ReactJs Interview Questions 2024 - Asked by TOP Companies

Updated May 28, 2024

Here you'll find the top 50+ React job interview questions and answers for freshers, beginners, frontend developers, junior developers as well as for experienced developers which might help you cracking your next interview.

ReactJs

ReactJs is a popular JavaScript library for building user interfaces. It is maintained by Facebook, and is widely used for building web applications, mobile apps, and other user interfaces. React allows developers to create reusable components, which can help make large applications easier to manage and maintain. It is designed to be efficient, declarative, and flexible, and can be used to create complex, dynamic user interfaces.

Looking to expand your knowledge on Javascript as well? Check out our comprehensive collection of JavaScript interview question and answers page to help you prepare for your next interview.

*Discover the answers by clicking on the questions.

1. How does React work?

React creates a virtual DOM. When the state changes in a component it first runs a "diffing" algorithm, which identifies what has changed in the virtual DOM. The second step is reconciliation, where it updates the DOM with the results of diff.

2. What are the advantages of using React?

- It is easy to know how a component is rendered, you just need to look at the render function.
- JSX makes it easy to read the code of your components. It is also really easy to see the layout, or how components are plugged/combined.
- You can render React on the server side. This improves SEO and performance.
- It is easy to test.

- You can use React with any framework you wish as it is only a view layer.

3. What is the difference between a Presentational component and a Container component?

Presentational components are concerned with how things look. They generally receive data and callbacks exclusively via props. These components rarely have their own state, but when they do it generally concerns UI state, as opposed to data state.

When your component just receives props and renders them to the page, this is a `stateless component`, for which a pure function can be used. These are also called dumb components or presentational components.

Container components are more concerned with how things work. These components provide the data and behavior to presentational or other container components. They define actions and provide these as callbacks to the presentational components. They are also often stateful as they serve as data sources.

4. What are the differences between a class component and functional component?

- The class component uses ES6 class syntax, and it extends React components with a render method that returns React elements.
- Functional components with hooks are purely JavaScript functions that also return React elements. Before the introduction of hooks, functional components were stateless.

5. What is the difference between state and props?

State is a data structure that starts with a default value when a Component mounts. It may be mutated across time, mostly as a result of user events.

Props (short for properties) are a Component's configuration. They are received from above and immutable as far as the Component receiving them is concerned. A Component cannot change its props, but it is responsible for putting together the props of its child Components. Callback functions can also be passed in as props.

6. What are the different lifecycle methods?

- `componentWillMount` (deprecated) - this is most commonly used for App configuration in your root component.
- `componentDidMount` - here you want to do all the setup you couldn't do without a DOM, and start getting all the data you need. Also if you want to set up eventListeners etc. this lifecycle hook is a good place to do that.
- `componentWillReceiveProps` (deprecated) - this lifecyclye acts on particular prop changes to trigger state transitions.
- `shouldComponentUpdate` - if you're worried about wasted rendersshouldComponentUpdate is a great place to improve performance as it allows you to prevent a rerender if component receives new prop.

shouldComponentUpdate should always return a booleanand based on what this is will determine if the component is rerendered or not.

- `componentWillUpdate` (deprecated) - rarely used. It can be used instead of componentWillReceiveProps on a component that also has shouldComponentUpdate (but no access to previous props).
- `componentDidUpdate` - also commonly used to update the DOM in response to prop or state changes.
- `componentWillUnmount` - enables you can cancel any outgoing network requests, or remove all event listeners associated with the component.

7. Explain React Hooks.

Hooks let you use more of React's features without having to use classes. The first hook that you will most likely encounter is useState. useState is a Hook that lets you add React state to function components. It returns an array with a getter and a setter.

The syntax looks like

```
const [count, setCount] = React.useState(0);

<button onClick={() => setCount(count + 1)}>Increase Count</button>;
```

The equivalent when using a class component would be.

```
this.state = {
  count: 0,
};

<button onClick={() => this.setState({ count: this.state.count + 1 })}>
  Increase Count
</button>;
```

The next hook you will most likely encounter is useEffect. The Effect Hook lets you perform side effects in function components. By passing an empty array as the second argument to useEffect is equivalent to using componentDidMount. If you pass a value to the array it will only call the useEffect function when the value in the array updates.

```
useEffect(() => {
  // do stuff when the component mounts
}, []);
```

8. Where in a React class component should you make an AJAX/API request?

`componentDidMount` is where an AJAX request should be made in a React component. This method will be executed when the component `mounts` (is added to the DOM) for the first time. This method is only executed once during the component's life. Importantly, you can't guarantee the AJAX request will have resolved before the component mounts. If it doesn't, that would mean

that you'd be trying to setState on an unmounted component, which would not work. Making your AJAX request in `componentDidMount` will guarantee that there is a component to update.

9. What are controlled components?

In HTML, form elements such as `<input>`, `<textarea>`, and `<select>` typically maintain their own state and update it based on user input. When a user submits a form the values from the mentioned elements are sent with the form. With React it works differently. The component containing the form will keep track of the value of the input in it's state and will re-render the component each time the callback function e.g. onChange is fired as the state will be updated. An input form element whose value is controlled by React in this way is called a `controlled component`.

10. What are refs used for in React?

Refs are used to get reference to a DOM node or an instance of a component in React. Good examples of when to use refs are for managing focus/text selection, triggering imperative animations, or integrating with third-party DOM libraries. You should avoid using string refs and inline ref callbacks. Callback refs are advised by React.

11. What is a higher order component?

A higher-order component is a function that takes a component and returns a new component. HOC's allow you to reuse code, logic and bootstrap abstraction. The most common is probably Redux's connect function. Beyond simply sharing utility libraries and simple composition, HOCs are the best way to share behavior between React Components. If you find yourself writing a lot of code in different places that does the same thing, you may be able to refactor that code into a reusable HOC.

12. What advantages are there in using arrow functions?

- Scope safety: Until arrow functions, every new function defined its own this value (a new object in the case of a constructor, undefined in strict mode function calls, the base object if the function is called as an "object method", etc.). An arrow function does not create its own this, the this value of the enclosing execution context is used.
- Compactness: Arrow functions are easier to read and write.
- Clarity: When almost everything is an arrow function, any regular function immediately sticks out for defining the scope. A developer can always look up the next-higher function statement to see what the Object is.

13. How would you prevent a class component from rendering?

Returning null from a component's render method means nothing will be displayed, but it does not affect the firing of the component's lifecycle methods.

If the amount of times the component re-renders is an issue, there are two options available. Manually implementing a check in the `shouldComponentUpdate` lifecycle method hook.

```
shouldComponentUpdate(nextProps, nextState){
  const allowRender = true;
  // Do some check here and assign decicison to allowRender
  return allowRender
}
```

Or using React.PureComponent instead of React.Component React.PureComponent implements shouldComponentUpdate() with a shallow prop and state comparison. This enables you to avoid re-rendering the component with the same props and state.

14. When rendering a list what is a key and what is it's purpose?

Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity. The best way to pick a key is to use a string that uniquely identifies a list item among its siblings. Most often you would use IDs from your data as keys. When you don't have stable IDs for rendered items, you may use the item index as a key as a last resort. It is not recommend to use indexes for keys if the items can reorder, as that would be slow.

