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?**

Роутинг в React обеспечивает навигацию и переходы между различными страницами и разделами сайта или приложения. Основная особенность роутинга – возможность осуществлять навигацию без перезагрузки страницы. Т.е. чтобы каждый раз не отправлять запрос на сервер и не загружать новую страницу, роутинг позволяет обновлять данные на странице динамически, только то, что нужно в данный момент.

Роутинг в React реализовывается с помощью следующих компонентов и хуков:

1. **<BrowserRouter></BrowserRouter>** - компонент, который оборачивает приложение и предоставляет контекст для реализации маршрутизации, т.е. настраивает основной маршрутизатор в приложении
2. **<Routes></Routes>** - компонент-контейнер, в котором определяются все маршруты в приложении, т.е. внутри него группируются все компоненты Route

**<Routes>**

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

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

**<Route path="/contact" element={<Contact />} />**

**</Routes>**

1. **<Route/>** - компонент, который определяет отдельный маршрут внутри **Routes** каждый **Route** имеет параметры **path** и **element**

**<Routes>**

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

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

**<Route path="/contact" element={<Contact />} />**

**</Routes>**

1. **<Link></Link>** - используется для создания ссылок для навигации между маршрутами в приложении

**<Link to="/">Главная</Link>**

**<Link to="/about">О нас</Link>**

**<Link to="/contact">Контакты</Link>**

1. **<NavLink></NavLink>** - позволяет создавать активные ссылки (отличие от простого **Link** в том, что **NavLink** добавляет класс active к ссылке, когда соответствующий маршрут активен). Это позволяет нам использовать класс как инструмент для стилизации страницы.

**<NavLink to="/" activeClassName="active">Главная</NavLink>**

**<NavLink to="/about" activeClassName="active">О нас</NavLink>**

**<NavLink to="/contact" activeClassName="active">Контакты</NavLink>**

1. **<Outlet/>** - компонент, который позволяет собрать определенный контент воедино, например:

Создаем компонент Layout, который будет ретурнить следующее:

**<header></header>**

**<Outlet/>**

**<footer></footer>**

Затем, в главном компоненте App прописываем роуты и оборачиваем в компонент выше роуты на наш контент:

**<Routes>**

**<Route path=’/’ element={<Layout/>}>**

**// Роуты на наш контент**

**</Route>**

**</Routes>**

1. **useNavigate()** – это хук, который позволяет перенаправлять пользователя по маршрутам.

**const Home = () => {**

**const navigate = useNavigate();**

**const handleClick = () => {**

**navigate('/about');**

**};**

**return (**

**<div>**

**<h1>Домашняя страница</h1>**

**<button onClick={handleClick}>Перейти к странице О нас</button>**

**</div>**

**);**

**};**

С помощью navigate можно сделать кнопку возврата назад:

**const navigate = useNavigate()**

**const goBack = () => navigate(-1)** – параметр у функции **navigate()**, которая является результатом вызова хука **useNavigate()** может быть как числовой (число соответствует тому, на сколько шагов перейти вперед (положительное число) или назад (отрицательное число)), так и абсолютный, то есть фактический URL-адрес куда перейти по выполнении этой функции, функцию можно передать или в кнопку по событию или в **<Link></Link>** по событию.

Еще один вариант применения – это компонент <Navigate/> и его использование для переадресации:

**<Route path=’/about’ element={<AboutPage/>}/>**

**<Route path=’/about-us’ element={<Navigate to=’/about’/>}/>**

Функция навигации может принимать в качестве параметра либо число, что означает навигацию назад/вперед по заданному числу шагов, либо строку с точным указанием куда перейти и тогда вторым параметром можно передать функцию NavigateOptions, которая принимает 2 параметра:

1. **replace** – по умолчанию **false**, то есть движение будет по истории, как в случае с числовым параметром выше, если задать его как true, то движение не будет записано в историю.
2. **state** – некие данные

С помощью navigate удобно переадресовывать пользователей на основе результата http-запроса.

1. **useParams()** – это хук, который позволяет получать параметры из маршрута.

**const UserProfile = () => {**

**const params = useParams();**

**return (**

**<div>**

**<h1>Профиль пользователя: {params.username}</h1>**

**</div>**

**);**

**};**

В этом примере использование хука позволяет получить некий объект **params**, в котором предположительно есть параметр **:username** и он выводится на страницу

Данный хук позволяет создавать динамические URL-адреса, например:

**<Route path=’about/:id’ element={<SinglePage/>}>**

**const { id } = useParams()**

**<Link to={`about/${id}/edit`}>Edit post</Link>**

1. **useLocation()** – это хук, который позволяет получать информацию о текущем **URL** и его параметрах.

