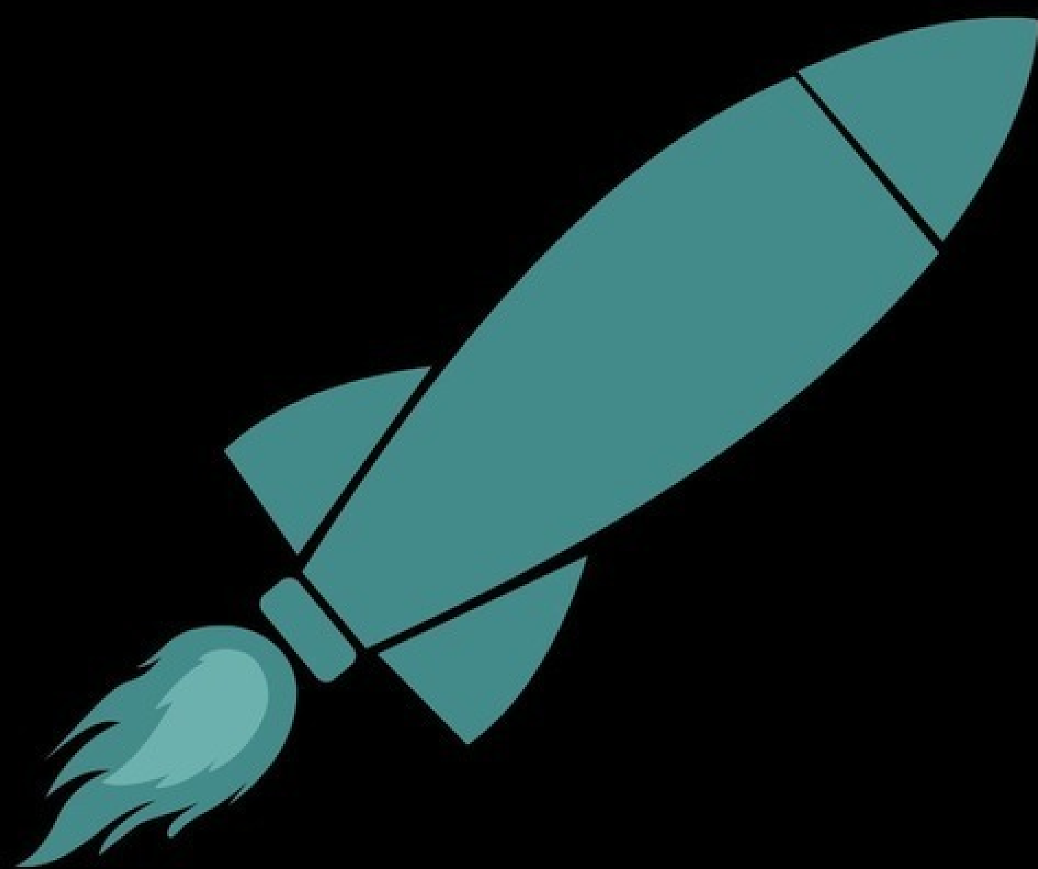
Ирина Кириченко

REACT и TYPESCRIPT: Практическое руководство

Быстрый старт



Ирина Кириченко

React и TypeScript: Практическое руководство. Быстрый старт

«Издательские решения»

Кириченко И.

React и TypeScript: Практическое руководство. Быстрый старт / И. Кириченко — «Издательские решения»,

ISBN 978-5-00-609759-9

Это руководство предназначено для тех, кто желает освоить создание веб-приложений, используя такие современные инструменты, как React и TypeScript. Оно предлагает последовательный путь в освоении этих технологий — от начальных концепций до более глубоких аспектов. Независимо от вашего уровня подготовки, предлагаются практические примеры и полезные советы, чтобы сделать ваши знания еще более глубокими и применимыми.

ISBN 978-5-00-609759-9

© Кириченко И.
© Издательские решения

Содержание

Содержание 8
Часть 1. Основы React и TypeScript 10
Глава 1. Введение в React и TypeScript 10
Заключение 12
Глава 2. Установка и настройка окружения разработки 13
2.1 Установка Node. js и npm 13
2.2 Настройка редактора кода 13
2.3 Создание проекта с использованием Create React App 13
2.4 Установка TypeScript 14
2.5 Заключение 14
Глава 3. Понятие компонента в React 16
3.1 Понятие компонента в React 16
3.2 Создание функционального компонента 16
3.3 Использование компонента в приложении 17
3.6 Файл README.md 21
3.7 Запуск приложения 22
3.8 Заключение 22
Глава 4. JSX и его синтаксис 23
4.1 Понятие JSX 23
4.2 Основы синтаксиса JSX 23
4.3 Рендеринг JSX 24
4.4 Преимущества JSX 25
4.5 Заключение 26
Глава 5. Работа с компонентами и их вложенность 27
5.1 Создание компонента 27
5.2 Использование компонентов 28
5.3 Вложенность компонентов 28
5.4 Компонент высшего порядка 30
5.5 Заключение 31
Глава 6. Основы использования props и state 33
6.1 Введение в props и state 33
6.2 Использование пропсов 35
6.5 Заключение 38
Глава 7. Методы жизненного цикла компонентов 39
7.1 Введение в методы жизненного цикла 39
7.2 Основные методы жизненного цикла 39
7.3 Пример использования методов жизненного цикла 39
7.4 Заключение 40
Глава 8. Автоматическое создание объектов props в React 41
8.1 Введение в объект props в React 41
8.2 Роль объекта props в компонентах 42
8.3 Создание объекта props 42
8.4 Доступ к свойствам через объект props 43
Конец ознакомительного фрагмента. 44

React и TypeScript: Практическое руководство Быстрый старт

Ирина Кириченко

© Ирина Кириченко, 2023

ISBN 978-5-0060-9759-9
Создано в интеллектуальной издательской системе Ridero

Все права защищены. Никакая часть этой книги не может быть воспроизведена, передана в какой-либо форме или любыми средствами, электронными или механическими, включая фотокопирование, запись или любые другие системы хранения и передачи информации, без предварительного письменного разрешения владельца авторских прав.

Это руководство предназначено для тех, кто желает освоить создание веб-приложений, используя такие современные инструменты, как React и TypeScript. Оно предлагает последовательный путь в освоении этих технологий – от начальных концепций до более глубоких аспектов. Независимо от вашего уровня подготовки, предлагаются практические примеры и полезные советы, чтобы сделать ваши знания еще более глубокими и применимыми.

Предполагается, что читатель уже обладает базовыми знаниями в JavaScript, HTML и CSS. Если вы новичок в этих технологиях, рекомендуется ознакомиться с их основами перед началом чтения.

Содержание

Часть 1: Основы React и TypeScript
– Введение в React и TypeScript 1
– Установка и настройка окружения разработки 6
– Понятие компонента в React 10
– JSX и его синтаксис 20
– Работа с компонентами и их вложенность 25
– Основы использования props и state 32
– Методы жизненного цикла компонентов 40
– Автоматическое создание объектов props в React 43
– Дополнительная информация:
Расширения файлов в React:.js,.jsx,.tsx 50
Обзор популярных React Hooks 51

Часть 2: Работа с формами и событиями
– Обработка событий в React 52
– Работа с формами и контролируемые компоненты 57
– Валидация ввода данных 61
– Управление состоянием и обновление компонентов 65
– Дополнительная информация:
Babel 71

Часть 3: Работа с данными и запросами
– Работа с API и запросы к серверу 72
– Обработка ответов и обновление состояния компонентов 78
– Асинхронное программирование 82
– Использование библиотек для упрощения работы с данными 91
– Дополнительная информация:
Методы HTTP «POST» и «GET» 99
JSON формат 101

Часть 4: Роутинг и навигация
– Введение в роутинг в React (React Router) 102
– Создание многостраничных приложений 105
– Динамическая навигация 110
– Дополнительная информация:
Метод map () в JavaScript 120
Метод reduce () в JavaScript 121

Часть 5: TypeScript в React
– Введение в TypeScript: Основные концепции и типы данных 122
– Добавление TypeScript в проект React 127
– Введение в основные концепции TypeScript:
Type, Interface и Generics 129
– Модули и пространства имен в TypeScript 134
– Дополнительная информация:
Области видимости в TypeScript 138

Часть 6: Углубленная типизация в React с TypeScript
– Декораторы в TypeScript 140
– Аннотации типов для компонентов и функций 143
– Типизация props и state компонентов 146
– Работа с событиями и обработчиками событий 150
– Использование дженериков (generics) в React 153
– Дополнительная информация:
Файлы с расширением. d. ts 157

Часть 7: Продвинутые темы
– Контекст и передача данных между компонентами 159
– Redux и управление глобальным состоянием 164
– Асинхронные операции с Redux Thunk 170
– Рефакторинг и лучшие практики 175
– Дополнительная информация:
Функция fetch 177

Часть 8: Тестирование и развертывание
– Тестирование компонентов
с использованием Jest и React Testing Library 178
– Автоматизация сборки и развертывания
с помощью инструментов, таких как Webpack и Babel 182
– Дополнительная информация:
Полезные библиотеки для стилизации React-приложений 185
Часть 9: Проекты и практика

Часть 1. Основы React и TypeScript

Глава 1. Введение в React и TypeScript

В мире веб-разработки существует множество разнообразных инструментов и технологий, React и TypeScript выделяются среди них как наиболее популярные и востребованные.

TypeScript – это язык программирования, который расширяет язык JavaScript, добавляя статическую типизацию. Это позволяет определять типы данных для переменных, параметров функций и других объектов в коде, добавляя статическую типизацию. Эта статическая типизация делает код более надежным и облегчает его поддержку и документирование.

