливаете состояние загрузки в значение `true`. В конце концов, когда запрос будет успешным, вы можете установить состояние загрузки обратно на значение `false`.

src/App.js

```
class App extends Component {  
  ...  
  
  setSearchTopStories(result) {  
    ...  
  
    this.setState({  
      results: {  
        ...results,  
        [searchKey]: { hits: updatedHits, page }  
      },  
      isLoading: false  
    });  
  }  
  
  fetchSearchTopStories(searchTerm, page = 0) {  
    this.setState({ isLoading: true });  
  
    axios(`#${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${page}&${PARAM_HPP}${DEFAULT_HPP}`)  
      .then(result => this._isMounted && this.setSearchTopStories(result.data))  
      .catch(error => this._isMounted && this.setState({ error }));  
  }  
  
  ...  
}
```

На последнем шаге вы будете использовать компонент Loading в App. Условная отрисовка на основе значения состояния загрузки будет определять, отображать ли компонент Loading или компонент Button. Последний компонент — это кнопка для получения дополнительных (больше) данных.

src/App.js

```
class App extends Component {  
  ...  
  
  render() {  
    const {  
      searchTerm,  
      results,  
      searchKey,  
      error,  
      isLoading  
    } = this.state;  
  
    ...  
  
    return (  
      <div className="page">  
        ...  
        <div className="interactions">  
          { isLoading  
            ? <Loading />  
            : <Button  
                onClick={() => this.fetchSearchTopStories(searchKey, page + 1)}  
              >  
              More  
            </Button>  
          }  
        </div>  
      </div>  
    );  
  }  
}
```

Первоначально компонент Loading будет отображаться при запуске приложения, потому что вы совершаете запрос в `componentDidMount()`. Компонент Table отсутствует, потому что список пуст. Когда ответ приходит от API Hacker News, показывается результат, состояние загрузки устанавливается на значение `false`, а компонент Loading исчезает. Вместо этого отображается кнопка «Больше историй» для получения ещё одной порции данных. Когда вы отправите запрос для получения дополнительных данных, кнопка снова исчезнет и вместо неё отобразится компонент Loading.

Упражнения:

- используйте библиотеку, такую как [Font Awesome¹⁴⁸](#), чтобы показать иконку загрузки вместо надписи “Загрузка ...”

¹⁴⁸<https://fontawesome.io/>

Компоненты высшего порядка

Компоненты высшего порядка (Higher-order components, НОС) — продвинутая концепция React. Компоненты высшего порядка эквивалентны функциям высшего порядка. Они принимают любые входные данные — чаще всего компонент, но ещё необязательные аргументы — и возвращают компонент в качестве возвращаемого значения. Возвращаемый компонент является расширенной версией переданного в НОС компонента и может использоваться в JSX.

Компоненты высшего порядка используются в разных случаях. Они могут подготавливать свойства, управлять состоянием или изменять представление компонента. Одним из вариантов использования может быть использование НОС в качестве помощника для условной отрисовки. Представьте, что у вас есть компонент `List`, отображающий список элементов или вообще ничего, потому что список пуст или равняется `null`. С использованием НОС можно ничего не отрисовывать, если нет списка. С другой стороны, компоненту простого `List` больше не нужно беспокоиться о несуществующем списке. Его задача только отрисовать список.

Давайте создадим простой НОС, который принимает компонент и возвращает компонент. Вы можете поместить его в файл `src/App.js`.

`src/App.js`

```
function withFoo(Component) {
  return function(props) {
    return <Component { ...props } />;
  }
}
```

Есть одно чистое соглашение — префикс имени НОС начинается с `with`. Поскольку вы используете JavaScript ES6, вы можете более кратко сделать НОС с помощью стрелочных функций из ES6.

`src/App.js`

```
const withFoo = (Component) => (props) =>
  <Component { ...props } />
```

В этом примере входной компонент будет оставаться таким же, как и выходной компонент. Ничего интересного не происходит. Он отрисовывает один и тот же экземпляр компонента и передаёт все свойства возвращаемому компоненту. Но это бесполезно. Давайте улучшим получаемый из НОС компонент. Компонент на выходе должен показывать компонент `Loading`, когда значение состояния загрузки равняется `true`, в противном случае он должен отображать переданный компонент. Отрисовка по условию — отличный вариант использования НОС.

src/App.js

```
const withLoading = (Component) => (props) =>
  props.isLoading
    ? <Loading />
    : <Component { ...props } />
```

В зависимости от значения свойства `isLoading` вы можете применить условную отрисовку. Функция вернёт компонент `Loading` или входной компонент.

В общем случае может быть очень эффективно использовать оператор расширения для объекта, например, для объекта `props` из предыдущего примера, при передаче его на вход компонента. Смотрите разницу в следующем фрагменте кода.

Код

```
// деструктуризация свойств перед их передачей
const { foo, bar } = props;
<SomeComponent foo={foo} bar={bar} />

// но можно использовать оператор расширения для передачи всех свойств объекта
<SomeComponent { ...props } />
```

Есть одна маленькая деталь, которую следует избегать: сейчас вы передаёте все свойства, включая `isLoading`, путём расширения объекта во входной компонент. Однако входной компонент может не поддерживать свойство `isLoading`, тогда вы можете использовать деструктуризацию оставшихся свойств из ES6 для решения этой проблемы.

src/App.js

```
const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading
    ? <Loading />
    : <Component { ...rest } />
```

Он выбирает одно свойство из объекта, но сохраняет оставшиеся свойства в объекте. Он также поддерживает работу с несколькими свойствами. Возможно, вы уже прочитали об этом в [деструктурирующем присваивании¹⁴⁹](#).

Теперь вы можете использовать НОС в своём JSX. В приложении может показываться либо кнопка «Больше историй», либо компонент `Loading`. Компонент `Loading` уже инкапсулирован в НОС, но отсутствует передаваемый компонент. В случае показа компонента `Button` или компонента `Loading`, компонент `Button` будет входным компонентом для НОС. Улучшенный выходной компонент — компонент `ButtonWithLoading`.

