**Top React Js interview questions to prepare From The video:**

**1) Hooks: Prepare this in order**

**- useState**

* useState is a React Hook that enables functional components to declare and manage state variables. It is used to create a local state variable and a corresponding function to update that state.

Syntax:

**const [state, setState] = useState(initialValue);**

* state: The current state value.
* setState: A function to update the state.
* initialValue: The initial value of the state.

**- useEffect**

* The **useEffect** hook in React is used for side effects in functional components. It allows you to perform actions like data fetching, subscriptions, or manually changing the DOM after the component has rendered. It takes two arguments: a function containing the code for the side effect, and an optional array of dependencies to control when the effect runs. This helps in managing the lifecycle of the component.

**useEffect(() => {**

**// Side effect code**

**// This code will run after the component has been rendered**

**// It may also clean up any resources when the component unmounts**

**// Return a cleanup function (optional) if needed**

**return () => {**

**// Cleanup code**

**};**

**}, [dependencies]);**

**- useContext**

* The useContext hook in React is used to consume values from the React context within functional components. Context provides a way to pass data through the component tree without having to pass props manually at every level.
* Props drilling to avoid

import React, { useContext } from 'react';

const MyContext = React.createContext();

function MyComponent() {

// Using useContext to access the value from the nearest MyContext.Provider

**const contextValue = useContext(MyContext);**

return (

<div>

<p>Value from Context: {contextValue}</p>

</div>

);

}

// MyContext.Provider is used to provide a value to the component tree

function App() {

return (

**<MyContext.Provider value="Hello from Context">**

<MyComponent />

</MyContext.Provider>

);

}

export default App;

**- useReducer**

* The useReducer hook in React is a state management hook that is used as an alternative to useState. It is particularly useful when managing complex state logic that involves multiple sub-values or when the next state depends on the previous state.
* It returns the current state and a dispatch function to update the state. The dispatch function accepts an action object that describes how the state should be updated.

import React, { useReducer } from 'react';

const initialState = /\* initial state value \*/;

// Reducer function takes the current state and an action, returns the new state

const reducer = (state, action) => {

switch (action.type) {

case 'INCREMENT':

return { count: state.count + 1 };

case 'DECREMENT':

return { count: state.count - 1 };

default:

return state;

}

};

function MyComponent() {

// useReducer returns the current state and a dispatch function

**const [state, dispatch] = useReducer(reducer, initialState);**

return (

<div>

<p>Count: {state.count}</p>

<button onClick={() => dispatch({ type: 'INCREMENT' })}>

Increment

</button>

<button onClick={() => dispatch({ type: 'DECREMENT' })}>

Decrement

</button>

</div>

);

}

export default MyComponent;

**- useMemo**

In React useMemo Hook returns a memoized value and prevents the application from unnecessary re-renders. It is useful in heavy computations and processes when using functional components.

Here's a simple example:

import React, { useState, useMemo } from 'react';

const ExampleComponent = ({ data }) => {

// Expensive calculation function

const calculateResult = (data) => {

// Some complex calculation based on 'data'

return data \* 2;

};

// Memoize the result based on 'data'

**const memoizedResult = useMemo(() => calculateResult(data), [data]);**

return (

<div>

<p>Data: {data}</p>

<p>Memoized Result: {memoizedResult}</p>

</div>

);

};

// Usage in a parent component

const ParentComponent = () => {

const [inputData, setInputData] = useState(5);

return (

<div>

<input

type="number"

value={inputData}

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

/>

<ExampleComponent data={inputData} />

</div>

);

};

In this example, the calculateResult function performs an expensive calculation. By using

useMemo, the result is memoized and only recalculated when the data

dependency changes, preventing unnecessary calculations on each render of

ExampleComponent

- **useCallback**

* The useCallback hook in React is used to memoize functions, preventing unnecessary re-creation of the function on every render. This can be beneficial when passing functions as props to child components, especially in scenarios where you want to optimize performance.
* useCallback is a React Hook that memoizes a callback function and returns a memoized version of the function. It also takes an array of dependencies as a second argument, and the memoized function will only be re-created if one of the dependencies has changed.