Преимущества использования TypeScript включают:

– Статическая типизация. TypeScript добавляет статическую типизацию к JavaScript, что позволяет обнаруживать и предотвращать множество ошибок на этапе разработки. Это особенно полезно в больших проектах, где сложно отслеживать типы данных и гарантировать их правильность. С помощью TypeScript можно определить типы данных для компонентов, состояния, пропсов[1] и других объектов, что делает код более надежным и легко читаемым.

– Улучшенная поддержка IDE[2]. TypeScript хорошо интегрируется с множеством современных интегрированных сред разработки (IDE): Visual Studio Code, WebStorm, Sublime Text, Atom, Eclipse, IntelliJ IDEA, NetBeans. Это обеспечивает доступ к таким функциям разработки, как автодополнение кода, анализ ошибок и подсказки по типам.

– Лучшая документация и понимание кода. Использование TypeScript улучшает самодокументируемость кода. Другие разработчики легче понимают, какие данные ожидаются, и какие функции должны выполняться в компонентах и модулях проекта.

– Рефакторинг[3]. TypeScript облегчает рефакторинг кода, так как IDE предоставляет инструменты для автоматической замены типов данных при переименовании переменных и изменении интерфейсов. Это ускоряет процесс обслуживания кода.

– Более безопасное состояние и пропсы. TypeScript позволяет строго типизировать состояние и пропсы в компонентах React, что уменьшает вероятность ошибок и облегчает их отслеживание.

– Интеграция со сторонними библиотеками. TypeScript поддерживает определение типов для сторонних библиотек, что позволяет использовать их в проектах и быть уверенными в том, что код будет правильно типизирован.

– Улучшенная работа в команде. Статическая типизация делает код более надежным, что особенно важно в совместной работе над проектами с другими разработчиками.

– Поддержка новых возможностей, таких как ECMAScript[4] и React. TypeScript быстро внедряет новые возможности JavaScript и React, что позволяет использовать последние технологические достижения в любом проекте.

---

[1] Пропсы (props) представляют собой механизм, с помощью которого компоненты React могут принимать данные и настраиваться извне.

[2] IDE (Integrated Development Environment) – программное обеспечение, предназначенное для разработки, отладки и управления кодом при создании программных приложений.

[3] Рефакторинг (refactoring) – это процесс улучшения структуры и качества кода программы без изменения её внешнего поведения. Основная цель рефакторинга – упростить код, сделать его более понятным, поддерживаемым и расширяемым, устранить дублирование кода и уменьшить технический долг

[4] ECMAScript (или сокращенно ES) – это стандартный набор правил, по которым описывается язык JavaScript. Он включает в себя синтаксис, типы данных, ключевые слова и другие элементы, необходимые для написания программ на JavaScript.

– Совместное использование React и TypeScript может улучшить качество кода, сделать его более надежным и облегчить его поддержку. Это особенно полезно в больших и сложных проектах, где строгость типов и управление состоянием играют важную роль.

React представляет собой библиотеку, которая позволяет разработчикам строить современные интерактивные веб-приложения без перезагрузки страницы. Выпущенный в 2013 году, React стал одним из самых популярных инструментов для фронтенд-разработки. Основная его идея заключается в разделении пользовательского интерфейса на множество маленьких компонентов, каждый из которых может быть разработан и объединен независимо друг от друга. Это подход делает код более модульным и управляемым, а так же обеспечивает быстрое обновление данных на стороне клиента без необходимости перезагрузки страницы.

Преимущества использования React по сравнению с другими фреймворками и библиотеками:

– Компонентный подход. React строится на компонентах, что способствует упорядоченности кода и его разбиению на небольшие модули. Таким образом, есть возможность создавать переиспользуемые компоненты, которые упрощают разработку и обслуживание приложения.

– Виртуальная DOM[5] (Модель объектов документа). React использует виртуальную DOM, что позволяет эффективно обновлять только те части интерфейса, которые изменились, вместо перерисовки всего дерева DOM. Это повышает производительность приложений.

– Синтаксис JSX[6] (JavaScript XML) Синтаксис JSX делает написание компонентов более читаемым и понятным. Он позволяет встраивать HTML-подобный код непосредственно в JavaScript.

– Однонаправленный поток данных[7]. React использует однонаправленный поток данных, что делает управление состоянием более прозрачным и предсказуемым.

– Композиция компонентов. Можно легко комбинировать и вкладывать компоненты друг в друга, создавая сложные пользовательские интерфейсы из простых компонентов. Это способствует модульности и переиспользованию кода, позволяя создавать масштабируемые и поддерживаемые приложения.

– Большое сообщество и экосистема. React имеет большое сообщество разработчиков и множество сторонних библиотек и инструментов, что облегчает разработку и расширение функциональности приложений.

– Поддержка серверного рендеринга[8]. React позволяет выполнять серверный рендеринг, что улучшает SEO и производительность веб-приложений.

– Гибкость и адаптивность. React не ограничивает разработчика в выборе других технологий и библиотек. Можно интегрировать React в различные стеки разработки.

– Официальные инструменты и документация. React предоставляет широкий спектр официальных инструментов, включая React DevTools[9] и Create React App[10], а также отличную документацию, что упрощает начало работы и разработку приложений.

---

[5] DOM – это структурное представление веб-страницы или документа в виде иерархии объектов, которое браузер использует для представления и манипуляции содержимым веб-страницы

[6] JSX – это специальный синтаксис, используемый в React (и некоторых других библиотеках), который объединяет структуру и стили, а также язык гипертекстовой разметки (HTML) в одном файле.

[7] однонаправленный поток данных (One-Way Data Flow) – это концепция, которая описывает способ передачи данных и управления состоянием в приложении, предполагая, что данные в приложении двигаются только в одном направлении, обычно от родительских компонентов к дочерним.

[8] Серверный рендеринг (Server-Side Rendering, SSR) – это метод разработки веб-приложений, при котором генерация HTML-кода для веб-страницы происходит на сервере, а не на стороне клиента (в браузере). Вместо того чтобы браузер загружал пустую HTML-страницу и затем заполнял ее данными и контентом с использованием JavaScript, при SSR сервер отправляет полностью готовую к отображению веб-страницу.

[9] React DevTools – это расширение для браузера и набор инструментов, предназначенных для отладки и анализа приложений, разработанных с использованием библиотеки React.

– Компоненты для мобильной разработки. Существует множество библиотек и фреймворков, таких как React Native, которые позволяют разрабатывать мобильные приложения с использованием тех же компонентов и навыков React.

В сочетании с TypeScript, React становится еще более мощным инструментом для создания современных веб-приложений с высокой производительностью и надежностью. Эти две технологии дополняют друг друга и позволяют создавать сложные приложения с четкой структурой и надежной типизацией данных.

Преимущества использования React и TypeScript вместе:

– Статическая типизация. TypeScript позволяет выявлять ошибки на этапе разработки, что сокращает количество багов в приложении.

– Интеллектуальное автодополнение. TypeScript предоставляет мощные инструменты автодополнения и подсказок, что упрощает работу с большими кодовыми базами.

– Большое сообщество и экосистема. React и TypeScript оба имеют активные сообщества и множество библиотек, что упрощает разработку и поддержку проектов.

– Изучение React и TypeScript предоставляет разработчикам целый набор навыков и инструментов для создания современных веб-приложений. Независимо от того, являетесь ли вы начинающим разработчиком или опытным специалистом, знание этих технологий открывает перед вами широкие перспективы.

– Работа в крупных компаниях. Множество крупных компаний используют React и TypeScript для разработки своих продуктов, и знание этих технологий делает вас более востребованным на рынке труда.

– Создание собственных проектов. React и TypeScript позволяют вам реализовать свои идеи и создать собственные веб-приложения.

– Улучшение качества кода. Статическая типизация и компонентный подход React помогут вам создавать более надежные и легко поддерживаемые приложения.

– Обучение и обмен опытом. Знание React и TypeScript открывает двери для обучения других и обмена опытом в сообществе разработчиков.

Заключение

React и TypeScript предоставляют возможность создавать веб-приложения, которые обладают не только красивыми и интуитивно понятными интерфейсами, но и высокой степенью надежности. Они позволяют разработчикам создавать гибкие и модульные приложения, что делает их код более поддерживаемым и масштабируемым.

Сочетание React и TypeScript позволяет создавать приложения, которые могут быть успешно применены в самых разных областях, от разработки веб-сайтов и онлайн-магазинов до масштабных корпоративных систем. Благодаря своей активной и поддерживаемой сообществом экосистеме, React и TypeScript остаются на переднем крае веб-технологий, гарантируя, что разработчики будут всегда в курсе последних тенденций и методов.

Приобретение навыков React и TypeScript дает разработчику конкурентное преимущество на рынке труда и способствует созданию приложений, которые удовлетворяют потребности пользователей и заказчиков.

[10] Create React App (CRA) – это инструмент командной строки, который упрощает создание, настройку и развертывание проектов, основанных на React.

Глава 2. Установка и настройка окружения разработки

Прежде чем начать разрабатывать приложения с использованием React и TypeScript, вам потребуется настроить ваше рабочее окружение. В этой главе рассмотрим шаги установки и настройки необходимых инструментов и библиотек для разработки веб-приложений.

2.1 Установка Node. js и npm

Node. js – это среда выполнения JavaScript, которая позволяет выполнять JavaScript на стороне сервера. Она также включает в себя пакетный менеджер npm (Node Package Manager), который используется для установки и управления сторонними пакетами и зависимостями проекта. При установке Node. js в комплекте с ним автоматически устанавливается и npm (Node Package Manager).

Шаги для установки Node. js и npm:

– Перейдите на официальный сайт Node. js (https://nodejs.org/) и загрузите установщик для вашей операционной системы (Windows, macOS, Linux).