¹⁴⁹https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

src/App.js

```
const Button = ({  
  onClick,  
  className = '',  
  children,  
}) =>  
  <button  
    onClick={onClick}  
    className={className}  
    type="button"  
>  
  {children}  
</button>  
  
const Loading = () =>  
  <div>Загрузка ...</div>  
  
const withLoading = (Component) => ({ isLoading, ...rest }) =>  
  isLoading  
  ? <Loading />  
  : <Component { ...rest } />  
  
const ButtonWithLoading = withLoading(Button);
```

Теперь всё установлено. В качестве последнего шага вам необходимо использовать компонент `ButtonWithLoading`, который получает состояние загрузки в виде дополнительного свойства. В то время как НОС использует свойство загрузки, все другие свойства передаются компоненту `Button`.

src/App.js

```
class App extends Component {  
  
  ...  
  
  render() {  
    ...  
  
    return (  
      <div className="page">  
        ...  
      <div className="interactions">  
        <ButtonWithLoading
```

```

    isLoading={isLoading}
    onClick={() => this.fetchSearchTopStories(searchKey, page + 1)}
  >
  Больше историй
</ButtonWithLoading>
</div>
</div>
);
}
}

```

Когда вы снова запустите свои тесты, вы заметите, что ваш тест снимком компонента App завершился неудачей. В командной строке различия могут выглядеть с помощью унифицированного формата diff следующим образом:

Командная строка

```

- <button
-   className=""
-   onClick={[Function]}
-   type="button"
- >
-   Больше историй
- </button>
+ <div>
+   Загрузка ...
+ </div>

```

Вы можете либо исправить компонент сейчас, когда вам кажется, что в нём что-то не так, либо можете принять новый снимок. Поскольку вы реализовали компонент Loading в этой главе, вы можете принять изменённый снимок в командной строке в интерактивном teste.

Компоненты высшего порядка — продвинутая методика в React. У неё есть несколько целей, таких как улучшенная возможность повторного использования компонентов, более сильная абстракция, композиция (компоновка) компонентов и управлением свойствами, состоянием и представлением. Не переживайте, если не сразу понимаете всё это. Потребуется время, чтобы привыкнуть к этому всему.

Я призываю вас прочитать [простое введение в компоненты высшего порядка¹⁵⁰](#). Статья даёт вам ещё один подход для их изучения, показывает элегантный способ использования их в функциональном программировании и, в частности, решает проблему условной отрисовки с компонентами высшего порядка.

¹⁵⁰<https://www.robinwieruch.de/gentle-introduction-higher-order-components/>

Упражнения:

- прочитайте [лёгкое введение в компоненты высшего порядка \(HOC\)](#)¹⁵¹
- поэкспериментируйте с HOC, который вы создали
- подумайте о случае использования, где другой HOC будет иметь смысл
 - реализуйте такой HOC, если в этом есть необходимость

¹⁵¹<https://www.robinwieruch.de/gentle-introduction-higher-order-components/>

Продвинутая сортировка

Вы уже реализовали поиск на стороне клиента и сервера. Поскольку у вас есть компонент Table, имеет смысл улучшить таблицу дополнительными возможностями. Как насчёт внедрения функциональности сортировки по каждому столбцу при нажатии на заголовки столбцов компонента Table?

Можно было бы написать свою собственную функцию сортировки, но лично я предпочитаю использовать вспомогательную библиотеку как раз для таких случаев. [Lodash¹⁵²](#) — одна из таких утилитарных библиотек, но вы можете использовать любую другую библиотеку, которая вам подходит. Давайте установим Lodash и воспользуемся её функциональностью сортировки.

Командная строка

```
npm install lodash
```

Теперь вы можете импортировать Lodash в файле *src/App.js*.

src/App.js

```
import React, { Component } from 'react';
import axios from 'axios';
import { sortBy } from 'lodash';
import './App.css';
```

У вас есть несколько столбцов в компоненте Table. Есть столбцы для заголовков, авторов, комментариев и очков. Вы можете определить функции сортировки, где каждая функция принимает список и возвращает список элементов, отсортированных по определённому свойству. Кроме того, вам понадобится одна функция сортировки по умолчанию, которая не сортирует, а возвращает обычный (неотсортированный) список. Это будет вашим первоначальным состоянием.

src/App.js

```
...
```

```
const SORTS = {
  NONE: list => list,
  TITLE: list => sortBy(list, 'title'),
  AUTHOR: list => sortBy(list, 'author'),
  COMMENTS: list => sortBy(list, 'num_comments').reverse(),
  POINTS: list => sortBy(list, 'points').reverse(),
};
```

¹⁵²<https://lodash.com/>

```
class App extends Component {  
  ...  
}  
...
```

Как вы можете увидеть сами, две функции сортировки возвращают список в обратном порядке. Это из-за того, что нужно отображение элементов с наибольшим количеством комментариев и очков вместо элементов с наименьшим количеством при сортировке списка в первый раз.

Объект SORTS позволяет вам ссылаться на любую функцию сортировки.

Снова ваш компонент App отвечает за сохранение состояния сортировки. Первоначальным состоянием будет функция сортировки по умолчанию, которая не сортирует элементы вообще, а возвращает входной список в качестве вывода.

src/App.js

```
this.state = {  
  results: null,  
  searchKey: '',  
  searchTerm: DEFAULT_QUERY,  
  error: null,  
  isLoading: false,  
  sortKey: 'NONE',  
};
```

После выбора другого ключа сортировка sortKey, скажем, AUTHOR, список будет отсортирован соответствующей функцией сортировки из объекта SORTS.

Теперь вы можете определить новый метод класса в компоненте App, который просто устанавливает sortKey в локальном состоянии компонента. После этого sortKey можно использовать для получения функции сортировки для её применения в вашем списке.

src/App.js

```
class App extends Component {  
  _isMounted = false;  
  
  constructor(props) {  
  
    ...  
  
    this.needsToSearchTopStories = this.needsToSearchTopStories.bind(this);  
    this.setSearchTopStories = this.setSearchTopStories.bind(this);
```

```
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
    this.onSort = this.onSort.bind(this);
}

...
onSort(sortKey) {
  this.setState({ sortKey });
}
...
}
```

Следующий шаг — передать метод и sortKey в ваш компонент Table.

src/App.js

```
class App extends Component {
```

```
  ...
  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading,
      sortKey
    } = this.state;
  ...
  return (
    <div className="page">
      ...
      <Table
        list={list}
        sortKey={sortKey}
        onSort={this.onSort}
      </Table>
    </div>
  );
}
```

```
        onDismiss={this.onDismiss}
      />
      ...
    </div>
  );
}
}
```

Компонент Table отвечает за сортировку списка. Он принимает одну из функций из объекта SORT по ключу sortKey и передаёт список в качестве входных данных. После этого он выводит отсортированной список.

src/App.js

```
const Table = ({
  list,
  sortKey,
  onSort,
  onDismiss
}) =>
<div className="table">
  {SORTS[sortKey](list).map(item =>
    <div key={item.objectID} className="table-row">
      ...
    </div>
  )}
</div>
```

Теоретически список будет отсортирован одной из функций. Но сортировка по умолчанию установлена на NONE, поэтому элементы не сортируются. До сих пор никто не выполнял метод onSort() для изменения sortKey. Расширим компонент Table, добавив заголовки столбцов, использующие компоненты Sort в столбцах для сортировки по каждому столбцу.

