1. **Что такое React?**

**React** - это библиотека JS для создания UI.

Главным преимуществом является компонентный подход, то есть разбитие UI на небольшие независимые компоненты, каждый из которых имеет свою логику и состояние.

**JSX (JS XML)** – синтаксис расширения для JS, которое используется в React для описания структуры UI, он позволяет встраивать JS-код в HTML-подобную разметку.

**NPM (Node Package Manager)** – это менеджер пакетов, который управляет зависимостями (библиотеками) приложения и позволяет устанавливать необходимые. Так же можно создать собственную библиотеку и опубликовать её, NPM это позволяет.

**WEB-интерфейс** – веб-страница или их совокупность, которые предоставляют UI для взаимодействия с сайтом или приложением.

**UI-интерфейс –** то, через что пользователь может взаимодействовать с сайтом или приложением.

**Библиотека –** набор готовых функций и инструментов, которые можно использовать при создании приложения или сайта на наше усмотрение.

**Фреймворк –** это структура и правила, которые сами определяют, как мы должны выстраивать приложение.

**CRUD –** аббревиатура, которая представляет основные http-методы (создание -> POST, чтение -> GET, обновление -> PUT, удаление -> DELETE)

1. **Рассказать про свойства в файлах package.json и package-lock.json**

В **package.json** лежит информация о всех установленные пакетах, зависимостях (библиотеках), скрипты сборки, основная информация о приложении (имя проекта, версия, приватность).

**package-lock.json** используется для фиксации версий всех установленных зависимостей в проекте, создается при установке пакетов с помощью NPM

1. **Рассказать про public и node\_modules**

В папке **public** хранятся статические ресурсы, которые должны отображаться в браузере без непосредственной обработки React-кодом.

В папке **node\_modules** хранятся все, установленные с помощью NPM, библиотеки

1. **Что такое Virtual DOM? Зачем он нужен?**

**VDOM** – это промежуточное представление реального DOM в памяти компьютера. React создает виртуальное дерево элементов (набор HTML-тегов), которое в точности соответствует реальному состоянию компонентов реального DOM. Затем сравнивает их и, вместо того, чтобы изменять каждый элемент в реальном DOM, React обновляет вирутальный DOM, а затем при накоплении минимального количества изменений, изменяет реальный DOM.

То есть **VDOM нужен для того**, чтобы оптимизировать изменения в реальном DOM и сделать программирование еще более удобным (не нужно беспокоиться о том, как будет обновляться реальный DOM и как компоненты будут отображаться в разных браузерах, React сам об этом позаботится).

1. **Расскажи что такое процесс Reconciliation?**

**Reconciliation** – это процесс, при котором React сравнивает предыдущее состояние компонентов с их новым состоянием и определяет, какие изменения нужно внести в UI.

Состоит из следующих процессов:

1. При первом рендере компонентов создается виртуальное дерево на основе JSX-разметки, создаются элементы разметки (теги) и устанавливается первоначальное состояние компонентов
2. React обновляет реальный DOM, отображая созданный виртуальный DOM на экране
3. После изменения состояния каких бы то ни было компонентов, React запускает процесс reconciliation и сравнивает предыдущий и новый виртуальный DOM
4. Сравниваются элементы дерева до изменений и после, далее React определяет, что нужно изменить/удалить, а что трогать не нужно
5. Вместо перерисовывания всего интерфейса, React применяет только необходимые изменения в реальном DOM
6. **Создать классовую компоненту.**

class User extends React.component {  
constructor (name, age) {  
this.name = name;

this.age = age;  
}

render (  
<div>  
`${this.name}, ${this.age}`

</div>  
)

}

1. **Что такое пропс? Для чего он используется? Написать пример**

class User extends React.component {  
constructor () {}

render() {  
<div>  
{this.props.list.split(‘’)}

<div/>  
}

}

function App () {  
return <User list ={[‘Ilya’, ‘Pasha’, ‘Nadya’]}/>  
}

**Пропсы** – это объекты, которые передаются в компоненты как параметры, позволяют передать данные из родительского компонента в дочерний. Через пропсы можно передавать любые типы данных (в том числе можно передать другой компонент), но пропсы нельзя менять в дочернем компоненте.

1. **Что такое деструктуризация пропсов и как он работает?**

Работает как и классическая деструктуризация при передаче в качестве параметра в функцию объекта.

1. **Какие проблемы могут возникнуть при изменении пропсов внутри компонента?**

В консоли будет ошибка, что пропсы только для чтения.