15. What is the purpose of `super(props)` ?

A child class constructor cannot make use of this until `super()` has been called. Also, ES2015 class constructors have to call `super()` if they are subclasses. The reason for passing props to `super()` is to enable you to access `this.props` in the constructor.

16. What is JSX?

- JSX is a syntax extension to JavaScript and comes with the full power of JavaScript. JSX produces React `elements`.
- You can embed any JavaScript expression in JSX by wrapping it in curly braces. After compilation, JSX expressions become regular JavaScript objects.
- This means that you can use JSX inside of `if` statements and `for loops`, assign it to variables, accept it as arguments, and return it from functions.

17. What is equivalent of the following using React.createElement?

```
const element = <h1 className="greeting">Hello, world!</h1>;
```

```
const element = React.createElement(
  "h1",
  { className: "greeting" },
  "Hello, world!"
);
```

18. What is redux?

- The basic idea of redux is that the entire application state is kept in a single store. The store is simply a javascript object.
- The only way to change the state is by sending actions from your application and then writing reducers for these actions that modify the state.
- The entire state transition is kept inside reducers and should not have any `side-effects`.

19. What is a store in redux?

The store is a javascript object that holds application state. Along with this it also has the following responsibilities:

- Allows access to state via `getState();`.
- Allows state to be updated via `dispatch(action);`.
- Registers listeners via `subscribe(listener);`.
- Handles unregistering of listeners via the function returned by `subscribe(listener)`.

20. Difference between action and reducer.

- Actions are plain javascript objects.
- They must have a type indicating the type of action being performed.
- In essence, actions are payloads of information that send data from your application to your store.

A reducer is simply a pure function that takes the previous state and an action, and returns the next state.

21. What is Redux Thunk used for?

- Redux thunk is middleware that allows you to write action creators that return a function instead of an action.
- The thunk can then be used to delay the dispatch of an action if a certain condition is met. This allows you to handle the asynchronous dispatching of actions.

22. Write a custom hook which can be used to debounce user's input.

```
//hook
const useDebounce = (value, delay) => {
  const [debouncedValue, setDebouncedValue] = useState(value);

  useEffect(() => {
    const timeout = setTimeout(() => {
      setDebouncedValue(value);
    }, delay);

    return () => {
      clearTimeout(timeout);
```

```
    };
  }, [value]);

  return debouncedValue;
};

//example
const Counter = () => {
  const [value, setValue] = useState(0);
  const lastValue = useDebounce(value, 1000);

  return (
    <div>
      <p>
        Current Value: {value} | Debounced Value: {lastValue}
      </p>
      <button onClick={() => setValue(value + 1)}>Increment</button>
    </div>
  );
};
```

23. Write a custom hook to copy text to clipboard.

```
// hook
function useCopyToClipboard(content) {
  const [isCopied, setIsCopied] = useState(false);

  const copy = useCallback(() => {
    navigator.clipboard
      .writeText(content)
      .then(() => setIsCopied(true))
      .then(() => setTimeout(() => setIsCopied(false), 1250))
      .catch((err) => alert(err));
  }, [content]);
  return [isCopied, copy];
}

// usage
export default function App() {
  const [isCopied, copy] = useCopyToClipboard("Text to copy!");
  return <button onClick={copy}>{isCopied ? "Copied!" : "Copy"}</button>;
}
```

24. How to Use the 'useId' Hook to generate unique ids.

- useId does not take any parameters.
- useId returns a unique ID string associated with this particular useId call

in this particular component.

```
//usage
import { useId } from "react";

const App = () => {
  const id = useId();

  return (
    <form>
      <label htmlFor={`email-${id}`}>Email</label>
      <input type="text" id={`email-${id}`} name="email" />

      <label htmlFor={`password-${id}`}>Password</label>
      <input type="password" id={`password-${id}`} name="password" />
    </form>
  );
};

//  Bad Practise - Don't use for key
const id = useId();

return posts.map((post) => <article key={id}>...</article>);
```

25. How to validate Props in React?

- We can use 'prop-types' package
- Earlier, till React v15.5 this was there as part of React iteslf

```
import PropTypes from "prop-types";

function MyComponent({ name }) {
  return <div>Hello, {name}</div>;
}

MyComponent.propTypes = {
  name: PropTypes.string,
};

export default MyComponent;
```

26. Give a practical example of Higher Order Component in react.

- Show a loader while a component waits for data

```
//HOC
function WithLoading(Component) {
  return function WihLoadingComponent({ isLoading, ...props }) {
    if (!isLoading) return <Component {...props} />;
```

```
    return <p>Please wait, fetching your data in no time...</p>;
  };
}
export default WithLoading;

//usage
import UserListComponent from "./UserListComponent.js"; //importing component
import WithLoading from "./withLoading.js"; //importing HOC
const ListWithLoading = WithLoading(UserListComponent); //connect component with HOC

const App = () => {
  const [loading, setLoading] = useState(true);
  const [users, setUsers] = useState([]);
  useEffect(() => {
    //fetch data
    const dataFromApi = ["this is coming from API call", "don't show loader"];
    //at this time loader will be shown in the UI using HOC
    //data fetched successfully
    setUsers([...dataFromApi]);
    setLoading(false);
  }, []);

  return <ListWithLoading isLoading={loading} users={users} />;
};
```

27. Why React's useDeferredValue hook is useful?

- 'useDeferredValue' is a React Hook that lets you defer updating a part of the UI.
- Basically it let you perform the debouncing technique with lesser code.

```
//usage
import { useState, useDeferredValue } from "react";
//userList component takes searchText to fetch user's list
import UserList from "./UserList.js";

export default function App() {
  const [searchText, setSearchText] = useState("");
  //pass searchText as default visible value in useDeferredValue
  const deferredQuery = useDeferredValue(searchText);

  return (
    <>
      <label>
        Search user:
        <input
          value={searchText}
```

```
          onChange={(e) => setSearchText(e.target.value)}
        />
      </label>
      <div>
        <UserList searchText={deferredQuery} />
      </div>
    </>
  );
}
```

29. How to detect 'click' outside React component?

```
export default function OutsideAlerter() {
  const clickMeDivRef = useRef(null);

  useEffect(() => {
    const handleClickOutside = (event) => {
      if (!ref?.current?.contains(event.target)) {
        alert("You clicked outside of me!");
      }
    };

    // Bind the event listener
    document.addEventListener("mousedown", handleClickOutside);

    return () => {
      // Unbind the event listener on clean up
      document.removeEventListener("mousedown", handleClickOutside);
    };
  }, [clickMeDivRef]);

  return <div ref={clickMeDivRef}>Clicked me?</div>;
}
```

30. Why do React component names have to start with capital letters?

In JSX, lowercase tag names are considered to be HTML tags. However, lowercase tag names with a dot (property accessor) aren't.