src/App.js

```
const Table = ({
  list,
  sortKey,
  onSort,
  onDismiss
}) =>
<div className="table">
  <div className="table-header">
    <span style={{ width: '40%' }}>
```

```
<Sort
  sortKey={'TITLE'}
  onSort={onSort}
>
  Заголовок
</Sort>
</span>
<span style={{ width: '30%' }}>
  <Sort
    sortKey={'AUTHOR'}
    onSort={onSort}
>
  Автор
</Sort>
</span>
<span style={{ width: '10%' }}>
  <Sort
    sortKey={'COMMENTS'}
    onSort={onSort}
>
  Комментарии
</Sort>
</span>
<span style={{ width: '10%' }}>
  <Sort
    sortKey={'POINTS'}
    onSort={onSort}
>
  Очки
</Sort>
</span>
<span style={{ width: '10%' }}>
  Архив
</span>
</div>
{SORTS[sortKey](list).map(item =>
  ...
)}
</div>
```

Каждый компонент Sort получает определённую функцию sortKey и общую функцию onSort(), передавая ей sortKey для установки конкретного ключа.

src/App.js

```
const Sort = ({ sortKey, onSort, children }) =>
  <Button onClick={() => onSort(sortKey)}>
    {children}
  </Button>
```

Как вы можете видеть, компонент Sort повторно использует общий компонент Button. При нажатии на кнопку каждому из компонентов Button передаётся sortKey, который будет установлен методом onSort(). Теперь есть возможность сортировать список при нажатии на заголовки столбцов.

Можно сделать ещё одно небольшое улучшение внешнего вида. Пока кнопка в заголовке столбца выглядит немного нелепо. Давайте дадим кнопке в компоненте Sort собственный CSS-класс, используя className.

src/App.js

```
const Sort = ({ sortKey, onSort, children }) =>
  <Button
    onClick={() => onSort(sortKey)}
    className="button-inline">
    {children}
  </Button>
```

Теперь кнопка выглядит красиво. Следующая цель заключалась бы в реализации обратной сортировки. Список должен быть отсортирован в обратном порядке по двойному нажатию на компонент Sort. Во-первых, нужно определить новое свойство для обратной сортировки с булевым значением. Сортировка может быть либо прямой, либо обратной.

src/App.js

```
this.state = {
  results: null,
  searchKey: '',
  searchTerm: DEFAULT_QUERY,
  error: null,
  isLoading: false,
  sortKey: 'NONE',
  isSortReverse: false,
};
```

Теперь в вашем методе сортировки вы можете вычислить, отсортирован ли список в обратном порядке. В обратном порядке, если значение состояния sortKey совпадает с переданным sortKey, а значение состояния обратного сортировки уже не равняется true.

src/App.js

```
onSort(sortKey) {
  const isSortReverse = this.state.sortKey === sortKey && !this.state.isSortReverse;
  this.setState({ sortKey, isSortReverse });
}
```

Снова вы можете передать свойство, указывающее на обратную сортировку в компонент Table.

src/App.js

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading,
      sortKey,
      isSortReverse
    } = this.state;

    ...

    return (
      <div className="page">
        ...
        <Table
          list={list}
          sortKey={sortKey}
          isSortReverse={isSortReverse}
          onSort={this.onSort}
          onDismiss={this.onDismiss}
        />
        ...
      </div>
    );
  }
}
```

Теперь для вычисления данных в компоненте Table необходима стрелочная функция.

src/App.js

```
const Table = ({  
    list,  
    sortKey,  
    isSortReverse,  
    onSort,  
    onDismiss  
) => {  
    const sortedList = SORTS[sortKey](list);  
    const reverseSortedList = isSortReverse  
        ? sortedList.reverse()  
        : sortedList;  
  
    return(  
        <div className="table">  
            <div className="table-header">  
                ...  
            </div>  
            {reverseSortedList.map(item =>  
                ...  
            )}  
        </div>  
    );  
}
```

Сортировка в обратном порядке должна сейчас работать.

И последнее, но не менее важное: вам нужно разобраться с одним открытым вопросом улучшения пользовательского интерфейса. Может ли пользователь определить, по какому столбцу сейчас происходит сортировка? Пока это невозможно. Давайте предоставим пользователю такую визуальную возможность.

Каждый компонент Sort уже получает свой конкретный sortKey. Его можно использовать для определения активной в данный момент сортировки. Вы можете передать sortKey из внутреннего состояния компонента в качестве активного ключа сортировки в свой компонент Sort.

src/App.js

```
const Table = ({  
    list,  
    sortKey,  
    isSortReverse,  
    onSort,  
    onDismiss  
) => {  
    const sortedList = SORTS[sortKey](list);  
    const reverseSortedList = isSortReverse  
        ? sortedList.reverse()  
        : sortedList;  
  
    return(  
        <div className="table">  
            <div className="table-header">  
                <span style={{ width: '40%' }}>  
                    <Sort  
                        sortKey={'TITLE'}  
                        onSort={onSort}  
                        activeSortKey={sortKey}>  
                    >  
                    Заголовок  
                </Sort>  
            </span>  
            <span style={{ width: '30%' }}>  
                <Sort  
                    sortKey={'AUTHOR'}  
                    onSort={onSort}  
                    activeSortKey={sortKey}>  
                >  
                    Автор  
                </Sort>  
            </span>  
            <span style={{ width: '10%' }}>  
                <Sort  
                    sortKey={'COMMENTS'}  
                    onSort={onSort}  
                    activeSortKey={sortKey}>  
                >  
                    Комментарии  
                </Sort>  
            </span>  
        </div>  
    );  
};
```

```
<span style={{ width: '10%' }}>
  <Sort
    sortKey={'POINTS'}
    onSort={onSort}
    activeSortKey={sortKey}>
  </Sort>
</span>
<span style={{ width: '10%' }}>
  Архив
</span>
</div>
{reverseSortedList.map(item =>
  ...
)}
</div>
);
}
```

Теперь в компоненте Sort можно узнать, основываясь на sortKey и activeSortKey, активен ли данный столбец сортировки или нет. Добавьте компоненту Sort ещё дополнительный атрибут className, применяемый при сортировке для визуального различия столбцов.