**const Page = () => {**

**const location = useLocation();**

**return (**

**<div>**

**<h1>Текущий URL: {location.pathname}</h1>**

**</div>**

**);**

**};**

Хук **useLocation()** возвращает объект Location, который содержит в себе следующие свойства:

1. **hash** – хэш текущего URL-адреса
2. **key** – уникальный ключ страницы
3. **pathname** – путь текущего URL-адреса (например /about)
4. **search** – строка запроса текущего URL-адреса
5. **state** – значение состояния местоположения, созданного с помощью **<Link state={}>** или **useNavigate ()**

Например:

**const navigate = useNavigate()**

**const goBack = () => navigate(‘/about’ , {state: 123})**

То есть, при переходе по ссылке **/about** в **state** объекта **location** запишется число **123**.

1. **Приватный роутинг?**

**const PrivateRoute = () => {  
const isAuth = // логика проверки авторизовался ли пользователь**

**return isAuth ? <Route {…props} /> : <Navigate to=’/login’ replace>  
}**

Далее компонентом PrivateRoute нужно обернуть те маршруты, которые должны быть защищены от неавторизованного пользователя:

**<Routes>**

**<Route path=’/login’ element={<LoginPage/>}/>**

**<PrivateRoute path=’/dashboard’ element={<DashboardPage/>}/>**

**</Routes>**

1. **Вложенный роутинг.**

Вложенный роутинг в тех случаях, когда нужно создать иерархию маршрутов, где одни маршруты будут вложены в другие.

**const App = () => {**

**return (**

**<div>**

**<h1>Мое приложение</h1>**

**<Routes>**

**<Route path="/" element={<HomePage />} />**

**<Route path="/about" element={<AboutPage />} />**

**<Route path="/products" element={<ProductsPage />}>**

**<Route path="/" element={<AllProducts />} />**

**<Route path="/:id" element={<ProductDetail />} />**

**</Route>**

**<Route path="/contact" element={<ContactPage />} />**

**</Routes>**

**</div>**

**);**

**};**

Для использования вложенных роутов внутри компонента ProductsPage можно воспользоваться компонентом Outlet.

Outlet определяет место, где будут отображаться вложенные роуты:

**const ProductsPage = () => {**

**return (**

**<div>**

**<h2>Страница продуктов</h2>**

**<ul>**

**<li>**

**<Link to="/">Все продукты</Link>**

**</li>**

**<li>**

**<Link to="/products/1">Продукт 1</Link>**

**</li>**

**<li>**

**<Link to="/products/2">Продукт 2</Link>**

**</li>**

**</ul>**

**<Outlet />**

**</div>**

**); };**

В этом примере компонент **ProductsPage** содержит ссылки на различные продукты. Компонент **Outlet** определяет место, где будут отображаться вложенные маршруты, т.е. компоненты **<AllProducts />** и **<ProductDetail />** в нашем случае.

1. **Про Storage.**

Существует несколько вариантов хранения данных в браузере:

1. **Local Storage** – данные хранятся локально в браузере (до 5МБ) и они будут сохранены после закрытия страницы или браузера, может быть полезен для хранения пользовательских настроек страницы (например темная тема)

Синтаксис:

1) запись данных: **localStorage.setItem(‘stringKey’, ‘stringValue’)**

2) чтение данных: **localStorage.getItem(‘stringKey’)**

1. **Cookie** – небольшие текстовые файлы (до 4КБ), которые веб-страницы отправляют и хранят компьютере пользователя через его веб-браузер, могут хранить информацию о состоянии приложения и обеспечивать персонализацию

Используются для хранения пользовательских настроек, данных об аутентификации и авторизации, а также для анализа поведения пользователя на сайте и предоставления персональных рекомендаций. Главное отличие от локального **стора** в том, что **куки** имеют срок хранения, который мы сами указываем, а так же то, что **куки** включаются в заголовки запросов автоматически, в то время как локальный **стор** нужно добавлять явно.

Установка производится через **npm**, если это пакетный модуль, либо можно использовать **JS API**.

1) установка cookie: **document.cookie = ‘user\_id=123; expires=Fri, 31 Dec 2023 23:59:59 GMT; path=’/’’**

2) удаление cookie: **document.cookie = ‘user\_id=; expires=Thu, 01 Jan 1970 00:00:00 GMT; path=’/’’**

1. **Session Storage** – еще один механизм хранения данных в браузере, однако в отличие от **local storage**, хранит данные только на время сеанса браузера, то есть данные будут удалены после закрытия вкладки или браузера (обычно в таких **сторах** хранят данные форм), так же имеет меньший объем памяти для хранения.

