**React**

1. **Что такое React?**
   * Это JavaScript библиотека для создания пользовательских интерфейсов. Основу кода составляют инкапсулированные компоненты, которые как кирпичики используется при построении сложных пользовательских интерфейсов. В основе библиотеки лежат такие понятия как виртуальныйDOM и JSX. Может работать на сервере (SSR) и на мобильных приложениях (React Native). Декларативен.
2. **Можно ли использовать React без webpack и create React app?**
   * Можно. Нужно подключить скрипты react, react-dom и js file с кодом в конец body. Чтобы иметь возможность использовать JSX нужно подключить еще и babel скрипт, добавив в скрипты с JSX type="text/babel". Следует помнить, что работа сайта при этом замедляется.

**JSX**

1. **Что такое JSX?**
   * JSX – Дополнение к синтаксису JavaScript. JS XML. Объединяет JS и HTML (логику и разметку) позволяя писать на HTML в React компонентах. Улучшает читабельность и производительность. Обязательно нужен babel или другой транспилятор. JSX использует стиль camelCase для свойств. Например, class становится className в JSX, а tabindex становится tabIndex.
2. **Можно ли использовать React без JSX?**
   * Да, JSX совсем необязателен, но значительно упрощает разработку и читабельность. Достаточно использовать JavaScript опираясь на React.createElement(). К слову, после транспиляции JSX представляет собой обычный JavaScript.
3. **Насколько безопасен ли JSX?**
   * Данные, введённые пользователем, можно безопасно использовать в JSX. По умолчанию React DOM экранирует все значения, включённые в JSX перед тем, как отрендерить их. Всё преобразуется в строчки, перед тем как быть отрендеренным. Это помогает предотвращать атаки межсайтовым скриптингом (XSS).
4. **Что такое React Фрагменты?** 
   * Возврат нескольких элементов из компонента является распространённой практикой в React. Фрагменты позволяют формировать список дочерних элементов, не создавая лишних узлов в DOM.
   * Полная запись <React.Fragment></ React.Fragment>
   * Сокращенная <></>
   * Единственный атрибут, фрагмента – ключ (key)
5. **Для чего всегда нужно импортировать React, даже там, где он не используется?**
   * Поскольку JSX компилируется в вызов React.createElement, библиотека React должна всегда быть в области видимости вашего JSX-кода.
   * К примеру, в данном коде оба импорта являются необходимыми, даже если на React и CustomButton нет прямых ссылок из JavaScript:

import React from 'react';

import CustomButton from './CustomButton';

function WarningButton() {

  // return React.createElement(CustomButton, {color: 'red'}, null);

  return <CustomButton color="red" />;

}

* + Исключением является только случаи, когда React загружается через тег script.

1. **Можно ли передавать в JSX пропсы JS выражения?**
   * Вы можете передавать любые JavaScript-выражения как пропсы, обернув их в {}.
   * Оператор if и цикл for не являются выражениями в JavaScript, поэтому их нельзя непосредственно использовать в JSX. Вместо этого, вы можете окружить ими JSX-код.
2. **Что может являться props.children и быть передано в JSX элемент?**
   * Строковые литералы - если вы поместите строку между открывающим и закрывающим тегом, то props.children будет равно этой строке.
   * JavaScript-выражения как дочерние компоненты
   * Функции как дочерние компоненты
   * Значения false, null, undefined и true — валидные дочерние компоненты. Просто они не рендерятся.
   * Вы можете смешивать различные типы потомков, скажем, использовать строковый литерал вместе с JSX-элементами.
3. **Возможно ли описать класс в React без использования ES6?**
   * Можно использовать модуль create-react-class

var createReactClass = require('create-react-class');

var Greeting = createReactClass({

  render: function() {

    return <h1>Привет, {this.props.name}</h1>;

  }

});

1. **Каковы особенности использования create-react-class в контексте его методов?**
   * В компонентах React, объявленных как классы ES6, методы следуют той же семантике, что и обычные классы ES6. Это означает, что они сами по себе не связывают this с экземпляром. Вам придётся явно использовать .bind(this) в конструкторе
   * Если вы используете createReactClass(), то это необязательно, так как все методы будут связаны:

var createReactClass = require('create-react-class');

var Greeting = createReactClass({

  render: function() {

    return <h1>Привет, {this.props.name}</h1>;

  }

});

var SayHello = createReactClass({

  getInitialState: function() {

    return {message: 'Привет!'};

  },

  handleClick: function() {

    alert(this.state.message);

  },

  render: function() {

    return (

      <button onClick={this.handleClick}>

        Поздороваться

      </button>

    );

  }

});

1. **Что такое примеси (mixins)?**
   * Иногда очень разные компоненты могут иметь общую функциональность. Иногда это называют [сквозной функциональностью](https://en.wikipedia.org/wiki/Cross-cutting_concern). createReactClass позволяет использовать для этого устаревшую систему mixins.
   * S6 запущен без поддержки примесей. Поэтому React не поддерживает примеси с классами ES6.
   * Кроме того, мы нашли множество проблем в кодовых базах, используя примеси, и не рекомендуем использовать их в коде.
2. **Для чего нужен <React.StrictMode>?**
   * StrictMode — инструмент для обнаружения потенциальных проблем в приложении. Строгий режим активирует дополнительные проверки и предупреждения для своих потомков. Строгий режим может быть включён для любой части приложения.
   * Проверки строгого режима работают только в режиме разработки; они не оказывают никакого эффекта в продакшен-сборке.
3. **Какие проверки делает StrictMode?**
   * Обнаружении небезопасных методов жизненного цикла
   * Предупреждении об использовании устаревшего API строковых реф
   * Предупреждении об использовании устаревшего метода findDOMNode
   * Обнаружении неожиданных побочных эффектов
   * Обнаружении устаревшего API контекста
   * Обеспечение переиспользованного состояния

**Работа с дом. Виртуальный дом (Virtual DOM). Refs.**

1. **Что такое Virtual DOM виртуальный дом и как он работает?**
   * Подход, при котором виртуальное представление дом хранится в памяти в виде JS объекта и этот виртуальный дом синхронизируется с настоящим, обновляя только те его части, где необходима повторная отрисовка. В React для этого используется библиотека React-DOM. Сам процесс называется **согласование(reconciliation).** Также React использует внутренние объекты называемые **волокна (fibers) –** js объекты, содержащие информацию об компоненте, входные параметры и результат.
2. **Как работает согласование в React?**
   * При работе с React вы можете понимать render() как функцию, которая создаёт дерево React-элементов в какой-то момент времени. При последующем обновлении состояния или пропсов функция render() вернёт новое дерево React-элементов.
   * Теперь React должен понять, как эффективно обновить UI, чтобы он совпадал с новейшим из деревьев.
   * Существует несколько общих решений алгоритмической проблемы трансформации одного дерева в другое за минимальное количество операций. Тем не менее, передовые алгоритмы имеют сложность порядка O(n3), где n — это число элементов в дереве.
   * Если бы мы использовали это в React, отображение 1000 элементов потребовало бы порядка миллиарда сравнений. Это слишком дорого. Взамен, **React реализует эвристический алгоритм O(n), который основывается на двух предположениях:**
   * Два элемента с разными типами произведут разные деревья.
   * Разработчик может указать, какие дочерние элементы могут оставаться стабильными между разными рендерами с помощью пропа key.
3. **Кратко опишите алгоритм сравнения двух деревьев в React?**
   * Поведение различается в зависимости от типов корневых элементов.
   * Всякий раз, когда корневые элементы имеют различные типы, React уничтожает старое дерево и строит новое с нуля. При уничтожении дерева старые DOM-узлы удаляются. Экземпляры компонента получают componentWillUnmount(). При построении нового дерева новые DOM-узлы вставляются в DOM. Экземпляры компонента получают UNSAFE\_componentWillMount(), а затем componentDidMount(). Любые компоненты, лежащие ниже корневого, также размонтируются, а их состояние уничтожится.
   * При сравнении двух React DOM-элементов одного типа, React смотрит на атрибуты обоих, сохраняет лежащий в основе этих элементов DOM-узел и обновляет только изменённые атрибуты.
   * Когда компонент обновляется, его экземпляр остаётся прежним, поэтому его состояние сохраняется между рендерами. React обновляет пропсы базового экземпляра компонента для соответствия новому элементу и вызывает UNSAFE\_componentWillReceiveProps(), UNSAFE\_componentWillUpdate и componentDidUpdate() на базовом экземпляре.
   * Далее вызывается метод render() и алгоритм сравнения рекурсивно обходит предыдущий и новый результаты.
   * По умолчанию при рекурсивном обходе дочерних элементов DOM-узла React проходит по обоим спискам потомков одновременно и создаёт мутацию, когда находит отличие. Когда у дочерних элементов есть ключи, React использует их, чтобы сопоставить потомков исходного дерева с потомками последующего дерева.
4. **Что такое render() и для чего он нужен?**
   * render() – единственный обязательный метод в классовом компоненте. При вызове он проверяет this.props и this.state и возвращает один из следующих вариантов:
   * **Элемент React**. Обычно создаётся с помощью JSX. Указывает React, что рендерить: DOM-узел или пользовательский компонент. Например, <div /> или <MyComponent />.
   * **Массивы и фрагменты**. Возвращает несколько элементов из render().
   * **Порталы**. Рендерит несколько дочерних элементов в другое поддерево DOM.
   * **Строки и числа**. Рендерит текстовые DOM-узлы.
   * **Booleans** **или** **null**. Ничего не рендерит. (Обычно необходим для поддержки паттерна return **test && <Child />**, где test — логическое значение.)
   * Функция render() должна быть чистой. Это означает, что она не изменяет состояние компонента, всегда возвращает один и тот же результат, не взаимодействует напрямую с браузером.
5. **Что такое Реф (ref)?**
   * Рефы дают возможность получить доступ к DOM-узлам или React-элементам, созданным в рендер-методе. Дают возможность императивно изменить дочерний элемент, обойдя обычный поток данных.
6. **Когда использовать рефы?**
   * Управление фокусом, выделение текста или воспроизведение медиа.
   * Императивный вызов анимаций.
   * Интеграция со сторонними DOM-библиотеками.
   * Работа с формами, построенными на неконтролируемых компонентах.
7. **Как создать реф?**
   * Рефы создаются с помощью React.createRef() или хука useRef() и прикрепляются к React-элементам через ref атрибут.

class MyComponent extends React.Component {

  constructor(props) {

    super(props);

    this.myRef = React.createRef();

  }

  render() {

    return <div ref={this.myRef} />;

  }

}

1. **Как получить доступ к рефу?**
   * Когда реф передаётся элементу в методе render, ссылка на данный узел доступна через свойство рефа current.

const node = this.myRef.current;

* + Для получения доступа к значению рефа (value) следует использовать myRef.current.value:

const node = this.myRef.current.value;

1. **Можно ли добавить реф к компоненте?**
   * Можно, но это сработает только случае, если компонент объявлен как классовый.
   * По умолчанию нельзя использовать атрибут ref с функциональными компонентами, потому что для них не создаётся экземпляров.
   * Если вам нужен реф на функциональный компонент, можете воспользоваться forwardRef (возможно вместе с useImperativeHandle хуком), либо превратить его в классовый компонент.
   * useImperativeHandle настраивает значение экземпляра, которое предоставляется родительским компонентам при использовании ref.

function FancyInput(props, ref) {

  const inputRef = useRef();

  useImperativeHandle(ref, () => ({

    focus: () => {

      inputRef.current.focus();

    }

  }));

  return <input ref={inputRef} ... />;

}

FancyInput = forwardRef(FancyInput);

1. **Можно ли передать DOM-рефы родительским компонентам?**
   * В общем случае, такой подход не рекомендуется, т. к. ведёт к нарушению инкапсуляции компонента, но иногда он может пригодиться для задания фокуса или измерения размеров, или положения дочернего DOM-узла.
   * Несмотря на то, что можно было бы добавить реф к дочернему компоненту, такое решение не является идеальным, т. к. вы получите экземпляр компонента вместо DOM-узла. Кроме того, это не сработает с функциональными компонентами.
   * Если вы работаете с React 16.3 или новее, мы рекомендуем использовать **перенаправление рефов** для таких случаев. **Перенаправление рефов позволяет компонентам осуществлять передачу рефа любого дочернего компонента как своего собственного**.
2. **Как использовать перенаправление рефов?**
   * Перенаправление рефов позволяет взять ref из атрибутов компонента, и передать («перенаправить») его одному из дочерних компонентов.
   * В данном примере мы используем React.forwardRef в компоненте FancyButton, чтобы получить реф и передать его в дочерний DOM-элемент button.

const FancyButton = React.forwardRef((props, ref) => (

  <button ref={ref} className="FancyButton">

    {props.children}

  </button>

));

// Теперь реф будет указывать непосредственно на DOM-узел button:

const ref = React.createRef();

<FancyButton ref={ref}>Click me!</FancyButton>;

* + Таким образом, когда мы будем применять FancyButton в других компонентах, мы сможем получить реф находящегося в нём DOM-узла button и использовать его так же, как если бы мы рендерили непосредственно button.

1. **Что такое колбэк-рефы и для чего они нужны?**
   * «колбэк-рефы» предоставляют более полный контроль над присвоением и сбросом рефов. Вместо того, чтобы передавать атрибут ref созданный с помощью createRef(), вы можете передать функцию. Данная функция получит экземпляр React-компонента или HTML DOM-элемент в качестве аргумента, которые потом могут быть сохранены или доступны в любом другом месте.
   * React вызовет ref колбэк с DOM-элементом при монтировании компонента, а также вызовет его со значением null при размонтировании. Рефы будут хранить актуальное значение перед вызовом методов componentDidMount или componentDidUpdate.

class CustomTextInput extends React.Component {

  constructor(props) {

    super(props);

    this.textInput = null;

    this.setTextInputRef = (element) => {

      this.textInput = element;

    };

    this.focusTextInput = () => {

      // Устанавливаем фокус на текстовом поле ввода с помощью чистого DOM API

      if (this.textInput) this.textInput.focus();

    };

  }

  componentDidMount() {

    // устанавливаем фокус на input при монтировании

    this.focusTextInput();

  }

  render() {

    // Используем колбэк в `ref`, чтобы сохранить ссылку на DOM-элемент

    // поля текстового ввода в поле экземпляра (например, this.textInput).

    return (

      <div>

        <input type="text" ref={this.setTextInputRef} />

        <input

          type="button"

          value="Focus the text input"

          onClick={this.focusTextInput}

        />

      </div>

    );

  }

}

**React элемент и React компонент**

1. **Что такое React элемент?**
   * JSX представляет собой объекты - Babel компилирует JSX в вызовы React.createElement(). React.createElement() проводит некоторые проверки и создаёт объект. Эти объекты называются React-элементами. Они содержат поля type (со значением например ‘h1’ или ‘div’), props и др. Элементы — мельчайшие кирпичики React-приложений.
2. **Можно ли мутировать** **React элемент?**
   * Элементы React иммутабельны. После создания элемента нельзя изменить его потомков или атрибуты. Элемент похож на кадр в фильме: он отражает состояние интерфейса в конкретный момент времени.
3. **Что такое React компонент?**
   * Ведут себя как обычные функции JavaScript. Они принимают произвольные входные данные (так называемые «пропсы») и возвращают React-элементы, описывающие, что мы хотим увидеть на экране. Позволяют разбить интерфейс на независимые части, их можно складывать вместе и использовать несколько раз. Существуют функциональные и классовые компоненты.
4. **Что такое функциональный компонент?**
   * это компонент, описанный как функция, она получает данные в одном объекте («пропсы») в качестве параметра и возвращает React-элемент. Буквально являются функцией.
5. **Что такое классовый компонент?**
   * это компонент, описанный как ES6 класс, который наследуется от React.Component. Имеет обязательный метод render(). Определяет свой стейт и получает пропсы используя constructor(). Обращается к пропсам и стейту через this. Также возвращает React элемент.
6. **Чем отличаются функциональный компонент от классового?**
   * С точки зрения React, эти два компонента эквивалентны. Основное отличие заключается в синтаксисе.
7. **Как ты понимаешь выражение React компонент — это чистая функция?**
   * React-компоненты обязаны вести себя как чистые функции по отношению к своим пропсам. Это одно из основополагающих правил React, которое нельзя нарушать.