- `<person />` compiles to React.createElement('person') (html tag)
- `<Person />` compiles to React.createElement(Person)
- `<obj.person />` compiles to React.createElement(obj.person)

```
// Wrong! This is a component and should be in uppercase.
function person(props) {
  // Correct! This usage of <div> is correct because div is a valid element.
  return <div>{props.isLearning ? "Great!" : "Call Mom!"}</div>;
}
```

```
function App() {
  // Wrong! React thinks <person /> is a HTML tag because it's not capitalized.
  return <person isLearning={true} />;
}

// Correct! This is a component and should be capitalized
function Person(props) {
  // Correct! This usage of <div> is correct because div is a valid element.
  return <div>{props.isLearning ? "Great!" : "Call Mom!"}</div>;
}

function App() {
  // Correct! React knows <Person /> is a component because it's capitalized.
  return <Person isLearning={true} />;
}
```

31. What is the difference between npx and npm?

- NPM is a package manager and can be used to install node.js packages.
- NPX is a tool to execute node.js packages.

It doesn't matter whether you installed that package globally or locally. NPX will temporarily install it and run it. NPM also can run packages if you configure a package.json file.

So if you want to check/run a node package quickly without installing it - use NPX.

'create-react-app' is a npm package that is expected to be run only once in a project's lifecycle. Hence, it is preferred to use npx to install and run it in a single step.

```
> npx create-react-app codinn
```

```
npM - Manager
```

```
npX - Execute
```

32. How to set focus on an input field after component mounts on UI?

```
import React, { useEffect, useRef } from "react";

const SearchPage = () => {
  const textInput = useRef(null);

  useEffect(() => {
    textInput.current.focus();
  }, []);

  return (
    <div>
```

```
      <input ref={textInput} type="text" />
    </div>
  );
};
```

33. How to programmatically navigate using latest React Router version?

```
//old - v5
import { useHistory } from "react-router-dom";

function HomeButton() {
  let history = useHistory();
  history.push('/some/path') here
};

//new - v6+
import { useNavigate } from "react-router-dom";

function SignupForm() {
  let navigate = useNavigate();

  async function handleSubmit(event) {
    event.preventDefault();
    await submitForm(event.target);
    navigate("../success", { replace: true });
  }

  return <form onSubmit={handleSubmit}>{/* ... */}</form>;
}

//or
import { redirect } from "react-router-dom";

const loader = async () => {
  const user = await getUser();
  if (!user) {
    return redirect("/login");
  }
};
```

34. What is React state batching? Guess the output.

Given Snippet

```
export default function Counter() {
  const [number, setNumber] = useState(0);

  return (
```

```
    <>
      <h1>{number}</h1>
      <button
        onClick={() => {
          setNumber(number + 1);
          setNumber(number + 1);
          setNumber(number + 1);
        }}
      >
        +3
      </button>
    </>
  );
}
```

Output

- on click of '+3' -> prints '1'
- or update state only once because of state batching concept

Why?

This lets you update multiple state variables without triggering too many re-renders.

But if you want to update anyways? That is - it need to print 3 on click of '+3'.

Pass the callback method to `setNumber`.

```
setNumber((n) => n + 1);

return (
  <>
    <h1>{number}</h1>
    <button
      onClick={() => {
        setNumber((n) => n + 1);
        setNumber((n) => n + 1);
        setNumber((n) => n + 1);
      }}
    >
      +3
    </button>
  </>
);
```

35. How to pass data between sibling components using React router?

Passing data between sibling components of React is possible using React Router `useParams` hook.

Parent component (usually App.js to define routes)

```
<Route path="/user/:id" element={<User />} />
```

```
import { useParams } from "react-router-dom";

const User = () => {
  let { id } = useParams();

  useEffect(() => {
    console.log(`/user/${id}`);
  }, []);

  // .....
};
```

36. How to access a global variable using useContext hook?

```
//1. create context
const GlobalLanguageContext = React.createContext(null);

const App = () => {
  const contextValue = { language: "EN" };

  return (
    //2. connect with all the child components under Provider
    //One time Config - Here in Provider's value prop you can pass
    //the value of your context global variable
    <GlobalLanguageContext.Provider value={contextValue}>
      <Child />
    </GlobalLanguageContext.Provider>
  );
};

const Child = () => {
  //3. use variable
  const { language } = React.useContext(GlobalLanguageContext);
  return <div>Application Language: {language}</div>;
};
```

37. What is the difference between useMemo and useCallback?

- useCallback gives you referential equality between renders for functions. And useMemo gives you referential equality between renders for values.
- useCallback and useMemo both expect a function and an array of dependencies. The difference is that useCallback returns its function when the dependencies change while useMemo calls its function and returns the result.

- useCallback returns its function uncalled so you can call it later, while useMemo calls its function and returns the result

38. Why you should prefer vite over create-react-app?

- Create React App (CRA) has long been the go-to tool for most developers to scaffold React projects and set up a dev server. It offers a modern build setup with no configuration.
- But, we see increased development and build time when the project size increases. This slow feedback loop affects developer's productivity and happiness.
- To address these issues, there is a new front-end tooling in the ecosystem: `Vite`.
- Unlike CRA, Vite does not build your entire application before serving, instead, it builds the application on demand. It also leverages the power of native ES modules, esbuild, and Rollup to improve development and build time.
- Vite is a next-generation, front-end tool that focuses on speed and performance.
- Vite is a development server that provides rich feature enhancements over native ES modules: fast Hot Module Replacement (HMR), pre-bundling, support for typescript, jsx, and dynamic import.
- A build command that bundles your code with Rollup, pre-configured to output optimized static assets for production.

39. What are the advantages of react-router?

- The major advantage of `react-router` is that the page does not have to be refreshed when a link to another page is clicked.
- It also allows us to use browser's `history` feature while preserving the right application view.
- Better user experience, animations and transitions can be easily implemented when switching between different components.
- React Router uses `dynamic routing` to ensure that routing is achieved as it is requested by the user. This also means that all the required components are also rendered without any flashes of white screen or page reload.
- The main components of `react-router` are: `BrowserRouter`, `Routes`, `Route`, `Link`.

40. How can you optimize performance in a ReactJS application?

- One way is to use the shouldComponentUpdate lifecycle method to prevent unnecessary re-renders of a component.
- Another way is to use the PureComponent class, which implements shouldComponentUpdate with a shallow comparison of props and state.
- Additionally, using the React.memo higher-order component can optimize the performance of functional components.

41. Write code for CRUD functionality in ReactJs?

To implement CRUD (create, read, update, delete) functionality in a React application using hooks, you can use the useState hook to manage the state of your application and the useEffect hook to handle side effects, such as making API calls to a server to create, read, update, or delete data.