1. **Что такое state и для чего используется?**

**State** – это объект простого JS, позволяющий отслеживать и обновлять данные компонента.

В классовых компонентах состояние хранится в свойстве state, которое является объетом, начальное состояние определяется в конструкторе явно.

class Counter extends Component {

constructor(props) {

super(props);

this.state = { count: 0, };

}

1. **setState. Написать пример изменения state.**

Для обновления состояния в классовых компонентах используют метод **setState(),** он принимает объект с обновленным значением состояния или функцию.

class Counter extends Component {

constructor(props) {

super(props);

this.state = { count: 0, };

}

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

// или

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

};

// Остальной код компонента

}

Второй вариант предпочтительнее, так как метод setState является асинхронным и такая запись гарантирует, что state будет обновлен.

1. **Параметры setState.**

Метод принимает в качестве параметра или объект с измененным состояниям или функцию, которая получает текущее состояние в качестве аргумента и возвращает новый объект с изменениями, вторым параметром является callback-функция, которая вызовется когда state будет обновлен.

1. **Что происходит при вызове setState?**

При вызове метода, React обновит состояние компонента и выполнит его перерисовку (то есть каждый вызов метода триггерит вызов метода render).

1. **Перечислить методы жизненного цикла.**
2. Методы монтирования
3. constructor()
4. render()
5. componentDidMount()
6. Методы обновления
7. shouldComponentUpdate(nextProps, nextState)
8. render()
9. componentDidUpdate(prevProps, prevState)
10. Методы размонтирования
11. componentWillUnmount()
12. Методы обработки ошибок
13. componentDidCatch(error, info)
14. **Рассказать про каждый из методов.**

Метод constructor() вызывается при создании экземпляра компонента, используется для инициализации состояния и привязки методов.

Метод render() отрисовывает компоненты и возвращает React элементы.

Метод componentDidMount() вызывается после рендеринга компонента и его добавления в DOM. Используется для выполнения запросов к серверу, подписки на события и других побочных действий.

Метод shouldComponentUpdate(nextProps, nextState) вызывается перед обновлением компонента, используется для оптимизации перерисовки компонента путем возврата false, если обновление не требуется.

Метод componentDidUpdate(prevProps, prevState) вызывается после обновления компонента и его перерендеринга, используется для выполнения дополнительных действий после обновления компонента, например, для выполнения запросов на сервер на основе новых пропсов.

Метод componentWillUnmount() вызывается перед удалением компонента из DOM, используется для выполнения необходимых очисток, отписки от событий и освобождения ресурсов.

Метод componentDidCatch(error, info) вызывается при возникновении ошибки в дочерних компонентах, используется для отлавливания и обработки ошибок

1. **Что нельзя делать в методе render()?**

Данный метод должен оставаться чистой функцией, то есть не вызывать изменения состояния компонента (не вызывать в нем метод setState), т.к. это спровоцирует циклический вызов метода render() и допускать вызова колбэков в родительском компоненте.

1. **Создать функциональную компоненту.**

const User = () => <div>Ilya</div>

1. **Как передать пропсы?**

Аналогично с классовыми компонентами, через атрибуты компонента, в самой компоненте они передаются в качестве аргументов функции, можно использовать деструктуризацию для того, чтобы постоянно не обращаться к ссылке на пропс.

1. **Как объявить state? useState. Какие параметры?**

const [count, setCount] = useState(0)

Хук useState возвращает массив, первым свойством которого является текущее состояние, вторым – функция для обновления состояния, в параметры хуку можно передать начальное состояние.

1. **Жизненный цикл, useEffect, написать каждый метод**

Для оптимизации работы функциональных компонент вместо методов жизненного цикла использует хук useEffect()

1. Монтирование компонента

useEffect(() => console.log(‘componentDidMount’), []) – отработает один раз при монтировании компонента, аналогично с componentDidMount(), передавая пустой массив зависимостей можно сделать этот хук методом монтирования

1. useEffect(() => console.log(‘componentDidUpdate’), […depencies]) – отработает при монтировании и затем, при изменении зависимостей, отработает как componentDidUpdate, вторым параметром передается массив зависимостей, пропсы или стэйты, с изменением которых будет обновляться компонент
2. useEffect(() => console.log(‘componentDidMount and componentDidUpdate’)) – выполняет функцию после каждого рендеринга компонента, то есть при каждом обновлении компонента, то есть используется метод componentDidUpdate он отрисовывается и затем срабатывает как componentDidMount
3. useEffect(() => {  
   return () => console.log(‘componentWillUnmount’)

}, []) – выполняет очистку и отписку от побочных эффектов при размонтировании компонента (удаления его из DOM), выполняется перед размонтированием компонента.

1. **Каким образом передаются данные из родительского компонента в дочерний?**

С помощью пропсов и контекста.