**Пропсы (props) и стейт (state)**

1. **Что такое пропсы (props)?**
   * props – свойства, данные, которые передаются от родительской к дочерней компоненте. Доступны только для чтения - компонент никогда не должен что-то записывать в свои пропсы — вне зависимости от того, функциональный он или классовый, так как компоненты – чистые функции и они не меняют свои входные данные. И не могут быть отправлены обратно – однопоточный поток данных.
2. **Что такое пропс (props) drilling?**
   * Пропс дриллинг (сверление) – это передача props через большое число компонент (которым они не нужны) к конечной компоненте получателю.
   * **Пути решения проблемы:**
   * Необходимо держать state так близко к компоненте ка это возможно. Следует избегать излишней разбивки компонент. Но это не значит держать все внутри одной компоненты. Вместо этого следует передать саму компоненту вместе с необходимыми ей props используя композиции.
   * Использовать React context. В малых и средних приложениях предпочтительно использовать его, чтобы не перегружать приложение Redux или Mobx.
   * Использовать recoil для средних приложений.
   * Использовать State management библиотеки – Redux, Mobx для крупных приложений.
3. **Что такое стейт (state)?**
   * Это внутренние состояние компоненты. Оно контролируется и доступно только конкретному компоненту. Объект, определяющий представление и поведение компоненты. Используется для представления динамических данных. Нельзя изменять напрямую. Нужно использовать setState().
4. **setState синхронный или асинхронный?**
   * Асинхронный. setState говорит React запустить следующую итерацию рендера, однако React может оптимизировать этот процесс и несколько вызовов setState приведут к одному рендеру.
5. **Что обозначает термин однонаправленный поток данных в React?**
   * Состояние компонент инкапсулировано, не важно функциональны они или классовые. Компонент может передать своё состояние вниз по дереву в виде пропсов дочерних компонентов. Но получив эти пропсы дочерний компонент не знает, откуда они взялись изначально — из состояния, пропсов или просто JavaScript-выражения.
   * Это, в общем, называется «нисходящим» («top-down») или «однонаправленным» («unidirectional») потоком данных. Состояние всегда принадлежит определённому компоненту, а любые производные этого состояния могут влиять только на компоненты, находящиеся «ниже» в дереве компонентов.
6. **Почему нельзя копировать props и перенося его в this.state?**
   * Это распространённая ошибка.

constructor(props) {

  super(props);

  // Не делайте этого!

  this.state = { color: props.color };

 }

* + Проблема в том, что это излишне и приводит к багам (обновления в пропе color не будут зафиксированы в состоянии). Вместо этого используйте this.props.color.

**Рендер-пропсы\***

1. **Что обозначает термин рендер-пропсы?**
   * Термин «рендер-проп» относится к возможности компонентов React разделять код между собой с помощью пропа, значение которого является функцией.
   * Компонент с рендер-пропом берёт функцию, которая возвращает React-элемент, и вызывает её вместо реализации собственного рендера.

<DataProvider render={data => (

  <h1>Привет, {data.target}</h1>

)}/>

* + рендер-проп — функция, которая сообщает компоненту что необходимо рендерить.

**Жизненный цикл (React Lifecycle)**

1. **Что такое жизненный цикл (Lifecycle) React компонента?**
   * Это период с момента первоначальный рендеринга компоненты (монтирования – mounting в DOM), до удаления DOM-узла, созданного компонентой (размонтирования – unmounting).
2. **Назови методы жизненного цикла (Lifecycle) компонента?**
   * **Монтирование (mounting)**
   * initialization (initial state or default props) / или вызов constructor()
   * componentWillMount() – перед рендерингом. УСТАРЕЛ
   * СЕЙЧАС- static getDerivedStateFromProps()
   * render() - рендер
   * componentDidMount() – запускается после того, как компонент отрендерился в DOM
   * **Обновление (updating)**
   * componentWillReceiveProps() – перед получением пропсов. УСТАРЕЛ.
   * СЕЙЧАС static getDerivedStateFromProps()
   * shouldComponentUpdate() - true or false, можно принудительно передать true для обновления
   * componentWillUpdate() – перед обновлением. УСТАРЕЛ
   * СЕЙЧАС getSnapshotBeforeUpdate()
   * render() - рендер
   * componentDidUpdate() – после обновления
   * **unmounting**
   * componentWillUnmount – после удаления компоненты. Для очистки памяти.
3. **Какой из методов жизненного цикла подходит для серверных запросов?**
   * componentDidMount() вызывается сразу после монтирования (то есть, вставки компонента в DOM). В этом методе должны происходить действия, которые требуют наличия DOM-узлов. Это хорошее место для создания сетевых запросов.
   * componentDidUpdate() вызывается сразу после обновления. Не вызывается при первом рендере. Метод позволяет работать с DOM при обновлении компонента. Также он подходит для выполнения таких сетевых запросов, которые выполняются на основании результата сравнения текущих пропсов с предыдущими.
4. **Можно ли вызвать setState в componentDidMount()?**
   * Это вызовет дополнительный рендер перед тем, как браузер обновит экран. Используйте этот подход с осторожностью, он может вызвать проблемы с производительностью. В большинстве случаев начальное состояние лучше объявить в constructor(). Однако, это может быть необходимо для случаев, когда нужно измерить размер или положение DOM-узла, на основе которого происходит рендер. Например, для модальных окон или всплывающих подсказок.

**Rendering и отрисовка (обновление DOM)**

1. **Что вызывает обновление rendering компонентов?**
   * setState у useState и useReducer – изменение ее состояния
   * rendering родительского компонента приведет к рендерингу дочернего через изменения props
   * в классовых компонентах есть функция forceUpdate()
2. **Что НЕ вызывает обновление rendering компонентов?**
   * useState и useReducer не приведут к обновлению если состояние не изменилось.
   * если мутировать состояние напрямую
3. **Как реализовать forceUpdate() в функциональной компоненте?**
   * const [ \_ , forceUpdate ] = useReducer( (x) => x + 1, 0 ) – можно также обернуть это в custom hook
   * Это может быть полезно при использовании Object.assign(obj, {‘new-field’: ‘value’}), который быстрее чем деструктуризация ( {…obj, new-field: value} ) НО не вызывает ререндеринг
4. **Тоже ли самое rendering и отрисовка (обновление DOM)?**
   * НЕТ – Компонент может заререндериться и без визуальных изменений. Рендеринг родительской компоненты вызовет рекурсивно рендеринг дочерних и те дочерние, что вернули тот же самый результат, т. е. не изменились - не будут перерисованы в DOM. Но React должен сделать ререндер, чтобы определить эти различия.

**События (events) в React**

1. **Чем отличается обработка событий в React от обработки в DOM c помощью JavaScript.**
   * События в React именуются в стиле camelCase вместо нижнего регистра.
   * В JSX передается функция как обработчик события вместо строки.

<button onclick="activateLasers()">

<button onClick={activateLasers}>

* + в React нельзя предотвратить обработчик события по умолчанию, вернув false. Следующий код не сработает. Нужен отдельный e.preventDefault();

<form onsubmit="console.log('Отправлена форма.'); return false">

* + React использует специальную кроссбраузерную обертку SyntheticEvent - все обработчики событий в React получают экземпляр SyntheticEvent. У неё такой же интерфейс, как и у нативного события, включая методы stopPropagation() и preventDefault().

1. **Для чего в React используется обертка SyntheticEvent?**
   * Эта кроссбраузерная обёртка помогает событиям работать одинаково во всех браузерах. React нормализует события так, чтобы они содержали одинаковые свойства во всех браузерах.
2. **Можно ли каким-то образом получить нативное браузерное событие?**
   * Да, нужно обратиться к атрибуту event.nativeEvent.
3. **На какой фазе вызываются обработчики событий в React?**
   * Обработчики вызываются на фазе всплытия (bubbling).
4. **Можно ли зарегистрировать событие на фазе перехвата (capture) в React? Как это сделать?**
   * Да, достаточно добавить Capture к имени события; например, вместо onClick использовать onClickCapture, чтобы обработать событие на фазе перехвата.
5. **Назовите некоторые из обработчиков событий в React, которые вы знаете?**
   * **Буфер**. onCopy onPaste. **Клавиатура** onKeyDown onKeyPress. **Фокус** onFocus onBlur. **Форма** - onChange onInput onInvalid. **Общие** onError onLoad. **Мышь** - onDoubleClick onDrag onDragEnd onDrop onMouseEnter onMouseMove onMouseOver. **Курсор** onPointerDown onPointerMove onGotPointerCapture. **Выбор** onSelect. **UI** onScroll. **Анимация** onAnimationStart onAnimationEnd onAnimationIteration и **другие события** в документации.
6. **Как правильно описать обработчик событий в классовой компоненте?**
   * В компоненте, определённом с помощью ES6-класса, в качестве обработчика события обычно выступает один из методов класса. Есть три способа описать обработчик.
   * Использовать привязку (bind) контекста внутри конструктора. При обращении к this в JSX-колбэках необходимо учитывать, что методы класса в JavaScript по умолчанию не привязаны к контексту:

class Toggle extends React.Component {

  constructor(props) {

    super(props);

    this.state = { isToggleOn: true };

    // Эта привязка обязательна для работы `this` в колбэке.

    this.handleClick = this.handleClick.bind(this);

  }

  handleClick() {

    this.setState((prevState) => ({

      isToggleOn: !prevState.isToggleOn,

    }));

  }

  render() {

    return (

      <button onClick={this.handleClick}>

        {this.state.isToggleOn ? 'Включено' : 'Выключено'}

      </button>

    );

  }

}

* + Следующий способ — это использовать синтаксис публичных полей класса чтобы правильно привязать колбэки. Такой синтаксис доступен в Create React App по умолчанию:

class LoggingButton extends React.Component {

  // Такой синтаксис гарантирует, что `this` привязан к handleClick.

  handleClick = () => {

    console.log('значение this:', this);

  };

  render() {

    return <button onClick={this.handleClick}>Нажми на меня</button>;

  }

}

* + Можно также использовать стрелочные функции, но стоит помнить, что при каждом рендере LoggingButton создаётся новый колбэк. Если этот колбэк попадает как проп в дочерние компоненты, эти компоненты могут быть отрендерены снова:

class LoggingButton extends React.Component {

  handleClick() {

    console.log('значение this:', this);

  }

  render() {

    // Такой синтаксис гарантирует, что `this` привязан к handleClick.

    return <button onClick={() => this.handleClick()}>Нажми на меня</button>;

  }

}

1. **Что такое пул событий в React и для чего нужен e.persist()?**
   * Сразу стоит отметить, что версия React 17 не использует пул событий. Он использовался в версиях 16 и ниже, а также в React Native.
   * Все события SyntheticEvent содержатся в пуле. Это означает, что объект SyntheticEvent будет повторно использован, а все его свойства будут очищены после вызова обработчика события. Следующий код не сработает:

function handleChange(e) {

  // Ничего не произойдёт, потому что объект события используется повторно.

  setTimeout(() => {

    console.log(e.target.value); // Слишком поздно!

  }, 100);

}

* + Если нужно обратиться к свойствам объекта события после выполнения обработчика события, то необходимо вызвать e.persist():

function handleChange(e) {

  // Останавливает React от сброса свойств объекта события:

  e.persist();

  setTimeout(() => {

    console.log(e.target.value); // Сработает

  }, 100);

}

**Условный рендеринг.**

1. **Что такое условный рендеринг?**
   * React позволяет разделить логику на независимые компоненты. Эти компоненты можно показывать или прятать в зависимости от текущего состояния. Условный рендеринг в React работает так же, как условные выражения работают в JavaScript:

function Greeting(props) {

  const isLoggedIn = props.isLoggedIn;

  if (isLoggedIn) {

    return <UserGreeting />;

  }

  return <GuestGreeting />;

}

1. **Что такое переменные-элементы в условном рендеринге?**
   * Это запись компоненты с пропсами внутрь переменной в методе render() и последующий ее рендер внутри фигурных скобок:

render() {

  const isLoggedIn = this.state.isLoggedIn;

  let button;

  if (isLoggedIn) {

    button = <LogoutButton onClick={this.handleLogoutClick} />;

  } else {

    button = <LoginButton onClick={this.handleLoginClick} />;

  }

  return (

    <div>

      <Greeting isLoggedIn={isLoggedIn} />

      {button}

    </div>

  );

}

1. **Можно ли использовать встроенные условия if с логическим оператором && внутри фигурных скобок JSX?**
   * Условная отрисовка с используем двойного && весьма распространена и хорошо читается. Можно добавлять JSX при условии, что какой-то стейт вернет true. В JavaScript-выражение true && expression всегда вычисляется как expression, а выражение false && expression — как false.

{

  unreadMessages && (

    <h2>У вас {unreadMessages.length} непрочитанных сообщений.</h2>

  );

}

1. **Можно ли использовать тернарные выражения внутри JSX?**
   * Да, JSX работает с тернарными операторами.

<div>

  Пользователь <b>{isLoggedIn ? 'сейчас' : 'не'}</b> на сайте.

</div>;

1. **Как спрятать компонент, который уже был отрендерен другим компонентом?**
   * Для этого достаточно вернуть null вместо того, что обычно возвращает компонент. Но сам факт возврата null из метода render компонента никак не влияет на срабатывание методов жизненного цикла компонента. Например, componentDidUpdate будет всё равно вызван.

**Ключи и списки**

1. **Что такое атрибут ключ и для чего он нужен?**
   * Ключ — это специальный строковый атрибут, который нужно указывать при создании списка элементов в JSX. Обычно такие списки создаются при помощи map().
   * Ключи помогают React определять, какие элементы были изменены, добавлены или удалены. Их необходимо указывать, чтобы React мог сопоставлять элементы массива с течением времени. Алгоритм сравнения React сравнивает не каждый элемент в списке, а только их ключи для ускорения работы.
   * Если не указать ключи, React выдаст напоминание в виде ошибки в консоль.
2. **Какие ключи использовать лучше всего?**
   * Лучше всего использовать ID из данных как ключи.
   * Когда у вас нет заданных ID для списка, то в крайнем случае можно использовать индекс элемента как ключ. Документация не рекомендует использовать индексы как ключи, если порядок элементов может поменяться. Это негативно скажется на производительности и может вызвать проблемы с состоянием компонента.
3. **Что будет если не указать ключи для элементов в списке?**
   * Если не указать ключи, React по умолчанию будет использовать индексы как ключи и выдаст напоминание в виде ошибки в консоль.
4. **Почему индексы ключи приводят к проблемам?**
   * Индексы как ключи – это антипаттерн. В случае если список сортируется, фильтруется и принимает дополнительные элементы. В этом случае React может быть запутан в том, какие пропсы принадлежат каким mapped компонентам и отобразить неверный UI.
5. **Что лучше использовать для ключей?**
   * Лучше использовать уникальные списки id, сгенерированные при создании элементов списка. Лучше использовать для этого специальные утилиты для генерации id – uuid, shortid, nanoid. Но не использовать их во время рендера.
6. **Должны ли ключи быть уникальными на уровне приложения?**
   * Нет, ключи внутри массива должны быть уникальными только среди своих соседних элементов.
7. **Что произойдет если использовать, например Math.random() для создания ключей?**
   * Нестабильные ключи ( например, произведённые с помощью Math.random() ) вызовут необязательное пересоздание многих экземпляров компонента и DOM-узлов, что может вызывать ухудшение производительности и потерю состояния у дочерних компонентов.

