1. Introduction to React JS

- **React** is a **JavaScript library** for building **user interfaces** (UIs), primarily for **single-page applications (SPAs)**.
- Developed by **Facebook** in 2013 and maintained by Meta (formerly Facebook) and a large community.
- React is **component-based**, allowing developers to create reusable and modular pieces of code.
- It uses a **virtual DOM** (Document Object Model) for efficient rendering.
- React follows a **"learn once, write anywhere"** principle, making it adaptable to web and mobile (with React Native).

2. Purpose of React JS

- **Simplifies UI Development**: React's component-based architecture simplifies building and maintaining complex user interfaces.
- **Reusable Components**: Encourages creating modular components that can be reused throughout the application.
- **Efficient Updates**: The **virtual DOM** optimizes rendering and updating of the UI.
- **Declarative Syntax**: Developers can describe how the UI should look based on the application's state.
- **Scalable**: Suitable for small applications to large-scale enterprise projects.
- **Ecosystem**: Integration with tools like Redux, React Router, and Next.js provides a robust ecosystem for development.

3. Challenges with HTML and JavaScript

- **Manual DOM Manipulation**: Updating the DOM directly with vanilla JavaScript becomes complex and error-prone as applications grow.
- **Code Reusability**: Traditional HTML and JavaScript lack efficient ways to create reusable UI components.
- **Performance Issues**: Frequent direct updates to the DOM can lead to performance bottlenecks.
- **State Management**: Managing application state becomes difficult in large applications.

- **Separation of Concerns**: Mixing HTML, CSS, and JavaScript can lead to messy and hard-to-maintain code.

4. Library vs Framework

- **Library**:
  - React is a **library**, not a framework.
  - Provides tools to build UIs but does not dictate the entire application structure.
  - Gives developers flexibility to choose other tools (e.g., routing, state management).
- **Framework**:
  - Frameworks like **Angular** or **Vue** offer a complete solution with rigid structures and rules.
  - Controls more aspects of development and often comes with built-in solutions for routing, state, and testing.

Key Difference:

- **Library** provides flexibility, while a **framework** imposes conventions and patterns.

5. React Features

1. **JSX (JavaScript XML)**:
   - Allows writing HTML-like syntax within JavaScript.
   - Simplifies the creation of React elements and components.
2. **Components**:
   - Building blocks of React applications.
   - Can be **functional** or **class-based**.
3. **Virtual DOM**:
   - React maintains a virtual copy of the real DOM.
   - Updates are first applied to the virtual DOM, and only the necessary parts of the real DOM are updated.
4. **One-Way Data Binding**:
   - Data flows in a single direction (from parent to child components).
   - Simplifies tracking changes and debugging.

5. **State and Props**:
    - **State**: Holds data within a component.
    - **Props**: Allows passing data from parent to child components.
6. **Hooks**:
    - Introduced in React 16.8.
    - Examples: `useState`, `useEffect`, `useContext` allow functional components to manage state and lifecycle methods.
7. **Lifecycle Methods**:
    - For class components: `componentDidMount`, `componentDidUpdate`, `componentWillUnmount`.
8. **React Router**:
    - A library for handling navigation and routing in React applications.

6. Why React JS

1. **Easy to Learn**:
    - Based on JavaScript and simple to understand for developers familiar with JS.
2. **Component Reusability**:
    - Break UI into smaller, reusable pieces for efficient development.
3. **Performance Optimization**:
    - Virtual DOM reduces costly DOM operations and enhances app speed.
4. **Strong Community and Ecosystem**:
    - Large number of libraries, tools, and resources.
5. **Declarative Programming**:
    - Describe **what** the UI should look like, and React takes care of **how** to render it.
6. **SEO Friendly**:
    - Can be combined with tools like **Next.js** for server-side rendering (SSR).
7. **Cross-Platform Development**:
    - Use **React Native** to build mobile applications with React concepts.

7. React Elements

- **React Elements** are the smallest building blocks of a React application.

- Unlike components, elements are **plain objects** representing what the UI should look like.
- Elements are **immutable** (cannot be changed once created).

Example of a React Element:

```
const element = <h1>Hello, world!</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

- The `element` is a React element representing the `<h1>` tag.
- `ReactDOM.render` takes the element and renders it to the DOM.

8. JSX and Its Rules in Depth

What is JSX?

- **JSX (JavaScript XML)** is a syntax extension for JavaScript.
- It looks like HTML but gets transpiled to JavaScript by tools like **Babel**.
- JSX makes writing React components more intuitive.

JSX Syntax Example:

```
const element = <h1>Hello, {name}!</h1>;
```

Key Rules for JSX:

1. **Return Single Parent Element**:
    - JSX expressions must have one root element.