– Запустите установщик и следуйте инструкциям на экране.

– После завершения установки, откройте терминал (командную строку) и выполните команду node -v, чтобы проверить версию Node. js, и npm -v, чтобы проверить версию npm. Если обе команды вернули версии, значит, установка прошла успешно.

– Добавьте полный путь к исполняемому файлу Node. js в переменную PATH вашей системы.

2.2 Настройка редактора кода

Для комфортной разработки с React и TypeScript, наилучшим выбором будет использование редактора кода, поддерживающего TypeScript. В этой книге мы будем использовать бесплатный редактор кода – Visual Studio Code, и все примеры будут представлены именно в нем.

Установите его, если у вас его еще нет, скачав с официального сайта.

Затем установите следующие его расширения:

– ESLint: Для проверки и форматирования кода.

– Prettier: Для автоматического форматирования кода.

– Reactjs code snippets: Для быстрого создания React-компонентов.

– Auto Complete Tag: Для автозакрытия HTML/XML тегов в процессе набора кода.

– Code Runner: Позволяет запускать код (в том числе, скрипты) прямо из редактора без переключения в терминал.

Для установки расширений откройте Visual Studio Code. Перейдите во вкладку «Extensions» (или нажмите Ctrl+Shift+X). В поисковой строке введите название и найдите соответствующий плагин. Нажмите кнопку «Install» (Установить) рядом с расширением.

После установки, убедитесь, что установленные вами плагины активированы.

2.3 Создание проекта с использованием Create React App

Create React App – это инструмент, который позволяет быстро создать новый проект React с предустановленной конфигурацией и набором инструментов для разработки. Это отличное решение, как для начинающих, так и для опытных разработчиков, позволяя сэкономить время и значительно упростить рабочий процесс.

Удобство использования Create React App:

– Быстрый старт. Разработка нового приложения на React занимает всего несколько минут. CRA создает стандартную структуру проекта и настраивает сборку.

– Готовая конфигурация. CRA предоставляет настройку для Webpack, Babel и других инструментов, сэкономив ваше время на ручной настройке.

– Автоматическое обновление зависимостей. CRA следит за обновлениями зависимостей и предупреждает о несовместимостях. Это позволяет легко поддерживать приложение в актуальном состоянии.

– Простое развертывание. CRA предоставляет средства для удобного развертывания приложения на различных хостинг-платформах.

– Простой интерфейс командной строки: CRA предоставляет команды, такие как npm start для разработки и npm build для создания оптимизированной версии приложения.

Таким образом, Create React App облегчает начало работы с проектами на React и позволяет разработчикам сосредотачиваться на написании кода и создании приложения, не тратя много времени на конфигурацию инструментов.

Для создания проекта с использованием Create React App выполните следующие шаги:

– Откройте терминал в вашем редакторе кода и выполните следующую команду для установки Create React App глобально на вашем компьютере:

npm install -g create-react-app

– Создайте новый проект React, заменяя my-app на имя вашего проекта:

npx create-react-app my-app

Эта команда создаст новую директорию my-app с начальной структурой проекта.

– Перейдите в директорию проекта:

cd my-app

– Запустите разработческий сервер:

npm start

Это запустит сервер разработки и откроет ваше приложение в браузере.

2.4 Установка TypeScript

Теперь, когда у вас есть проект React, созданный с помощью Create React App, вы можете добавить поддержку TypeScript.

Шаги для установки TypeScript в проект:

– Остановите разработческий сервер, если он запущен, нажав Ctrl + C в терминале.

– Выполните следующую команду для установки TypeScript и связанных инструментов:

npm install – save typescript @types/node @types/react @types/react-dom @types/jest

– Переименуйте файлы в вашем проекте с расширением. js в. tsx[11], чтобы использовать синтаксис TypeScript.

Теперь вы можете продолжить разработку вашего приложения с поддержкой TypeScript.

Примечание: Создать приложение на React и Typescript можно сразу, задав команду: npx create-react-app – template typescript my-app

2.5 Заключение

Настройка разработочного окружения для React и TypeScript представляет собой важный первый этап при создании веб-приложений. Этот процесс включает в себя установку Node. js и npm, а также использование инструмента Create React App, который делает разработку более удобной и эффективной. Node. js обеспечивает среду выполнения JavaScript на сервере,

[11] .tsx обозначает файлы, содержащие код на языке TypeScript с использованием синтаксиса JSX.

а npm – управление пакетами, необходимыми для проекта. Кроме того, настройка TypeScript добавляет строгую типизацию, способствуя предсказуемости и надежности кода. Это особенно важно при работе с крупными и сложными проектами.

После завершения этой начальной конфигурации вы будете готовы приступить к созданию компонентов и приложений с использованием React и TypeScript. Эта начальная конфигурация обеспечивает вам устойчивую основу для разработки.

В следующих главах мы подробно рассмотрим техники создания мощных веб-приложений в стеке React и TypeScript.

Глава 3. Понятие компонента в React

После установки и настройки окружения разработки для React и TypeScript, давайте начнем создавать свой первый компонент в React. В этой главе мы рассмотрим базовые шаги по созданию и отображению компонента.

3.1 Понятие компонента в React

В React, компонент – это основная строительная единица пользовательского интерфейса. Он представляет собой независимую и переиспользуемую часть интерфейса, которая может содержать в себе как структуру (HTML-элементы), так и логику.

Компоненты в React могут быть разделены на два типа:

– Функциональные компоненты: Это функции, которые принимают входные данные (props) и возвращают JSX, определяющий структуру компонента.

– Классовые компоненты: Это классы, которые наследуются от React.Component и могут содержать состояние (state) и методы жизненного цикла.

Выбор между классовыми и функциональными компонентами в React зависит от конкретных требований проекта и предпочтений разработчика. Однако, начиная с версии React 16.8, появились хуки (например, useState), которые обеспечивают функциональным компонентам возможности, ранее доступные только классовым компонентам.

Функциональные компоненты с хуками обычно имеют несколько преимуществ:

– Краткость кода. Функциональные компоненты с использованием хуков обычно более компактны и могут быть проще для понимания.

– Читаемость. Хуки обеспечивают локальное состояние и другие возможности функциональным компонентам, что делает код более декларативным и легко читаемым.

– Проще тестирование. Функциональные компоненты обычно более просты в тестировании. Вы можете тестировать их, используя библиотеки тестирования, такие как Jest, без создания экземпляра класса.

Однако, есть сценарии, в которых классовые компоненты могут быть предпочтительными:

– Жизненный цикл компонентов. Если вам нужен доступ к методам жизненного цикла (например, componentDidMount, componentDidUpdate), то классовые компоненты могут быть более подходящим выбором.

– Классовые свойства. В классовых компонентах вы можете использовать классовые свойства для хранения данных без необходимости использования this.setState.

– Интеграция с библиотеками. Некоторые сторонние библиотеки и устаревший код могут предпочитать использование классовых компонентов.

В конечном итоге, выбор между классовыми и функциональными компонентами зависит от вашего стиля кодирования, требований проекта и командных предпочтений. В современных проектах, особенно если вы начинаете новый проект или обновляете старый, функциональные компоненты с хуками обычно являются более современным и предпочтительным вариантом.

3.2 Создание функционального компонента

Давайте создадим простой функциональный компонент в React.

– В вашем проекте React откройте папку src.

– Внутри папки src создайте новый файл с расширением. tsx (например, MyComponent. tsx).

– Откройте созданный файл в вашем редакторе кода.
– Напишите следующий код, чтобы создать простой функциональный компонент:

```
import React from ’react’
function MyComponent () {
  return (
    <div>
      <h1> Привет, это мой первый компонент React! </h1>
    </div>
)
}
export default MyComponent;
```

Давайте разберемся, что произошло:
– Мы импортировали библиотеку React.
– Создали функциональный компонент MyComponent, который возвращает JSX.
– Вернули JSX, который представляет собой div с заголовком. Несмотря на схожесть с HTML, JSX является спецификацией React, применяемой для создания пользовательского интерфейса.
– Использовали export default, чтобы сделать компонент доступным для импорта в других частях вашего приложения.

3.3 Использование компонента в приложении

Теперь, когда у нас есть компонент, давайте научимся его использовать.
Откройте файл src/App. tsx (не забудьте переименовать js в tsx).
Импортируйте ваш компонент в этот файл:

```
import React from ’react’
import MyComponent from». /MyComponent’
function App () {
  return (
    <div>
      <h1> Мое приложение React </h1>
      <MyComponent />
    </div>
);
}
export default App
```

Теперь ваш компонент MyComponent будет отображаться внутри компонента App.
Примечание 1:
Вместо обычной функции допустимо использовать стрелочную.
Примечание 2:
В новой версии React (с версии 17.0.0) в большинстве случаев не требуется явно импортировать React из библиотеки react. Это связано с тем, что в новой версии компилятор Babel встраивает необходимые вызовы React автоматически в JSX без явного импорта. Однако если у вас есть компоненты, в которых используется состояние или классовый подход, то вам всё

так же необходимо импортировать React в файл. Для удобства в этой книге мы всегда будем прописывать строку импорта.

– Прописываем index. js и App. js

«index. js» и «App. js» – это пользовательские компоненты, созданные для React-приложения. Это два важных компонента, которые играют разные роли:

– index. js (или index. tsx). Это точка входа в React-приложение. Этот файл обычно является стартовой точкой, с которой начинается выполнение приложения. Он отвечает за инициализацию React и рендеринг корневого компонента приложения в HTML-элементе на веб-странице. В этом файле используется функция ReactDOM.render для монтирования приложения в DOM.