```
src/App.js
-----
const Sort = ({

  sortKey,
  activeSortKey,
  onSort,
  children
}) => {
  const sortClass = ['button-inline'];

  if (sortKey === activeSortKey) {
    sortClass.push('button-active');
  }

  return (
    <Button
      onClick={() => onSort(sortKey)}
      className={sortClass.join(' ')}
    >
      {children}
    
```

```
    </Button>
  );
}
```

Подобный способ определения CSS-классов, используя константу `sortClass` немного неуклюжий, не считаете так? Существует замечательная небольшая библиотека для элегантной установки CSS-классов. Сначала нужно её установить.

Командная строка

```
npm install classnames
```

И теперь нужно импортировать её в верху файла `src/App.js`.

src/App.js

```
import React, { Component } from 'react';
import axios from 'axios';
import { sortBy } from 'lodash';
import classNames from 'classnames';
import './App.css';
```

Теперь вы можете воспользоваться ею для определения CSS-классов в зависимости от условий через всё тот же атрибут `className` компонента.

src/App.js

```
const Sort = ({  
  sortKey,  
  activeSortKey,  
  onSort,  
  children  
}) => {  
  const sortClass = classNames(  
    'button-inline',  
    { 'button-active': sortKey === activeSortKey }  
  );  
  
  return (  
    <Button  
      onClick={() => onSort(sortKey)}  
      className={sortClass}  
    >  
      {children}  
    </Button>  
  );  
};
```

```
        </Button>
    );
}
```

Теперь же, когда вы выполните тесты, то увидите неудачные снимки тестов, а также не прошедшие модульные тесты компонента Table. Поскольку вы изменили внешний вид компонентов, вы можете принять эти снимки. Но вам нужно исправить модульный тест. В вашем файле *src/App.test.js* вам необходимо указать ключи *sortKey* и *isSortReverse* с логическим значением для компонента Table.

src/App.test.js

...

```
describe('Table', () => {
  const props = {
    list: [
      { title: '1', author: '1', num_comments: 1, points: 2, objectID: 'y' },
      { title: '2', author: '2', num_comments: 1, points: 2, objectID: 'z' },
    ],
    sortKey: 'TITLE',
    isSortReverse: false,
  };
  ...
});
```

Ещё раз вам может потребоваться принять неудачные снимки компонента Table, потому что были расширены его свойства.

Наконец, работа с расширенной сортировкой завершена.

Упражнения:

- используйте библиотеку, такую как [Font Awesome¹⁵³](#), чтобы указать на применяемую (прямую или обратную) сортировку
 - это может быть иконка стрелки вверх или стрелки вниз рядом с каждым заголовком в компоненте Sort
- узнайте подробнее про [библиотеку classnames¹⁵⁴](#)

¹⁵³<https://fontawesome.io/>

¹⁵⁴<https://github.com/JedWatson/classnames>

Вы изучили продвинутые технологии компонентов React! Давайте повторим последние пройденные темы:

- React
 - атрибут `ref` для ссылок на DOM-узлы
 - Компоненты высшего порядка — это распространённый способ создания продвинутых компонентов
 - внедрение расширенных взаимодействий в React
 - назначение CSS-классов по условию с помощью небольшой вспомогательной библиотеки `classNames`
- ES6
 - деструктуризация оставшихся свойств для разделения объектов и массивов

Исходный код можно найти в [официальном репозитории](#)¹⁵⁵.

¹⁵⁵<https://github.com/the-road-to-learn-react/hackernews-client/tree/5.5>

Управление состоянием в React и за его пределами

Вы уже изучили основы управления состоянием в React в предыдущих главах. В данной главе мы немножко углубимся в эту тему. Вы узнаете о передовых практиках, как их применять, а также как использовать стороннюю библиотеку для управления состоянием.

Подъём состояния

В вашем приложении только компонент App — классовый компонент с состоянием. Он обрабатывает все состояния в приложении и содержит много логики в своих методах. Возможно, вы заметили, что передаёте много свойств компоненту Table. Большинство из них используются только в компоненте Table. В заключение можно утверждать, что нет никакого смысла в том, что компонент App знает обо всём этом.

Вся функциональность сортировки используется только в компоненте Table. Вы можете перенести её в компонент Table, потому что компонент App вообще не должен знать об этом. Процесс рефакторинга подсостояния от одного компонента к другому известен как *подъём состояния*. В вашем случае вам нужно переместить состояние, неиспользуемое в компоненте App, в компонент Table. Состояние перемещается от родительского к дочернему компоненту.

Для работы с состоянием и методами класса в компоненте Table, его нужно преобразовать в компонент ES6-класса. Рефакторинг из функционального компонента без состояния в компонент класса ES6 — это просто.

Компонент Table как функциональный компонент без состояния выглядит примерно так:

src/App.js

```
const Table = ({  
    list,  
    sortKey,  
    isSortReverse,  
    onSort,  
    onDismiss  
) => {  
    const sortedList = SORTS[sortKey](list);  
    const reverseSortedList = isSortReverse  
        ? sortedList.reverse()  
        : sortedList;  
  
    return(  
        ...  
    );  
}
```

Компонент Table в виде класса ES6 компонента будет таким:

src/Table.js

```
class Table extends Component {
  render() {
    const {
      list,
      sortKey,
      isSortReverse,
      onSort,
      onDismiss
    } = this.props;

    const sortedList = SORTS[sortKey](list);
    const reverseSortedList = isSortReverse
      ? sortedList.reverse()
      : sortedList;

    return (
      ...
    );
  }
}
```

Поскольку вам нужно работать с состоянием и методами в своём компоненте, вам нужно добавить конструктор и начальное состояние.

src/App.js

```
class Table extends Component {
  constructor(props) {
    super(props);

    this.state = {};
  }

  render() {
    ...
  }
}
```

Теперь вы можете переместить состояния и методы класса, относящиеся к функциональности сортировки из вашего компонента App в компонент Table.

src/Table.js

```
class Table extends Component {
  constructor(props) {
    super(props);

    this.state = {
      sortKey: 'NONE',
      isSortReverse: false,
    };

    this.onSort = this.onSort.bind(this);
  }

  onSort(sortKey) {
    const isSortReverse = this.state.sortKey === sortKey && !this.state.isSortRevers\
e;
    this.setState({ sortKey, isSortReverse });
  }

  render() {
    ...
  }
}
```

Не забудьте удалить перемещённое состояние и метод onSort() из компонента App.

src/App.js

```
class App extends Component {
  _isMounted = false;

  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
      error: null,
      isLoading: false,
    };
  }

  this.setSearchTopStories = this.setSearchTopStories.bind(this);
```

```
this.fetchSearchTopStories = this.fetchSearchTopStories.bind(this);
this.onDismiss = this.onDismiss.bind(this);
this.onSearchSubmit = this.onSearchSubmit.bind(this);
this.onSearchChange = this.onSearchChange.bind(this);
this.needsToSearchTopStories = this.needsToSearchTopStories.bind(this);
}

...
}
```

Кроме того, вы можете сделать API компонента Table более лёгким. Удалите свойства, которые передаются ему из компонента App, потому что теперь они обрабатываются внутри компонента Table.

src/App.js

```
class App extends Component {

    ...

    render() {
        const {
            searchTerm,
            results,
            searchKey,
            error,
            isLoading
        } = this.state;