import React, { useCallback } from 'react';

const MyComponent = ({ onClick }) => {

// Using useCallback to memoize the onClick function

const memoizedOnClick = useCallback(() => {

// Function logic

console.log('Button clicked!');

}, []); // Empty dependency array means the function will not change unless the component is recreated

return (

<button onClick={memoizedOnClick}>

Click me

</button>

);

};

export default MyComponent;

**Example**:

* In this example, the useCallback hook is used to memoize the onClick function, preventing it from being recreated on every render. The empty dependency array ([]) indicates that the memoized function will never change unless the component is recreated. This can be particularly useful when passing the callback as a prop to child components, as it helps optimize performance.
* Keep in mind that using useCallback is generally beneficial in performance optimizations and should be used judiciously. If the function doesn't have any dependencies and doesn't need to be memoized, using a regular function declaration is usually sufficient.

- **useRef**

**Definition**:

* useRef is a React Hook that returns a mutable object called a "ref." This ref object has a current property, which can be assigned any value. The current property persists between renders and can be used to store mutable values.

import React, { useRef } from 'react';

function MyComponent() {

// Creating a ref

**const myRef = useRef(initialValue);**

// Accessing the current value of the ref

console.log(myRef.current);

return (

// JSX and component logic

);

}

export default MyComponent;

**Example**:

In this example, the useRef hook is used to create a ref called myRef with an optional initial value (initialValue). The current property of the ref is then accessed to read or update the mutable value.

import React, { useRef, useEffect } from 'react';

function TextInput() {

// Creating a ref to store the input element

**const inputRef = useRef(null);**

useEffect(() => {

// Focusing on the input element when the component mounts

**inputRef.current.focus();**

}, []);

return (

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

);

}

export default TextInput;

In this example, the inputRef is used to store a reference to the input element. The useEffect hook is then used to focus on the input element when the component mounts, showcasing one common use case for useRef.

2) Higher Order Components (HOC):

- What?

HOCs are functions that take a component and return a new component with additional props or behavior. They are a way to reuse component logic.

- When?

- Why?

- How?

// Higher-Order Component

const withEnhancements = (WrappedComponent) => {

return (props) => {

// Add additional props or behavior here

const enhancedProps = {

...props,

newProp: "This is a new prop",

};

// Return the wrapped component with enhancements

return <WrappedComponent {...enhancedProps} />;

};

};

// Functional Component

const MyComponent = (props) => {

return (

<div>

<p>{props.text}</p>

<p>{props.newProp}</p>

</div>

);

};

// Use the HOC to enhance the functional component

const EnhancedComponent = withEnhancements(MyComponent);

// Now, EnhancedComponent has the additional prop from the HOC

3) Life Cycle Methods of Components:

- Class Components

- Mounting

- Updating

- Unmounting

4) State management (all about data):

- State/Props

- Props drilling

- Context

5) Redux or Zustand:

- How redux works?

Certainly! Redux is a state management library commonly used with React to manage the state of an application in a predictable way. Here's a brief overview of how Redux works in a React application:

1. \*\*Store\*\*: The state of the entire application is stored in a single JavaScript object called the "store." This store is created using the `createStore` function from the Redux library.

```javascript

import { createStore } from 'redux';

const store = createStore(reducer);

```

2. \*\*Reducer\*\*: A reducer is a pure function that takes the current state and an action as arguments, and returns a new state. The reducer specifies how the state should change in response to different actions.

```javascript

const initialState = /\* initial state \*/;

function reducer(state = initialState, action) {

// Handle different actions and return new state

switch (action.type) {

case 'INCREMENT':

return { count: state.count + 1 };

// Other cases...

default:

return state;

}

}