– App. js (или App. tsx) – это корневой компонент React-приложения. Он представляет собой основу приложения, внутри которого определяется его структура и логика. Обычно компонент «App» содержит маршрутизацию (если это не одностраничное приложение), заголовок, меню и контейнер для других компонентов, формирующих страницу.

– Пример index. js:

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import». /index. css'
import App from». /App'
import reportWebVitals from». /reportWebVitals'

const root = ReactDOM.createRoot(document.getElementById
('root') as HTMLElement)
root.render (
<React.StrictMode>
  <App />
</React.StrictMode>
);
reportWebVitals ()
```

reportWebVitals – это функция, предоставляемая Create React App для измерения производительности приложения. Она отправляет данные о производительности на сервер Google Analytics (по умолчанию), что позволяет отслеживать метрики производительности приложения в реальном времени. Эти метрики могут включать в себя время загрузки страницы, время рендеринга компонентов, использование памяти и другие аспекты производительности. Они могут помочь выявить узкие места в приложении и улучшить его производительность. Эта функция не является обязательной и вы можете упустить её в своем приложении. Таким образом, если вы не планируете использовать reportWebVitals для отслеживания производительности, вы можете безопасно удалить соответствующие импорты и вызовы функции из index. js. Это не повлияет на основной функционал вашего приложения.

<React.StrictMode> – это компонент, предоставляемый React, который помогает выявлять потенциальные проблемы в компонентах и их потомках. Он не влияет на продакшен-сборку[12], но помогает разработчикам рано обнаруживать и исправлять проблемы.

Он может выявить следующие виды проблем:

[12] Продакшен-сборка (production build) – это версия вашего программного продукта, предназначенная для развертывания и использования в реальной эксплуатационной среде.

– Устаревшие методы жизненного цикла. Предупреждает, если ваш компонент использует устаревшие методы жизненного цикла, которые могут быть удалены в будущих версиях React.

– Побочные эффекты при рендеринге. Если код при рендеринге компонента вызывает побочные эффекты (например, изменение состояния, которое влияет на сам компонент), <React.StrictMode> поможет обнаружить такие сценарии.

– Использование несовместимых API. Предупреждает о применении устаревших или несовместимых с версией React API в приложении.

Запись document.getElementById (’root’) as HTMLElement означает, что мы пытаемся получить элемент с идентификатором ’root’ из DOM (Document Object Model) веб-страницы и привести его к типу HTMLElement. Этот код представляет собой так называемый «Type Assertion»[13] в TypeScript, который используется для явного указания типа переменной.

При этом:

– document – это объект, представляющий веб-страницу в браузере.

– getElementById (’root’) – это метод объекта document, который пытается найти элемент на веб-странице с указанным идентификатором ’root’.

– as HTMLElement – это TypeScript-синтаксис, который используется для явного указания типа переменной. В данном случае, мы явно указываем, что результат вызова getElementById (’root’) должен быть интерпретирован как объект типа HTMLElement.

Обычно это используется, когда TypeScript не может определить тип элемента автоматически, и мы хотим явно указать, какой тип ожидается. В данном случае, мы ожидаем, что элемент с идентификатором ’root’ является элементом типа HTMLElement.

Пример «App. js»:

```
import React from «react»
import». /App. css»

function App () {
  return (
    <div className=«App»>
      <header>
        <h1> Мое React-приложение </h1>
      </header>
      <main>
        {/* Здесь может быть контент вашего приложения */}
      </main>
    </div>
);
}

export default App
```

– Заполняем файл tsconfig. json

Файл tsconfig. json – это конфигурационный файл для TypeScript, который используется для настройки параметров компиляции и поведения TypeScript-компилятора[14] (tsc). Этот файл

[13] Type Assertion (также известное как Type Casting) – это способ в языке TypeScript явно указать компилятору, какой тип данных должен быть присвоен переменной или выражению. Это позволяет переопределить или уточнить тип данных, который TypeScript выводит автоматически, когда это необходимо.

[14] TypeScript-компилятор – это инструмент, предоставляемый TypeScript, который преобразует исходный код, написанный

обычно располагается в корневой директории проекта и определяет, как TypeScript должен обрабатывать и компилировать исходный код. В целом, tsconfig. json позволяет настроить проект так, чтобы TypeScript понимал, как правильно компилировать код в JavaScript.

В файле tsconfig. json можно определить ряд важных параметров и настроек для компиляции TypeScript. Вот некоторые из них:

– compilerOptions: Этот раздел определяет параметры компиляции TypeScript. Некоторые распространенные опции включают:

– target: Определяет версию JavaScript, на которую будет транспилирован TypeScript-код (например, «ES5» или «ES6»).

– module: Указывает, как TypeScript должен обрабатывать модули (например, «CommonJS», «ESNext»).

– outDir: Задает директорию, в которую будут скомпилированы выходные файлы.

– strict: Включает или отключает строгую типизацию.

– jsx: Определяет, как TypeScript должен обрабатывать JSX (например, «react» или «preserve»).

– include и exclude: Эти опции определяют, какие файлы TypeScript должны включаться в процесс компиляции (include) и какие файлы исключаться (exclude).

– extends: Позволяет использовать другой конфигурационный файл в качестве базового и переопределить или дополнить его настройки.

– files и include: Определяют, какие файлы исходного кода TypeScript должны быть включены в компиляцию. Файлы перечислены в виде массива строк с путями к файлам.

– exclude: Определяет, какие файлы исходного кода TypeScript следует исключить из компиляции. Это также представлено в виде массива строк с путями к файлам.

– baseUrl и paths: Определяют настройки для алиасов путей к модулям, что может упростить импорт файлов в TypeScript.

При настройке tsconfig. json в своем проекте, убедитесь, что параметры соответствуют вашим требованиям, и что ваш код успешно компилируется и работает в соответствии с ожиданиями. Этот файл является важной частью инфраструктуры TypeScript-проекта и помогает обеспечить более точное и надежное развитие приложения. Правильная настройка tsconfig. json может также значительно улучшить процесс совместной разработки и обеспечить соблюдение стандартов кодирования в вашем проекте. В корне с проектом создайте файл tsconfig. json и напишите в него следующий код:

```
{
  «compilerOptions»: {
    «target»: «es5»,
    «lib»: [
      «dom»,
"dom.iterable»,
      «esnext»
],
    «allowJs»: true,
    «skipLibCheck»: true,
    «esModuleInterop»: true,
    «allowSyntheticDefaultImports»: true,
    «strict»: true,
    «forceConsistentCasingInFileNames»: true,
```

на TypeScript, в эквивалентный JavaScript-код.

```
            «noFallthroughCasesInSwitch»: true,
            «module»: «esnext»,
            «moduleResolution»: «node»,
            «resolveJsonModule»: true,
            «isolatedModules»: true,
            «noEmit»: true,
            «jsx»: «react-jsx»
        },
          «include»: [
            «src»

]
}
```

3.6 Файл README.md

Файл README.md в проекте на TypeScript (или любом другом проекте) обычно служит для предоставления информации о проекте, его использовании и внутренней структуре. Этот файл предназначен для документирования проекта и обеспечения ясности для других разработчиков, которые могут работать с ним. В README.md вы можете включить описание проекта, инструкции по установке и запуску, а также примеры использования и важную информацию о зависимостях[15] и лицензии. Кроме того, README.md может содержать ссылки на документацию, рекомендации по структуре проекта, а также контактные данные для связи с авторами или поддержкой проекта, делая его ценным ресурсом для совместной разработки и использования.

Обычно в файл README.md для проекта на TypeScript включают следующее:

– Заголовок и описание. Начните файл README.md с заголовка, который описывает ваш проект. Затем предоставьте краткое описание проекта, которое объясняет, что ваше приложение делает.

– Установка. Опишите, как установить и настроить ваш проект. Укажите, какие зависимости нужно установить, какой пакетный менеджер[16] использовать (например, npm или yarn) и какие команды выполнить.

– Использование. Предоставьте примеры кода или инструкции о том, как использовать ваше приложение. Объясните, как запустить приложение, какие команды или параметры доступны.

– Примеры. Если ваш проект включает в себя примеры кода, покажите их здесь. Это может быть особенно полезно для других разработчиков, чтобы быстро понять, как использовать вашу библиотеку или приложение.

– Структура проекта[17]. Поясните структуру каталогов и файлов в вашем проекте. Это поможет другим разработчикам быстро ориентироваться в коде.

---

[15] Зависимости (Dependencies) в React и TypeScript проекте представляют собой сторонние библиотеки, модули и ресурсы, которые проект использует для выполнения определенных функций.

[16] Пакетный менеджер (Package Manager) – это инструмент, используемый в разработке программного обеспечения для управления зависимостями и пакетами, необходимыми для проекта. Он позволяет разработчикам легко устанавливать, обновлять, удалить и управлять библиотеками и модулями, которые используются в их приложениях.

[17] Структура проекта (Project Structure) – это организация и распределение файлов, папок и ресурсов внутри программного проекта.

– Лицензия. Укажите информацию о лицензии, в соответствии с которой распространяется ваш проект. Это важно для определения правил использования и распространения вашего кода.

– Ссылки и контакты. Если у вас есть веб-сайт, репозиторий[18] на GitHub или другие ресурсы, связанные с проектом, укажите их здесь. Также предоставьте контактную информацию для обратной связи или вопросов.

Пример простейшего файла README.md для проекта на TypeScript:

 # Проект на TypeScript

Этот проект представляет собой пример README.md файла для проекта на TypeScript.

Установка

1. Установите зависимости с помощью npm:

```bash

npm install

## 3.7 Запуск приложения

Убедитесь, что ваш разработческий сервер остановлен (если запущен), иначе введите Ctrl + C в терминале.