1. **Как передать данные из дочернего компонета в родительский?**

Необходимо создать в родительском компоненте callback-функцию и передать ее в качестве пропса дочернему компоненту, а затем, в дочернем вызвать ее по событию и передать в дочернем компоненте этой функции какое-то значение в качестве аргумента.

function App () {  
const callback = (param) => console.log(param)

return (  
 <div>

<Test callback = {callback}/>

</div>

)  
}

const Test = ({callback}) => {

return (  
<div>

<button onClick = {() => callback(2)}>Click</button>

</div>  
)

}

1. **Какие есть события?**

onClick, obDblClick, onChange, onFocus, onBlur, onKeyDown, onKeyPress, onSubmit, onScroll и т.д.

1. **Как передать параметры в обработчик события?**

<button onClick={(e) => deleteRow(id, e)}>Удалить строку</button>

Передать можно с помощью анонимной функции, которой мы передаем в качестве параметров то, что нужно и вызываем нужный метод с этими параметрами.

1. **Как реализовать условный рендеринг компонентов в React?**

Реализуется с помощью тернарного оператора, логического оператора && либо классического if..else

Суть в том, что при определенных условиях (должно быть true) у нас отрендериться один компонент, но если false, то рендериться другой компонент.

Например рендеринг сообщений об успешном входе на сайт или сообщение о том, что нужно залогиниться:

function UserGreeting(props) {

return <h1>С возвращением!</h1>;

}

function GuestGreeting(props) {

return <h1>Войдите, пожалуйста.</h1>;

}

function Greeting(props) {

const isLoggedIn = props.isLoggedIn;

if (isLoggedIn) {

return <UserGreeting />;

}

return <GuestGreeting />;

}

1. **useContext. Для чего используется? Написать пример.**

Context предоставляет способ **передачи данных через дерево компонентов без явной передачи пропсов от родительских компонентов к дочерним**. Это особенно полезно, когда необходимо передать данные глубоко в иерархии компонентов, избегая "просачивания" пропсов через несколько промежуточных компонентов.

То есть вместо того, чтобы передавать через каждый компонент по дереву пропсы, без явного их использования, мы можем передать сразу в нужный компонент. Обычная передача пропсов через несколько ветвей дерева приводит к усложнению кода, затруднение переиспользования.

Порядок таков:

1. Создание контекста:

Контекст лучше создавать в новой компоненте

import React from 'react';

const MyContext = React.createContext();

export default MyContext

1. Предоставление значения контекста

Затем нужно предоставить значение контекста внутри компонента-провайдера, он обертывает дерево компонентов, которым нужен доступ к контексту

import React from 'react';

import MyContext from './MyContext';

import ComponentA from './ComponentA';

const App = () => {

const value = 'Hello, Context!';

return (

<MyContext.Provider value={value}>

<ComponentA />

</MyContext.Provider> );

}

export default App;

1. Использование хука useContext

Теперь можно использовать хук внутри функционального компонента, чтобы получить доступ к значению контекста

import React, { useContext } from 'react';

import MyContext from './MyContext';

import ComponentB from './ComponentB';

function ComponentA() {

const value = useContext(MyContext);

return (

<div>

<h2>Component A</h2>

<p>Received value from Context: {value}</p>

<ComponentB />

</div> );

}

export default ComponentA;

1. Подписка на контекст (еще один вариант)

Еще один вариант, можно использовать MyContext.Consumer

import React from 'react';

import MyContext from './MyContext';

import ComponentB from './ComponentB';

function ComponentA() {

return (

<div>

<h2>Component A</h2>

<MyContext.Consumer>

{(value) => (

<p>Received value from Context: {value}</p>

)}

</MyContext.Consumer>

<ComponentB />

</div> );

}

export default ComponentA;

1. **Разница между компонентом и элементом?**

Компонент это функция, которая возвращает React-элементы.

1. **Что такое refs? Для чего они используются? Написать пример.**

Refs (далее просто “ссылки”) предоставляет способ доступа к DOM-узлам или React-элементам, созданным в методе render().

Есть несколько хороших примеров использования ссылок:

1. Управление фокусом, выделение текста
2. Выполнение анимаций
3. Интеграция со сторонними библиотеками, взаимодействующие с DOM.

Например, управление фокусом:

import React, { useEffect, useState, Fragment, useRef } from 'react'

const Test = () => {

const [text, setText] = useState('')

const textInput = useRef(null)

const valid = () => {

textInput.current.focus()

}

return (

<div>

<input ref={textInput} value={text} />

<button onClick={() => valid()}> Click to focus </button>

</div>

)

}

export default Test;

1. **Что такое Fragment? Написать пример**.