Синтаксис аналогичен **localStorage**, но **sessionStorage.**

1. localForage – JS библиотека, предоставляющая удобный асинхронный интерфейс для работы с локальными хранилищами в браузерах

Установка через **npm: (npm localforage)**

Методы аналогичны встроенным **storage**.

1. **High Order Components.**

**HOC –** это функции высшего порядка, которые принимают компонент в качестве параметра и возвращают его же с расширенным функционалом. Они помогают избежать дублирования кода, управлять состоянием, обеспечивают абстракцию поведения и **позволяют добавлять функциональность к компонентам без изменения их кода**.

**Проблемы, которые решают HOC:**

**Логическое переиспользование**: HOC позволяют выделить общую логику компонентов, такую как обработка аутентификации или получение данных, и применять ее ко множеству компонентов без дублирования кода.

**Прокинуть дополнительные props**: HOC позволяют добавлять или модифицировать props компонентов. Это удобно, например, для передачи данных из контекста или оборачивания компонентов в дополнительные компоненты высшего порядка.

**Обертывание компонентов**: HOC могут использоваться для обертывания компонентов дополнительным функциональным слоем, например, для логирования, манипуляции с состоянием и т.д.

**Создание HOC:**

**const withLogger = (WrappedComponent) => {**

**return (props) => {**

**console.log('Component rendered:', WrappedComponent.name);**

**return <WrappedComponent {...props} />;**

**};**

**};**

Прокидывание дополнительных пропсов:

**const withExtraProps = (WrappedComponent) => {**

**return (props) => {**

**const extraProps = { additionalProp: "I'm extra!" };**

**return <WrappedComponent {...props} {...extraProps} />;**

**};**

**};**

1. **Создание форм с помощью библиотек Formik и React-hook-form.**

**1) React-hook-form**

Установка с помощью npm: **npm install react-hook-form**

Хуки для работы с формами:

1. **useForm()** – хук, который возвращает следующие методы:
2. **register()** - этот метод позволяет зарегистрировать элемент ввода или выбора и применить правила проверки к форме React Hook.

Все правила проверки основаны на стандарте HTML и допускают использование пользовательских методов проверки.

Первым параметром является имя ключа, по которому запишется значение, введенное пользователем в поле ввода, эта пара записывается в специальный объект, который хранит данные формы. Вторым параметром передается объект, в котором можно настроить правила валидации для формы, например:

1. **required** – поле обязательно для ввода (либо boolean, либо текст ошибки, который можно вывести в случае невалидных данных)
2. **max/min** – максимальное/минимальное числовое значение, которое можно принять для этого ввода
3. **maxLength/minLength** – максимальная/минимальная длина значения в поле ввода, должно быть числом
4. **pattern** – позволяет добавить регулярное выражения для валидации
5. **validate** – кастомная функция для валидации
6. **unregister()** – отменяет регистрацию формы
7. **formState()** - этот объект содержит информацию обо всем состоянии формы. Это помогает нам отслеживать взаимодействие пользователя с вашим приложением-формой

Два полезных свойства:

1. errors – объект, содержащий ошибки
2. isValid – проверяет форму на валидность
3. **watch()** – этот метод отслеживает указанные входные данные и возвращает их значения
4. **handleSubmit()** – эта функция получает данные формы, если они валидны
5. **reset()** – сброс всего состояния формы, ссылок на поля и подписок
6. и другие…

Так же при вызове этого хука можно передать ему следующие параметры:

1. mode – на что будут реагировать правила валидации, например onChange, onBlur, onSubmit (по умолчанию), onTouched, all
2. reValidateMode
3. defaultValues
4. values
5. errors – объект, который используется для валидации (например проверить, правильно ли заполнено поле)
6. resetOptions
7. и другие…
8. **useController()** – хук, который возвращает компонент Controller.
9. <Controller/> - компонент, который облегчает работу со сторонними UI-библиотеками

Свойства, которые можно передать в компонент:

1. name – имя переменной, куда запишется значение поля ввода
2. control – объект, который является результатом вызова useForm()
3. rules – правила валидации, по аналогии с options у register()
4. render – функция, которая возвращает React элемент и позволяющая прикреплять события и значения к компоненту
5. useFormContext()
6. <FormProvider/>
7. useWatch()
8. useFormState()
9. <ErrorMessage/>
10. useFieldArray()

**register() и <Controller/> нельзя использовать вместе.**

**2) Formik**