```
// Correct
return (
  <div>
    <h1>Hello</h1>
    <p>World</p>
  </div>
```

```
);

// Incorrect
return (
  <h1>Hello</h1>
  <p>World</p> // This will throw an error.
);
```

2. **Self-Closing Tags**:
3. Elements without children must be self-closed.

```
<img src="logo.png" />
```

4. **JavaScript Expressions in `{}`**:
    - Embed JavaScript expressions within `{}`.

```
const name = "Alice";
return <h1>Hello, {name}!</h1>;
```

5. **Use `className` Instead of `class`**:
    - `class` is a reserved keyword in JavaScript.

```
 <div className="container"></div>
```

6. **Conditional Rendering**:
    - Use ternary operators for simple conditions.

```
const isLoggedIn = true;
return <h1>{isLoggedIn ? "Welcome back!" : "Please log
in"}</h1>;
```

7. **Fragments**:
    - Use `<React.Fragment>` or `<>...</>` to group elements without adding extra nodes to the DOM.

```
<>
<h1>Hello</h1>
<p>World</p>
</>
```

Interview Questions

Introduction to React JS

1. What is React JS? Why is it called a library rather than a framework?
2. Who developed React, and when was it released?
3. Explain the concept of a Virtual DOM in React. How does it differ from the real DOM?
4. What are some typical use cases for React JS?

Purpose of React JS

1. Why is React widely used for building web applications?
2. How does React improve the performance of web applications?

3. What problem does React solve compared to traditional JavaScript development?
4. Can React be used for mobile development? If yes, how?

Challenges with HTML and JS

1. What are some challenges of using plain HTML and JavaScript for building user interfaces?
2. How does React address issues like DOM manipulation and state management in vanilla JavaScript?
3. Why is scalability a problem with HTML and JS, and how does React solve it?

Library and Framework

1. What is the difference between a library and a framework? Is React a library or a framework?
2. How does React compare to frameworks like Angular and Vue.js?
3. What makes React more flexible compared to frameworks?
4. Why is React often referred to as a ‘view’ library in the MVC architecture?

React Features

1. What are the key features of React JS?
2. Explain the concept of `Components` in React and their importance.
3. What are `props` and `state` in React? How do they differ?
4. What is unidirectional data flow in React, and why is it important?
5. How does React implement declarative programming?

Why React JS

1. Why should we use React over other JavaScript libraries or frameworks?
2. Explain React's reusability feature and its impact on development.
3. What makes React easy to integrate with other libraries or frameworks?
4. How does React simplify testing and debugging compared to other frameworks?

React Elements

1. What are React elements, and how do they differ from React components?
2. What is the difference between functional and class components in React?
3. How do React elements describe what you want to see on the screen?
4. Explain the `React.createElement` function and its significance.

JSX and its Rules in Depth

1. What is JSX, and why is it used in React?
2. What are the differences between JSX and HTML?
3. Can browsers understand JSX directly? If not, how is it converted into JavaScript?
4. What are the rules of JSX syntax? Provide examples for each rule.
5. Why must JSX expressions be wrapped in a single parent element?
6. Explain how to use dynamic expressions inside JSX.
7. What is the role of `className` in JSX? Why can't we use the `class` attribute?
8. How are inline styles applied in JSX?
9. What are self-closing tags in JSX, and why are they required?
10. Explain the use of curly braces `{}` in JSX.

Components and Its Purposes

Definition:

- Components are the building blocks of a React application.
- They enable the UI to be divided into reusable, independent pieces, each encapsulating its logic and structure.

Types of Components:

1. Functional Components: Stateless, represented as functions.

```
function Welcome(props) {
    return <h1>Hello, {props.name}!</h1>;
}
```

2. Class Components: Stateful, represented as ES6 classes.