Нужен для того, чтобы сгруппировать список дочерних элементов без добавления дополнительных DOM-элементов.

const ChildComponent = () => <Fragment>

<h1>123</h1>

<h1>456</h1>

</Fragment>

Есть так же сокращенная запись, единственный нюанс – при ее использовании нельзя использовать атрибуты.

const ChildComponent = () => <>

<h1>123</h1>

<h1>456</h1>

</>

1. **Что такое управляемые компоненты (controlled components)?**

Разница между контролируемой и неконтролируемой в том, что в контролируемой мы можем отслеживать то, что вводит пользователь, присваивая результат ввода, например в инпуте, переменной и проверяя корректность введенных данных.

const Text = () => {  
const [text, setText] = useState(‘’)

return <>

<input onChange = {(event) => setText(event.target.value)} value = {text}/>

</>

}

В React, параметр **event** в callback-функции **представляет объект события**, который генерируется при изменении значения элемента формы или компонента. **event.target.value** содержит **текущее значение** **input** элемента.

1. **Как установить значение по умолчанию для таких компонентов?**

Мы можем задать значение по умолчанию при создании стэйта.

const Text = () => {  
const [text, setText] = useState(‘Введите значение’)

return <>

<input onChange = {(event) => setText(event.target.value)} value = {text}/>

</>

}

1. **В чем смысл специального атрибута key? Написать пример.**

Нужен для того, чтобы React понимал какие элементы нужно добавить, изменить или удалить. Ключи нужно определять непосредственно внутри массивов. Если не указать ключи, в консоли увидим сообщение. Ключи **оптимизируют работу с элементами массивов**, уменьшают количество ненужных удалений и созданий элементов. Добавляя key, мы помогаем механизму reconciliation тем, что с key он сверяет не попарно, а ищет компоненты с тем же key.

const List = ({list}) => {  
return <div>{list.map(item, index => <p key = {index}>{item}</p>)}

}

Но использовать индекс элемента массива не лучшая идея, лучше пользоваться библиотекой uuid.

1. **shouldComponentUpdate(). Для чего нужен?**

Он нужен для оптимизации рендеринга, то есть чтобы не ререндерить каждый раз компонент, когда это не нужно (например мы хотим выводить на экран только четные числа).

shouldComponentUpdate(nextProps, nextState) {

// Прописываем логику, когда компонент должен ререндериться

}

Данный метод возвращает true или false, если нужно ререндерить – true, иначе false. По умолчанию возвращает true.

Им лучше не злоупотреблять, лучше использовать Pure Components, которые так же следят за изменением пропсов и стэйта.

1. **Что такое Pure Components?**

Это компонент, который вызывает ререндеринг только при изменении пропсов или стэйт. Он нужен для оптимизации работы.

Class User extends React.PureComponent {}

В Pure Components метод shouldComponentUpdate() по умолчанию возвращает false.

1. **React.memo. Для чего используется? Написать пример.**

**React.memo** — это функция высшего порядка (Higher Order Component) в библиотеке React, которая оптимизирует производительность функциональных компонентов. Аналог shouldComponentUpdate() в функциональных компонентах.

Данный метод использует механизм мемоизации, чтобы запоминать предыдущий результат рендеринга компонента и сравнивать его с текущими пропсами. Если пропсы не изменились, он предотвращает повторный рендеринг компонента, возвращая сохраненный результат.

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

// Компонент будет перерисовываться только при изменении пропсов return ( <div> {props.text} </div> ); });

Также данный метод принимает второй необязательный параметр в виде функции сравнения, которая позволяет контролировать, какие пропсы должны считаться равными, если функция сравнения не указана, React будет использовать поверхностное сравнение для пропсов.

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

// Компонент будет перерисовываться только при изменении пропсов с помощью

// compareFunction

return ( <div> {props.text} </div> );}, (prevProps, nextProps) => {

// compareFunction - пользовательская функция сравнения пропсов

// Возвращает true, если пропсы должны считаться равными и компонент не должен перерисовываться });

const Text = () => {  
const [text, setText] = useState(‘Введите значение’)

return <>

<input onChange = {(event) => setText(event.target.value)} value = {text}/>

</>

}

export default React.memo(Text)

Используется для того, чтобы без надобности (например в компоненте ничего вообще не изменяется, но он заново рендериться в другом компоненте, т.к. в том компоненте изменения происходят) не ререндерить компонент.

1. **useMemo. Для чего используется? Написать пример.**

Данный хук предназначен для мемоизации вычислений в функциональных компонентах. Он позволяет сохранять результаты вычислений и повторно использовать их, если входные значения (зависимости) не изменились.

const handleClick = useMemo(() => {  
return count % 2 === 0? count: count - 1

}, [count])

1. **Зачем нужен роутинг в веб-приложениях и как его реализовать в React?**