Запустите разработческий сервер снова с помощью команды:

npm start

Откройте браузер и перейдите по адресу http://localhost:3000. Вы должны увидеть ваш компонент MyComponent отображенным внутри приложения.

Если в коде будут обнаружены ошибки – они отобразятся на странице с указанием на саму ошибку и на строку, где она расположена. Этот шаг важен для проверки функциональности вашего приложения в реальной среде разработки. Обратите также внимание на сообщения в консоли браузера и на подчеркнутые красным строки кода в редакторе, чтобы оперативно выявить и исправить возможные проблемы.

## 3.8 Заключение

Создание компонентов является фундаментальной частью разработки приложений на React. Компоненты позволяют разбивать интерфейс на маленькие и переиспользуемые части, упрощая код и улучшая структуру проекта.

Выбор между функциональными и классовыми компонентами зависит от ваших потребностей и предпочтений. В современном React функциональные компоненты и хуки стали более популярными и часто являются предпочтительным выбором.

Создание компонентов в React – ключевой этап при разработке приложений, и глубокое понимание этой концепции поможет вам создавать более удобные, модульные и легко обслуживаемые приложения.

В следующей главе мы погрузимся в изучение JSX и его синтаксиса, что позволит нам создавать более выразительные и мощные пользовательские интерфейсы.

[18] Репозиторий (Repository) – это хранилище данных, которое используется для сохранения, управления и отслеживания версий файлов и компонентов в рамках проекта разработки программного обеспечения.
```

Глава 4. JSX и его синтаксис

JSX (JavaScript XML) представляет собой особый синтаксис в JavaScript, который применяется в React для создания структуры пользовательского интерфейса. Вместо того чтобы писать код на чистом JavaScript для создания элементов, можно использовать JSX для более удобной и декларативной разметки компонентов.

В данной главе мы исследуем мир JSX и более детально рассмотрим его синтаксис. Это поможет вам освоить процесс создания компонентов React и научиться выражать их структуру. Понимание JSX является ключевым навыком для разработчиков React, позволяя создавать сложные пользовательские интерфейсы.

4.1 Понятие JSX

JSX – это синтаксическое расширение JavaScript, которое позволяет писать HTML-подобный код внутри JavaScript. Этот код компилируется в обычный JavaScript, который React может понимать и использовать для создания элементов пользовательского интерфейса. Когда мы пишем код в JSX, он будет преобразован в обычный JavaScript, который React может понимать и использовать для построения интерфейсных элементов. Проще говоря, JSX – это инструмент, позволяющий нам описывать компоненты более наглядно и эффективно, чем просто с помощью JavaScript.

Пример JSX:

```
const element = <h1> Hello, World! </h1>;
```

В этом примере <h1> Hello, World! </h1> – это JSX элемент, который представляет собой заголовок первого уровня.

4.2 Основы синтаксиса JSX

В этом разделе мы изучим основы синтаксиса JSX, который используется в React для создания интерфейсов. JSX напоминает HTML, но имеет некоторые исключения и интересные особенности:

– Теги. В JSX можно создавать элементы с использованием тегов, как в HTML. Например, для создания блока текста используется тег <div>, для заголовка – <h1>, а для изображения – <img>. Однако есть некоторые исключения, такие как использование <className> вместо <class> для определения классов элементов.

– Значения атрибутов. Вы также можете добавлять атрибуты к JSX элементам, как в HTML. Например, для указания источника изображения и его альтернативного текста используются атрибуты src и alt.

– Вложенные элементы. JSX позволяет создавать вложенные элементы, аналогично HTML. Это значит, что вы можете размещать одни элементы внутри других:

```
<div>
<h1> Заголовок </h1>
<p> Параграф текста </p>
</div>
```

– Интерполяция. Одной из мощных особенностей JSX является возможность вставки значений JavaScript внутрь элементов с использованием фигурных скобок {}. Это называется интерполяцией. Например, вы можете вставить значение переменной name в текст:

```
const name = «John'
const greeting = <p> Привет, {name}! </p>;
```

Некоторые исключения в синтаксисе JSX включают замену class на className, как уже упоминалось ранее, и также замену for на htmlFor при работе с атрибутами class и for. Эти изменения внесены для избежания конфликтов с ключевыми словами JavaScript и HTML.

4.3 Рендеринг JSX

Для отображения JSX на веб-странице, необходимо использовать React – библиотеку, которая предоставляет компоненты, создаваемые с использованием JSX, и затем рендерит их на веб-странице с помощью специализированных функций. Этот подход позволяет создавать динамичные и масштабируемые веб-приложения, которые реагируют на пользовательские действия и изменения данных.

Пример рендеринга JSX в React:

```
import React from 'react'
import ReactDOM from 'react-dom'
const element = <h1> Hello, React! </h1>
ReactDOM.render (element, document.getElementById ('root'))
<h1> Hello, React! </h1>
ReactDOM.render ()
```

Пояснения:

– JSX элемент <h1> Hello, React! </h1> сохраняется в переменной element.

– ReactDOM – это библиотека, предоставляемая React, которая используется для взаимодействия с DOM (Document Object Model) в веб-приложениях. Она позволяет React-компонентам отображаться и обновляться в браузере путем управления виртуальным DOM и его согласованием с реальным DOM.

– ReactDOM.render () используется для отображения element в элементе с идентификатором root на веб-странице.

Основные функции ReactDOM включают:

– ReactDOM.render (element, container): Эта функция используется для отображения React-элемента или компонента в заданном контейнере (обычно в div или другом HTML-элементе). Она инициализирует процесс создания виртуального DOM и его синхронизацию с реальным DOM.

– ReactDOM. hydrate (element, container): Эта функция аналогична ReactDOM.render, но предназначена для гидратации (hydration) существующего серверного рендеринга[19]. Она используется, когда React приложение запускается на стороне клиента и должно восстановить состояние, созданное на сервере.

---

[19] Гидратация существующего серверного рендеринга означает, что когда веб-страница создается на сервере и отправляется на ваш компьютер, она уже содержит часть информации о том, как должен выглядеть интерфейс.

– ReactDOM. unmountComponentAtNode (container): Эта функция используется для отключения (размонтирования) React-компонента, который был ранее отображен в заданном контейнере.

– Другие функции, такие как ReactDOM.createPortal, которые позволяют встраивать React-компоненты вне их обычной иерархии DOM.

Примечание: Виртуальный DOM (Document Object Model) и реальный DOM представляют собой две разные концепции, связанные с манипуляциями интерфейсом в веб-разработке.

Реальный DOM:

– Реальный DOM – это фактическое представление структуры веб-страницы, доступное в браузере. Он представляет дерево объектов, где каждый узел представляет собой часть страницы.

– Изменения в реальном DOM могут быть медленными и затратными. Когда происходит изменение, например, добавление или удаление элемента, браузер вынужден пересчитывать весь макет и перерисовывать страницу.

Виртуальный DOM:

– Виртуальный DOM – это абстрактное представление структуры веб-страницы, существующее в памяти программы (обычно на языке JavaScript). Он является копией реального DOM.

– Манипуляции с виртуальным DOM происходят быстрее, так как это операции в памяти программы. Вместо того чтобы изменять реальный DOM сразу, изменения происходят в виртуальном DOM.

– После внесения изменений в виртуальный DOM сравнивается с реальным DOM, и только изменения применяются к фактической структуре. Это позволяет сделать процесс обновления более эффективным, так как браузеру не нужно пересчитывать и перерисовывать всю страницу, а только те части, которые изменились.

Таким образом, виртуальный DOM служит промежуточным этапом для оптимизации процесса манипуляций интерфейсом в React и других библиотеках, сокращая нагрузку на реальный DOM.

4.4 Преимущества JSX

Использование JSX в React обладает рядом преимуществ:

– Читаемость кода. JSX делает код более читаемым и похожим на HTML, что упрощает понимание структуры интерфейса. Разработчики могут легко определить иерархию элементов, что сделает поддержку и рефакторинг кода более удобными.

– Интерполяция данных. Вы можете вставлять переменные и выражения JavaScript в JSX с использованием фигурных скобок {}, что делает динамическое создание интерфейса простым. Это позволяет вам связывать компоненты с данными и динамически обновлять контент на веб-странице.

– Компоненты. JSX позволяет создавать компоненты, которые можно переиспользовать в разных частях приложения. Это способствует модульности и повторному использованию кода, что сокращает дублирование и упрощает обслуживание приложения.

Использование JSX в React упрощает разработку интерфейсов, делая код более выразительным и функциональным.

4.5 Заключение

JSX – это мощное средство для описания пользовательского интерфейса в React, позволяющее разработчикам создавать динамические, интерактивные и легко сопровождаемые веб-приложения. Оно объединяет в себе удобство декларативного описания интерфейса с мощью JavaScript, что делает его популярным выбором для программистов в области веб-разработки.

В следующих главах мы продолжим рассматривать JSX и изучим его более глубокие возможности, такие как передача пропсов (свойств) между компонентами, обработка событий, управление состоянием и многое другое.

Глава 5. Работа с компонентами и их вложенность

В этой главе рассмотрим более детально компоненты React и изучим, как создавать, использовать и вкладывать их друг в друга. Компоненты являются основными строительными блоками React-приложений и играют важную роль в организации кода и создании модульных интерфейсов.

5.1 Создание компонента

Создание компонента в React – это простой процесс. Существуют функциональный или классовый компоненты.

– Функциональный компонент основан на функциях JavaScript и представляет из себя функцию, которая принимает входные данные (props) и возвращает JSX элемент, который будет отображаться на экране. Функциональные компоненты стали более популярными с появлением React Hooks[20] и обладают следующими ключевыми характеристиками:

– Простота и чистота. Функциональные компоненты обычно более краткие и легче читаемы, чем классовые компоненты. Они представляют из себя обычные JavaScript-функции, что делает код более понятным.

– Использование Hooks. С появлением React Hooks (таких как: useState, useEffect и многие другие) функциональные компоненты могут управлять состоянием и жизненным циклом компонента так же, как классовые компоненты. Это позволяет функциональным компонентам выполнять разнообразные задачи, например, такие как: управление состоянием, выполнение побочных эффектов и многое другое.

– Отсутствие this. В функциональных компонентах отсутствует ключевое слово this, что устраняет проблемы с областью видимости и контекстом, связанными с классовыми компонентами.

– Большая производительность. Функциональные компоненты могут иметь более высокую производительность из-за отсутствия накладных расходов, связанных с созданием экземпляров классов.

– Легко тестируемы. Функциональные компоненты часто более легко поддаются модульному тестированию, так как их поведение зависит только от входных данных (props) и их вывода (возвращаемого JSX).

Пример функционального компонента:

```
import React from 'react'
function MyComponent () {
  return <h1> Привет, это мой первый компонент! </h1>
}
export default MyComponent
```

Функциональные компоненты являются предпочтительным способом создания компонентов в React.

– Классовый компонент определен как класс JavaScript и расширяет базовый класс React.Component. Классовые компоненты имеют следующие ключевые характеристики:

[20] React Hooks – это функции, предоставляемые библиотекой React, которые позволяют добавлять состояние и другие возможности React в функциональные компоненты.

– Состояние (State). Классовые компоненты имеют доступ к локальному состоянию, которое может быть использовано для хранения и управления данными, изменяющимися во времени. Состояние компонента может быть изменено с помощью метода setState, и изменения состояния приводят к перерисовке компонента.

– Жизненный цикл (Lifecycle). Классовые компоненты поддерживают разнообразные методы жизненного цикла, такие как componentDidMount, componentDidUpdate, и componentWillUnmount. Жизненный цикл компонента – это специальные методы, которые вызываются в разные моменты времени в жизни компонента. Например, componentDidMount вызывается после того, как компонент был добавлен к DOM. Эти методы позволяют выполнять действия при монтировании, обновлении и размонтировании компонента.

– Пропсы (Props). Классовые компоненты могут принимать входные данные, передаваемые через свойство props. Пропсы представляют собой данные, которые компонент может использовать для настройки своего поведения или отображения.

– Контекст (Context). Классовые компоненты могут использовать механизм контекста React для передачи данных глубоко в иерархии компонентов без необходимости передавать пропсы через каждый уровень.

Пример классового компонента:

```
import React, {Component} from ’react’
class MyComponent extends Component {
render () {
return <h1> Привет, это мой компонент! </h1>
}
}
export default MyComponent
```

5.2 Использование компонентов

Чтобы использовать компонент в другом компоненте или приложении, вы должны его импортировать и затем использовать как тег JSX.

Пример использования функционального компонента:

```
import React from ’react’
import MyComponent from». /MyComponent’
function App () {
  return (
    <div>
      <h1> Мое приложение React </h1>
      <MyComponent />
    </div>
);
}
export default App
```

5.3 Вложенность компонентов

Одним из сильных сторон React является возможность вложения компонентов друг в друга для создания более сложных интерфейсов путем объединения более простых компо-

нентов в более крупные структуры. Вложенность компонентов помогает в организации кода, повторном использовании и поддерживаемости приложения. Вот некоторые важные аспекты вложенности компонентов в React:

– Иерархия компонентов. В React компоненты могут быть вложены друг в друга, создавая иерархию. Эта иерархия может быть глубокой и сложной, и она определяет, как компоненты взаимодействуют друг с другом.

– Пропсы (Props) и состояние (State). Вложенные компоненты могут обмениваться данными через пропсы (входные данные) и состояние (локальные данные). Родительский компонент может передать данные дочернему компоненту через пропсы, что позволяет дочернему компоненту отображать эти данные. Дочерний компонент может также передавать информацию обратно в родительский компонент с помощью обратных вызовов.

– Поддержка композиции. Вложенность компонентов позволяет создавать компоненты, которые могут быть переиспользованы в разных частях приложения. Можно создавать маленькие компоненты, которые решают конкретные задачи, и затем объединять их в более крупные компоненты.

– Разделение ответственности. Разделение функциональности между компонентами позволяет каждому компоненту сосредотачиваться на выполнении конкретных задач. Это улучшает читаемость и обеспечивает модульность кода.

– Управление структурой и стилями. Вложенные компоненты также могут использоваться для создания сложных макетов и структур интерфейса. Каждый компонент может быть стилизован и настроен независимо, что способствует упрощению управления стилями.

Пример вложенности компонентов:

```
import React from 'react'
function Header () {
  return <h1> Заголовок </h1>
}
function Sidebar () {
  return (
    <div>
      <h2> Боковая панель </h2>
    </div>
)
}
function App () {
  return (
    <div>
      <Header />
      <Sidebar />
      <p> Основное содержание </p>
    </div>
)
}
export default App
```

В этом примере Header и Sidebar являются дочерними компонентами, вложенными в компонент App. Это позволяет легко организовывать структуру интерфейса и создавать компоненты, которые можно переиспользовать.

5.4 Компонент высшего порядка

Компонент высшего порядка (Higher Order Component, HOC) в React – это функция, которая принимает компонент и возвращает новый компонент с дополнительной функциональностью.

Допустим, у вас есть компонент, но вам нужно добавить ему какие-то общие функции или свойства, которые могли бы понадобиться в разных частях приложения. Вместо того чтобы копировать и вставлять один и тот же код в разные места, вы можете использовать HOC. Эта функция «обернет» компонент, добавив к нему нужные функции или свойства.

Другими словами, HOC позволяет повторно использовать код и легко расширять функциональность компонентов.

Например, если у вас есть компонент для отображения информации о пользователе, а вы хотите добавить к нему функцию загрузки данных из сети, HOC может помочь сделать это без необходимости изменения самого компонента.

Пример: у вас есть компонент UserComponent, который отображает имя пользователя:

```
import React from 'react'

const UserComponent = (props) => {
  return <div> Привет, {props.name}! </div>;
}

export default UserComponent
```

Теперь вы хотите добавить к этому компоненту возможность загрузки данных о пользователе из сети. Для этого мы можем использовать HOC:

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

const withDataFetching = (WrappedComponent) => {
  return class extends Component {
    constructor (props) {
      super (props);
this.state = {
        data: null,
        loading: true,
}
}

    async componentDidMount () {
      try {
        const response = await fetch(this.props. url)
        const data = await response. json ()
this.setState ({data, loading: false})
} catch (error) {
console.error («Ошибка:», error)
this.setState ({loading: false})
}
```

```
}

  render () {
    return (
      <WrappedComponent
{…this.props}
data={this.state. data}
loading={this.state. loading}
      />
)
}
}
}

export default withDataFetching
```

Здесь мы должны создать HOC withDataFetching, который загружает данные из указанного URL и передает их в обернутый компонент. После этого мы можем использовать его с нашим UserComponent:

```
import React from 'react'
import withDataFetching from». /withDataFetching'

const UserComponent = (props) => {
  return (
    <div>
      {props. loading? (
        «Загрузка…»
): (
        <div>
          Привет, {props.data.name}!
          <p> Email: {props. data. email} </p>
        </div>
)}
    </div>
);
}

export default withDataFetching (UserComponent)
```

Теперь UserComponent получает данные из сети благодаря HOC withDataFetching

5.5 Заключение

В этой главе мы изучили, как создавать компоненты в React и использовать их в приложениях. Мы освоили, как создавать функциональные компоненты и классовые компоненты, а также как использовать их в составе наших приложений.

Одним из наиболее важных аспектов, который мы рассмотрели, является вложенность компонентов. Мы поняли, как важно правильно структурировать компоненты в иерархии,

чтобы сделать наш код чище и более организованным. Вложенность позволяет нам создавать маленькие, переиспользуемые компоненты, которые могут быть объединены в более крупные структуры, создавая мощные и гибкие интерфейсы.

Эта глава была началом нашего путешествия в мире React, и впереди нас ждут еще много интересных тем и возможностей. Мы будем изучать более сложные концепции, такие как управление состоянием, маршрутизация, обработка событий и многое другое. Мы также рассмотрим передовые техники разработки с использованием React и узнаем, как создавать масштабируемые и производительные приложения.

Глава 6. Основы использования props и state

В React props и state – это два основных механизма для управления данными, обновления пользовательского интерфейса и взаимодействия с компонентами. В этой главе мы рассмотрим, как использовать props и state для передачи и управления данными в компонентах React.

6.1 Введение в props и state

Props (сокращение от «properties») представляют собой механизм передачи данных в компоненты React. Они представляют собой входные данные, получаемые компонентом от его родительского компонента. Props передаются компоненту в виде атрибутов и становятся доступными внутри компонента, как свойства.

Важно отметить, что props являются неизменяемыми, что означает, что однажды переданные данные нельзя изменить внутри компонента. Это позволяет передавать инструкции или информацию от родительского компонента к его дочерним компонентам, определяя, как компонент должен себя вести или какие данные отобразить.

Пример: Если у вас есть компонент «Кнопка», то пропс может содержать информацию о том, какой текст должен отображаться на кнопке. Вы передаете этот текст в компонент, и компонент использует его для отображения на кнопке.

Преимущества использования props:

– Расширяемость. Props являются расширяемым и удобным способом передачи данных в компоненты, особенно, когда у вас есть множество разных пропсов. При использовании props вы также можете определить значения по умолчанию и лучше структурировать ваш компонент.