```

3. \*\*Actions\*\*: Actions are plain JavaScript objects that describe what should change in the application's state. They have a `type` property that indicates the type of action and additional payload data if needed.

```javascript

const incrementAction = { type: 'INCREMENT' };

```

4. \*\*Dispatch\*\*: To update the state, you dispatch actions to the Redux store using the `dispatch` method.

```javascript

store.dispatch(incrementAction);

```

5. \*\*Subscribe\*\*: Components that need to be aware of state changes can subscribe to the Redux store. This allows them to be notified whenever the state changes.

```javascript

store.subscribe(() => {

// Update component state or trigger a re-render

});

```

6. \*\*Connect\*\*: React components can connect to the Redux store using the `connect` function provided by the `react-redux` library. This allows components to access the state and dispatch actions.

```javascript

import { connect } from 'react-redux';

const mapStateToProps = (state) => ({

count: state.count,

});

const mapDispatchToProps = (dispatch) => ({

increment: () => dispatch({ type: 'INCREMENT' }),

});

export default connect(mapStateToProps, mapDispatchToProps)(YourComponent);

```

In summary, Redux provides a predictable state container for managing the state of a React application, making it easier to understand and manage the flow of data in a complex application.

- Why?

- When?

- Redux Toolkit (RTK)

(slice is reducer in with inital state)

6) Custom Hooks:

- When to use?

- Code

- Why? (to make code clean, maintainable, readable, reusable)

1. Lazy Loading (Very imp and highly asked):

 lazy loading means that a component or a part of code must get loaded when it is required. It is also referred to as code splitting and data fetching.

- Code splitting

- Chunking

Lazy loading in React allows you to split your code into smaller chunks, loading only the code that is needed for a particular part of your application when it is actually required. This can improve the initial loading time of your application

- Suspense

Suspense is a React feature that helps handle components that are not ready to render yet, such as those that need to fetch data asynchronously. It allows you to "suspend" rendering until the required data or resources are available, providing a better user experience.

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

const Customer = lazy(() => import("./components/User.js"));

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

const App = (()=>{

  return (

   <Suspense fallback="loading">

    <Customer></Customer>

    <Admin></Admin>

   </Suspense>

  );

})

8) Virtual DOM:

- Reconciliation Algorithm

- React Fiber

- Renders

- Diff algorithm

- How does render work?

9) SSR vs CSR (important) :

- What?

- Difference

- SEO and performance (SSR)

****SSR (Server-Side Rendering):****

* With SSR, the initial rendering of the React components occurs on the server side.
* The server sends fully rendered HTML to the client, which improves initial page load performance.
* It is beneficial for SEO (Search Engine Optimization) since search engines can easily index the content.

****CSR (Client-Side Rendering):****

* With CSR, the initial rendering of the React components happens on the client side, usually in the browser.
* The server sends a minimal HTML document and JavaScript bundles to the client, which then handles rendering and data fetching.
* CSR can provide a smoother user experience for subsequent navigation once the initial load is complete.

Choosing between SSR and CSR depends on the specific requirements of your application. SSR is often preferred for better SEO and faster initial page loads, while CSR can provide a more dynamic and responsive user experience after the initial load.

10) Routing (Role-based access control-RBAC)

- react-router

- How do you manage protected routes?

- How do you handle routes?

- query params

- Dynamic routing

11) Testing

- React Testing Library

- Unit Testing

Hack for interview: Try to mention that the code you wrote is testable and try to write test cases.

12) Async Tasks

- API Calls

- useEffect in depth

- Events

- Promises

- setTimeout

13) Reusability, Readability, modularity, testability (Coding Practices)

14) Performance

- Lazy loading

- Asset optimization (how do you optimize js, css code)

- Writing optimized code

- Bundler

- CDN / Server level

- Rendering of components

15) Styling

- Tailwind

- StyleX

- Bootstrap

- Material UI

- Ant UI

- CSS / SCSS

16) Accessibility

17) Security