**React формы. Управляемые и неуправляемые компоненты.**

1. **Что такое управляемая (контролируемая) форма?**
   * Форма является управляемой, если ее элементы (input, textarea, select) являются управляемыми компонентами. Для ее input элементов созданы state, state задан в качестве value и при изменении value вызывается setState в обработчике событий onChange. Состояние React компонента становится единственным источником истины. Значение элементов формы input в этом случае будет контролировать React.
2. **Что такое управляемая компонента?**
   * управляемая компонента хранит свои данных во внешнем state, например <input>, <textarea> и <select> хранят свои значения в state формы. Обновляет это значение только через вызов setState() которое передается в onChange(). Значение элемента формы input в этом случае будет контролировать React.
3. **Что такое неуправляемая компонента?**
   * неуправляемая компонента хранит данные формы прямо в DOM. В HTML элементы формы, такие как <input>, <textarea> и <select>, обычно сами управляют своим состоянием и обновляют его, когда пользователь вводит данные. Для форм, построенных на неуправляемых компонентах, не нужно писать обработчики событий, а читать значения напрямую из DOM используя рефы (ref).
4. **Когда следует использовать неуправляемые компоненты, а когда управляемые?**
   * В большинстве случаев React документация рекомендует использовать управляемые компоненты. Но существуют кейсы и для неуправляемых компонент.
   * Для форм лучше использовать контролируемые компоненты, для них доступна валидация прямо во время ввода значений в inputs – до нажатия submit. Валидировать поля удобно с помощью useEffect() в зависимостях [ ] которого нужно указать inputName. Исключением может быть только инпут с типом file.
   * В React <input type="file">, который позволяет пользователю выбрать один или несколько файлов из дискового устройства, всегда является неуправляемым компонентом, потому что его значение может быть установлено только пользователем, а не программным путём.
   * В случае работы с модальными окнами – могут быть uncontrolled and controlled. Оба получают { children } через пропсы и выводят их внутри себя через { children }. Uncontrolled сам содержит свой state и процесит открытие закрытие изнутри компоненты. Controlled modal контролируется снаружи и получает переменную shouldShow и функцию onRequestClose() которые контролируют модальное окно.
   * Работа с onboarding flows (входящие потоки) – это просто компонента, которая отображает разные шаги в onboarding процессе. Эта компонента пошагово опрашивает пользователя собирая данные в объект. Может быть uncontrolled and controlled. Controlled передает управление в родительскую компоненту, что позволяет отобразить дополнительные шаги или скрыть ненужные.
   * неуправляемые, если вы переносите существующую кодовую базу в React, или когда работаете над интеграцией React-приложения с другой библиотекой.
5. **Есть ли особенности в работе с тегом select в React?**
   * Да, в управляемом select значение по умолчанию будет задано в его атрибуте value={this.state.value}. Вместо атрибута selected у option в неуправляемом select. Чтобы выбрать несколько опций в select по умолчанию можно отправить в атрибут value массив и добавить атрибут multiple={true}.
6. **Создайте форму с контролируемыми компонентами.**

import React, { useState } from 'react';

export const ControlledForm = () => {

  const [input, setInput] = useState('');

  const [textarea, setTextarea] = useState('');

  const [select, setSelect] = useState('');

  const submitHandler = (e) => {

    e.preventDefault();

    console.log({ input: input, textarea: textarea, select: select });

  };

  return (

    <form onSubmit={(e) => submitHandler(e)}>

      Regular input

      <input

        type="text"

        value={input}

        onChange={(e) => setInput(e.target.value)}

      />

      Text area

      <textarea

        type="text"

        value={textarea}

        onChange={(e) => setTextarea(e.target.value)}

      />

      Select

      <select value={select} onChange={(e) => setSelect(e.target.value)}>

        <option value="red">red</option>

        <option value="green">green</option>

      </select>

      <button type="submit">Submit</button>

    </form>

  );

};

1. **Создайте классовую форму с контролируемыми компонентами.**

import React from 'react';

export class ClassControlledForm {

  constructor(props) {

    super(props);

    this.state = { input: '' };

    this.handleChange = this.handleChange.bind(this);

    this.handleSubmit = this.handleSubmit.bind(this);

  }

  handleChange(event) {

    this.setState({ value: event.target.value });

  }

  handleSubmit(event) {

    event.preventDefault();

    console.log({ input: this.state.input });

  }

  render() {

    return (

      <form onSubmit={this.handleSubmit}>

        <input

          type="text"

          value={this.state.value}

          onChange={this.handleChange}

        />

        <input type="submit" value="Submit" />

      </form>

    );

  }

}

1. **Создайте форму с неконтролируемыми компонентами.**

import React, { useRef } from 'react';

export const UncontrolledForm = () => {

  const input = useRef();

  const textarea = useRef();

  const select = useRef();

  const submitHandler = (e) => {

    e.preventDefault();

    console.log({

      input: input.current.value,

      textarea: textarea.current.value,

      select: select.current.value,

    });

  };

  return (

    <form onSubmit={(e) => submitHandler(e)}>

      Regular input

      <input ref={input} type="text" />

      textarea

      <textarea ref={textarea}></textarea>

      select

      <select ref={select}>

        <option value="red">red</option>

        <option value="green">green</option>

      </select>

      <button type="submit">submit</button>

    </form>

  );

};

1. **Как заблокировать редактирование input?** 
   * Достаточно задать его атрибуту value начальное значение отличное от null или undefined.
2. **Какие дополнительные возможности, упрощающие работу с формами, вы знаете?**
   * Существуют уже готовые библиотеки позволяющие строить формы с настроенной валидацией полей и прочими упрощающими жизнь функциями. Например Formik.
   * Также существуют готовые в отношении стилей и не только разнообразные компоненты форм. Например, библиотека React.bootstrap.

**Подъем состояния**

1. **Что такое подъем состояния в React?**
   * Когда несколько компонент должны отображать одни и те же изменяющиеся данные то state рекомендуется поднимать до ближайшего общего предка. Вместо того, чтобы пытаться синхронизировать состояние между различными компонентами, вы должны полагаться на однонаправленный поток данных.
2. **Как реализовать использование единого стейта несколькими компонентами?**
   * Поднять общий state до ближайшего общего предка. Использовать однонаправленный поток данных от родителя к потомкам.

**Композиция и наследование**

1. **Что такое композиция в React?**
   * Передача компонент в качестве пропсов (children). Передача происходит по средствам вложенного синтаксиса, когда родительская компонента охватывает дочерние как html элемент.
2. **Что такое prop children?**
   * Специальный проп children содержит в себе переданные в компоненту дочерние компоненты. Достаточно написать {children} или {props.children}, чтобы вывести их.
3. **Что такое композиция - вставка?**
   * Некоторые компоненты не знают своих потомков заранее. Это особенно характерно для таких компонентов, как Sidebar или Dialog, которые представляют из себя как бы «коробку», в которую можно что-то положить. Для таких компонентов рекомендуется использовать специальный проп children, который передаст дочерние элементы сразу на вывод:

function FancyBorder(props) {

  return (

    <div className={'FancyBorder FancyBorder-' + props.color}>

      {props.children}

    </div>

  );

}

* + Это позволит передать компоненту произвольные дочерние элементы, вложив их в JSX:

function WelcomeDialog() {

  return (

    <FancyBorder color="blue">

      <h1 className="Dialog-title">Добро пожаловать</h1>

      <p className="Dialog-message">

        Спасибо, что посетили наш космический корабль!

      </p>

    </FancyBorder>

  );

}

* + Необязательно использовать children – можно использовать свой формат:

function SplitPane(props) {

  return (

    <div className="SplitPane">

      <div className="SplitPane-left">{props.left}</div>

      <div className="SplitPane-right">{props.right}</div>

    </div>

  );

}

function App() {

  return <SplitPane left={<Contacts />} right={<Chat />} />;

}

1. **Что такое композиция - специализация?**
   * Некоторые компоненты можно рассматривать как «частные случаи» других компонентов. Например, WelcomeDialog может быть частным случаем Dialog. В React это можно сделать через композицию, где «частный» вариант компонента рендерит более «общий» и настраивает его с помощью пропсов:

function Dialog(props) {

  return (

    <FancyBorder color="blue">

      <h1 className="Dialog-title">{props.title}</h1>

      <p className="Dialog-message">{props.message}</p>

    </FancyBorder>

  );

}

function WelcomeDialog() {

  return (

    <Dialog

      title="Добро пожаловать"

      message="Спасибо, что посетили наш космический корабль!"

    />

  );

}

1. **Что насчет наследования?**
   * Согласно документации: в Facebook мы используем React в тысячах компонентов, и не находили случаев, когда бы рекомендовали создавать иерархии наследования компонентов. Пропсы и композиция дают вам всю гибкость, необходимую для настройки внешнего вида и поведения компонента явным и безопасным способом.

**Доступность контента (Accessibility)**

1. **Работают ли HTML атрибуты aria-\* в JSX? Каково их отличие от других атрибутов.**
   * все HTML-атрибуты aria-\* полностью поддерживаются в JSX. Но несмотря на то, что большинство DOM-свойств и атрибутов в React пишутся в стиле camelCase, атрибуты aria-\* должны быть написаны с разделением дефисами.
2. **Каковы особенности семантической верстки с JSX.**
   * Чтобы не нарушать семантическую версту стоит использовать React фрагменты <></> при рендеринге компонент.
   * элемент label в JSX также должен иметь сноску на элемент, к которому он относится, однако, стоит помнить про cameCase:

<label htmlFor="namedInput">Имя:</label>

<input id="namedInput" type="text" name="name"/>

* + Чтобы управлять фокусом в React, можно использовать рефы на DOM-элементы.

**Code splitting (разделение кода)**

1. **Что обозначает термин code splitting (разделение кода) в React?**
   * В первую очередь это обозначает что вместо отправления всего кода React приложения клиенту целиком, мы отправляем его по частям по мере необходимости. Это техника обеспечивает производительность и экономит ресурсы так как позволяет уменьшить величину кода, которую необходимо загрузить клиенту. Реализуется через { lazy, Suspense } from ‘react’. Реализация:

const One = lazy(() => import('./One'));

const Two = lazy(() => import('./Two'));

const Three = lazy(() => import('./Three'));

export const About = () => {

  return (

    <>

      <h1>About</h1>

      <Suspense fallback={<p>Loading Components...</p>}>

        <One />

        <Two />

        <Three />

      </Suspense>

    </>

  );

};

* + Компоненты внутри Suspense будут загружены в виде трех отдельных чанков (chunk) только после того, как пользователь перейдет на About page.
  + Также можно разделить код основываясь на Routes, (роутах). Ведь, по сути, мы не обязаны загружать код до того, как пользователь перейдет на определенную страницу.

1. **Что следует помнить при работе с lazy()?**
   * Следует помнить, что lazy работает только с экспортом по умолчанию (export default).
   * Если модуль, который требуется импортировать, использует именованный экспорт, можно создать промежуточный модуль, который повторно экспортирует его как модуль по умолчанию.

// ManyComponents.js

export const MyComponent = /\* ... \*/;

export const MyUnusedComponent = /\* ... \*/;

// MyComponent.js

export { MyComponent as default } from "./ManyComponents.js";

// MyApp.js

import React, { lazy } from 'react';

const MyComponent = lazy(() => import("./MyComponent.js"));

1. **Для чего нужен атрибут fallback у компонента Suspense?**
   * Этот атрибут принимает любой JSX, который будет отображен во время ленивой загрузки. Например, простое <p>Loading components…</p>
2. **Как разделить код основываясь на Routes (роутах)?**
   * Достаточно обернуть в Suspense наши Routes (роуты) или Switch если используется 5 версия. И импортировать с помощью lazy() компоненты передаваемы в каждый Route (роут). При переходе на каждый из Route (роутов) будет подгружаться соответствующий chunk (чанк).

import React, { Suspense, lazy } from 'react';

import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

const Home = lazy(() => import('./routes/Home'));

const About = lazy(() => import('./routes/About'));

const App = () => (

  <Router>

    <Suspense fallback={<div>Loading...</div>}>

      <Routes>

        <Route path="/" element={<Home />} />

        <Route path="/about" element={<About />} />

      </Routes>

    </Suspense>

  </Router>

);

1. **Когда НЕ следует использовать разделение кода (code splitting)?**
   * Не имеет смысла использовать разделение кода при работе с небольшими компонентами.
   * Также нет смысла использовать разделение кода внутри компоненты, которая уже была подвержена этому процессу ранее. Если только в ней нет дополнительной логики по скрытию отображению компонент внутри нее.
2. **Что делать если какой-то чанк кода не загрузился?**
   * В паре с разделением кода следует использовать предохранители (ErrorBoundary). Они покажут запасной UI и уберегут от размонтирования всего React приложения.

**Контекст (Context)**

1. **Что такое контекст в React?**
   * контекст разработан для передачи данных, которые можно назвать «глобальными» для всего дерева React компонентов. Позволяет избежать передачи пропсов в промежуточные компоненты. В основе стоят Providers – компоненты хранящие данные(state) и Consumers – компоненты принимающие данные(state).
2. **Когда следует использовать контекст?**
   * Контекст разработан для передачи данных, которые можно назвать «глобальными» для всего дерева React-компонентов (например, текущий аутентифицированный пользователь, UI-тема или выбранный язык).
   * Если говорить про использование контекста в стейт менеджменте, то он подойдет для небольших приложений.
3. **Опишите простое применение контекста в классовой компоненте.**
   * Нужно использовать SomeContext = React.createContext(value), <SomeContext.provider value=””> static contextType = ThemeContext в компоненте потребителе.

const ThemeContext = React.createContext('light');

export class App extends React.Component {

  render() {

    return (

      <ThemeContext.Provider value="dark">

        <Toolbar />

      </ThemeContext.Provider>

    );

  }

}

// Компонент, который находится в середине,

// больше не должен явно передавать тему вниз.

function Toolbar() {

  return (

    <div>

      <ThemedButton />

    </div>

  );

}

class ThemedButton extends React.Component {

  // Определяем contextType, чтобы получить значение контекста.

  static contextType = ThemeContext;

  render() {

    return <Button theme={this.context} />;

  }

}

1. **Напишите пример использования контекста с использованием useState() внутри provider?**
   * 1. Создать отдельные файл SomeContext.js, в котором создать глобальный контекст:

import { createContext } from 'react';

export const CounterContext = createContext();

* + 2. Создать в отдельном файле Context.Provider. Зачастую в нем понадобится useState():

import React, { useState } from 'react';

import { CounterContext } from './CounterContext';

export const CounterProvider = ({ children }) => {

  const [numberOfClicks, setNumberOfClicks] = useState(0);

  const increment = (amount) => {

    setNumberOfClicks(numberOfClicks + 1);

  };

  return (

    <CounterContext.Provider value={{ numberOfClicks, increment }}>

      {children}

    </CounterContext.Provider>

  );

};

* + 2\*. Также можно просто обернуть в <Context.Provider value={{ someValue, orSomeObj }}> без использования отдельного файла, в случае простого контекста.
  + 3. Передача доступа. Оборачиваем нужную компоненту в **Context.Provider** (CounterProvider):

import { CounterButton } from './CounterButton';

import './App.css';

import { CounterProvider } from './CounterProvider';

const App = () => {

  return (

    <CounterProvider>

      <h1>State Management Example</h1>

      <CounterButton />

    </CounterProvider>

  );

};

export default App;

* + 4. Использование. Теперь достаточно вызвать useContext(SomeContext) хук и передать в него контекст:

import { useContext } from 'react';

import { CounterContext } from './CounterContext';

export const CounterButton = () => {

  const { numberOfClicks, increment } = useContext(CounterContext);

  return (

    <>

      <p>You have clicked the button {numberOfClicks} times.</p>

      <button onClick={() => increment()}>Click</button>

    </>

  );

};

1. **Почему не стоит использовать контекст?**
   * это усложняет повторное использование компонентов. Если вы хотите избавиться от передачи некоторых пропсов на множество уровней вниз, обычно композиция компонентов является более простым решением, чем контекст.
2. **Могут ли быть провайдеры (Context.Provider) вложенными друг в друга?**
   * Компоненты Provider могут быть вложены друг в друга, переопределяя значение контекста глубже в дереве.
3. **Как влияет изменение value контекста на рендеринг вложенных в него компонент?**
   * Все потребители, которые являются потомками Provider, будут повторно рендериться, как только проп value у Provider изменится. Потребитель (включая .contextType и useContext) перерендерится при изменении контекста, даже если его родитель, не использующий данный контекст, блокирует повторные рендеры с помощью **shouldComponentUpdate**.
4. **Что такое Context.Consumer?**
   * Consumer — это React-компонент, который подписывается на изменения контекста. В свою очередь, использование этого компонента позволяет вам подписаться на контекст в функциональном компоненте.