Here is an example of how you might implement CRUD functionality in a React component using hooks:

```
import React, { useState, useEffect } from "react";

function App() {
  // useState hook to manage the state of our items
  const [items, setItems] = useState([]);

  // useEffect hook to fetch the items from an API
  useEffect(() => {
    fetch("https://my-api.com/items")
      .then((response) => response.json())
      .then((data) => setItems(data));
  }, []);

  // helper function to add a new item
  const addItem = (name) => {
    const newItem = { name };
    setItems([...items, newItem]);
  };

  // helper function to update an item
  const updateItem = (index, name) => {
    const updatedItems = [...items];
    updatedItems[index] = { name };
    setItems(updatedItems);
  };

  // helper function to delete an item
  const deleteItem = (index) => {
    const updatedItems = [...items];
    updatedItems.splice(index, 1);
    setItems(updatedItems);
  };

  // render the items in a list
  return (
    <ul>
      {items.map((item, index) => (
```

```
        <li key={index}>
          {item.name}
          <button onClick={() => updateItem(index, "updated name")}>
            Update
          </button>
          <button onClick={() => deleteItem(index)}>Delete</button>
        </li>
      ))}
      <button onClick={() => addItem("new item")}>Add item</button>
    </ul>
  );
}
```

42. What is a hook in React and why are they useful?

A hook in React is a function that allows developers to use state and other React features without writing a class. This makes it possible to use these features in functional components, which can be easier to write and understand than class-based components.

43. What are some common hooks that are used in React?

Some common hooks that are used in React include useState, useEffect, and useContext. The useState hook allows a functional component to have local state, the useEffect hook allows a functional component to perform side effects, and the useContext hook allows a functional component to access values from the nearest context provider.

44. Can you use hooks inside a class-based component?

No, hooks can only be used inside functional components. If you need to use state or other React features in a class-based component, you will need to use a class component.

45. How do you test a component that uses hooks?

You can test a component that uses hooks by using the act utility from the react-testing-library package. This utility allows you to simulate the effects of React's reconciliation process, which is necessary for hooks to work correctly. You can then use standard Jest or Enzyme assertions to verify the behavior of your component.

46. What is the useEffect hook used for?

The useEffect hook is used for performing side effects in functional components. This can include things like data fetching, setting up subscriptions, or manually changing the DOM. The useEffect hook is called after the component renders, and can be used to ensure that your component stays up-to-date with any relevant data or dependencies.

47. Create a simple custom hook in React?

To create a custom hook in React, you can use the useState hook to add local state to a functional component. Here's an example:

```
import { useState } from "react";

function useCounter() {
  const [count, setCount] = useState(0);

  function increment() {
    setCount(count + 1);
  }

  return { count, increment };
}
```

This hook adds a count state and an increment function to a component. To use this hook in a component, you can call it at the top of the component function, like this:

```
function MyComponent() {
  const { count, increment } = useCounter();

  return (
    <div>
      <p>The count is {count}.</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

Now, whenever the increment button is clicked, the count state will be updated and the component will re-render with the new value.

48. What is the difference between useEffect and useLayoutEffect?

Here is an example of how you might use useEffect and useLayoutEffect in a React component:

```
import React, { useState, useEffect, useLayoutEffect } from "react";

function MyComponent() {
  const [count, setCount] = useState(0);

  // useEffect runs after the render cycle has completed
  useEffect(() => {
    // This code will run every time the component renders,
    // after the render is complete.
    console.log("useEffect running");
  });
```

```
  // useLayoutEffect runs synchronously immediately after the render cycle
  useLayoutEffect(() => {
    // This code will run every time the component renders,
    // before the browser has a chance to paint the update to the screen.
    // Be careful! This can cause visual inconsistencies.
    console.log("useLayoutEffect running");
  });

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

In this example, when the Increment button is clicked, the useEffect hook will run after the component has been updated and re-rendered, whereas the useLayoutEffect hook will run before the update is painted to the screen. This means that if you were to use useLayoutEffect to update the UI, the user might see the UI update before the update is complete, which can cause visual inconsistencies. useEffect, on the other hand, runs after the update is complete and is therefore safer to use for updating the UI.

49. Why virtual DOM is faster to update than real DOM?

    - The virtual DOM is faster to update than the real DOM because React uses a clever technique to minimize the number of updates that need to be made to the real DOM.
    - When you update the virtual DOM, React will compare the new virtual DOM with the old one, determine which parts have changed, and then update the real DOM accordingly. This means that only the parts of the DOM that actually need to be changed are updated, which is much faster than updating the entire DOM every time there is a change.
    - Furthermore, the virtual DOM is implemented in JavaScript, which is generally faster to execute than the native code that is used to manipulate the real DOM.
    - This means that React can perform updates to the virtual DOM quickly, and then use the resulting diff to make efficient updates to the real DOM.

Overall, the use of the virtual DOM allows React to make efficient updates to the UI, which results in a faster and more responsive user experience.

50. Can you explain the difference between a pure and impure function, and why it matters in the context of React?

In React, a pure function is a function that returns the same output for the same set of inputs, regardless of when it is called. An impure function, on the other hand, is a function that may produce different outputs for the same set of inputs, depending on when it is called or other factors.

Here is an example of a pure function in React:

```
function addNumbers(a, b) {
  return a + b;
}
```

This function takes in two numbers, a and b, and returns their sum. This function will always return the same result for the same input, regardless of when it is called or what state the component is in.

Here is an example of an impure function in React:

```
function getRandomNumber() {
  return Math.random();
}
```

This function returns a random number every time it is called. Because the output of this function depends on factors outside of its control (in this case, the current time and a random seed), it is considered an impure function.

In general, pure functions are preferred in React because they are easier to reason about and test. Impure functions, on the other hand, can introduce unpredictable behavior and make your code more difficult to understand.

51. Explain Styled Component in React with example?

Styled Components is a library for React and React Native that allows you to write actual CSS code to style your components. It allows you to write your styles in a declarative way alongside your components, rather than having to maintain separate style sheets.

Here is an example of using Styled Components in a React component:

```
import styled from "styled-components";

const Button = styled.button`
  background: palevioletred;
  border-radius: 3px;
  border: none;
  color: white;
`;

function MyComponent() {
  return <Button>Click me!</Button>;
}
```

In this example, the Button component is styled with a pale violet red background and white text. The styles are written within a template literal and are applied to the button element. When the Button component is rendered, it will have these styles applied to it.

Styled Components allows you to easily customize your styles based on props passed to the component. For example:

```
const Button = styled.button`
  background: ${(props) => (props.primary ? "palevioletred" : "white")};
  border-radius: 3px;
  border: none;
  color: ${(props) => (props.primary ? "white" : "palevioletred")};
`;

function MyComponent() {
  return (
    <div>
      <Button>Click me!</Button>
      <Button primary>Click me!</Button>
    </div>
  );
}
```

In this example, the Button component has a customizable background and text color based on the primary prop. The first Button will have a white background and pale violet red text, while the second Button will have a pale violet red background and white text.

52. Styled-Components vs Inline Styling in React?

It really depends on your specific needs and preferences. Both inline styling and Styled Components have their own advantages and disadvantages, and the best choice for you will depend on the requirements of your project.

Inline styling refers to the practice of applying styles directly to elements using the style attribute. In React, this can be done using the style prop on elements. For example:

```
function MyComponent() {
  return <div style={{ color: "red", fontSize: "20px" }}>Hello, World!</div>;
}
```

One advantage of inline styling is that it can be very simple to use and understand. There's no need to import additional libraries or set up complex configurations. Inline styling also allows you to easily apply styles based on props or state, which can be very useful in certain situations.

However, inline styling can also have some drawbacks. It can make your code more cluttered and harder to read, especially for complex styles. It can also be

more difficult to reuse styles across different components, as you would need to copy and paste the style objects between components.