        ...

        return (
            <div className="page">
                ...
                { error
                    ? <div className="interactions">
                        <p>Что-то пошло не так.</p>
                    </div>
                    : <Table
                        list={list}
                        onDismiss={this.onDismiss}
                    />
                }
            </div>
        );
    }
}
```

```
    ...
  </div>
);
}
}
```

Теперь в компоненте Table вы можете использовать внутренний метод `onSort()` и внутреннее состояние Table.

src/App.js

```
class Table extends Component {

  // ...

  render() {
    const {
      list,
      onDismiss
    } = this.props;

    const {
      sortKey,
      isSortReverse,
    } = this.state;

    const sortedList = SORTS[sortKey](list);
    const reverseSortedList = isSortReverse
      ? sortedList.reverse()
      : sortedList;

    return(
      <div className="table">
        <div className="table-header">
          <span style={{ width: '40%' }}>
            <Sort
              sortKey={'TITLE'}
              onSort={this.onSort}
              activeSortKey={sortKey}
            >
              Заголовок
            </Sort>
          </span>
          <span style={{ width: '30%' }}>
```

```
<Sort
  sortKey={'AUTHOR'}
  onSort={this.onSort}
  activeSortKey={sortKey}
>
  Автор
</Sort>
</span>
<span style={{ width: '10%' }}>
  <Sort
    sortKey={'COMMENTS'}
    onSort={this.onSort}
    activeSortKey={sortKey}
>
  Комментарии
</Sort>
</span>
<span style={{ width: '10%' }}>
  <Sort
    sortKey={'POINTS'}
    onSort={this.onSort}
    activeSortKey={sortKey}
>
  Очки
</Sort>
</span>
<span style={{ width: '10%' }}>
  Архив
</span>
</div>
{ reverseSortedList.map((item) =>
  ...
)
}
</div>
);
}
}
```

Ваше приложение по-прежнему должно работать, хотя вы сделали большой рефакторинг. Вы переместили функциональность и состояние ближе к другому компоненту. Другие компоненты снова стали более лёгкими. Кроме того, API компонента стал более лёгким, поскольку он внутренне выполняет функции сортировки.

Процесс подъёма состояния может идти по обратному пути: от дочернего к родительскому компоненту. Это называется *подъём состояния вверх*. Представьте, что вы имеете дело с внутренним состоянием в дочернем компоненте. Теперь вы хотите удовлетворить требование по отображению состояния в своём родительском компоненте. Вам нужно будет поднять состояние к вашему родительскому компоненту. Но это идёт ещё дальше. Представьте, что вы хотите показать состояние в одноуровневом компоненте вашего дочернего компонента. Снова вам нужно будет поднять состояние до вашего родительского компонента. Родительский компонент имеет дело с внутренним состоянием, но предоставляет его для обоих дочерних компонентов.

Упражнения:

- прочитайте подробнее про [поднятие состояния в React](#)¹⁵⁶
- прочитайте больше про поднятие состояния в [статье про изучение React перед использованием Redux](#)¹⁵⁷

¹⁵⁶<https://ru.react.js.org/docs/lifting-state-up.html>

¹⁵⁷<https://www.robinwieruch.de/learn-react-before-using-redux/>

Пересмотр: setState()

До сих пор вы использовали React `setState()` для управления состоянием вашего внутреннего компонента. Вы можете передать объект функции, в которой вы можете частично обновить внутреннее состояние.

Код

```
this.setState({ foo: bar });
```

Но `setState()` принимает не только объект. Второй вариант использования этого метода включает передачу функции для обновления состояния.

Код

```
this.setState((prevState, props) => {
  ...
});
```

Когда это может пригодиться? Существует один важный случай использования, когда имеет смысл передать функцию вместо объекта. Это когда вы обновляете состояние в зависимости от предыдущего состояния или свойства. Если вы не используете функцию, управление внутренним состоянием может вызвать баги (ошибки).

Но почему это вызывает баги при использовании объекта вместо функции, когда обновление зависит от предыдущего состояния или свойства? Метод React `setState()` — асинхронный. React группирует вызовы `setState()` и выполняет их рано или поздно. Может случиться так, что предыдущее состояние или свойство изменились между тем, когда вы будете использовать его в вызове `setState()`.

Код

```
const { fooCount } = this.state;
const { barCount } = this.props;
this.setState({ count: fooCount + barCount });
```

Представьте, что `fooCount` и `barCount`, таким образом, состояние или свойство, изменяются где-то ещё асинхронно, когда вы вызываете `setState()`. В растущем приложении у вас будет более одного вызова `setState()` в приложении. Поскольку `setState()` выполняется асинхронно, вы можете использовать его в примере на устаревших значениях.

С помощью функционального подхода функция `setState()` — колбэк, который работает с состоянием и свойствами во время выполнения колбэк-функции. Даже при том, что `setState()` является асинхронным, с функцией он принимает состояние и свойства в момент его выполнения.

Код

```
this.setState((prevState, props) => {
  const { fooCount } = prevState;
  const { barCount } = props;
  return { count: fooCount + barCount };
});
```

Теперь вернёмся к вашему коду, чтобы исправить это поведение. Вместе мы исправим его для одного места, где используется `setState()` и полагается на состояние или свойства. Впоследствии вы сможете исправить это и в других местах.

Метод `setSearchTopStories()` опирается на предыдущее состояние и, следовательно, является прекрасным примером использования функции вместо объекта в `setState()`. Сейчас так выглядит следующий фрагмент кода.

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;
  const { searchKey, results } = this.state;

  const oldHits = results && results[searchKey]
    ? results[searchKey].hits
    : [];

  const updatedHits = [
    ...oldHits,
    ...hits
  ];

  this.setState({
    results: {
      ...results,
      [searchKey]: { hits: updatedHits, page }
    },
    isLoading: false
  });
}
```

Вы извлекаете значения из состояния, но обновляете состояние в зависимости от предыдущего состояния асинхронно. Теперь вы можете использовать функциональный подход для предотвращения ошибок из-за устаревшего состояния.