<MyContext.Consumer>

  {value => /\* отрендерить что-то, используя значение контекста \*/}

</MyContext.Consumer>

* + Consumer принимает функцию в качестве дочернего компонента. Эта функция принимает текущее значение контекста и возвращает React-компонент. Передаваемый аргумент value будет равен ближайшему (вверх по дереву) значению этого контекста, а именно пропу value компонента Provider.
  + Если такого компонента Provider не существует, аргумент value будет равен значению **defaultValue**, которое было передано в **createContext**().

1. **Для чего нужен Context.displayName?**
   * Объекту Context можно задать строковое свойство displayName. React DevTools использует это свойство при отображении контекста.
2. **Как использовать несколько контекстов одновременно?**
   * Чтобы последующие рендеры (связанные с контекстом) были быстрыми, React делает каждого потребителя контекста отдельным компонентом в дереве.

// Контекст UI-темы, со светлым значением по умолчанию

const ThemeContext = React.createContext('light');

// Контекст активного пользователя

const UserContext = React.createContext({

  name: 'Guest',

});

class App extends React.Component {

  render() {

    const { signedInUser, theme } = this.props;

    // Компонент App, который предоставляет начальные значения контекстов

    return (

      <ThemeContext.Provider value={theme}>

        <UserContext.Provider value={signedInUser}>

          <Layout />

        </UserContext.Provider>

      </ThemeContext.Provider>

    );

  }

}

function Layout() {

  return (

    <div>

      <Sidebar />

      <Content />

    </div>

  );

}

// Компонент, который может использовать несколько контекстов

function Content() {

  return (

    <ThemeContext.Consumer>

      {(theme) => (

        <UserContext.Consumer>

          {(user) => <ProfilePage user={user} theme={theme} />}

        </UserContext.Consumer>

      )}

    </ThemeContext.Consumer>

  );

}

**Хуки.**

1. **Что такое хуки?**
   * Хуки — это функции, с помощью которых вы можете «подцепиться» к состоянию и методам жизненного цикла React из функциональных компонентов. Хуки не работают внутри классов — они дают вам возможность использовать React без классов.
2. **Расскажите про useState() хук?**
   * Необходим, чтобы наделить функциональный компонент состоянием. React будет хранить это состояние между рендерами. Единственный аргумент useState — это начальное состояние. Хук состояния можно использовать в компоненте более одного раза.
3. **Расскажите про useEffect() хук?**
   * Хук эффекта даёт вам возможность выполнять побочные эффекты в функциональном компоненте. Позволяет получить доступ к таким методам жизненного цикла как componentDidMount(), componentDidUpdate(), componentWillUnmount(). Побочные эффекты могут быть со сбросом и не требующие сброса.
   * Иногда мы хотим выполнить дополнительный код после того, как React обновил DOM. Сетевые запросы, изменения DOM вручную, логирование — всё это примеры эффектов, которые не требуют сброса.
4. **Как с помощью useEffect() обратиться к каждому из методов жизненного цикла?**
   * Если депенденси аррей (dependency array) у useEffect() пуст то колбек, переданный в useEffect() вызовется только один раз, в componentDidMount().
   * Если в депенденси аррей добавлены какие зависимости то колбек, переданный в useEffect() будет вызываться каждый раз при их изменении. В данном случае предоставляется доступ к componentDidUpdate().
   * Из переданного в useEffect() колбека есть возможность возвращать что-то из колбека, например выполнить очистку, удалить слушатели событий и таким образом получить доступ к componentWillUnmount()
5. **В чем основное отличие в использовании useEffect и методов жизненного цикла?**
   * В отличие от componentDidMount или componentDidUpdate, эффекты, запланированные с помощью useEffect, не блокируют браузер при попытке обновить экран. Ваше приложение будет быстрее реагировать на действия пользователя, даже когда эффект ещё не закончился.
6. **Чем отличается useEffect() от useLayoutEffect()?**
   * Большинству эффектов не нужно работать в синхронном режиме. Есть редкие случаи, когда им всё же нужно это делать (например, измерять раскладку), но для этого мы разработали специальный хук useLayoutEffect с точно таким же API, как и у useEffect.
7. **Какие существуют советы при работе с useEffect()?**
   * используйте разные хуки для разных задач. Это даёт нам возможность разделять разную несвязанную между собой логику между разными эффектами. С помощью хуков мы можем разделить наш код основываясь на том, что он делает, а не по принципам методов жизненного цикла.
   * оптимизация производительности за счёт пропуска эффектов. Нужно использовать депенденси аррей, чтобы не запускать useEffect() при каждом рендере, так как это может вызвать проблему с производительностью.
8. **Можно ли использовать функции внутри useEffect(), описанные снаружи?**
   * Лучше всего описывать функции внутри useEffect.
   * Можно попробовать поместить функцию снаружи компонента. В таком случае она гарантированно не будет ссылаться на пропсы и состояние, так что её не требуется перечислять в списке зависимостей.
   * Если функция, которую вы вызываете, является чистым вычислением и её безопасно вызывать во время рендера, вы можете вызвать её снаружи эффекта, а не внутри, и сделать эффект зависящим от результата этого вызова, а не от самой функции.
   * В крайнем случае можете добавить саму функцию в список зависимостей эффекта, но обернув её определение в хук useCallback. Тогда функция будет оставаться неизменной, до тех пор, пока не изменятся её зависимости:

function ProductPage({ productId }) {

  // ✅ Оборачиваем в useCallback, чтобы избежать изменений при каждом рендере

  const fetchProduct = useCallback(() => {

    // ... Что-то делаем с productId ...

  }, [productId]); // ✅ Перечисляем все зависимости useCallback

  return <ProductDetails fetchProduct={fetchProduct} />;

}

function ProductDetails({ fetchProduct }) {

  useEffect(() => {

    fetchProduct();

  }, [fetchProduct]); // ✅ Перечисляем все зависимости useEffect

  // ...

}

1. **Где можно использовать хуки?**
   * В функциональных компонентах
   * В пользовательских хуках.
   * Следуя этому правилу, можно гарантировать, что вся логика состояния компонента чётко видна из исходного кода.
2. **Существуют ли дополнительные проверки правильности использования хуков?**
   * Да, ребята из React выпустили специальный плагин для линтера (ESLint) - eslint-plugin-react-hooks, который принуждает к соблюдению основных правил при написании хуков.
3. **Для чего нужны пользовательские хуки?**
   * Создание пользовательских хуков позволяет вам перенести логику компонентов в функции, которые можно повторно использовать.
   * Когда одинаковую логику состояния нужно повторно использовать в нескольких компонентах, в React традиционно применялись рендер-пропсы и компоненты высшего порядка. Рассмотрим, как хуки решают многие из тех же задач, не добавляя лишних компонентов в ваше дерево.
4. Напишите простой пользовательский хук.
   * В отличие от React-компонента, пользовательский хук не обязательно должен иметь конкретную сигнатуру. Мы можем решить, что он принимает в качестве аргументов, и должен ли он что-либо возвращать.

import { useState, useEffect } from 'react';

function useFriendStatus(friendID) {

  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {

    function handleStatusChange(status) {

      setIsOnline(status.isOnline);

    }

    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);

    return () => {

      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);

    };

  });

  return isOnline;

}

* + Цель нашего хука useFriendStatus — подписать нас на статус друга. Поэтому он принимает в качестве аргумента friendID и возвращает статус друга в сети:

function useFriendStatus(friendID) {

  const [isOnline, setIsOnline] = useState(null);

  // ...

  return isOnline;

}

1. **Напишите код, который использует ваш пользовательский хук.**
   * Вначале нашей целью было удалить дублированную логику из компонентов FriendStatus и FriendListItem. Они оба хотят знать, есть ли друг в сети.
   * Теперь, когда мы извлекли эту логику в хук useFriendStatus, мы можем его использовать:

function FriendStatus(props) {

  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {

    return 'Загрузка...';

  }

  return isOnline ? 'В сети' : 'Не в сети';

}

function FriendListItem(props) {

  const isOnline = useFriendStatus(props.friend.id);

  return (

    <li style={{ color: isOnline ? 'green' : 'black' }}>{props.friend.name}</li>

  );

}

* + Всё, что мы сделали, это извлекли общий код в отдельную функцию. Пользовательские хуки — это скорее соглашение, соответствующее дизайну хуков, нежели чем возможность самого React.

1. Можно ли обмениваться информаций между обычным и пользовательским хуком используя параметры?
   * Да, поскольку хуки являются функциями, мы можем передавать информацию между ними.

function ChatRecipientPicker() {

  const [recipientID, setRecipientID] = useState(1);

  const isRecipientOnline = useFriendStatus(recipientID);

  return (

    <>

      <Circle color={isRecipientOnline ? 'green' : 'red'} />

      <select

        value={recipientID}

        onChange={(e) => setRecipientID(Number(e.target.value))}

      >

        {friendList.map((friend) => (

          <option key={friend.id} value={friend.id}>

            {friend.name}

          </option>

        ))}

      </select>

    </>

  );

}

* + Мы сохраняем выбранный в настоящее время идентификатор друга в переменной состояния recipientID и обновляем его, если пользователь выбирает другого друга в <select>.

**memo, useMemo и useCallback**

1. **В чем разница между memo и useMemo?**
   * memo – это компонент высшего порядка, а useMemo – это хук возвращающий меморизированное значение функции. Оба нужны для повышения производительности. memo подходит для случаев, когда компонент рендерит одинаковый результат при одних и тех же значениях пропсов в этом случае результат будет меморизирован.
2. **Как работает и применяется memo?**
   * Если ваш компонент всегда рендерит одно и то же при неменяющихся пропсах, вы можете обернуть его в вызов React.memo для повышения производительности в некоторых случаях, мемоизируя тем самым результат. Это значит, что React будет использовать результат последнего рендера, избегая повторного рендеринга.

const MyComponent = React.memo(function MyComponent(props) {

  /\* рендер с использованием пропсов \*/

});

* + React.memo затрагивает только изменения пропсов. Если функциональный компонент обёрнут в React.memo и использует useState, useReducer или useContext, он будет повторно рендериться при изменении состояния или контекста.
  + По умолчанию он поверхностно сравнивает вложенные объекты в объекте props. Если вы хотите контролировать сравнение, вы можете передать свою функцию сравнения в качестве второго аргумента.

function MyComponent(props) {

  /\* рендер с использованием пропсов \*/

}

function areEqual(prevProps, nextProps) {

  /\*

  возвращает true, если nextProps рендерит

  тот же результат что и prevProps,

  иначе возвращает false

  \*/

}

export default React.memo(MyComponent, areEqual);

1. **Как работает и применяет useMemo?**
   * Возвращает меморизированное значение функции, которая делает некоторые затратные вычисления. Эта оптимизация позволяет избежать дорогостоящих вычислений при каждом рендере. useMemo будет повторно вычислять меморизированное значение только тогда, когда будет изменено значение зависимости. const isPrime = useMemo( () => checkPrime(num), [num] )
2. **Что такое useCallback хук (hook)?**
   * Это хук позволяющий меморизировать callback и вызывать его повторно только при изменении dependency array.
3. **Какие есть проблемы с рендеренгом в работе с контекст?**
   * Контекст использует сравнение по ссылкам, чтобы определить, когда запускать последующий рендер. Из-за этого существуют некоторые подводные камни, например, случайные повторные рендеры потребителей, при перерендере родителя Provider-компонента. При передаче в value объекта. он будет создаваться каждый раз. Один из вариантов решения этой проблемы — хранение этого объекта в состоянии родительского компонента:

class App extends React.Component {

  constructor(props) {

    super(props);

    this.state = {

      value: { something: 'something' },

    };

  }

  render() {

    return (

      <MyContext.Provider value={this.state.value}>

        <Toolbar />

      </MyContext.Provider>

    );

  }

}

**Предохранители (компоненты Error Boundary)**

1. **Что такое предохранители (компоненты Error Boundary)?**
   * это компоненты React, которые отлавливают ошибки JavaScript в любом месте деревьев их дочерних компонентов, сохраняют их в журнале ошибок и выводят запасной UI вместо рухнувшего дерева компонентов. Предохранители отлавливают ошибки при рендеринге, в методах жизненного цикла и конструкторах деревьев компонентов, расположенных под ними.
   * Предохранители работают как JavaScript-блоки catch {}, но только для компонентов.
   * Классовый компонент является предохранителем, если он включает хотя бы один из следующих методов жизненного цикла: static getDerivedStateFromError() или componentDidCatch()
   * Только классовые компоненты могут выступать в роли предохранителей
2. **В каких случаях** **предохранители (компоненты Error Boundary) не сработают(не поймают ошибки)?**
   * в обработчиках событий;
   * в асинхронном коде (например колбэках из setTimeout или requestAnimationFrame);
   * в серверном рендеринге (Server-side rendering);
   * в самом предохранителе (а не в его дочерних компонентах).
3. **Напишите предохранитель (компонент Error Boundary).**

export default class ErrorBoundary extends React.Component {

  constructor(props) {

    super(props);

    this.state = { hasError: false };

  }

  static getDerivedStateFromError(error) {

    // Обновить состояние с тем, чтобы следующий рендер показал запасной UI.

    return { hasError: true };

  }

  componentDidCatch(error, errorInfo) {

    // Можно также сохранить информацию об ошибке в соответствующую службу журнала ошибок

    logErrorToMyService(error, errorInfo);

  }

  render() {

    if (this.state.hasError) {

      // Можно отрендерить запасной UI произвольного вида

      return <h1>Что-то пошло не так.</h1>;

    }

    return this.props.children;

  }

}

1. **Где стоит размещать предохранитель (Error Boundary)?**
   * Степень охвата кода предохранителями остаётся на усмотрение разработчиков. Можно защитить им навигационные (route) компоненты верхнего уровня, чтобы выводить пользователю сообщение «Что-то пошло не так», как это часто делают при обработке ошибок серверные фреймворки. Или можно охватить индивидуальными предохранителями отдельные виджеты, чтобы помешать им уронить всё приложение.
   * Более детальный охват предохранителями обеспечит лучший опыт взаимодействия с приложением при возникновении ошибок.
2. **Что будет если ошибка не будет отловлена не одним из предохранителей (Error Boundary)?**
   * Начиная с React 16, ошибки, не отловленные ни одним из предохранителей, будут приводить к размонтированию всего дерева компонентов React.
   * Хотя принятие этого решения и вызвало споры, бо́льшим злом будет вывести некорректный UI, чем удалить его целиком. К примеру, в приложении типа Messenger, вывод поломанного UI может привести к тому, что пользователь отправит сообщение не тому адресату. Аналогично, будет хуже, если приложение для проведения платежей выведет пользователю неправильную сумму платежа, чем если оно не выведет вообще ничего.
3. **Как отловить ошибку в обработчике событий в React?**
   * Чтобы отловить ошибку в обработчике событий, пользуйтесь обычной JavaScript-конструкцией try / catch

handleClick() {

  try {

    // Делаем что-то, что сгенерирует ошибку

  } catch (error) {

    this.setState({ error });

  }

}

1. **Что такое unstable\_handleError?**
   * React 15 включал очень ограниченную поддержку предохранителей с другим названием метода: unstable\_handleError. Этот метод больше не работает и вам нужно будет заменить его на componentDidCatch в своем коде, начиная с первого бета-релиза React 16.