```
class Welcome extends React.Component {
    render() {
        return <h1>Hello, {this.props.name}!</h1>;
    }
}
```

Purpose:

- **Reusability**: Break down the application into smaller, manageable pieces.
- **Maintainability**: Easier to read, debug, and test.
- **Separation of Concerns**: Encapsulates logic and UI in one unit.

Props

Definition:

- Props (short for properties) are read-only data passed from a parent component to a child component.

Key Points:

- Immutable: Cannot be modified by the child component.
- Used for passing data and event handlers.

Example:

```
function Greeting(props) {
    return <h1>Hello, {props.name}!</h1>;
}

function App() {
    return <Greeting name="Alice" />;
}
```

- Here, `name` is a prop passed to the `Greeting` component.

Development of Static Web Page Using React

Steps:

1. Install React and create a project using `create-react-app` or `Vite`.
2. Create components for each section of the webpage (e.g., Header, Footer, MainContent).
3. Use JSX to define the structure and CSS for styling.

Example:

```
function Header() {
    return <header><h1>Welcome to My Website</h1></header>;
}

function MainContent() {
    return (
        <main>
            <p>This is a static webpage created using React.</p>
        </main>
    );
}

function Footer() {
    return <footer>Copyright 2024</footer>;
}
```

```
function App() {
    return (
        <div>
            <Header />
            <MainContent />
            <Footer />
        </div>
    );
}
```

Introduction to Hooks and Its Purpose

Definition:

- Hooks are functions that let you "hook into" React state and lifecycle features from functional components.

Purpose:

- Replace class components for state and lifecycle management.
- Simplify component logic and make it reusable.

Common Hooks:

1. **useState**: Manage state.
2. **useEffect**: Handle side effects like data fetching.
3. **useContext**: Access context values.

useState()

Definition:

- A Hook used to add state to functional components.

Syntax:

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

Example:

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

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

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

State and setState

State:

- An object that holds data that can change over time.
- Used in both class and functional components.

setState:

- A method used to update the state in class components or returned by `useState` in functional components.

Example:

```
// Functional Component
function Counter() {
    const [count, setCount] = useState(0);

    return (
        <button onClick={() => setCount(count + 1)}>
```

```
            Count: {count}
        </button>
    );
}
```

```
// Class Component
class Counter extends React.Component {
    constructor(props) {
        super(props);
        this.state = { count: 0 };
    }

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

    render() {
        return <button onClick={this.increment}>Count:
{this.state.count}</button>;
    }
}
```

Profile Page with Light and Dark Theme

Steps:

1. Create a context for theme management.
2. Toggle between light and dark themes using a button.
3. Apply conditional styles based on the theme.**Example**:

```
import React, { createContext, useContext, useState } from 'react';

const ThemeContext = createContext();
```

```
function ThemeProvider({ children }) {
    const [theme, setTheme] = useState('light');

    const toggleTheme = () => {
        setTheme(theme === 'light' ? 'dark' : 'light');
    };
    return (
        <ThemeContext.Provider value={{ theme, toggleTheme }}>
            {children}
        </ThemeContext.Provider>
    );
}
```

```
function ProfilePage() {
    const { theme, toggleTheme } = useContext(ThemeContext);

    return (
        <div className={theme === 'light' ? 'bg-white text-black' :
'bg-black text-white'}>
            <h1>Profile Page</h1>
            <button onClick={toggleTheme}>Toggle Theme</button>
        </div>
    );
}

function App() {
    return (
        <ThemeProvider>
            <ProfilePage />
        </ThemeProvider>
    );
}
```

setState() in Depth

Key Points:

- `setState` in class components can be asynchronous.

- Use the functional form of `setState` for updates dependent on the previous state.

Example:

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

Batching:

- React batches multiple `setState` calls for performance optimization.

Pitfalls:

- Avoid directly mutating the state.