Styled Components is a library that allows you to define styles using actual CSS syntax and apply them to React components. It allows you to write your styles in a declarative way alongside your components, rather than having to maintain separate style sheets. Here's an example of using Styled Components in a React component:

```
import styled from "styled-components";

const Button = styled.button`
  background: palevioletred;
  border-radius: 3px;
  border: none;
  color: white;
`;

function MyComponent() {
  return <Button>Click me!</Button>;
}
```

One advantage of Styled Components is that it helps to keep your styles organized and modular. Instead of having a separate CSS file for each component, you can define the styles directly within the component itself. This can make it easier to understand and maintain your code, as everything related to the component is kept in one place.

Styled Components also allows you to easily customize your styles based on props passed to the component, and to define complex styles using standard CSS syntax.

However, Styled Components does require an additional library to be installed and imported, which can add some complexity to your project. It may also have a slightly higher learning curve for developers who are not familiar with CSS-in-JS libraries.

Ultimately, the choice between inline styling and Styled Components will depend on your specific needs and preferences. If you're looking for a quick and easy way to apply simple styles, inline styling may be the way to go. If you want more control and flexibility over your styles, and are willing to invest some time in learning a new library, Styled Components may be a better choice.

Guess the Output

53. What is the output of the following code snippet when the "Click me" button is clicked twice?

```
function App() {
```

```
  const [count, setCount] = React.useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

Answer

The output would be "You clicked 2 times".

54. What is the output of the following code snippet when the "Increment age" button is clicked three times?

```
function App() {
  const [state, setState] = React.useState({
    name: "John",
    age: 30,
  });

  return (
    <div>
      <p>
        My name is {state.name} and I am {state.age} years old
      </p>
      <button onClick={() => setState({ ...state, age: state.age + 1 })}>
        Increment age
      </button>
    </div>
  );
}
```

Answer

The output would be "My name is John and I am 33 years old".

55. What is the output of the following code snippet when the "Add hobby" button is clicked twice and then the page is refreshed?

```
function App() {
  const [state, setState] = React.useState({
    name: "John",
    age: 30,
    hobbies: ["reading", "running"],
  });
```

```
  return (
    <div>
      <p>
        My name is {state.name} and I am {state.age} years old
      </p>
      <ul>
        {state.hobbies.map((hobby) => (
          <li key={hobby}>{hobby}</li>
        ))}
      </ul>
      <button
        onClick={() =>
          setState({ ...state, hobbies: [...state.hobbies, "swimming"] })
        }
      >
        Add hobby
      </button>
    </div>
  );
}
```

Answer

The output would be a list with two items: "reading" and "running". The state of the component is reset when the page is refreshed, so the hobbies list would only contain the original two items after the refresh.

56. What is the output of the following code snippet when the "Increment" button is clicked twice and then the "Reset" button is clicked once?

```
import { useEffect, useState } from "react";

function App() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log("count updated");
  }, []);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setCount(0)}>Reset</button>
    </div>
  );
```

```
}
```

Answer The output would be:

```
count updated
```

The `useEffect` hook is called every time the component is rendered, but the dependencies array is empty. This means that the effect will only be run on the initial render of the component, and not on subsequent renders. Since the `Increment` button is clicked twice and the component is re-rendered each time, the effect is not run. However, when the `Reset` button is clicked and the component is re-rendered with a new value for count, the effect is run and the string `count updated` is logged to the console.

57. Create your own useState hook for your new vanilla javascript project. Here is an example of a `useState` hook function that you can use in a `vanilla` JavaScript project:

```
function useState(initialState) {
  let state = initialState;
  function setState(newState) {
    //we can also add few conditions to validate the data.
    state = newState;
    render(); //your custom method to trigger page refresh on state change
  }
  return [state, setState];
}
```

Here is an example of a `useState` hook which might be implemented in a vanilla JavaScript project.

The useState function takes an `initial state` as an argument, and returns an array with the current state and a function to update it.

The state is maintained in a `closure` so that it can be accessed and updated by the component functions.

For the render function on the setState is not something that you would typically include in a vanilla implementation, since is related to some kind of framework, but in your case you may replace it with your specific render function.

58. Write a custom hook which can be used to apply dark or light mode.

```
//custom hook

const useDarkMode = () => {
  const [isDarkMode, setIsDarkMode] = useState(
    localStorage.getItem("isDarkMode") === "true"
  );

  // function to toggle
```

```
  const toggleDarkMode = () => {
    setIsDarkMode((prevMode) => {
      localStorage.setItem("isDarkMode", !prevMode);
      return !prevMode;
    });
  };
  return { isDarkMode, toggleDarkMode };
};

export default useDarkMode;

//usage
// note- there should be different classes to toggle between dark and light mode
// here, ex- 'dark-mode', 'light-mode'

const App = () => {
  const { isDarkMode, toggleDarkMode } = useDarkMode();

  return (
    <div className={isDarkMode ? "dark-mode" : "light-mode"}>
      <button onClick={toggleDarkMode}>
        {isDarkMode ? "Switch to Light Mode" : "Switch to Dark Mode"}
      </button>
    </div>
  );
};
export default App;
```

Here's an example of a custom hook that can toggle between dark and light mode, and also uses local storage to persist the theme across different sessions in a single file.

This is a simple example, you can improve this by adding more styles, and you can also use it in multiple components.

59. How to access the latest value of a text input field in a React component using the 'useRef' hook?

Answer:

You can access the latest value of a text input field in a React component using the `useRef` hook as follows:

```
import React, { useRef } from "react";

const InputComponent = () => {
  const inputRef = useRef(null);

  const handleClick = () => {
```

```
    console.log(inputRef.current.value);
  };

  return (
    <div>
      <input type="text" ref={inputRef} />
      <button onClick={handleClick}>Show value </button>
    </div>
  );
};
```

60. How to create a counter that increments every second using the 'useRef' hook?

Answer:

You can create a counter that increments every second using the `useRef` hook as follows:

```
import React, { useState, useRef, useEffect } from "react";

const Counter = () => {
  const [count, setCount] = useState(0);
  const intervalRef = useRef();

  useEffect(() => {
    intervalRef.current = setInterval(() => {
      setCount((count) => count + 1);
    }, 1000);

    return () => {
      clearInterval(intervalRef.current);
    };
  }, []);

  return <h1>{count}</h1>;
};
```

61. How to implement a simple dropdown menu using the 'useRef' hook?

Answer:

You can implement a simple dropdown menu using the `useRef` hook as follows:

```
import React, { useState, useRef } from "react";

const Dropdown = () => {
  const [isOpen, setIsOpen] = useState(false);
  const dropdownRef = useRef(null);
```