**Компоненты высшего порядка - HOC (high order components)\***

1. **Что такое ХОК (HOC) - (high order component)?**
   * Компонент высшего порядка (Higher-Order Component, HOC) — это один из продвинутых способов для повторного использования логики.
   * Компонент высшего порядка — **это функция, которая принимает компонент и возвращает новый компонент**. HOC ничего не меняет и не наследует поведение оборачиваемого компонента, вместо этого HOC оборачивает оригинальный компонент в контейнер посредством композиции.
   * HOC является чистой функцией без побочных эффектов.
   * Название функции начинается с маленькой буквы, потому что внутри них почти никогда не используется JSX напрямую.

const EnhancedComponent = higherOrderComponent(WrappedComponent);

1. **Как применяются ХОК (HOC) - (high order component)?**
   * Применения HOC: добавление дополнительной функциональности существующим компонентам; Передача поведения между несколькими компонентами. Загрузка данных. Повторное использование и инкапсуляция кода.
2. **В чем отличие между контейнерной компонентой и HOC?**
   * при помощи контейнеров мы обычно разделяем общую функциональность от частной. Например, в контейнере мы будем управлять внутренним состоянием или подпиской на внешние ресурсы, и через пропсы передавать данные в компоненты, ответственные за рендер UI. При реализации HOC мы тоже используем контейнеры. Можно сказать, что HOC — это инструмент для параметризированного создания контейнеров.
3. **Напишите простой пример HOC.**
   * Вот компонент более высокого порядка, который преобразует и возвращает имена пользователей в верхнем регистре:

const hoc = (WrappedComponent) => (props) => {

  return (

    <div>

      <WrappedComponent {...props}>

        {props.children.toUpperCase()}

      </WrappedComponent>

    </div>

  )

}

* + Этот компонент более высокого порядка принимает в качестве аргумента WrappedComponent. Затем он возвращает новый компонент с переданными ему реквизитами, создавая элемент React. Мы вызываем .toUpperCase() для props.children, чтобы преобразовать переданные props.children в верхний регистр.
  + Чтобы использовать этот компонент более высокого порядка, нам нужно создать компонент, который принимает реквизит и отображает дочерние компоненты.

const Username = (props) => (

  <div>{props.children}</div>

)

* + Затем мы оборачиваем Username в компонент более высокого порядка. Давайте сохраним это в переменной:

const UpperCaseUsername = hoc(Username)

1. **Что такое Partially applied components? Напишите простую Partially applied component.**
   * Partially applied components позволяют взять более базовые компоненты и путем передачи подмножества пропсов этих компонент создать более специфические компоненты. Следующий шаг в концепции композиций.

export const Button = (props) => {

  return (

    <>

      <button

        style={{

          backgroundColor: props.backgroundColor,

          width: '300px',

          minHeight: '200px',

        }}

      >

        Press

      </button>

    </>

  );

};

const partiallyApply = (Component, partialProps) => {

  return (props) => {

    return <Component {...props} {...partialProps} />;

  };

};

export const DangerButton = partiallyApply(Button, { backgroundColor: 'red' });

**React Design Patterns**

1. **Что такое Design Patterns в React?**
   * Это выработанные эффективные решения для популярных проблем, возникающих при разработке. Улучшают производительность и простоту поддержки кода.
2. **Какие** **Паттерны и подходы при разработке (design patterns) в React вы знаете?**
   * Создание reusable layouts components (использование styled-components) – компоненты не должны знать где они будут отображены. Часть стилей им можно передавать через пропсы (props).

const Styled = styled.div`

  flex: ${(props) => props.weight};

`;

* + Создание компонент оберток. Например, так называемые SplitScreens (композиций)
  + List and items. List определяет только логику общего для items контейнера. Уникальная стилистика, обработка и вывод данных определяется внутри ListItem.
  + Modals – создается базовое универсальное модальное окно состоящие из компонент фона и тела. Внутри него размещаются {children} передаваемые внутрь Modal в качестве пропсов.
  + Реиспользование Reusing сложной логики между компонентами – контейнерные компоненты (Container Components).
  + Всевозможное повсеместное использование ХОК (HOC) - (high order component), композиций.
  + Использование кастом хуков (custom hooks).
  + Работа с формами – controlled and uncontrolled components
  + Внедрение функциональных подходов.

1. **Что такое инверсия управления в React?**
   * может сделать код чище во многих случаях, уменьшая количество пропсов, которые нужно передавать через ваше приложение, и давая больше контроля корневым компонентам. Однако, такое решение не всегда подходит. Перемещая больше сложной логики вверх по дереву, вы перегружаете вышестоящие компоненты.
2. **Что такое SplitScreen?**
   * Это базовая компонент обертка (композиция – вставка), принимает в себя две равнозначные компоненты в качестве { children }. И, например занимаемые ими части экрана ( flex: 1 и flex: 3 )

// снаружи

<SplitScreen>

  <LeftChildren />

  <RightChildren />

</SplitScreen>;

// внутри

export const SplitScreen = ({ children }) => {

  const [left, right] = children;

  return <>{left}</>;

};

1. **Что такое компонента контейнер (Container Component)?**
   * Этокомпонент, который берет на себя загрузку данных и их последующее распределение между его {children} компонентами. Может использовать внутри useEffect с async функцией или принимать ее в виде пропсов. Возвращает

<>

  {React.Children.map(children, (child) => {

    if (React.isValidElement(child)) {

      return React.cloneElement(child, { [resourceName]: state });

    }

    return child;

  })}

</>;

1. **Что такое кастом хуков (custom hooks)?**
   * Кастом хуки (сustom рooks) – специальные хуки, которые комбинируют внутри себя стандартные хуки и способствуют повторному использованию кода, а также его инкапсуляции.
2. **Внедрение каких функциональных подходов оправдано и используется в React?**
   * Чистые функции (pure functions) - Каждый раз функция возвращает одинаковый результат, когда она вызывается с тем же набором аргументов, нет побочных эффектов. Это позволяет держать функции независимыми от внешних данных.
   * Функции как объекты первого класса – когда функции трактуются как объекты, т. е. могут быть переданы другим функциям и их можно вернуть из функций. В React функциональные компоненты functional components соответствуют этому правилу. HOC high order components - функциональный подход, описанный в концепции функциями первого класса.
   * Component compositions -это когда мы берем несколько разных компонент и объединяем их в одну. Своеобразный функциональный аналог и замена наследования в ООП

export const DangerButton = (props) => {

  return <Button {...props} color="red" />;

};

* + Partially applied components – позволяют взять более базовые компоненты и путем передачи подмножества пропсов этих компонент создать более специфические компоненты. Следующий шаг в концепции композиций, через использование HOC.

**Типизация. Статическая типизация в React. Flow. TypeScript. PropTypes.**

1. **Для чего нужна типизация и какие инструменты для этого вы знаете?**
   * Типизация позволяет позволяют отлавливать большую часть ошибок ещё до исполнения кода.
   * Типизация также улучшает читабельность и поддержку кода.
   * Существенно упрощает разработку предоставляя подсказки, автодополнение и другие возможности.
   * Для крупных приложений следует использовать TypeScript и Flow. Для небольших достаточно использовать PropTypes.
2. **Что вы можете сказать о Flow?**
   * Flow — это библиотека для статической типизации JavaScript, разработанная в Facebook и часто применяемая в связке с React. Flow расширяет возможности JavaScript, добавляя аннотации типов для переменных, функций и React-компонентов.
3. **Что вы можете сказать о Typescript?**
   * Строгая типизация уменьшает количество потенциальных ошибок, которые могли бы возникнуть при разработке на JavaScript.
   * TypeScript реализует многие концепции, которые свойствены объектно-ориентированным языкам, как, например, наследование, полиморфизм, инкапсуляция и модификаторы доступа
   * Потенциал TypeScript позволяет быстрее и проще писать большие сложные комплексные программы, соответственно их легче поддерживать, развивать, масштабировать и тестировать
4. **Что такое проп тайпс (PropTypes)?**
   * React предоставляет встроенные возможности для проверки типов. Для запуска этой проверки на пропсах компонента вам нужно использовать специальное свойство propTypes.
   * Проверка может быть применена и к функциональным компонентам, или к компонентам, созданным с помощью React.memo или React.forwardRef.

import PropTypes from 'prop-types';

class Greeting extends React.Component {

  render() {

    return (

      <h1>Привет, {this.props.name}</h1>

    );

  }

}

Greeting.propTypes = {

  name: PropTypes.string

};

* + По соображениям производительности propTypes проверяются только в режиме разработки.
  + Можно задать значение по умолчанию:

// Задание значений по умолчанию для пропсов:

Greeting.defaultProps = {

  name: 'Незнакомец',

};

**Routing**:

1. **Что в React принято использовать для роутинга на стороне фронт-енда?**
   * Для роутинга в React используется библиотека React Router. В ней доступны все нужные решения для роутинга в том числе приватный роутинг.
2. **Как начать работу с роутингом?**
   * Необходимо установить react-router-dom – актуальная версия – 6: $ npm install react-router-dom@6
   * Импортировать <BrowserRouter>, обернуть в него, например <App/> компонент и начать описывать роуты.
3. **Что такое Routes и Switch?** 
   * Служат для одного и того же – они группируют роуты, которые располагаются внутри них. Отличается только синтаксис внутри. Switch использовался в предыдущих версиях React Router. Сейчас используется Routes.
4. **Как описать роут?**
   * Нужно использовать импортируемый компонент **Route** <**Route** **path**='/' **element**={<**Component** />} /> - **path** - путь, в котором может быть **:id** и компонент, который будет отображен по этому пути.
5. **Для чего нужен Link? Чем он отличается от NavLink?**
   * <**Link** to='/' > import { **Link** } from **'react-router-dom'**; - используется для перехода с одного роута на другой.
   * <**NavLink** **to**="/" **activeClassName**="**active**" /> - специальная версия <**Link** /> которая позволяет передавать активный класс, он сработает в случае совпадения **URL** с <**NavLink** **to**="/">
6. **Что такое Outlet и для чего он нужен?**
   * <**Outlet**> - для отображение вложенного Route внутри другого Route. Создает как бы страницу в странице и отображает ее только при совпадении URL. В роутинге создаем вложенный Route:

<Route path="/со-слешем" element={<Element />}>

  <Route path="без-слеша" element={<Element />} />

</Route>

* + Внутри компоненты указанной в родительском Route вставляем Outlet - в этом месте отобразится компонента, переданная во вложенный <Route />.

import { Outlet } from 'react-router-dom';

<div>

  <Outlet />

</div>;

1. **Для чего используется хук useHistory()?**
   * useHistory() – предоставляет доступ к объекту history получая доступ к его методам и свойства.
   * let history = useHistory();
   * history.push(новый-адрес),
   * history.go(n),
   * history.length,
   * history.location.pathname,
   * history.location.hesh - the hash key(#) also known as anchor
2. **Для чего нужен редирект?**
   * <Redirect to='/путь-со-слешем' > - используется рядом с Route для редиректов с одного URL на другой.
3. **Для чего используется хук useParams()?**
   * useParams() хук позволяет получать данные из URL, которые находятся после “/:" /:данные
   * const { id } = useParams();

**React performance**

1. **Как повлиять на производительность React приложения?**
   * Если вдруг по какой-то причине React подгружается через теги скриптов, перейти на любой из сборщиков.
   * Использовать минифицированную продакшн сборку (production build). Команда npm run build.
   * Использовать разделение кода - Suspense в связке с lazy().
   * Использовать memo, а также хуки useCallback и useMemo в функциональных компонентах.
   * Использовать React.PureComponent или метод shouldComponentUpdate() в классовых компонентах.
   * Использовать виртуализацию длинных списков: react-window и react-virtualized — это популярные библиотеки для оконного доступа. Они предоставляют несколько повторно используемых компонентов для отображения списков.
   * Использовать SSR Server-Side Rendering.
   * Стоит избегать мутирования значений объектов в логике и всегда возвращать новый объект. Для этого использовать методы, возвращающие новый объект, например map() или операторы типа spread ( … ).
   * Стоит попробовать воспользоваться инструментом разработки Profiler в Chrome dev tools для анализа производительности компонентов.
2. **Что такое Pure Component?**
   * Компонент является чистым, если он гарантированно возвращает один и тот же результат при одинаковых пропсах и состоянии. Чистые компоненты имеют лучшую производительность за счет поверхностного сравнения пропсов и стейта. Для классового компонента есть метод **shouldComponentUpdate**() – это необязательный метод и если он вернет false, то React пропустит рендеринг компонента. Может содержать любую логику сравнения пропсов и стейта. Также есть classPureComponent он может быть использован (вместо обычного в связке shouldComponentUpdate)**.** Вместо этих штук в функциональных компонентах используются memo.
3. **Напишите классовую компоненту с реализованным методом shouldComponentUpdate().**
   * Если единственный случай изменения вашего компонента — это когда переменная props.color или state.count изменяются, вы могли бы выполнить проверку в shouldComponentUpdate следующим образом:

class CounterButton extends React.Component {

  constructor(props) {

    super(props);

    this.state = {count: 1};

  }

  shouldComponentUpdate(nextProps, nextState) {

    if (this.props.color !== nextProps.color) {

      return true;

    }

    if (this.state.count !== nextState.count) {

      return true;

    }

    return false;

  }

  render() {

    return (

      <button

        color={this.props.color}

        onClick={() => this.setState(state => ({count: state.count + 1}))}>

        Счётчик: {this.state.count}

      </button>

    );

  }

}

* + В этом коде shouldComponentUpdate — это простая проверка на наличие каких-либо изменений в props.color или state.count. Если эти значения не изменяются, то компонент не обновляется.

1. **Напишите пример чистой компоненты используя React.PureComponent.**
   * Если ваш компонент стал более сложным, вы можете использовать аналогичный паттерн «поверхностного сравнения» между всеми полями props и state, чтобы определить должен ли обновиться компонент. Этот механизм достаточно распространён, поэтому React предоставляет вспомогательную функцию для работы с ним — просто наследуйтесь от React.PureComponent.

class CounterButton extends React.PureComponent {

  constructor(props) {

    super(props);

    this.state = { count: 1 };

  }

  render() {

    return (

      <button

        color={this.props.color}

        onClick={() => this.setState((state) => ({ count: state.count + 1 }))}

      >

        Счётчик: {this.state.count}

      </button>

    );

  }

}

* + В большинстве случаев вы можете использовать React.PureComponent вместо написания собственного shouldComponentUpdate. Но он делает только поверхностное сравнение, поэтому его нельзя использовать, если пропсы и состояние могут измениться таким образом, который не сможет быть обнаружен при поверхностном сравнении.

1. **Как работает Profiler?**
   * Profiler измеряет то, как часто рендерится React-приложение и какова «стоимость» этого. Его задача — помочь найти медленные части приложения, которые можно оптимизировать (например, через мемоизацию).
2. **Как использовать Profiler?**
   * Profiler может быть добавлен в любую часть React-дерева для измерения стоимости рендеринга этой части. Он принимает два пропа: id (string) и колбэк onRender (function), который React вызывает каждый раз, когда компонент внутри дерева «фиксирует» обновление.
   * Например, так выглядит процесс профилирования компонента Navigation и его дочерних компонентов:

render(

  <App>

    <Profiler id="Navigation" onRender={callback}>

      <Navigation {...props} />

    </Profiler>

    <Main {...props} />

  </App>

);

1. **Для чего нужен атрибут onRender у Profiler?**
   * Profiler принимает функцию onRender в качестве пропа. React вызывает эту функцию каждый раз, когда компонент внутри профилируемого дерева «фиксирует» изменение. Эта функция принимает параметры, которые описывают, что было отрендерено и сколько времени это заняло.