```
// Incorrect
this.state.count = this.state.count + 1;
// Correct
this.setState({ count: this.state.count + 1 });
```

Interview Questions

1. Components and Their Purposes

1. What are React components, and how are they classified? Provide examples.
2. How do functional components differ from class components? Why are functional components preferred?
3. Can a React component return multiple elements? How can you achieve this?
4. What is the purpose of `defaultProps` in a React component?
5. Explain the lifecycle methods available in class components. Are they available in functional components?

2. Props

1. What are props in React? How are they different from state?
2. How do you pass props to a child component? Provide an example.
3. What happens if a prop is not passed to a component, but it is required? How can you handle this?
4. Can you modify props inside a component? Why or why not?
5. What are prop types in React, and how do you validate them?

3. Development of Static Web Page Using React

1. How do you structure a static web page in React?
2. What is the role of JSX in developing static pages using React?
3. How can you apply styles to React components? Name three methods.
4. How do you include images and assets in a React static web page?
5. Explain the importance of using reusable components while building a static web page

4. Introduction to Hooks and Their Purpose

1. What are React hooks, and why were they introduced?
2. Name three commonly used hooks in React and explain their purpose.
3. Can you use hooks in class components? Why or why not?
4. What are the rules of hooks? Why are they important?
5. How do hooks simplify state and lifecycle management in React?

5. useState()

1. What is the purpose of the `useState` hook in React?
2. How do you initialize state with the `useState` hook? Provide an example.
3. Can you update multiple state variables using `useState`? How?
4. What happens if you call the `setState` function without updating the state value?
5. Explain how you can use the `useState` hook to toggle a boolean value.

6. State and setState

1. What is state in React, and how is it different from props?
2. How does `setState` work in class components? How does it differ from the `useState` hook?
3. Why is state considered immutable in React? How do you update it correctly?
4. Explain the difference between synchronous and asynchronous updates to state.
5. How would you update state based on the previous state?

7. Profile Page with Light and Dark Theme

1. How can you implement a light and dark theme toggle in React?
2. What are some best practices for managing theme-related state in a React application?
3. How can you persist a user's theme preference across sessions?
4. Explain how CSS variables can help in creating a theme system in React.
5. How would you structure the components for a profile page that supports theme toggling?

8. setState() in Depth

1. Explain how `setState` works in React. Why is it asynchronous?
2. How can you pass a callback to `setState` to execute code after state updates?
3. What is the difference between updating state with an object and with a function in `setState`?
4. How do batched updates work in React? Provide an example.
5. What are common pitfalls when using `setState`, and how can you avoid them?

Consuming the API

Definition: Consuming an API involves fetching data from a server and utilizing it in your React components to display or manipulate the data dynamically.

Steps to Consume an API in React:

1. Use `fetch()` or `axios` to make an API call.
2. Handle asynchronous operations with `async/await` or `.then()`.
3. Store the data in state variables using `useState`.
4. Render the data in the component.

Example:

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

const UsersList = () => {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/users')
      .then((response) => response.json())
      .then((data) => {
        setUsers(data);
        setLoading(false);
      })
      .catch((error) => console.error('Error fetching users:',
error));
  }, []);

  if (loading) return <p>Loading...</p>;

  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
```

```
  );
};

export default UsersList;
```

Component and Its Relationship

Definition: Components are the building blocks of a React application. Relationships between components often involve parent-child or sibling communication.

Parent to Child: Passing data via `props`. **Child to Parent:** Using callback functions passed as props. **Sibling Communication:** Managed via a shared parent component or Context API.

Example:

```
const Parent = () => {
  const [message, setMessage] = useState('Hello from Parent');

  return (
    <div>
      <Child message={message} />
    </div>
  );
};

const Child = ({ message }) => {
  return <p>{message}</p>;
};
```

Conditional Rendering

Definition: Render specific components or elements based on certain conditions.

Techniques:

1. **Ternary Operator**
2. **Short-Circuiting**

3. `if` **Statements in Functions**

Example:

```
const LoginStatus = ({ isLoggedIn }) => {
  return (
    <div>
      {isLoggedIn ? <p>Welcome back!</p> : <p>Please log in.</p>}
    </div>
  );
};
```

List Rendering

Definition: Render a list of data dynamically using JavaScript's array methods like `map()`.

```
Example:
const Products = ({ items }) => {
  return (
    <ul>
      {items.map((item) => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  );
};
```

List and Keys

Definition: Keys are unique identifiers assigned to elements when rendering lists. React uses keys to optimize updates.

Best Practices:

1. Use a unique `id` if available.
2. Avoid using indexes as keys unless the list is static.

Example:

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

Data Sharing Between Components

Techniques:

1. **Props:** Pass data from parent to child.
2. **Callback Functions:** Allow children to send data back to parents.
3. **Context API or State Management Libraries:** Share data across multiple components.

Example:

```
const Parent = () => {
  const [data, setData] = useState('Hello from Parent');

  return (
    <Child data={data} onUpdate={(newData) => setData(newData)} />
  );
};
```

```
const Child = ({ data, onUpdate }) => {
  return (
    <div>
      <p>{data}</p>
      <button onClick={() => onUpdate('Data updated by
Child')}>Update</button>
    </div>
  );
};
```

Context API

Definition: Context API provides a way to pass data through the component tree without manually passing props at every level.

Steps:

1. Create a context using `React.createContext`.
2. Wrap the provider around the components that need access.
3. Consume the context in child components.