Пример:

```
function UserProfile (props) {
  return (
    <div>
      <p> Name: {props.name} </p>
      <p> Age: {props.age} </p>
      {/* Другие свойства */}
    </div>


)
}
```

– Читаемость кода. Передача данных через props делает компонент более читаемым, так как вы видите, какие данные он использует, необходимо только взглянуть на его декларацию.

– Значения по умолчанию: Вы можете задать значения по умолчанию для props, что полезно, если некоторые данные не передаются. Например, если props.age не был передан, вы можете использовать значение по умолчанию.

Пример:

```
function UserProfile (props) {
  const age = props.age || 25 // Значение по умолчанию
  return (
    <div>
      <p> Name: {props.name} </p>
```

```
      <p> Age: {age} </p>
     </div>

)
}
```

Хотя слово «props» является стандартным и широко используется в сообществе React, вы можете использовать любое другое слово в качестве аргумента функции компонента. Это полностью зависит от вас и вашего стиля кодирования.

Например, вместо props можно использовать data, parameters, options или любое другое имя, которое вам кажется более подходящим для вашей ситуации. Важно помнить, что выбранное вами имя аргумента будет использоваться для доступа к пропсам внутри компонента, поэтому оно должно быть понятным и соответствовать содержанию.

Пример с использованием другого имени для аргумента:

```
function PersonInfo (data) {
  return (
   <div>
    <p> Имя: {data.name} </p>
    <p> Возраст: {data.age} </p>
   </div>

)
}
```

При использовании компонента:

```
<PersonInfo name=«Алиса» age= {30} />
```

Стоит помнить, что использование слова «props» для аргумента функции компонента – это стандартная практика в сообществе React, и она делает ваш код более читаемым и понятным для других разработчиков, знакомых с React. Большинство разработчиков ожидают видеть props как стандартное имя для пропсов, и это может упростить совместную работу над проектами.

Конечно, если у вас есть особые обстоятельства или предпочтения, и вы хотите использовать другое имя, вы можете сделать это. Однако это может усложнить совместную работу и понимание вашего кода другими членами команды.

State – это специальный объект внутри компонента, который используется для хранения и отслеживания изменяющихся данных. Стейт позволяет компоненту «запоминать» данные и перерисовывать себя, когда эти данные меняются.

Пример: Если у вас есть компонент «Счетчик», то стейт может содержать текущее значение счетчика. Когда пользователь нажимает на кнопку увеличения счетчика, стейт изменяется, и компонент обновляется, чтобы отобразить новое значение счетчика.

```
import React, {useState} from 'react'
function Counter () {
 const [count, setCount] = useState (0)

 const increment = () => {
  setCount (count +1)
```

```
}

  return (
    <div>
      <p> Текущее значение: {count} </p>
      <button onClick= {increment}> Увеличить </button>
    </div>
)
}
function App () {
  return <Counter />
}

export default App
```

count в данном случае представляет начальное состояние (значение), которое мы указываем в useState, и setCount – это функция, которую мы используем для обновления этого состояния. В нашем случае, increment вызывает setCount, чтобы увеличить значение count.

Итак, обощим:

Пропсы (Props) – это данные, которые вы передаете в компонент извне, как параметры.

Стейт (State) – это данные, которые компонент «запоминает» и использует для отслеживания изменений и перерисовки себя.

6.2 Использование пропсов

Пропсы используются для передачи информации, такой как текст, числа, функции или объекты, от одного компонента к другому. Пропсы делают компоненты в React многоразовыми и настраиваемыми.

Вот основные аспекты работы с пропсами в React:

– Передача данных. Пропсы позволяют родительскому компоненту передавать данные в дочерний компонент. Эти данные могут быть переданы в виде атрибутов (параметров) при использовании компонента в JSX.

– Нестрогое чтение (доступ). Дочерний компонент может получить доступ к пропсам, используя props (для функциональных компонентов) или this.props (для классовых компонентов).

– Пропсы только для чтения. Пропсы являются только для чтения, что означает, что дочерний компонент не может изменять значения пропсов. Они предназначены только для отображения данных.

– Использование по умолчанию. Вы можете предоставить значения по умолчанию для пропсов, которые будут использоваться, если соответствующие пропсы не переданы из родительского компонента.

– Проверка типов (Type Checking). React позволяет проводить проверку типов пропсов с помощью PropTypes (для функциональных компонентов) или propTypes (для классовых компонентов). Это помогает предотвратить ошибки типов во время выполнения.

Пример использования:

```
import React from 'react'
function Welcome (props) {
```

```
  return <h1> Привет, {props.name}! </h1>
}
function App () {
  return <Welcome name=«John» />
}
export default App
```

В этом примере компонент App передает name=«John» в дочерний компонент Welcome через props.

Использование пропсов позволяет создавать компоненты, которые могут быть легко настраиваемыми и переиспользуемыми в различных контекстах

– Использование state

Стейт (state) в React представляет собой объект, который содержит данные, влияющие на отображение компонента, и используется для хранения информации, которая может изменяться со временем и должна быть учтена при перерисовке компонента.

Важные аспекты работы со стейтом в React:

– Локальный для компонента. Стейт обычно является локальным для компонента, что означает, что каждый компонент имеет свой собственный стейт. Это помогает изолировать данные и логику между компонентами.

– Инициализация. Стейт может быть инициализирован в конструкторе компонента при использовании классовых компонентов или с использованием хука useState в функциональных компонентах.

– Изменение стейта. Стейт можно изменять с помощью метода setState (для классовых компонентов) или функции, возвращаемой хуком useState (для функциональных компонентов). При изменении стейта React автоматически перерисовывает компонент.

– Асинхронность. Вы должны быть осторожны при изменении стейта, так как операции по его изменению могут быть асинхронными. React может объединять несколько обновлений стейта для оптимизации производительности.

– Неизменяемость (Immutability). Рекомендуется не изменять стейт напрямую, а создавать новый объект стейта с обновленными данными. Это помогает предотвратить мутацию стейта и упростить отслеживание изменений.

– Прокидывание стейта. Стейт может быть передан дочерним компонентам через пропсы, что позволяет им отображать и использовать данные из него.

Таким образом, state используется для хранения данных, которые могут изменяться и влиять на отображение компонента. Вы можете инициализировать стейт в конструкторе компонента и изменять его с помощью метода setState ().

Пример использования:

```
import React, {Component} from 'react'
class Counter extends Component {
  constructor (props) {
    super (props)
this.state = {count: 0}
}
  incrementCount = () => {
this.setState ({count: this.state.count +1})
}
  render () {
```

```
      return (
        <div>
          <p> Счетчик: {this.state.count} </p>
          <button onClick={this.incrementCount}> Увеличить </button>
        </div>
  )
  }
  }
  function App () {
    return <Counter />
  }
  export default App
```

Рассмотрим код подробней.

Конструктор – это особая функция в React-компонентах, которая выполняется при создании нового экземпляра компонента. Она принимает props в качестве параметра и вызывает базовый конструктор через super (props), чтобы унаследовать функциональность родительского компонента. В конструкторе обычно инициализируют начальное состояние компонента и выполняют другие подготовительные операции.

Однако, в современных версиях React большинство разработчиков предпочитают использовать более современный синтаксис с хуками (useState, useEffect, и т.д.), который обеспечивает более читаемый и функциональный способ управления состоянием и эффектами. Хуки стали стандартом в React и рекомендуются для большинства проектов.

Тем не менее, знание о конструкторе важно, так как в некоторых случаях (например, в классовых компонентах) он может использоваться для настройки компонента или выполнения других специфических операций.

– constructor (props): Мы объявляем конструктор для нашего компонента и передаем ему props (свойства), которые компонент получит от родительского компонента. Таким образом, props содержат информацию о данных, которые компонент может использовать.

– super (props): Эта строка вызывает конструктор родительского класса (класса React.Component). Она необходима, чтобы наш компонент мог правильно наследовать функциональность React и корректно обрабатывать переданные свойства (props).

– this.state = {count: 0};: Здесь мы инициализируем локальное состояние компонента. В данном случае, мы создаем переменную count и устанавливаем ее значение на 0. Состояние – это способ для компонента React хранить и отслеживать изменения данных, которые могут влиять на отображение на веб-странице.

Итак, в этом коде мы подготавливаем наш компонент к работе, передавая ему свойства и инициализируя начальное состояние. Это важные шаги, которые делают наш компонент готовым к использованию и реагированию на изменения данных.

В этом примере компонент Counter (счетчик) имеет внутренний state, который используется для отслеживания количества нажатий на кнопку «Увеличить». Метод setState () обновляет состояние компонента, что влечет за собой перерендеринг.

– Ипользование пропсов и стейтов

Пропсы (props) позволяют передавать информацию от одного компонента к другому, что делает взаимодействие между компонентами в React гибким и модульным. С использованием состояния (state), вы можете управлять внутренними данными компонента, такими как значения полей ввода или флаги активации, что позволяет компоненту реагировать на пользовательские действия и динамически изменять свое отображение в соответствии с этими данными.

Важно помнить, что props – это данные, которые передаются сверху вниз и не изменяются внутри компонента, в то время как state – это изменяемые данные, управляемые самим компонентом.

Используйте пропсы (props), чтобы передавать данные от родительского компонента к дочернему компоненту.

Используйте стейт (state), чтобы управлять внутренним состоянием компонента, которое может изменяться и влиять на его отображение.

Помните, что props – это неизменяемые, а state – изменяемые данные.

6.5 Заключение

Использование props и state в React является фундаментом для создания динамичных и интерактивных пользовательских интерфейсов. Пропсы позволяют компонентам обмениваться данными и структурировать приложение в виде множества многоразовых и переиспользуемых компонентов. Операция о своевременном обновлении состояния компонента через state является ключевой для создания отзывчивых приложений, которые реагируют на действия пользователя. Умение совмещать работу с props и state открывает широкие возможности для разработки сложных приложений.