**Server-Side Rendering (SSR)**

1. **Что такое Server-Side Rendering (SSR)?**
   * Нормальное поведение обычного React приложения – клиент делает запрос на сервер и загружает index.html который чаще всего не информативен и содержит только базовую html структуру. По сути, он пуст для пользователя. Далее этот файл указывает браузеру загрузить React scripts, которые уже запустят рендеринг всех html элементов на странице. Однако с Server-Side renderingserver – тот, кто заботится о запуске скриптов и рендеринге всех html элементов.
2. **Чем отличает** **Client-Side Rendering от Client-Side Rendering?**

|  |  |
| --- | --- |
| **Client-Side Rendering** | **Client-Side Rendering** |
| Renders app to HTML in the user’s browser | Renders app to HTML on the server и просто шлет законченный HTML документ клиенту |
| **Процесс:**  Loads index.html from server,  loads JavaScript bundle from server,  runs bundle,  display app,  loads data | **Процесс:**  runs JavaScript bundle,  loads data (much easier because we are on the server)  creates HTML document,  send it to client side |
| **Pluses:**  Less нагрузка на server | **Pluses:**  Faster user experience  Better for SEO |
| **Minuses:**  generally slower user experience | **Minuses:**  More strain on the server |

1. **Для чего нужен метод renderToString из библиотеки 'react-dom/server'?**
   * Данный метод позволяет преобразовать React компоненту к строке, чтобы в последующем передать ее в ответ (response) пользователю с остальным html.
2. **Напишите самый базовый пример Server-Side Rendering (SSR)?**
   * Basic Server-Side rendering express example:

// Express

import express from 'express';

// React

import React from 'react';

import { renderToString } from 'react-dom/server';

// Components

import { Home } from './src/pages/Home';

const app = express();

const app = express();

app.use(express.static('./build', { index: false })); // Staticky serve the files inside build folder

// but don’t load base index.html by default

app.get('/\*', (req, res) => {

const reactApp = renderToString(<Home />);

return res.send(`

<html>

<body>

<div id="root">${reactApp}</div>

</body>

</html>

`);

});

app.listen(8080, console.log('Server is listening on port 8080'));

1. **Для чего нужен метод ReactDOM.hydrate()?**
   * ReactDOM.hydrate вместо ReactDOM.render внутри index.js позволит добавлять React в получаемые с сервера HTML. Hydrate takes pre-rendered HTML that we're going to get from the server and adds React to it. Despite of we're getting an HTML from the server, we still want to behave and rerender it like ReactApp.
2. **Для чего используется StaticRouter компонент?**
   * StaticRouter компонент, импортируемый из 'react-router-dom' используется на сервере при работе с SSR. В данный компонент на сервере оборачивается <App /> зеркальным образом, как и на фронте.
   * Также необходимо передать ему атрибут location из реквеста (request)
3. **Напишите полноценный пример Server-Side Rendering (SSR).**
   * **Express**. Use **npx nodemon --exec npx babel-node server.js** command to run the **server** with **nodemon** and **babel**.
   * Также необходимо перенести <BrowserRouter> </BrowserRouter> обертку из App в index.js для нормальной работы приложения с SSR:

import React from 'react';

import ReactDOM from 'react-dom';

import { BrowserRouter } from 'react-router-dom';

import App from './App';

// hydrate takes pre-rendered HTML that we're going to get from the server and adds React to it

// Despite of we're getting an HTML from the server, we still want to behave and rerender like ReactApp

ReactDOM.hydrate(

  <React.StrictMode>

    <BrowserRouter>

      <App />

    </BrowserRouter>

  </React.StrictMode>,

  document.getElementById('root')

);

* + Worked example of SSR server.js file:

/ Express

import express from 'express';

import path from 'path';

import fs from 'fs';

// React

import React from 'react';

import { renderToString } from 'react-dom/server';

import { StaticRouter } from 'react-router-dom';

// Components

import App from './src/App';

const app = express();

// Staticky serve the files inside build folder but don’t load base index.html by default

app.use(express.static('./build', { index: false }));

app.get('/\*', (req, res) => {

  const reactApp = renderToString(

    <StaticRouter location={req.url}>

      <App />

    </StaticRouter>

  );

  const templateFile = path.resolve('./build/index.html');

  fs.readFile(templateFile, 'utf8', (err, data) => {

    if (err) {

      return res.status(500).send(err);

    }

    return res.send(

      data.replace('<div id="root"></div>', `<div id="root">${reactApp}</div>`)

    );

  });

});

app.listen(8080, () => {

  console.log('Server is listening on port 8080');

});

* + styled components don’t work with SSR. We need to add addition code to the server to make it work. And also, don’t forget to add {{ styles }} to index.html для успешной замены: .replace('{{ styles }}', sheet.getStyleTags()).

// Styles

import { ServerStyleSheet } from 'styled-components';

…

app.get('/\*', (req, res) => {

  const sheet = new ServerStyleSheet();

  const reactApp = renderToString(

    sheet.collectStyles(

      <StaticRouter location={req.url}>

        <App />

      </StaticRouter>

    )

  );

…

return res.send(

      data

        .replace('<div id="root"></div>', `<div id="root">${reactApp}</div>`)

.replace('{{ styles }}', sheet.getStyleTags())

    );

…

* + Следует также помнить, что **window** и **document** не будут работать при **SSR**.

**Порталы (Portals). Веб-компоненты.**

1. **Что такое портал?**
   * Порталы позволяют рендерить дочерние элементы в DOM-узел, который находится вне DOM-иерархии родительского компонента.

render() {

  // React \*не\* создаёт новый div. Он рендерит дочерние элементы в `domNode`.

  // `domNode` — это любой валидный DOM-узел, находящийся в любом месте в DOM.

  return ReactDOM.createPortal(

    this.props.children,

    domNode

  );

}

1. **Может ли React быть использован с веб-компонентами?**
   * Можете использовать React в своих веб-компонентах, или использовать веб-компоненты в React, или и то, и другое.
   * Чтобы получить доступ к необходимому API веб-компонентов, необходимо использовать реф для взаимодействия с DOM-узлом напрямую. Если вы используете сторонние веб-компоненты, лучшим решением будет создать React-компонент и использовать его как обёртку для веб-компонента.
   * События, созданные веб-компонентами, могут неправильно распространяться через дерево React-компонентов. Вам нужно вручную добавить обработчики для таких событий в собственные React-компоненты.
2. **Что следует помнить при интеграции React с плагинами, изменяющими DOM?**
   * React не знает про изменения DOM, которые сделаны вне React. Он определяет обновления на основе своего внутреннего представления, и если одни и те же DOM-узлы управляются другими библиотеками, то это нарушает работу React без возможности её восстановления.

ReactDOM.createPortal(child, container);

* + Первый аргумент (child) — это любой React-компонент, который может быть отрендерен, такой как элемент, строка или фрагмент. Следующий аргумент (container) — это DOM-элемент.

render() {

  // React \*не\* создаёт новый div. Он рендерит дочерние элементы в `domNode`.

  // `domNode` — это любой валидный DOM-узел, находящийся в любом месте в DOM.

  return ReactDOM.createPortal(

    this.props.children,

    domNode

  );

}

1. **Как применяются порталы?**
   * Обычно, когда вы возвращаете элемент из рендер-метода компонента, он монтируется в DOM как дочерний элемент ближайшего родительского узла. Но иногда требуется поместить потомка в другое место в DOM
   * Типовой случай применения порталов — когда в родительском компоненте заданы стили overflow: hidden или z-index, но вам нужно чтобы дочерний элемент визуально выходил за рамки своего контейнера.
   * Например, диалоги, всплывающие карточки и всплывающие подсказки.
2. **Как ведет себя контекст и всплытия событий с порталами?**
   * Как уже было сказано, портал может находиться в любом месте DOM-дерева. Несмотря на это, во всех других аспектах он ведёт себя как обычный React-компонент. Такие возможности, как контекст, работают привычным образом, даже если потомок является порталом, поскольку сам портал всё ещё **находится в React-дереве**, несмотря на его расположение в DOM-дереве.
   * Так же работает и всплытие событий. Событие, сгенерированное изнутри портала, будет распространяться к родителям **в содержащем React-дереве**, даже если эти элементы не являются родительскими в DOM-дереве.

**Стейт менеджмент (State Management)**

1. **Какие подходы для стейт менеджмента вы посоветуете в зависимости от размера стейта приложения?**
   * small state – useState, Context
   * medium state – Recoil
   * Large state – Redux, MobX

**Flux**

1. **Что такое Flux?**
   * **Flux** – архитектурный **паттерн, не библиотека** обеспечивающий однонаправленный поток данных – данные всегда исходят from **view** to the **modal**. Сам по себе, Flux — это схема обработки данных в вашем приложении.
2. **Ключевая концепция Flux?**
   * **однонаправленный поток данных**

Graphical user interface, diagram, text

Description automatically generated

1. **Что такое создатель действия (action creator) в Flux?**
   * Судя по названию, он отвечает за создание действий, т. е. “пути”, по которому должны пройти все изменения и взаимодействия. Когда бы вы ни захотели изменить состояние приложения или изменить представление данных, вы должны вызвать действие.
   * Он создает действие с типом и полезной нагрузкой. Тип — один из тех, которые вы определили в вашей системе (обычно — список констант). Пример такого действия будет чем-то вроде MESSAGE\_CREATE или MESSAGE\_READ.
2. **Что такое диспетчер в Flux?**
   * Диспетчер, по сути, — это реестр обратных вызовов. Он хранит список всех хранилищ, которым нужно отправлять действия. Как только действие приходит от его создателя, диспетчер отправляет его по разным хранилищам.
   * Если вам нужно установить зависимости между хранилищами (чтобы одно обновлялось раньше другого), вы можете указать диспетчеру управлять ими с помощью waitFor().
3. **Чем диспетчер Flux отличается от других архитектур?**
   * Действие отправляется во все зарегистрированные хранилища независимо от типа действия. Это значит, что хранилище не просто подписывается на действие, оно “слышит” обо всех действиях и фильтрует, о чем стоит позаботиться, а о чем — нет.
4. **Что такое хранилище в Flux?**
   * Оно содержит все состояния приложения, а также всю логику изменений состояния. Все изменения состояний должны быть сделаны им лично. И вы не можете напрямую запросить, чтобы состояние изменилось.
   * Чтобы запросить изменение состояния, вы должны пройти надлежащую процедуру… т. е. подтвердить действие через связку “создатель действия/диспетчер”.
   * Как было упомянуто выше, если хранилище зарегистрировано вместе с диспетчером, то ему будут отправлены все действия. Внутри хранилища обычно находится переключатель состояния, который анализирует тип действия и решает, нужно ли хранилищу реагировать на него. Если хранилище должно отреагировать на действие, оно поймет, какие именно изменения нужно сделать, основываясь на этом действии, и обновит состояние.
   * Как только хранилище сделало изменение состояния, оно вызовет событие изменения. Это уведомит контроллер о том, что состояние было изменено.
5. **Что такое контроллер и представление (view) в FLUX?**
   * Представления(view) в ответе за получение состояния и отображение его для пользователя так же, как и за принятие пользовательских данных.
   * Представление — это докладчик. Он не беспокоится ни о чем в приложении, он просто знает, что есть поступившие ему данные, и как их форматировать для понятного человеку вывода (используя HTML)
   * Контроллер похож на менеджера среднего звена между хранилищем и представлением. Хранилище говорит ему, когда состояния изменилось. Контроллер собирает новое состояние и затем отправляет обновленное состояние по всем связанным представлениям.
6. **Как взаимодействую между собой** **создатель действия (action creator), диспетчер, хранилище, контроллер и представление?**
   * **Настройка**.
   * Хранилища дают диспетчеру знать, что они хотят быть уведомлены о произошедшем действии.
   * Затем контроллер запрашивает у хранилища последнее состояние.
   * После того, как хранилища передают состояние контроллеру, они также передают его своим дочерним представлениям для отображения.
   * Контроллер просит хранилища держать его в курсе, когда состояние изменяется.
   * **Действие**.
   * Как только настройка окончена, приложение готово принимать пользовательские данные. Так что давайте вызовем действие с помощью пользователя, сделавшего изменение.
   * Мы запустим поток данных через действие пользователя.
   * Представление говорит создателю действия приготовить его.
   * Создатель действия форматирует его и отправляет диспетчеру.
   * Диспетчер отправляет действие хранилищам последовательно. Каждое хранилище уведомляется обо всех действиях. Потом оно решает, иметь с ним дело или нет, и соответственно изменяет состояние.
   * Как только состояние изменилось, хранилище дает об этом знать всем подписанным на него контроллерам.
   * Эти контроллеры затем запросят у хранилища обновленное состояние.
   * После того как хранилище отдает состояние, контроллер передаст его дочерним представлениям запрос на повторное отображение согласно новому состоянию.

**Redux**

1. **Что такое Redux?**
   * Это библиотека для state менеджмента, используется для контроля состояния всего JS приложения основана на архитектуре Flux.
2. **Назовите три основных принципа Redux?**
   * Единственный источник правды - состояние всего вашего приложения сохранено в дереве объектов внутри одного стора.
   * Состояние только для чтения - единственный способ изменить состояние — это применить экшен — объект, который описывает, что случится.
   * Мутации (reducers) написаны как чисты функции - для определения того, как дерево состояния будет трансформировано экшенами, вы пишете чистые редюсеры.
   * Все состояние вашего приложения сохранено в объекте внутри одного стора (**store**). Единственный способ изменить дерево состояния — это вызвать экшена (**action**)\* — объект, описывающий то, что случилось. Чтобы указать, каким образом экшены преобразовывают дерево состояния, вы пишете чистые редюсеры (**reducers**).
3. **Какого главное отличие Redux от Flux?**
   * Redux не имеет Диспетчера (Dispatcher) или поддержки множества сторов. Вместо этого есть только один стор с одной корневой функцией-редюсером. Когда ваше приложение разрастется, вместо добавления сторов, вы разделяете корневой редюсер на более мелкие редюсеры, которые независимо друг от друга обслуживают разные части дерева состояния. Это аналогично тому, что в React приложении есть только один корневой компонент, состоящий из множества мелких компонентов
   * Другим важным отличием от Flux является то, что Redux предполагает, что вы никогда не изменяете ваши данные напрямую. Вы запросто можете использовать простые объекты и массивы для состояния, но строго рекомендуется изменять их через редюсеры.
4. **Что такое Экшены (actions) в Redux?**
   * Экшены — это структуры, которые передают данные из вашего приложения в стор. Они являются единственными источниками информации для стора. Вы отправляете их в стор, используя метод store.dispatch().
   * Например, вот экшен, которое представляет добавление нового пункта в список дел:

const ADD\_TODO = 'ADD\_TODO'

{

  type: ADD\_TODO,

  text: 'Build my first Redux app'

}

* + Экшены — это обычные JavaScript-объекты. Экшены должны иметь поле type, которое указывает на тип исполняемого экшена. Типы должны быть, как правило, заданы, как строковые константы. После того, как ваше приложение разрастется, вы можете захотеть переместить их в отдельный модуль.

import { ADD\_TODO, REMOVE\_TODO } from '../actionTypes';

1. **Что такое Генераторы экшенов (Action Creators)?**
   * Генераторы экшенов (Action Creators) — не что иное, как функции, которые создают экшены. Довольно просто путать термины “action” и “action creator,” поэтому постарайтесь использовать правильный термин.
   * В Redux генераторы экшенов (action creators) просто возвращают action:

function addTodo(text) {

  return {

    type: ADD\_TODO,

    text,

  };

}

* + Это делает их более переносимыми и легкими для тестирования.