Example:

```
import React, { createContext, useContext, useState } from 'react';

const ThemeContext = createContext();

const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState('light');
  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};
```

```
const Header = () => {
  const { theme, setTheme } = useContext(ThemeContext);
  return (
    <button onClick={() => setTheme(theme === 'light' ? 'dark' :
'light')}>
      Toggle Theme
    </button>
  );
};

const App = () => (
  <ThemeProvider>
    <Header />
  </ThemeProvider>
);

export default App;
```

Interview Questions

1. Integrating React With APIs

1. How can you fetch data from an API in React? Explain with an example.
2. What is the role of the `useEffect` hook in API integration?
3. What are the common methods used to handle asynchronous API calls in React?
4. How do you display a loading spinner or placeholder while fetching data from an API?
5. What is the difference between fetching data on the client side and server side?

2. Consuming the API

1. How do you handle errors while consuming an API in React?
2. What is the difference between `fetch` and `axios` when consuming an API in React?
3. How can you consume paginated data from an API in React?
4. How would you structure a React component to display data fetched from an API?
5. What are CORS issues, and how can you handle them when consuming APIs?

3. Component and Its Relationship

1. What is the difference between parent and child components in React?
2. How can a parent component pass data to a child component?
3. How do sibling components communicate in React?
4. What is the purpose of "lifting state up" in React?
5. How would you manage relationships between components in a large React application?

4. Conditional Rendering

1. What is conditional rendering in React, and why is it useful?
2. How can you use the ternary operator for conditional rendering in React?
3. Explain how logical `&&` can be used for conditional rendering.
4. How do you conditionally render components based on API data?
5. What is the difference between rendering `null` and `false` in React?

5. List Rendering

1. How do you render a list of items in React using the `map` function?
2. What are the key attributes in list rendering, and why are they important?
3. How can you render a list of components conditionally?
4. How do you handle dynamic data in list rendering?
5. How would you handle user interaction (e.g., click events) on list items?

6. List and Keys

1. What is the purpose of the `key` prop in React lists?
2. What happens if you don’t provide a unique `key` in a list?
3. Can the index of an array be used as a `key`? When is it appropriate to do so?
4. How does React use the `key` prop to optimize rendering?
5. What are the common pitfalls when assigning `key` props in React?

7. Data Sharing Between Components

1. How can you pass data from a parent component to a child component?
2. What is "prop drilling," and how can it be avoided in React?
3. How do you share state between sibling components in React?

4. What is the difference between context and props for data sharing?
5. How can custom hooks be used to share data between components?

8. Context API

1. What is the React Context API, and when should you use it?
2. How do you create and consume a context in React?
3. What are the differences between `React.createContext` and `useContext`?
4. How can you provide default values for context in React?
5. What are some common use cases for the Context API?

---

1. useContext() Hook

The `useContext` hook allows you to consume a React Context in functional components.

Example:

```
import React, { createContext, useContext } from 'react';

const ThemeContext = createContext();

const ThemedComponent = () => {
  const theme = useContext(ThemeContext);
  return <div style={{ backgroundColor: theme === 'dark' ? '#333' :
'#fff', color: theme === 'dark' ? '#fff' : '#000' }}>Theme:
{theme}</div>;
};

const App = () => {
  return (
    <ThemeContext.Provider value="dark">
      <ThemedComponent />
    </ThemeContext.Provider>
  );
};

export default App;
```

Key Points:

- Simplifies prop drilling by directly accessing context.
- Should be used for shared global state (e.g., themes, auth).

2. Form Management

Managing forms involves handling user input, validation, and submission.

Controlled Form Example:

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

const ControlledForm = () => {
  const [email, setEmail] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    alert(`Submitted Email: ${email}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>Email:
        <input type="email" value={email} onChange={(e)
=> setEmail(e.target.value)} />
      </label>
      <button type="submit">Submit</button>
    </form>
  );
};

export default ControlledForm;
```

Key Points:

- Use controlled inputs to bind the input field value to the component state.
- Validation can be handled during `onChange` or `onSubmit`.

3. useRef() Hook

The `useRef` hook is used to directly access and manipulate DOM elements or persist a mutable value without causing re-renders.

Example:

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

const FocusInput = () => {
  const inputRef = useRef();

  const handleFocus = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} type="text"
placeholder="Type here..." />
      <button onClick={handleFocus}>Focus
Input</button>
    </div>
  );
};
```

export default FocusInput;

Key Points:

- Does not trigger re-renders when updated.
- Commonly used for accessing DOM elements or storing mutable variables.