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;

  this.setState(prevState => {
    ...
  });
}
```

Вы можете переместить весь блок, который вы уже внедрили в эту функцию. Вам нужно только изменить, что вы работаете уже с prevState, а не с this.state.

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;

  this.setState(prevState => {
    const { searchKey, results } = prevState;

    const oldHits = results[searchKey]
      ? results[searchKey].hits
      : [];

    const updatedHits = [
      ...oldHits,
      ...hits
    ];

    return {
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      },
      isLoading: false
    };
  });
}
```

Это исправит проблему с устаревшим состоянием. Однако есть ещё одно улучшение. Поскольку это функция, её можно извлечь для улучшения читабельности. Это ещё одно преимущество использования функции перед объектом — функция может работать вне компонента. Но вам нужно использовать функцию высшего порядка, чтобы передать результат. В конце концов, вы хотите обновить состояние на основе полученного результата из API.

src/App.js

```
setSearchTopStories(result) {
  const { hits, page } = result;
  this.setState(updateSearchTopStoriesState(hits, page));
}
```

Функция updateSearchTopStoriesState() должна возвращать функцию. Это функция высшего порядка. Вы можете определить эту функцию высшего порядка вне вашего компонента App. Обратите внимание на то, как теперь меняется объявление функции.

src/App.js

```
const updateSearchTopStoriesState = (hits, page) => (prevState) => {
  const { searchKey, results } = prevState;

  const oldHits = results && results[searchKey]
    ? results[searchKey].hits
    : [];

  const updatedHits = [
    ...oldHits,
    ...hits
  ];

  return {
    results: {
      ...results,
      [searchKey]: { hits: updatedHits, page }
    },
    isLoading: false
  };
};

class App extends Component {
  ...
}
```

Вот и всё. Использование функции вместо объекта в setState() исправляет потенциальные ошибки, а также повышает читаемость и поддержку вашего кода. Кроме того, код становится тестируемым вне компонента приложения. Вы можете экспортить его и написать тест в виде упражнения.

Упражнение:

- прочитать больше о том, как [в React правильно использовать состояние](#)¹⁵⁸
- экспортить updateSearchTopStoriesState из файла
 - напишите тест для него, который передаёт данные (истории, страницы) и составленное предыдущее состояние и, наконец, ожидает новое состояние
- улучшите методы setState() для использования функции
 - но только тогда, когда это имеет смысл, т.е. когда есть зависимость от свойства или состояния
- снова запустите свои тесты и убедитесь, что всё в порядке

¹⁵⁸<https://ru.react.js.org/docs/state-and-lifecycle.html#%D0%9F%D1%80%D0%B0%D0%B2%D0%B8%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5-%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%D1%81%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D1%8F>

Укroщение состояния

Предыдущие главы показали, что управление состоянием может быть важной темой в более крупных приложениях. В целом, не только React, но и много фреймворков SPA противостоят этому. За последние годы приложения стали более сложными. Одна из серьёзных проблем в веб-приложениях в настоящее время — укroщение и контроль состояния.

По сравнению с другими решениями, React уже сделал большой шаг вперёд. Однонаправленный поток данных и простой API для управления состоянием в компоненте являются незаменимыми. Эти концепции упрощают управление состоянием и изменениями в вашем состоянии на уровне компонентов и в определённой степени на уровне приложения.

В условиях роста приложения становится всё труднее размышлять об изменениях состояния. Вы можете допустить баги, работая с устаревшим состоянием при использовании объекта вместо функции в `setState()`. Вы должны поднять состояние, чтобы поделиться необходимым или скрыть ненужное состояние между компонентами. Может случиться так, что компонент должен поднять состояние, потому что его родственный (одноуровневый) компонент зависит от него. Возможно, компонент находится далеко в дереве компонентов и, следовательно, вам нужно разделить это состояние по всему дереву компонентов. И наконец, компоненты в большей степени задействованы в управлении состоянием. Но ведь основная ответственность компонентов должна представлять пользовательский интерфейс, не так ли?

Из-за всех этих причин существуют независимые решения по управлению состоянием и его сохранению. Эти решения используются не только в React. Однако это то, что делает экосистему React таким мощным местом. Вы можете использовать различные решения проблем, связанных с состоянием. Чтобы решить проблему масштабирования управления состоянием, вы, возможно, слышали о библиотеках [Redux¹⁵⁹](#) или [MobX¹⁶⁰](#). Вы можете использовать любое из этих решений в приложении React. Они включают расширения, такие как [react-redux¹⁶¹](#) и [mobx-react¹⁶²](#), чтобы интегрировать их в слой представления React.

Redux и MobX выходят за рамки данной книги. Когда вы закончите книгу, вы получите руководство о том, как вы можете продолжать изучать React и его экосистему. Одним из путей обучения может стать изучение Redux. Прежде чем вы погрузитесь в тему внешнего управления состоянием, я могу порекомендовать прочитать [этую статью¹⁶³](#). Она призвана дать вам лучшее понимание того, как изучать внешнее управление состоянием.

Упражнения:

- узнайте больше о [внешнем управлении состоянием и о том, как его изучать¹⁶⁴](#)

¹⁵⁹<http://redux.js.org/docs/introduction/>

¹⁶⁰<https://mobx.js.org/>

¹⁶¹<https://github.com/reactjs/react-redux>

¹⁶²<https://github.com/mobxjs/mobx-react>

¹⁶³<https://www.robinwieruch.de/redux-mobx-confusion/>

¹⁶⁴<https://www.robinwieruch.de/redux-mobx-confusion/>

- ознакомьтесь с моей второй книгой про управление состоянием в React¹⁶⁵

¹⁶⁵<https://roadtoreact.com/>

Вы изучили некоторые продвинутые техники управления состоянием в React! Давайте повторим последние темы:

- React
 - поднятие состояния вверх и вниз до нужных компонентов
 - `setState()` может использовать функцию в качестве аргумента для предотвращения ошибок, связанных с устаревшим состоянием
 - существующие сторонние решения, которые помогут вам приручить состояние

Исходный код можно найти в [официальном репозитории¹⁶⁶](#).

¹⁶⁶<https://github.com/the-road-to-learn-react/hackernews-client/tree/5.6>

Заключительные шаги к развертыванию в продакшнене

В последних главах будет показано, как развернуть приложение в продакшнене. Вы будете использовать бесплатный хостинг Heroku. По пути развертывания (далее — деплой) вашего приложения вы узнаете больше о *create-react-app*.

Команда eject

Следующий шаг и знание **не требуются** для деплоя приложения на продакшен. Тем не менее, я хочу объяснить это. *create-react-app* поставляется с одной возможностью, чтобы оставаться расширяемым, но и также предотвращать привязку к поставщику (*vendor lock-in*). Привязка к поставщику обычно происходит, когда вы покупаете, привязываетесь к технологии без лёгкой замены её использования на другую в будущем. К счастью, в *create-react-app* есть такая возможность, называемая «*eject*».

В вашем *package.json* вы найдёте скрипты для запуска (*start*), выполнения тестов (*test*) и сборки (*build*) приложения. Но есть ещё последний скрипт — *eject*. Вы можете его попробовать, но учтите, что обратного пути уже не будет. Это **односторонняя операция**. Как только вы выполните *eject*, пути назад не будет, это означает, что вы больше не сможете использовать *create-react-app*! Если вы только начали изучать React, нет смысла оставлять удобное окружение, предоставляемое *create-react-app*, т.к. выполнение данной команды раскроет все зависимости и конфигурационные файлы, созданные и используемые утилитой *create-react-app*.