1. **Чем отличаются генераторы экшенов (action creators) в Redux от Flux?**
   * В традиционной реализации Flux, генераторы экшенов (action creators) при выполнении часто вызывают dispatch, примерно так:

function addTodoWithDispatch(text) {

  const action = {

    type: ADD\_TODO,

    text,

  };

  dispatch(action);

}

* + В Redux это не так. Вместо того чтобы на самом деле начать отправку, передайте результат в функцию dispatch():

dispatch(addTodo(text));

dispatch(completeTodo(index));

1. **Напишите простой файл с actions.**
   * Типичный файл actions.js может выглядеть так:

// типы экшенов (могут храниться в отдельном файле, если их много.)

export const ADD\_TODO = 'ADD\_TODO';

export const TOGGLE\_TODO = 'TOGGLE\_TODO';

export const SET\_VISIBILITY\_FILTER = 'SET\_VISIBILITY\_FILTER';

// генераторы экшенов

export function addTodo(text) {

  return { type: ADD\_TODO, text };

}

export function toggleTodo(index) {

  return { type: TOGGLE\_TODO, index };

}

export function setVisibilityFilter(filter) {

  return { type: SET\_VISIBILITY\_FILTER, filter };

}

1. **Как получить доступ к функции dispatch(), чтобы передать в нее action creator?**
   * Доступ к функции dispatch() может быть получен непосредственно из стора (store) store.dispatch(), но, что более вероятно, вы будете получать доступ к ней при помощи чего-то типа connect() из react-redux.
2. **Что такое reducer в Redux?**
   * Редьюсеры определяют, как состояние приложения изменяется в ответ на экшены, отправленные в стор. Помните, что экшены только описывают, что произошло, но не описывают, как изменяется состояние приложения.
   * это чистые функции, которые берут предыдущее состояние и экшен и возвращают новое состояние. Чистая функция в формате (previousState, action) => newState. Он описывает то, как экшен преобразовывает состояние в следующее состояние. Нельзя изменять объект состояния напрямую, а нужно возвращать новый объект, если состояние изменилось.
3. **Что нельзя делать в редьюсере?**
   * Непосредственно изменять то, что пришло в аргументах функции;
   * Выполнять какие-либо сайд-эффекты: обращаться к API или осуществлять переход по роутам;
   * Вызывать не чистые функции, например Date.now() или Math.random().
4. **Какой формат может быть у state в Redux?**
   * Формат состояния зависит от вас: это может быть примитивом, массивом, объектом или даже структурой данных Immutable.js.
5. **Что такое Immutable?**
   * Immutable — это JavaScript-библиотека, позволяющая использовать неизменяемые структуры данных. Immutable и большинство похожих библиотек ортогональны Redux. Не стесняйтесь использовать их вместе.
   * Для Redux не важно, как вы храните состояние — простым объектом, объектом Immutable или как-то еще.
6. **Напишите простой редюсер (reducer).**
   * Чаще всего reducer представляет из switch – case обернутый в чистую функцию, принимающую state и action.

function todoApp(state = initialState, action) {

  switch (action.type) {

    case SET\_VISIBILITY\_FILTER:

      return Object.assign({}, state, {

        visibilityFilter: action.filter

      })

    default:

      return state

  }

}

1. **Что такое Redux стор (store) и как его создать?**
   * Redux store – это объект, который хранит состояние всего приложения. Единый источник истины. Его API - { subscribe, dispatch, getState }.
   * Создание стора:

import { createStore } from 'redux';

let store = createStore(counter);

1. **Что вы знаете об экосистеме Redux?**
   * Redux — это небольшая библиотека, в которой соглашения и API были тщательно продуманы для удобного создания инструментов и подключения расширений экосистемы.
   * **Адаптеры (Bindings):**
   * react-redux — React
   * ng-redux — Angular
   * **Промежуточное ПО (Middleware)**
   * redux-thunk — Самый легкий способ создавать асинхронные генераторы экшенов (action creators)
   * redux-saga — An alternative side effect model for Redux apps и др.
   * **Маршрутизация (Routing)**
   * react-router-redux — Ruthlessly simple bindings to keep React Router and Redux in sync
   * redial — Universal data fetching and route lifecycle management for React that works great with Redux
   * **Компоненты (Components)**
   * redux-form — Поддержка Redux-состояний (stores) для html-форм, использующихся в React
   * **Утилиты (Utilities)**
   * reselect — Простая библиотека "селекторов", нашедшая вдохновение в геттерах NuclearJS
   * **Инструменты разработчика (Developer Tools)**
   * Redux DevTools — Логирование экшенов с UI для путешествий во времени, горячая перезагрузка и обработка ошибок для редьюсеров, впервые представлено на React Europe
   * **И это только малая часть. Вся экосистема есть в русской документации на гитхабе.**

**styled-components**

1. **Что такое styled-components?**
   * Это библиотека для удобной стилизации при создании компонент. Все стили можно напрямую писать в JavaScript в обратных кавычках и создавать таким образом компоненты со стилями. Их впоследствии можно наследовать использовать в коде. Также доступна JavaScript логика внутри описания стилей.
2. **Как использовать пропсы (props) внутри styled-components?**
   * Нужно использовать колбек внутри стилей:

const Styled = styled.div`

  flex: ${(props) => props.weight};

`;

* + Состояние хранится в едином **state** – дереве объектов. State предоставляет методы для доступа к состоянию **mapStateToProps**(**state**) => data: selector(state) где selector это указание к полю объекта в state. И методы к его изменению **mapDispatchToProps**(**dispatch**) => **dispatch**(**actionCreator**(**payload**)). **States** управляемых **reducer** может быть несколько, они объединяются при помощи **combineReducers**(). При помощи **configureStore** () можно подключить **persistedReducer**() – который предотвращает **refresh** данных в **store** при обновлении страницы, **composeWithDevTools** – который позволяет использовать **Chrome extensions** и визуально наблюдать работу Redux приложения в браузере, и **applyMiddleware**(**thunk**) который позволяет подключать дополнительные библиотеки для работы с асинхронными запросами, такие как **thunk**, **saga** и др.
  + **Пример.** Если изменить что-то, например в **user profile** в **React** вернется **копия всего старого state** с новыми изменениями не просто модификация. Это может показаться чрезмерным, но есть серьезные преимущества в виде **redo functionality** и **debagging**. Позволяет снизить сложность средних и крупных приложений.
  + **export const configureStore = () => createStore(persistedReducer, composeWithDevTools(applyMiddleware(thunk)));**
  + **Преимущество** **Redux:**
    - **Разделение ответственности**. Предсказуемость результатов (**одно хранилище** – **один источник данных**). Проще в обслуживании при укрупнении приложения – **понятная** **структура**. **Инструменты** **разработчика**. **Тестируемость**.
    - У **Redux** **стандартизирован подход к управлению state**. **State** всегда организован по одному и тому же **pattern**, это означает, что если ты знаешь **Redux** ты легко сможешь перепрыгнуть в абсолютно **новый код** и со старта понимать как происходят **манипуляции** с **данными**.
  + **Недостатки** **Redux:**
    - Каждый раз при вызове **action** создается новый state – это может быть дорогостояще.
* **Immutable.JS** – популярная библиотека которая помогает улучшить производительность. Но у нее, как и у всего есть свои недостатки – нельзя совмещать immutable objects (которые создаются с помощью библиотеки) с обычными JS объектами. Есть пара методов, которые предоставляют взаимодействие, но в обмен на производительность.
* <**Provider** **store**={**store**}> оборачивает <**App** /> компонент для подключения **Redux** **store** к приложению.

**Selectors**

* **Selectors – функции,** хранящиеся в отдельном файле (для удобства), принимают **state как аргумент** и **возвращают** **state.someValue или state**.**someReducer**.**someValue**. Могут быть использованы только в **useSelector(selectorName)** хуке.
  + **Если есть Selectors,** то **useSelector(selectorName) –** импортируемый из react-redux хук для получения необходимых кверей из store.
  + **Если нет Selectors,** то **useSelector**((**state**) => **state**.**someReducer**.**someValue**) – импортируемый из react-redux хук для получения необходимых кверей из store.
* **reselect** – удобная библиотека для создания **selector. import { createSelector } from 'reselect'; альтернатива useSelector() хуку:**
  + **export const getTodos = (state) => state.todos.data;**
  + **export const getIncompleteTodos = createSelector(getTodos, (todos) => todos.filter((todo) => !todo.isCompleted));**
* **useDispatch** – импортируемый из **react-redux** хук **const** **dispatch** = **useDispatch**(); используется для вызова **dispatch** и передачи в него **action-creators**.
* **connect** – функция, из **react-redux**, позволяющая подключить компонент к **Redux**.
  + **export default connect**(**mapStateToProps**, **mapDispatchToProps**)(**Component**);
* **mapStateToProps –** is maps a redux state to props that passed down to the component that it’s connected to.
* **middleware –** структура вызывающая **third-party endpoints,** которые случаются когда мы вызываем **action** и перед тем как **action** достигнет **reducer.** Обычно используются для logging, calling an API, загрузки данных и так далее.
  + **пример middleware. Это прежде всего функция:**
    - **const middleware = store => next => action => { do something } For example:**

export const logger = (store) => (next) => (action) => {

  console.log('dispatching', action);

  const result = next(action);

  console.log('next state', store.getState());

  return result;

};

* + - * **store – Redux store**
      * **next function** говорит **redux** продолжитьобработкуследующего **middleware**
      * **action –** последнийаргумент**,** которыйбудетотправленв **reducer**

Во время исполнения **middleware task ты вызовешь next() on the action** что определяет будет ли вызван **middleware** или приложение отправится в **reducer**

**MobX**

* **MobX** - обращается к тем, у кого есть знания в **ООП**. Использует **observer** **patten**, что позволяет системе cтроить **dependency tree** среди разных частей state и позволяет производить свободный рендеринг компонент отслеживая что должно быть мутировано. Дает возможность to manage state, но все равно нужно заниматься организацией кода в форме **stores** и **controllers**. **Store** представляет собой не **JSON объект а spreadsheet with derivations**. Использует концепты **Derivations** and **Reactions**.
  + **Derivation –** любое значение, вычисляемое автоматическииз **state.** Чтобы вызвать **Derivation,** нужноиспользовать **decorator. Decorator** позволяет быть добавленным к объекту без изменения других объектов.
  + **Плюсы MobX**:
    - нет привязки к организационной структуре использования actions and reducers, что дает flexibility выбирать разные архитектурные подходы.
    - можно мутировать store. **Immutability isn’t the preferred practice for using MobX.**
* **Apollo Client** – еще одна альтернатива **Redux**. Удобно при использовании в связке с **GraphQL**. Преимуществом **GraphQL** – когда мы делаем запрос на сервер to get data мы можем вызвать их в той форме, в которой хотим их получить. Можем выбрать необходимые данные из объекта и их форму.

**React recoil**

* **React recoil** – **React** library for **medium state** management. **Recoil** defines a **directed graph orthogonal to** but also **intrinsic** and **attached** to your **React** **tree**. **State** changes flow from the **roots** of this **graph** (which we call **atoms**) through **pure** **functions** (which we call **selectors**) and into components.

Использование:

1. Установить библиотеку: **npm install recoil**
2. Импортировать **RecoilRoot** и обернуть компоненту в **<RecoilRoot></RecoilRoot>**

import { RecoilRoot } from 'recoil';

import { CounterButton } from './CounterButton';

import './App.css';

const App = () => {

  return (

<RecoilRoot>

      <h1>State Management Example</h1>

      <CounterButton />

    </RecoilRoot>

  );

};

1. Создать recoil state/states с помощью atom.

import { atom } from 'recoil';

export const counterState = atom({

key: 'counterState',

default: 0,

});

import { atom } from 'recoil';

export const incrementByState = atom({

  key: 'incrementByState',

  default: 1,

});

**key** – string that Recoil uses to store this state behind the scenes. **default** – default value for state.

1. Использовать внутри компоненты. Что-среднее между **useState** and **useContext**.

import { useRecoilState } from 'recoil';

import { counterState } from './counterState';

import { incrementByState } from './incrementByState';

export const CounterButton = () => {

  const [numberOfClicks, setNumberOfClicks] = useRecoilState(counterState);

  const [incrementBy, setIncrementBy] = useRecoilState(incrementByState);

  return (

    <>

      <p>You have clicked the button {numberOfClicks} times.</p>

      <label>

        Increment By:

        <input

          value={incrementBy}

          onChange={(e) => setIncrementBy(Number(e.target.value))}

          type="number"

        />

      </label>

      <button onClick={() => setNumberOfClicks(numberOfClicks + incrementBy)}>

        Click

      </button>

    </>

  );

};

* **Recoil state** автоматически **шарится** **между компонентами** находящимися внутри **<RecoilRoot>.**
* В случае, когда не нужен setState из **useRecoilState**() то следует использовать другой хук – **useRecoilValue**()
* **Recoil selector –** they take the fundamental values that are expressed as atoms and they transform them in some way or combine them into another value.

**SCSS/SASS**

1. **Что такое SASS?**
   * Это препроцессор для CSS. Конвертирует SCSS файлы в CSS при это расширяя возможности CSS. Использует слегка отличный от CSS синтаксис.
2. **Назовите основные возможности SASS?**
   * Переменные. Думайте о переменных, как о способе хранения информации, которую вы хотите использовать на протяжении написания всех стилей проекта. Вы можете хранить в переменных цвета, стеки шрифтов или любые другие значения CSS, которые вы хотите использовать. Чтобы создать переменную в Sass нужно использовать символ $.
   * Вложенности (nesting). Sass позволит вам вкладывать CSS селекторы таким же образом, как и в визуальной иерархии HTML. Но помните, что чрезмерное количество вложенностей делает ваш документ менее читабельным и воспринимаемым, что считается плохой практикой.
   * Импорт (import). Sass берет идею импорта файлов через директиву @import, но вместо создания отдельного HTTP-запроса Sass импортирует указанный в директиве файл в тот, где он вызывается, т. е. на выходе получается один CSS-файл, скомпилированный из нескольких фрагментов.
   * Фрагментирование. Вы можете создавать фрагменты Sass-файла, которые будут содержать в себе небольшие отрывки CSS, которые можно будет использовать в других Sass-файлах. Это отличный способ сделать ваш CSS модульным, а также облегчить его обслуживание. Фрагмент — это простой Sass-файл, имя которого начинается с нижнего подчеркивания, например, \_partial.scss. Нижнее подчеркивание в имени Sass-файла говорит компилятору о том, что это только фрагмент и он не должен компилироваться в CSS. Фрагменты Sass подключаются при помощи директивы @import.
   * Миксины (примеси). Миксины позволяют создавать группы деклараций CSS, которые вам придется использовать по нескольку раз на сайте. Вы даже можете передавать переменные в миксины, чтобы сделать их более гибкими. Так же хорошо использовать миксины для вендорных префиксов.

**@mixin** **transform**($property) {

-webkit-transform: $property;

-ms-transform: $property;

**transform**: $property;

}

**.box** { **@include** **transform**(**rotate**(30deg)); }

**// в CSS это будет преобразовано к:**

**.box** {

**-webkit-transform**: rotate(30deg);

**-ms-transform**: rotate(30deg);

**transform**: rotate(30deg);

}

* + Расширение/Наследование. Это одна из самых полезных функций Sass. Используя директиву @extend можно наследовать наборы свойств CSS от одного селектора другому. Это позволяет держать ваш Sass-файл в «чистоте».