```
  const handleClickOutside = (event) => {
    if (dropdownRef.current && !dropdownRef.current.contains(event.target)) {
      setIsOpen(false);
    }
  };

  useEffect(() => {
    document.addEventListener("mousedown", handleClickOutside);
    return () => {
      document.removeEventListener("mousedown", handleClickOutside);
    };
  }, []);

  return (
    <div ref={dropdownRef}>
      <button onClick={() => setIsOpen(!isOpen)}>Dropdown</button>
      {isOpen && (
        <ul>
          <li>Option 1</li>
          <li>Option 2</li>
          <li>Option 3</li>
        </ul>
      )}
    </div>
  );
};
```

62. Can you explain the React architecture in one sentence?

`Think You Know React? Try Answering this!`

Answer:

React is like a virtual Lego set where each component is a brick that can be combined in various ways to build complex user interfaces.

63. How does React know what to render?

`Think You Know React? Try Answering this!`

Answer:

React uses a virtual DOM to keep track of the state of the application and determine what changes need to be made to the real DOM to reflect those changes.

64. If React was a food, what food would it be?

`Think You Know React? Try Answering this!`

Answer:

React would be like a sushi platter, where each piece of sushi is a component and can be combined in different ways to create a unique and satisfying user experience.

65. How would you explain the concept of "lifting state up" in React?
    Answer:

Lifting state up is the process of moving state from a lower-level component to a higher-level component in the React component hierarchy.

This is done to share state between sibling components that do not have a direct parent-child relationship.

By lifting state up to a common ancestor component, we can avoid prop drilling and make the application more efficient and easier to maintain.

```
import React, { useState } from "react";

function App() {
  const [count, setCount] = useState(0);

  const incrementCount = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <h1>Count: {count}</h1>
      <CounterButton incrementCount={incrementCount} />
      <ResetButton setCount={setCount} />
    </div>
  );
}

function CounterButton({ incrementCount }) {
  return <button onClick={incrementCount}>Increment Count</button>;
}

function ResetButton({ setCount }) {
  return <button onClick={() => setCount(0)}>Reset Count</button>;
}
```

By lifting the count state up to the App component, we can share this state between the CounterButton and ResetButton components without having to pass it down as props through multiple levels of components.

This makes the code cleaner and more efficient, and avoids prop drilling.

66. How does Next.js differ from React.js, and what benefits does it provide for building web applications?

Answer:

Next.js is a framework built on top of React.js that provides additional features for building server-side rendered web applications.

One of the main differences between Next.js and React.js is that Next.js provides server-side rendering out of the box, which allows for faster initial page loads and better search engine optimization.

Next.js also provides features like automatic code splitting and optimized performance for production builds, which can make it easier to build and deploy large-scale applications.

Additionally, Next.js provides support for static site generation, which allows for even faster load times and improved user experiences.

Overall, Next.js provides a more complete solution for building modern web applications than React.js alone, and can be especially beneficial for larger applications that require server-side rendering and other advanced features.

67. Scenario Based -

You have been tasked with creating a form component that allows users to submit data to an API endpoint using React. The form should include the following fields:

```
Name (required)
Email (required)
Message (required)
Checkbox (optional)
```

When the user submits the form, the data should be sent to the API endpoint as a POST request. If the request is successful, the form should be reset and a success message should be displayed. If the request fails, an error message should be displayed.

Write a functional React component that implements the above requirements.

Answer:

```
import React, { useState } from "react";

function ContactForm() {
  const [formData, setFormData] = useState({
    name: "",
    email: "",
    message: "",
    checkbox: false,
  });
  const [submitting, setSubmitting] = useState(false);
  const [submitted, setSubmitted] = useState(false);
  const [error, setError] = useState("");
```

```
const handleChange = (e) => {
  const { name, value, checked } = e.target;
  setFormData({
    ...formData,
    [name]: name === "checkbox" ? checked : value,
  });
};

const handleSubmit = async (e) => {
  e.preventDefault();
  setSubmitting(true);

  try {
    const response = await fetch("https://example.com/api/contact", {
      method: "POST",
      body: JSON.stringify(formData),
      headers: {
        "Content-Type": "application/json",
      },
    });
    if (!response.ok) {
      throw new Error("Error submitting form");
    }
    setFormData({
      name: "",
      email: "",
      message: "",
      checkbox: false,
    });
    setSubmitted(true);
    setError("");
  } catch (err) {
    setError(err.message);
  } finally {
    setSubmitting(false);
  }
};

return (
  <form onSubmit={handleSubmit}>
    <label htmlFor="name">Name</label>
    <input
      type="text"
      id="name"
      name="name"
```

```
          value={formData.name}
          onChange={handleChange}
          required
        />

        <label htmlFor="email">Email</label>
        <input
          type="email"
          id="email"
          name="email"
          value={formData.email}
          onChange={handleChange}
          required
        />

        <label htmlFor="message">Message</label>
        <textarea
          id="message"
          name="message"
          value={formData.message}
          onChange={handleChange}
          required
        />

        <label htmlFor="checkbox">
          <input
            type="checkbox"
            id="checkbox"
            name="checkbox"
            checked={formData.checkbox}
            onChange={handleChange}
          />
          Checkbox
        </label>

        {error && <div>{error}</div>}
        {submitting ? (
          <div>Submitting...</div>
        ) : submitted ? (
          <div>Form submitted successfully!</div>
        ) : (
          <button type="submit">Submit</button>
        )}
      </form>
    );
}
```

```
export default ContactForm;
```

This form component uses the useState hook to manage the form data, submission status, and error state. When the form is submitted, it sends a POST request to the specified API endpoint using fetch(), and displays either a success message, an error message, or a loading spinner depending on the submission status.

68. Scenario Based -

You have been given a requirement to build a simple to-do list application in React. The application should allow the user to add, edit, and delete tasks from the list. Each task should have a title, a description, and a priority level (high, medium, low). When a task is marked as completed, it should be displayed with a strikethrough.

Write a React component that implements the above requirements. You may use any state management library of your choice (e.g. Redux, MobX, Context API).

Answer:

```
import React, { useState } from "react";

function TodoList() {
  const [tasks, setTasks] = useState([]);

  const [newTask, setNewTask] = useState({
    title: "",
    description: "",
    priority: "medium",
    completed: false,
  });

  const handleNewTaskChange = (e) => {
    const { name, value } = e.target;
    setNewTask({
      ...newTask,
      [name]: value,
    });
  };

  const handleAddTask = (e) => {
    e.preventDefault();
    setTasks([...tasks, newTask]);
    setNewTask({
      title: "",
      description: "",
```

```
    priority: "medium",
    completed: false,
  });
};

const handleDeleteTask = (index) => {
  const newTasks = [...tasks];
  newTasks.splice(index, 1);
  setTasks(newTasks);
};

const handleEditTask = (index, updatedTask) => {
  const newTasks = [...tasks];
  newTasks[index] = updatedTask;
  setTasks(newTasks);
};

const toggleCompleted = (index) => {
  const newTasks = [...tasks];
  newTasks[index].completed = !newTasks[index].completed;
  setTasks(newTasks);
};

return (
  <div>
    <form onSubmit={handleAddTask}>
      <input
        type="text"
        name="title"
        placeholder="Task title"
        value={newTask.title}
        onChange={handleNewTaskChange}
      />
      <textarea
        name="description"
        placeholder="Task description"
        value={newTask.description}
        onChange={handleNewTaskChange}
      />
      <select
        name="priority"
        value={newTask.priority}
        onChange={handleNewTaskChange}
      >
        <option value="high">High</option>
        <option value="medium">Medium</option>
```