Если вы запустите `npm run eject`, команда скопирует всю конфигурацию в новый каталог *config/*, а зависимости в файл *package.json*. Таким образом вы сконвертируете весь проект в собственную настройку для сборки вашего приложения со всеми используемыми инструментами, включая Babel и Webpack; в итоге получится так, словно вы никогда и не использовали *create-react-app*. После этого у вас будет полный контроль над всеми этими инструментами и вы будете точно знать, что использовалось под капотом *create-react-app* при сборки вашего приложения.

В официальной документации говорится, что *create-react-app* подходит для малых и средних проектов. Поэтому не считайте, что вы обязательно должны использовать команду «*eject*».

Упражнения:

- прочитайте подробнее про команду [eject¹⁶⁷](#)

¹⁶⁷<https://github.com/facebook/create-react-app/blob/master/packages/react-scripts/template/README.md#npm-run-eject>

Деплой приложения

В конце концов, ни одно приложение не должно оставаться локально на компьютере. Вы хотите показать его всему миру. Heroku — это платформа как услуга (Platform as a Service, PaaS), где вы можете разместить ваше приложение. Они предлагают безболезненную интеграцию с React. В частности, приложение, созданное с помощью *create-react-app*, можно развернуть за считанные минуты. Это деплой приложения без всякой настройки конфигурации, что следует философии *create-react-app*.

Перед деплоем приложения в Heroku необходимо выполнить два шага:

- установите [CLI Heroku¹⁶⁸](#)
- зарегистрируйте [бесплатный аккаунт в Heroku¹⁶⁹](#)

Если у вас установлен Homebrew, вы можете установить CLI Heroku из командной строки:

Командная строка

```
brew update  
brew install heroku-toolbelt
```

Теперь можно воспользоваться git и CLI Heroku для деплоя вашего приложения.

Командная строка

```
git init  
heroku create -b https://github.com/mars/create-react-app-buildpack.git  
git add .  
git commit -m "react-create-app on Heroku"  
git push heroku master  
heroku open
```

Вот и всё. Надеюсь, ваше приложение запущено и работает. При возникновении проблем можно ознакомиться со следующими ресурсами:

- [Основы Git и GitHub¹⁷⁰](#)
- [Деплой React-приложения без какой-либо конфигурации¹⁷¹](#)
- [Пакет для сборки приложения, созданного с помощью create-react-app, на Heroku¹⁷²](#)

¹⁶⁸<https://devcenter.heroku.com/articles/heroku-cli>

¹⁶⁹<https://www.heroku.com/>

¹⁷⁰<https://www.robinwieruch.de/git-essential-commands/>

¹⁷¹<https://blog.heroku.com/deploying-react-with-zero-configuration>

¹⁷²<https://github.com/mars/create-react-app-buildpack>

Краткий обзор

Это была последняя глава книги. Я надеюсь, что вам понравилось читать данную книгу и что она помогла вам в изучении React. Если вам понравилась книга, поделитесь ею с вашими друзьями в целях научить их React. Её следует распространять в виде небольшого подарка. Кроме того, для меня бы многое значило, если бы вы уделили несколько минут своего времени для написания отзыва о книге на [Amazon¹⁷³](#) или на [Goodreads¹⁷⁴](#).

Итак, что мне теперь делать после прочтения этой книги? Вы можете либо расширить приложение по своему желанию или попробовать сделать собственный проект с использованием React. Прежде чем вы погрузитесь в другую книгу, курс или какой-либо обучающий материал, вам нужно создать собственный реальный проект на React. Сделайте его в течение одной недели, разверните где-нибудь на продакшене, и свяжитесь со [мной¹⁷⁵](#) или другими заинтересованными людьми для демонстрации того, что вы сделали, используя React. Мне любопытно, что вы создадите после того, как прочтете книгу.

Если вы в поисках того, как ещё можно расширить своё приложение, я могу порекомендовать несколько путей обучения после того, как вы использовали только обычный React в этой книге:

- **Управление состоянием:** Вы использовали `this.setState()` и `this.state` в React для получения доступа к локальному состоянию компонента и управления им. Это прекрасное начало. Однако в более крупном приложении вы непременно столкнётесь с [ограничениями локального состояния компонента React¹⁷⁶](#). Поэтому можно использовать стороннюю библиотеку управления состоянием, например [Redux](#) или [MobX¹⁷⁷](#). На сайте курса [Путь к изучению React¹⁷⁸](#), вы найдёте курс «Укрощение состояния в React» (*Taming the State in React*), который учит расширенному управлению состоянием в React, используя Redux и MobX. В этом курсе есть также электронная книга, но я рекомендую кроме неё погрузиться в исходный код и просмотр скринкастов. Если вам понравилась эта книга, вы обязательно должны заказать «Укрощение состояния в React».
- **Подключение к базе данных и/или аутентификация:** В растущем приложении React вы рано или поздно будете сохранять данные. Данные должны храниться в базе данных, чтобы они оставались доступными после завершения сессии браузера и могли использоваться различными пользователями вашего приложения. Самый простой способ

¹⁷³<https://www.amazon.com/dp/B077HJFCQX>

¹⁷⁴<https://www.goodreads.com/book/show/37503118-the-road-to-learn-react>

¹⁷⁵<https://twitter.com/rwieruch>

¹⁷⁶<https://www.robinwieruch.de/learn-react-before-using-redux/>

¹⁷⁷<https://www.robinwieruch.de/redux-mobx-confusion/>

¹⁷⁸<https://roadtoreact.com/>

добавить базу данных — использовать Firebase. В [этой всеобъемлющей обучающей статье¹⁷⁹](#) вы найдёте пошаговое руководство по использованию аутентификации Firebase (регистрация, вход, выход и т.д.) в контексте React. Кроме того, вы будете использовать базу данных Firebase в режиме реального времени для хранения пользовательских сущностей. После этого решать вам, хранить больше данных в базе данных, которые необходимы вашему приложению.