*/\* This CSS will print because %message-shared is extended. \*/*

%message-shared {

**border**: 1px solid #ccc;

**padding**: 10px;

**color**: #333;

}

*// This CSS won't print because %equal-heights is never extended.*

%equal-heights {

**display**: flex;

**flex-wrap**: wrap;

}

**.message** {

**@extend** %message-shared;

}

**.success** {

**@extend** %message-shared;

**border-color**: green;

}

**.error** {

**@extend** %message-shared;

**border-color**: red;

}

**.warning** {

**@extend** %message-shared;

**border-color**: yellow;

}

* + Математические операторы. Sass имеет несколько стандартных математических операторов, таких как +, -, \*, / и %.

**.container** {

**width**: 100%;

}

article**[**role**=**"main"**]** {

**float**: left;

**width**: 600px **/** 960px **\*** 100%;

}

aside**[**role**=**"complementary"**]** {

**float**: right;

**width**: 300px **/** 960px **\*** 100%;

}

**JEST**

1. **Что такое JEST?**
   * Jest — это восхитительный фреймворк для тестирования JavaScript с акцентом на простоту.
   * Работает с проектами, использующими Babel, TypeScript, Node, React, Angular, Vue и многое другое!
2. **Приведите примеры сопоставлений в JEST (примеры обычных тестирований).**
   * **Стандартные сопоставления**. Самый простой способ проверить значение — при помощи точного равенства.
   * test('два плюс два равно четыре', () => {  
      expect(2 + 2).toBe(4);  
     });
   * Или
   * test('сложение положительных чисел не равно нулю', () => {  
      for (let a = 1; a < 10; a++) {  
      for (let b = 1; b < 10; b++) {  
      expect(a + b).not.toBe(0);  
      }  
      }  
     });
   * **Сопоставления с типами null, undefined, false**.
     1. toBeNull соответствует только null
     2. toBeUndefined соответствует только undefined
     3. toBeDefined является противоположностью toBeUndefined
     4. toBeTruthy соответствует всему, что if инструкция рассматривает как true
     5. toBeFalsy соответствует всему, что if инструкция рассматривает как false
   * **Числа**. Большинство способов сравнения чисел имеют эквивалентные вычислители.
   * test('два плюс два', () => {  
      const value = 2 + 2;  
      expect(value).toBeGreaterThan(3);  
      expect(value).toBeGreaterThanOrEqual(3.5);  
      expect(value).toBeLessThan(5);  
      expect(value).toBeLessThanOrEqual(4.5);  
       
      *// toBe и toEqual эквивалентны по отношению к числам*  
      expect(value).toBe(4);  
      expect(value).toEqual(4);  
     });
   * **Строки**. Для сопоставления строк с регулярными выражениями, используйте toMatch:
   * test('в команде нет места Я', () => {  
      expect('команда').not.toMatch(/Я/);  
     });  
     test('но есть "ася" в Васе', () => {  
      expect('Вася').toMatch(/ася/);  
     });
   * **Массивы и перебираемые объекты**. Вы можете проверить, содержит ли массив или итерируемый объект конкретное значение, используя toContain:
   * const shoppingList = [  
      'diapers',  
      'kleenex',  
      'trash bags',  
      'paper towels',  
      'milk',  
     ];  
       
     test('the shopping list has milk on it', () => {  
      expect(shoppingList).toContain('milk');  
      expect(new Set(shoppingList)).toContain('milk');  
     });
3. **Как происходить тестирование асинхронного кода в JEST?**
   * <https://jestjs.io/ru/docs/asynchronous>
4. **Тестирование React приложений с помощью снимков.**
   * Тестирование методом снимков. Создадим тест использующий снимок для компонента Link

Link.js

import {useState} from 'react';

const STATUS = {

  HOVERED: 'hovered',

  NORMAL: 'normal',

};

export default function Link({page, children}) {

  const [status, setStatus] = useState(STATUS.NORMAL);

  const onMouseEnter = () => {

    setStatus(STATUS.HOVERED);

  };

  const onMouseLeave = () => {

    setStatus(STATUS.NORMAL);

  };

  return (

    <a

      className={status}

      href={page || '#'}

      onMouseEnter={onMouseEnter}

      onMouseLeave={onMouseLeave}

    >

      {children}

    </a>

  );

}

* + Теперь используем рендерер тестов React и функции создания снимков Jest для взаимодействия с компонентом и захвата результата его отображения, а также создания файла снимка:

Link.test.js

import renderer from 'react-test-renderer';

import Link from '../Link';

it('changes the class when hovered', () => {

  const component = renderer.create(

    <Link page="http://www.facebook.com">Facebook</Link>,

  );

  let tree = component.toJSON();

  expect(tree).toMatchSnapshot();

  // manually trigger the callback

  renderer.act(() => {

    tree.props.onMouseEnter();

  });

  // re-rendering

  tree = component.toJSON();

  expect(tree).toMatchSnapshot();

  // manually trigger the callback

  renderer.act(() => {

    tree.props.onMouseLeave();

  });

  // re-rendering

  tree = component.toJSON();

  expect(tree).toMatchSnapshot();

});

1. **Тестирование DOM в React.**
   * Для тестирования DOM можно использовать. React-testing-library, Enzyme, or React's TestUtils. Следующие два примера используют react-testing-library и Enzyme.
   * Давайте имплементируем чекбокс, который переключается между двумя лейблами:

CheckboxWithLabel.js

import {useState} from 'react';

export default function CheckboxWithLabel({labelOn, labelOff}) {

  const [isChecked, setIsChecked] = useState(false);

  const onChange = () => {

    setIsChecked(!isChecked);

  };

  return (

    <label>

      <input type="checkbox" checked={isChecked} onChange={onChange} />

      {isChecked ? labelOn : labelOff}

    </label>

  );

}

* + test

\_\_tests\_\_/CheckboxWithLabel-test.js

import {cleanup, fireEvent, render} from '@testing-library/react';

import CheckboxWithLabel from '../CheckboxWithLabel';

// Note: running cleanup afterEach is done automatically for you in @testing-library/react@9.0.0 or higher

// unmount and cleanup DOM after the test is finished.

afterEach(cleanup);

it('CheckboxWithLabel changes the text after click', () => {

  const {queryByLabelText, getByLabelText} = render(

    <CheckboxWithLabel labelOn="On" labelOff="Off" />,

  );

  expect(queryByLabelText(/off/i)).toBeTruthy();

  fireEvent.click(getByLabelText(/off/i));

  expect(queryByLabelText(/on/i)).toBeTruthy();

});

**Next.js**

1. **Опишите основные фичи (преимущества) Next.js.** 
   * Встроенная поддержка CSS. Есть инструменты CSS-in-JS, но Next.js выходит со своим собственным предложением — styled-jsx, а также поддерживает множество методологий стилизации.
   * Автоматическая поддержка TypeScript. Если вам нравится писать код на TypeScript, с Next.js у вас буквально есть автоматическая поддержка конфигурации и компиляции TypeScript.
   * Техника множественной выборки данных. Он поддерживает SSG и/или SSR. Вы можете использовать один или другой, или оба.
   * Маршрутизация файловой системы. Переход с одной страницы на другую поддерживается файловой системой вашего приложения. Вам не нужна специальная библиотека для обработки маршрутизации.
   * Есть еще много других функций, например, использование экспериментальных функций ES, таких как необязательная последовательность, отсутствие импорта React везде, где вы используете JSX, поддержка API типа next/head, он помогает управлять заголовками HTML-документа и так далее.
2. **Как устроена базовая маршрутизация в Next.js?**
   * Next.js имеет файловую систему маршрутизации на основе страниц. Каждая созданная страница автоматически становится маршрутом. Например, pages/books.js станет маршрутом /book.

- pages

  - index.js # url: /

  - books.js # url: /books

  - profile.js # url: /profile

* + Чтобы структурировать папку, и иметь такой маршрут, как /dashboard/user/:id, вам нужны вложенные папки:

- pages

  - index.js

  - dashboard

    - index.js

    - user

      - [id].js # dynamic id for each user

1. **Что такое индексные маршруты в Next.js?**
   * В папке pages автоматически создается страница index.js которая перенаправляется в начальную точку вашего приложения /. У вас могут быть разные index.js страницы, но по одной в каждой папке. Вам не обязательно это делать, но это помогает определить начальную точку ваших маршрутов и избежать некоторой избыточности в именах.

- pages

  - index.js

  - users

    - index.js

    - [user].js

* + Есть два индексных маршрута / и /users. В папке users можно назвать индексный маршрут users.js. Тогда его путь будет /users/users, если это удобно для чтения. В противном случае вы можете использовать индексный маршрут.

1. **Что такое и как описываются сегменты динамического маршрута в Next.js?**
   * Сегменты URL-адреса не всегда определенны. Иногда просто невозможно сказать, что будет в процессе разработки. Именно здесь на помощь приходят сегменты динамического маршрута.
   * В последнем примере :id является динамическим сегментом в URL-адресе /dashboard/user/:id. id определяет пользователя, который будет находиться на странице в данный момент.
   * Динамическая часть может появляться в любом месте вложенных маршрутов:

- pages

  - dashboard

    - user

      - [id].js

          - profile

* + Данный пример предоставляет путь /dashboard/user/:id/profile, который ведет к странице профиля пользователя с определенным идентификатором.

1. **Как происходит перемещение между страницами в Next.js?**
   * Перемещение между страницами в приложениях может выполняться с помощью компонента Link, экспортированного с помощью next/link.

export default function Users({users) {

  return (

    <div>

      <Link href="/">Home</Link>

      <Link href="/profile">Profile</Link>

      <Link href="/settings">

        <a> Settings </a>

      </Link>

      <Link href="/users">

        <a> Settings </a>

      </Link>

      <Link href="/users/bob">

        <a> Settings </a>

      </Link>

    </div>

  )

}

* + Свойство: as. Значение по умолчанию: такое же как href. Описание: указывает, что отображать в адресной строке браузера.
  + Свойство: passHref. Значение по умолчанию: false. Описание: заставляет компонет Link передать свойство href своему дочернему элементу.
  + Свойство: prefetch. Значение по умолчанию: true. Описание: позволяет Next.js заблаговременно получать страницы, которые в настоящее время находятся в области просмотра, даже до их посещения, для более быстрого перехода между ними.
  + Свойство: replace. Значение по умолчанию: false. Описание: заменяет текущую историю переходов вместо того, чтобы помещать новый URL-адрес в стек истории.
  + Свойство: scroll. Значение по умолчанию: true. Описание: после перехода новая страница должна быть прокручена вверх.

1. **Какие методы стилизации использует Next.js?**
   * Next.js поставляется с тремя стандартными методами стилизации, глобальным CSS, модулями CSS и styled-jsx.
2. **Что можете сказать о форматировании кода и линтинге в Next.js?**
   * ESLint и Prettier. Если последний идеально форматирует, то первый выравнивает вашу кодовую базу.
3. **Статические файлы в Next.js?**
   * На некоторых или нескольких этапах жизненного цикла вашего приложения Next.js вам понадобятся какие-либо файлы. Это могут быть иконки, собственные шрифты, изображения и т. д. В Next.js это называется Обслуживание статических файлов и есть единственный источник для них – папка public. Документация Next.js предупреждают: не называйте каталог public иначе.
4. Получение данных.
   * Предварительный рендеринг — важная часть того, как работает Next.js, а также то, что делает его быстрым. По умолчанию Next.js выполняет предварительный рендеринг каждой страницы, заранее генерируя HTML- код каждой страницы вместе с минимальным кодом JavaScript, который необходимо запустить, с помощью процесса, известного как Hydration.
   * При этом существует две формы предварительного рендеринга: 1) Статическая генерация (SG), 2) Отрисовка на сервере (SSR).
   * Разница между ними заключается в извлечении данных. Для SG данные извлекаются во время сборки и повторно используются при каждом запросе (что делает его быстрее, поскольку его можно кэшировать), тогда как в SSR данные выбираются при каждом запросе.
   * В обоих случаях Next.js предлагает специальные функции для получения данных.
   * getStaticProps — используется в SG, когда содержимое страницы зависит от внешних данных.
   * getStaticPaths — используется в SG, когда пути к страницам зависят от внешних данных.
   * getServerSideProps — используется в рендеринге на стороне сервера.
5. getStaticProps
   * getStaticProps является родственным getStaticPaths и используется в Static Generation. Это асинхронная функция, с помощью которой вы можете извлекать внешние данные и возвращать их в качестве поддержки для компонента по умолчанию на странице.
   * Данные возвращаются как объект props и неявно сопоставляются со свойством компонента экспорта по умолчанию на странице.

export default function AccountsPage({accounts}) {

  return (

    <div>

      <h1>Bank Accounts</h1>

      {accounts.map((account) => (

        <div key={account.id}>

          <p>{account.Description}</p>

        </div>

      ))}

    </div>

  )

}

export async function getStaticProps() {

  // This is a real endpoint

  const res = await fetch('https://sampleapis.com/fakebank/api/Accounts');

  const accounts = await res.json();

  return {

    props: {

      accounts: accounts.slice(0, 10),

    },

  };

}

1. getStaticPaths
   * Представьте, у вас есть приложение Next.js со страницами States и state, которые отображают список штатов в Соединенных Штатах и все государство соответственно. У вас может быть структура папок, которая выглядит так:

- pages

  - index.js

  - states

    - index.js # url: /states

    - [id].js # url /states/[id].js

* + Вы создаете, [id].js чтобы показать одно состояние на основе их id. Таким образом, содержимое страницы (возвращаемые данные getStaticProps) будет зависеть от путей к страницам (возвращаемых данных getStaticPaths). Сначала создадим компоненты <States/>.

// The states will be passed as a prop from getStaticProps

export default function States({states}) {

  // We'll render the states here

}

export async function getStaticProps() {

  // This is a real endpoint.

  const res = await fetch(`https://sampleapis.com/the-states/api/the-states`);

  const states = await res.json();

  // We return states as a prop to <States/>

  return {

    props: {

      states

    }

  };

}

* + Теперь давайте создадим динамическую страницу для одного состояния. Это причина, по которой у нас есть [id].js, чтобы мы могли сопоставить путь /states/1, или /states/2 где 1 и 2 — это id в [id].js.

// We start by expecting a state prop from getStaticProps

export default function State({ state }) {

    // We'll render the states here

}

// getStaticProps has a params prop that will expose the name given to the

// dynamic path, in this case, `id` that can be used to fetch each state by id.

export async function getStaticProps({ params }) {

  const res = await fetch(

    `https://sampleapis.com/the-states/api/the-states?id=${params.id}`

  );

  const state = await res.json();

  return {

    props: {

      state: state[0]

    }

  };

}

* + Если вы попытаетесь запустить код как есть, вы получите сообщение: Error: getStaticPaths is required for dynamic SSG pages and is missing for /states/[id].

// The state component

// getStaticProps function

// getStaticPaths

export async function getStaticPaths() {

  // Fetch the list of states

  const res = await fetch("https://sampleapis.com/the-states/api/the-states");

  const states = await res.json();

  // Create a path from their ids: `/states/1`, `/states/2` ...

  const paths = states.map((state) => `/states/${state.id}`);

  // Return paths, fallback is necessary, false means unrecognize paths will

  // render a 404 page

  return { paths, fallback: false };

}

* + С помощью paths возвращенного от getStaticPaths, getStaticProps будет известно и его params параметры будут заполнены необходимыми значениями, как и id в этом случае.

1. **Что такое styled-jsx?**
   * Стилизованный JSX — это библиотека CSS-in-JS, которая позволяет вам писать инкапсулированный и ограниченный CSS для стилизации ваших компонентов. Стили, которые вы вводите для одного компонента, не повлияют на другие компоненты, позволяя вам добавлять, изменять и удалять стили, не беспокоясь о непредвиденных побочных эффектах.