```
          <option value="low">Low</option>
        </select>
        <button type="submit">Add Task</button>
      </form>

      <ul>
        {tasks.map((task, index) => (
          <li key={index}>
            <h3
              style={{
                textDecoration: task.completed ? "line-through" : "none",
              }}
            >
              {task.title}
            </h3>
            <p>{task.description}</p>
            <div>
              <button onClick={() => handleDeleteTask(index)}>Delete</button>
              <button onClick={() => toggleCompleted(index)}>
                {task.completed ? "Mark Incomplete" : "Mark Complete"}
              </button>
              <button
                onClick={() => {
                  const updatedTask = prompt("Enter updated task:");
                  if (updatedTask) {
                    handleEditTask(index, {
                      ...task,
                      title: updatedTask,
                    });
                  }
                }}
              >
                Edit
              </button>
            </div>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default TodoList;
```

69. Scenario Based -

You have been assigned to create a registration form component in React. The

form should include fields for the user to enter their name, email, and password.

Implement form validation to ensure that the following conditions are met:

- The name field is required and should contain only alphabetic characters.
- The email field is required and should be a valid email address format.
- The password field is required and should have a minimum length of 8 characters.

Write a React component that implements the above requirements.

Solution -

```
import React, { useState } from "react";

function RegistrationForm() {
  const [formData, setFormData] = useState({
    name: "",
    email: "",
    password: "",
  });

  const [errors, setErrors] = useState({
    name: "",
    email: "",
    password: "",
  });

  const handleInputChange = (e) => {
    const { name, value } = e.target;
    setFormData({
      ...formData,
      [name]: value,
    });
  };

  const validateForm = () => {
    let isValid = true;
    const newErrors = {
      name: "",
      email: "",
      password: "",
    };

    if (formData.name.trim() === "") {
      newErrors.name = "Name is required";
      isValid = false;
    } else if (!/^[a-zA-Z]+$/.test(formData.name)) {
```

```
    newErrors.name = "Name should only contain alphabetic characters";
    isValid = false;
  }

  if (formData.email.trim() === "") {
    newErrors.email = "Email is required";
    isValid = false;
  } else if (
    !/^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$/i.test(formData.email)
  ) {
    newErrors.email = "Invalid email address";
    isValid = false;
  }

  if (formData.password.trim() === "") {
    newErrors.password = "Password is required";
    isValid = false;
  } else if (formData.password.length < 8) {
    newErrors.password = "Password should be at least 8 characters long";
    isValid = false;
  }

  setErrors(newErrors);
  return isValid;
};

const handleSubmit = (e) => {
  e.preventDefault();

  if (validateForm()) {
    // Form is valid, proceed with registration logic
    // e.g., submit data to server or perform necessary actions
    console.log("Form submitted successfully:", formData);
  }
};

return (
  <div>
    <h2>Registration Form</h2>
    <form onSubmit={handleSubmit}>
      <div>
        <label>Name:</label>
        <input
          type="text"
          name="name"
          value={formData.name}
```

```
            onChange={handleInputChange}
          />
          {errors.name && <span>{errors.name}</span>}
        </div>

        <div>
          <label>Email:</label>
          <input
            type="email"
            name="email"
            value={formData.email}
            onChange={handleInputChange}
          />
          {errors.email && <span>{errors.email}</span>}
        </div>

        <div>
          <label>Password:</label>
          <input
            type="password"
            name="password"
            value={formData.password}
            onChange={handleInputChange}
          />
          {errors.password && <span>{errors.password}</span>}
        </div>

        <button type="submit">Register</button>
      </form>
    </div>
  );
}

export default RegistrationForm;
```

The component uses React hooks to manage form data and validation errors.

The form fields include name, email, and password, and the component ensures that each field meets the specified validation criteria before allowing form submission.

70. Scenario Based - Controlled Input with Delayed Value Display

Create a React component that consists of an input field.

The component should be controlled, meaning its value is determined by React state.

However, when the user types in the input field, there should be a 2-second

delay before the displayed value updates.

If the user types new characters within this 2-second interval, the display update should be delayed again by 2 seconds.

Only after 2 seconds of inactivity, the displayed value should be updated with the latest input.

Implement the above behavior in the React component.

Solution -

```
import React, { useState, useEffect } from "react";

function DelayedInput() {
  const [inputValue, setInputValue] = useState("");
  const [displayValue, setDisplayValue] = useState("");

  useEffect(() => {
    let timeoutId = null;

    // Update the display value after a 2-second delay
    const delayedUpdateDisplay = () => {
      timeoutId = setTimeout(() => {
        setDisplayValue(inputValue);
      }, 2000);
    };

    // Clear the previous timeout when the user types within 2 seconds
    clearTimeout(timeoutId);

    // Initiate the delayed update only when the user stops typing
    delayedUpdateDisplay();

    // Clean up the timeout on component unmount
    return () => {
      clearTimeout(timeoutId);
    };
  }, [inputValue]);

  const handleInputChange = (e) => {
    setInputValue(e.target.value);
  };

  return (
    <div>
      <h2>Delayed Input</h2>
      <input type="text" value={inputValue} onChange={handleInputChange} />
```

```
      <p>Display Value: {displayValue}</p>
    </div>
  );
}

export default DelayedInput;
```

The component `DelayedInput` uses React hooks such as `useEffect` to achieve the delayed value display functionality. As the user types, the `inputValue` state is updated immediately, but the displayed value (`displayValue`) is updated after a 2-second delay.

71. Scenario Based - Dynamic Nested List Rendering

Create a React component that renders a nested list from a given array of objects. Each object can have a `name` property and a nested `children` property, which is an array of objects with the same structure.

The depth of nesting is unknown and can vary for different objects.

Implement the React component to render the nested list based on the provided data.

Example Data:

```
const data = [
  {
    name: "Item 1",
    children: [
      {
        name: "Subitem 1.1",
        children: [
          { name: "Subsubitem 1.1.1", children: [] },
          { name: "Subsubitem 1.1.2", children: [] },
        ],
      },
      { name: "Subitem 1.2", children: [] },
    ],
  },
  {
    name: "Item 2",
    children: [
      { name: "Subitem 2.1", children: [] },
      { name: "Subitem 2.2", children: [] },
    ],
  },
];
```

Render the nested list using the provided data.

Solution -

```
import React from "react";

function NestedList({ data }) {
  const renderNestedItems = (items) => {
    return (
      <ul>
        {items.map((item, index) => (
          <li key={index}>
            {item.name}
            {item.children.length > 0 && renderNestedItems(item.children)}
          </li>
        ))}
      </ul>
    );
  };

  return (
    <div>
      <h2>Nested List</h2>
      {renderNestedItems(data)}
    </div>
  );
}

export default NestedList;
```

The component `NestedList` recursively renders a nested list using the provided `data` prop. It checks if the current item has children and, if so, calls the `renderNestedItems` function recursively to render the nested list.

72. Scenario Based - Async Data Fetch and Rendering

Create a React component that fetches data from a given API endpoint and renders it as a list.

The API endpoint returns an array of objects, each containing an `id` and a `name`. However, there is a 2-second delay before the API responds.

Implement the React component to fetch the data from the API and display it as a list.

API Endpoint: `https://jsonplaceholder.typicode.com/users`

Solution -