- **Инструменты Webpack и Babel:** В книге вы использовали *create-react-app* для создания приложения. В какой-то момент, когда вы овладеете React, вы, возможно, захотите изучить инструменты, которые используются внутри него. Это позволит вам создать собственный проект без применения *create-react-app*. Я могу порекомендовать последовать минимальной настройке [Webpack и Babel¹⁸⁰](#). Позже, вы можете добавить больше инструментов к уже существующим. Например, вы можете [использовать ESLint¹⁸¹](#), чтобы придерживаться единого стиля кода в приложении.
- **Синтаксис компонентов React:** Возможности и передовые практики по реализации компонентов React со временем меняются. Вы найдёте много способов написания компонентов React, особенно классовых компонентов React, на других учебных ресурсах. Вы можете посмотреть [этот репозиторий на GitHub¹⁸²](#), чтобы узнать об альтернативном способе написания классов-компонентов в React. Используя объявления полей класса, вы можете писать их ещё более лаконично в будущем.
- **Другие проекты:** Изучив чистый React, всегда хорошо сначала применить полученные знания в своих проектах, прежде чем начинать изучать что-то новое. Вы можете написать собственную игру в крестики-нолики или простой калькулятор на React. Есть много обучающих статей, которые используют только React, чтобы создать что-то захватывающее. Посмотрите мою статью про создание [постстраничный список с бесконечной прокруткой¹⁸³](#), просмотр твитов из ленты Twitter¹⁸⁴ или подключение React-приложения к Stripe для снятия денег¹⁸⁵. Экспериментируйте с этими мини-приложениями, чтобы научиться уверенно работать с React.
- **Компоненты пользовательского интерфейса (UI):** Вы совершаете ошибку, слишком рано представив библиотеку для пользовательского интерфейса компонентов в вашем проекте. Во-первых, вам нужно знать, как с нуля реализовать и использовать выпадающий список, чекбокс или диалоговое окно, используя стандартные HTML-элементы, после чего реализовать их в React. У большинства этих компонентов будет собственное локальное состояние. Например, у чекбокса будут свойства состояния, указывающие, отмечен он или нет. Таким образом, вы должны реализовать их как управляемые компоненты. После того, как у вас есть все основные реализации UI, вы можете задуматься над созданием библиотеки компонентов пользовательского интерфейса, собрав в неё чекбоксы, диалоговые окна и всё то, что вы реализовали в виде React-компонентов.

¹⁷⁹<https://www.robinwieruch.de/complete-firebase-authentication-react-tutorial/>

¹⁸⁰<https://www.robinwieruch.de/minimal-react-webpack-babel-setup/>

¹⁸¹<https://www.robinwieruch.de/react-eslint-webpack-babel/>

¹⁸²<https://github.com/the-road-to-learn-react/react-alternative-class-component-syntax>

¹⁸³<https://www.robinwieruch.de/react-paginated-list/>

¹⁸⁴<https://www.robinwieruch.de/react-svg-patterns/>

¹⁸⁵<https://www.robinwieruch.de/react-express-stripe-payment/>

- **Организация кода:** По пути изучения книги вы наткнулись на одну главу про организацию кода. Вы можете применить полученные значения сейчас, если ещё не сделали этого. Организуйте ваши компоненты в структурированные файлы и каталоги (модули). Кроме того, это поможет вам понять и изучить принципы разделения кода, повторного использования, дальнейшей поддержки и проектирования API модуля. В конце концов ваше приложение будет увеличиваться, и вам так или иначе придётся разделять его по модулям. Так что лучше начать делать это прямо сейчас.
- **Тестирование:** Книга только лишь прошлась по тестированию. Если вы не знакомы с общей темой тестирования, вы можете глубже погрузиться в концепции модульного тестирования и интеграционного тестирования, особенно в контексте приложений React. Я бы порекомендовал для этого использовать такие инструменты, как Enzyme и Jest, чтобы совершенствовать ваш подход к тестированию с помощью модульных тестов и тестов снимками в React.
- **Маршрутизация:** Вы можете реализовать маршрутизацию для своего приложения с помощью [react-router¹⁸⁶](#). До сих пор в приложении была только одна страница. React Router поможет вам иметь несколько страниц, доступных по нескольким URL-адресам. Когда вы добавляете маршрутизацию в приложение, вам не нужны дополнительные запросы к веб-серверу для получения следующей страницы. Маршрутизатор сделает всё за вас на стороне клиента. Так что придётся замарать руки и настроить маршрутизацию в своём приложении.
- **Проверка типов:** В одной главе вы использовали PropTypes в React для определения интерфейсов компонента. Это распространённая хорошая практика для предотвращения ошибок. Но PropTypes проверяются только во время выполнения. Вы можете пойти дальше и добавить статическую проверку типов, выполняемую во время компиляции. [TypeScript¹⁸⁷](#) — один из таких популярных подходов. Но в экосистеме React люди часто используют [Flow¹⁸⁸](#). Я могу порекомендовать дать шанс Flow, если вы заинтересованы в том, чтобы сделать ваше приложение более надёжным.
- **React Native:** [React Native¹⁸⁹](#) доставляет ваши приложения на мобильные устройства. Вы можете применить свои знания из React для создания приложений для iOS и Android. Кривая обучения, как только вы научились React, не должна быть слишком крутой в React Native. Обе технологии разделяют одни и те же принципы. На мобильном устройстве вы будете сталкиваться только с компонентами макета, отличными от тех, к которым вы привыкли в веб-приложениях.

В общем, я приглашаю вас посетить мой [сайт¹⁹⁰](#) для получения более интересных тем по веб-разработке и разработке программного обеспечения в принципе. Вы можете [подписаться на мою информационную рассылку¹⁹¹](#) для получения обновлений о новых статьях, книгах и

¹⁸⁶<https://github.com/ReactTraining/react-router>

¹⁸⁷<https://www.typescriptlang.org/>

¹⁸⁸<https://flowtype.org/>

¹⁸⁹<https://facebook.github.io/react-native/>

¹⁹⁰<https://www.robinwieruch.de>

¹⁹¹<https://www.getrevue.co/profile/rwieruch>

курсах. Кроме того, сайт курса [Путь к изучению React¹⁹²](#) предлагает более сложные курсы, чтобы узнать об экосистеме React. Вам стоит их увидеть!

И последнее, но не менее важное: я надеюсь найти больше [меценатов¹⁹³](#), у которых есть возможность поддерживать мой контент. Есть много студентов, которые не могут позволить себе оплатить образовательный контент. Вот почему я размещаю много своего контента бесплатно. Поддержите меня в моих делах в качестве моего патрона, и я смогу продолжать поддерживать эти трудозатраты, чтобы бесплатно обучать других.

Напомню ещё раз, если вы получили удовольствие от книги, я хочу, чтобы вы на мгновение подумали о человеке, которому могла бы понравиться данная книга, чтобы научиться React. Познакомьтесь с этим человеком и поделитесь с ним этой книгой. Это бы много значило для меня. Книга предназначена для других. Со временем она будет улучшаться, когда больше людей прочитают её и поделятся своими отзывами со мной. Я надеюсь также увидеть ваши отклики, отзывы или оценки!

Большое вам спасибо за то, что вы прочитали «Путь к изучению React».

Робин

¹⁹²<https://roadtoreact.com>

¹⁹³<https://www.patreon.com/rwieruch>