В следующих главах, мы более подробно рассмотрим эти концепции и узнаем, как они взаимодействуют между собой для создания мощных React-приложений.

Глава 7. Методы жизненного цикла компонентов

Методы жизненного цикла компонентов в React предоставляют возможность управлять различными аспектами поведения компонента на разных этапах его существования. В этой главе мы рассмотрим основные методы жизненного цикла компонентов и как их использовать.

7.1 Введение в методы жизненного цикла

Методы жизненного цикла – это специальные функции, предоставляемые React, которые вызываются автоматически на различных этапах развития компонента. Эти методы позволяют встраивать логику и выполнить определенные действия в разные моменты времени, когда компонент создается, обновляется или удаляется. Таким образом, методы жизненного цикла компонента предоставляют разработчикам управление процессами и позволяют выполнять различные задачи в зависимости от текущего состояния компонента. Важно знать, когда и как использовать каждый из этих методов для эффективной работы с React-компонентами.

7.2 Основные методы жизненного цикла

В React компоненты проходят через различные этапы своего жизненного цикла, которые тесно связаны с процессами монтирования (создания и добавления в DOM) и размонтирования (удаления из DOM). Рассмотрим эти этапы подробнее:

Монтирование (Mounting):

– constructor (): Вызывается при создании объекта компонента. Здесь происходит инициализация состояния и привязка методов.

– static getDerivedStateFromProps (): Метод, вызываемый перед render, позволяющий компоненту обновить своё внутреннее состояние на основе изменений в свойствах.

– render (): Отвечает за отображение компонента, возвращая элементы для отображения в интерфейсе.

– componentDidMount (): Вызывается сразу после добавления компонента в DOM. Подходит для выполнения действий, которые требуют наличия компонента в DOM, например, запросов к серверу.

Размонтирование (Unmounting):

– componentWillUnmount (): Вызывается перед удалением компонента из DOM. Здесь происходит очистка ресурсов, таких как отмена запросов или удаление подписок.

Эти этапы жизненного цикла предоставляют точки вставки для кода, который должен выполняться при создании и удалении компонента. Дополнительно, React предоставляет другие важные методы жизненного цикла, такие как componentDidUpdate, который вызывается после обновления компонента и предоставляет возможность реагировать на изменения в props или state.

7.3 Пример использования методов жизненного цикла

Рассмотрим пример использования методов жизненного цикла:

```
import React, {Component} from 'react'
class Timer extends Component {
  constructor (props) {
    super (props)
```

```
this.state = {seconds: 0}
}
  componentDidMount () {
this.intervalId = setInterval (() => {
this.setState ({seconds: this.state.seconds +1})
}, 1000)
}
  componentWillUnmount () {
clearInterval(this.intervalId)
}
  render () {
    return <p> Секунды: {this.state.seconds} </p>
}
}
function App () {
  return <Timer />
}
export default App
```

В этом примере:

– В методе constructor инициализируется начальное состояние компонента.

– В методе componentDidMount устанавливается интервал, который каждую секунду увеличивает значение seconds в состоянии.

– В методе componentWillUnmount интервал очищается перед удалением компонента из DOM, чтобы избежать утечек памяти.

7.4 Заключение

Методы жизненного цикла компонентов React позволяют управлять поведением компонента на разных этапах его жизни. Методы монтирования, обновления и размонтирования предоставляют ключевые точки внедрения для инициализации, реагирования на изменения и освобождения ресурсов.

Методы жизненного цикла служат не только инструментами технического управления компонентами, но и предоставляют возможности для оптимизации производительности, а также для реализации сложных логик и сценариев. Их грамотное использование способствует созданию более эффективных, надежных и легко поддерживаемых React-приложений.

В следующих главах мы рассмотрим конкретные практические сценарии использования этих методов, а также обсудим лучшие практики, которые помогут вам максимально извлечь пользу из возможностей, предоставляемых жизненным циклом React-компонентов.

Глава 8. Автоматическое создание объектов props в React

Автоматическое создание объектов props – это инструмент, который упрощает передачу данных между компонентами. Давайте рассмотрим, как этот подход может ускорить разработку и сделать ваш код более чистым и читаемым.

8.1 Введение в объект props в React

В React, props (сокращение от «properties» или «свойства») – это специальный объект, который используется для передачи данных и настроек от родительского компонента к дочернему компоненту. Он представляет собой набор свойств, которые доступны дочернему компоненту для использования. Эти свойства передаются в компонент в виде атрибутов JSX при его использовании.

Рассмотрим простой пример. У нас есть родительский компонент ParentComponent и дочерний компонент ChildComponent. Мы хотим передать строковое свойство message из родительского компонента в дочерний компонент для отображения:

```
// Родительский компонент
function ParentComponent () {
   return <ChildComponent message=«Привет, мир!» />
}

  // Дочерний компонент
  function ChildComponent (props) {
    return <div>{props.message} </div>

}
```

В приведенном примере, свойство message передается из родительского компонента ParentComponent в дочерний компонент ChildComponent следующим образом:

– В родительском компоненте, при использовании компонента ChildComponent, мы добавляем атрибут message и устанавливаем его значение в «Привет, мир!».

– Внутри дочернего компонента ChildComponent, это свойство становится доступным через объект props. мы можем получить доступ к нему, обратившись к props.message и использовать его для отображения внутри компонента.

Таким образом, через объект props в React можно сделать доступным любое свойство, которое вы определите и передадите из родительского компонента в дочерний компонент. Объект props предоставляет интерфейс для передачи данных между компонентами и позволяет динамически настраивать компоненты.

Через props вы можете передавать не только данные (такие как строки, числа, объекты), но и функции, обработчики событий и другие настройки компонента.

Что касается области видимости, то React обеспечивает уровень доступа к свойствам компонента через объект props, который может быть рассмотрен как «публичный интерфейс» компонента. Дочерний компонент не имеет доступа к свойствам родительского компонента напрямую, за исключением тех свойств, которые были явно переданы через props. Это обеспечивает инкапсуляцию[21] и изоляцию компонентов и упрощает их переиспользование.

---

[21] Инкапсуляция – это упаковка данных и методов, работающих с этими данными, в единый объект (класс или модуль),

8.2 Роль объекта props в компонентах

В React компоненты организованы в древовидную структуру, где один компонент может быть родительским по отношению к другому. В React каждый компонент «знает» о своих дочерних компонентах, но дочерние компоненты не «знают» о своих родительских компонентах напрямую. Вместо этого родительские компоненты могут передавать данные и свойства своим дочерним компонентам через props.

Для того, чтобы определить, является ли текущий компонент родительским, можно воспользоваться следующими методами:

– Иерархия компонентов. Родительский компонент – это тот, который находится на более высоком уровне иерархии компонентов, и от которого исходят данные или свойства для одного или нескольких дочерних компонентов. Дочерний компонент, наоборот, находится внутри родительского компонента.

– Анализ кода. Если вы изучаете код приложения, родительский компонент обычно тот, который решает, какие данные передавать дочерним компонентам через props и какие обработчики событий передавать для взаимодействия с дочерними компонентами.

– Свойство children. Родительский компонент может использовать свойство children, чтобы передать дочерним компонентам элементы JSX внутри компонента. Если вы видите, что в родительском компоненте есть использование props.children, это может быть признаком того, что компонент решает, какие компоненты или элементы JSX вставлять внутри себя.

Таким образом, объект props в React позволяет передавать данные и настройки между компонентами и делает компоненты более гибкими и переиспользуемыми.

8.3 Создание объекта props

React автоматически создает объект props для каждой инстанции компонента на основе атрибутов JSX, которые вы передаете при создании компонента. Объект props представляет собой набор свойств и их значений, которые могут быть использованы внутри компонента. Давайте разберем, как это происходит:

– Создание компонента. При создании компонента в React, например, с использованием функционального компонента, вы определяете его сигнатуру, включая ожидаемые свойства.

Например:

```
function MyComponent (props) {
//…
}
```

– Использование компонента в JSX. При использовании компонента в JSX, вы передаете свойства, используя атрибуты компонента. Например:

```
<MyComponent name=«John» age= {30} />
```

В этом примере мы передаем два свойства (name и age) в компонент MyComponent.

– Создание объекта props: React автоматически создает объект props, который содержит переданные свойства. В данном случае, объект props будет выглядеть следующим образом:

```
{
name: «John»,
age: 30
```

и скрытие деталей реализации от внешнего мира. Это позволяет ограничить доступ к данным и методам объекта, предоставив только определенный интерфейс (публичные методы и свойства) для взаимодействия с ним.

```
}
```

Использование свойств внутри компонента: Внутри компонента вы можете обращаться к свойствам через объект props.

Например:

```
function MyComponent (props) {
  return (
    <div>
      <p> Name: {props.name} </p>
      <p> Age: {props.age} </p>
    </div>

)
}
```

В этом примере компонент MyComponent использует свойства name и age, которые были переданы через props.

Таким образом, React автоматически создает объект props для каждой инстанции компонента на основе переданных атрибутов JSX, и этот объект становится доступным внутри компонента для доступа к переданным свойствам.

8.4 Доступ к свойствам через объект props

В React, чтобы получить доступ к свойствам из объекта props внутри компонента, вы можете использовать обычную точечную нотацию, так как мы это делали ранее. Однако также существует более удобный способ с использованием деструктуризации[22] объекта props.

[22] Деструктуризация – это способ извлечь данные из структуры данных, таких как массивы или объекты, используя более компактный и удобный синтаксис.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, купив полную легальную версию на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.