```
import React, { useState, useEffect } from "react";

function DataList() {
```

```
  const [data, setData] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch(
          "https://jsonplaceholder.typicode.com/users"
        );
        const json = await response.json();
        // Simulate a 2-second delay before setting the data
        setTimeout(() => {
          setData(json);
          setLoading(false);
        }, 2000);
      } catch (error) {
        console.error("Error fetching data:", error);
        setLoading(false);
      }
    };

    fetchData();
  }, []);

  return (
    <div>
      <h2>Data List</h2>
      {loading ? (
        <p>Loading...</p>
      ) : (
        <ul>
          {data.map((item) => (
            <li key={item.id}>{item.name}</li>
          ))}
        </ul>
      )}
    </div>
  );
}

export default DataList;
```

The component `DataList` uses `useEffect` hook to fetch data from the provided API endpoint.

While waiting for the API response, it displays a loading message. After a 2-second delay (simulated using `setTimeout`), the fetched data is rendered as a

list.

73. Scenario Based - Managing Focus with useRef

Create a React component that includes an input field and a button. When the button is clicked, it should focus on the input field.

Implement this functionality using the `useRef` hook.

Solution:

```
import React, { useRef } from "react";

function FocusInput() {
  const inputRef = useRef(null);

  const handleFocusButtonClick = () => {
    // Focus on the input field
    if (inputRef.current) {
      inputRef.current.focus();
    }
  };

  return (
    <div>
      <h2>Focus Input</h2>
      <input ref={inputRef} type="text" placeholder="Enter text" />
      <button onClick={handleFocusButtonClick}>Focus Input</button>
    </div>
  );
}

export default FocusInput;
```

In this component, we use the `useRef` hook to create a reference (`inputRef`) to the input element. When the button is clicked, the `handleFocusButtonClick` function is called, and it focuses on the input element by calling `inputRef.current.focus()`. This allows us to programmatically manage the focus of the input field.

74. What is a higher-order component in React?

A higher-order component acts as a container for other components. This helps to keep components simple and enables re-usability. They are generally used when multiple components have to use a common logic.

Solution:

```
import React, { useState, useEffect } from "react";

const CommentList = () => {
  const [comments, setComments] = useState(DataSource.getComments());
```

```
  useEffect(() => {
    const handleChange = () => {
      setComments(DataSource.getComments());
    };

    // Subscribe to changes
    DataSource.addChangeListener(handleChange);

    // Clean up listener
    return () => {
      DataSource.removeChangeListener(handleChange);
    };
  }, []); // Empty dependency array to mimic componentDidMount and componentWillUnmount

  return (
    <div>
      {comments.map((comment) => (
        <Comment comment={comment} key={comment.id} />
      ))}
    </div>
  );
};

export default CommentList;
```

The CommentList component is a React functional component that displays a list of comments. It utilizes the useState hook to manage the comments state and the useEffect hook to subscribe to changes in a global data source (DataSource). When the component mounts, it fetches initial comments and sets up a change listener, updating the state whenever the data source changes. The component renders a list of Comment components based on the comments in its state, each uniquely identified by its id. The change listener is removed when the component unmounts to avoid memory leaks.

75. Explain the role of keys in React lists and the potential issues that can arise without them. Provide a code example to demonstrate their usage.

Answer:

Keys in React lists serve a crucial purpose for efficient rendering and reconciliation.

They enable React to identify which items in a list have changed, been added, or removed, and update the DOM accordingly.

Without keys, React may perform unnecessary re-renders of components, leading to performance issues and potential bugs.

Here are key points to remember:

- **Uniqueness:** Each item in a list should have a unique key.
- **Stability:** Keys should remain consistent across re-renders unless the items themselves change identity.
- **Data Attributes:** Using data attributes from the items themselves is often a good approach for key assignment.

Example:

```
import React from "react";

const TodoList = ({ todos }) => {
  return (
    <ul>
      {todos.map((todo, index) => (
        <li key={todo.id}>{todo.text}</li>
      ))}
    </ul>
  );
};
```

Explanation:

- The `TodoList` component renders a list of `Todo` items.
- Each `Todo` item is assigned a unique key using its `id` property.
- React uses these keys to track changes and apply updates efficiently.

76. Scenario Based - Browser's Local Storage

Imagine you are working on a task management application using React. You want to implement a feature where the user's tasks are saved locally so that even if they refresh the page or close the browser, their tasks remain intact.

How would you achieve this using local storage in React?

Solution:

```
import React, { useState, useEffect } from "react";
import TaskList from "./TaskList";

const App = () => {
  const [tasks, setTasks] = useState([]);

  useEffect(() => {
    const storedTasks = localStorage.getItem("tasks");
    if (storedTasks) {
      setTasks(JSON.parse(storedTasks));
    }
  }, []);
```

```
  useEffect(() => {
    localStorage.setItem("tasks", JSON.stringify(tasks));
  }, [tasks]);

  const addTask = (newTask) => {
    setTasks([...tasks, newTask]);
  };

  const deleteTask = (taskId) => {
    const updatedTasks = tasks.filter((task) => task.id !== taskId);
    setTasks(updatedTasks);
  };

  return (
    <div className="App">
      <h1>Task Management App</h1>
      <TaskList tasks={tasks} addTask={addTask} deleteTask={deleteTask} />
    </div>
  );
};

export default App;
```

In this component, tasks are stored in local storage under the key 'tasks', and they are loaded into the state when the component mounts. Changes to the tasks state are automatically synced with local storage using another useEffect hook.

77. Scenario Based - Browser's Session Storage

Imagine you are developing a shopping cart feature for an e-commerce website using React. You want to implement a feature where the user's cart items are saved temporarily during their session. If they navigate away from the cart page and come back, their cart items should still be there, but if they close the browser, the cart should be reset.

How would you achieve this using session storage in React?

Solution:

```
import React, { useState, useEffect } from "react";
import Cart from "./Cart";

const App = () => {
  const [cartItems, setCartItems] = useState([]);

  useEffect(() => {
    const storedCartItems = sessionStorage.getItem("cartItems");
    if (storedCartItems) {
```

```
      setCartItems(JSON.parse(storedCartItems));
    }
  }, []);

  useEffect(() => {
    sessionStorage.setItem("cartItems", JSON.stringify(cartItems));
  }, [cartItems]);

  const addItemToCart = (newItem) => {
    setCartItems([...cartItems, newItem]);
  };

  const removeItemFromCart = (itemId) => {
    const updatedCartItems = cartItems.filter((item) => item.id !== itemId);
    setCartItems(updatedCartItems);
  };

  return (
    <div className="App">
      <h1>Shopping Cart</h1>
      <Cart
        cartItems={cartItems}
        addItemToCart={addItemToCart}
        removeItemFromCart={removeItemFromCart}
      />
    </div>
  );
};

export default App;
```

In this component, cart items are stored in session storage under the key 'cartItems', and they are loaded into the state when the component mounts. Changes to the cartItems state are automatically synced with session storage using another useEffect hook.