4. Controlled and Uncontrolled Components

Controlled Components:

- Value is controlled by React state.

Example:

```
const ControlledComponent = () => {
  const [value, setValue] = useState('');
  return <input value={value} onChange={(e) =>
setValue(e.target.value)} />;
};
```

Uncontrolled Components:

- Value is managed by the DOM.

Example:

```
const UncontrolledComponent = () => {
  const inputRef = useRef();

  const handleClick = () => {
    alert(inputRef.current.value);
  };

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

5. React Routing

React Router allows navigation between views without refreshing the page.

Example:

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

const Home = () => <h2>Home</h2>;
const About = () => <h2>About</h2>;

const App = () => (
  <Router>
    <nav>
      <Link to="/">Home</Link> | <Link
to="/about">About</Link>
    </nav>
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
    </Routes>
  </Router>
);

export default App;
```

Key Points:

- Use `Link` for navigation.
- Define routes using `Route` within `Routes`.

6. Lifecycle of Components

React class components have lifecycle methods grouped into phases:

- **Mounting:** `constructor`, `componentDidMount`
- **Updating:** `componentDidUpdate`
- **Unmounting:** `componentWillUnmount`

In functional components, the `useEffect` hook handles lifecycle-like behavior.

7. Phases of Lifecycle

1. **Mounting:** When a component is created and inserted into the DOM.
2. **Updating:** When the state or props change.
3. **Unmounting:** When the component is removed from the DOM.

8. useEffect() Hook

The `useEffect` hook manages side effects in functional components.

Example:

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

const Timer = () => {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const timer = setInterval(() => setCount((prev) => prev +
1), 1000);
    return () => clearInterval(timer); // Cleanup function
  }, []);

  return <div>Timer: {count}s</div>;
};

export default Timer;
```

Key Points:

- Runs after render by default.
- Dependencies array controls when the effect runs.
- Cleanup function handles resource cleanup.

Interview Questions

1. useContext() Hook

1. What is the purpose of the `useContext` hook in React?
2. How does `useContext` differ from passing props in React?
3. Explain the steps to use the `useContext` hook with an example.
4. What are some common use cases for the `useContext` hook?
5. How does `useContext` work with nested providers in React?

2. Form Management

1. What are the differences between controlled and uncontrolled components in React forms?
2. How do you handle form validation in React?
3. What are the advantages of using libraries like `Formik` or `React Hook Form` for form management?
4. How can you dynamically add or remove form fields in React?
5. How do you handle submitting forms and processing the submitted data in React?

3. useRef() Hook

1. What is the purpose of the `useRef` hook in React?
2. How can you use `useRef` to access a DOM element?
3. What is the difference between `useRef` and `useState`?
4. How would you use `useRef` to maintain state across renders without causing re-renders?
5. Explain a practical use case of `useRef` in a functional component.

4. Controlled and Uncontrolled Components

1. What are controlled components in React, and how do they work?
2. What are uncontrolled components, and when should they be used?
3. Explain the pros and cons of controlled vs. uncontrolled components.
4. How do you handle default values in controlled and uncontrolled components?
5. How can you manage form elements efficiently using controlled components?

5. React Routing

1. What is React Router, and why is it used?
2. How do you set up routing in a React application?
3. What is the difference between `<BrowserRouter>` and `<HashRouter>`?
4. Explain how to implement nested routes in React Router.
5. How do you handle route parameters and query strings in React Router?

6. Lifecycle of Components

1. What is the component lifecycle in React?
2. What is the purpose of the `componentDidMount` lifecycle method in class components?
3. How does React's lifecycle differ between functional and class components?
4. What is the difference between `componentDidUpdate` and `componentWillUnmount`?
5. How has the introduction of hooks changed the way lifecycle events are managed?

7. Phases of Lifecycle

1. What are the three main phases of the React component lifecycle?
2. What happens during the "mounting" phase of a React component?
3. What is the role of the "updating" phase in the lifecycle?
4. What happens during the "unmounting" phase of a component?
5. How does React optimize the rendering process during lifecycle phases?

8. useEffect() Hook

1. What is the purpose of the `useEffect` hook in React?
2. How can you use `useEffect` to mimic the behavior of `componentDidMount`?
3. Explain the difference between running `useEffect` with no dependencies, an empty array, and specific dependencies.
4. How do you clean up side effects in `useEffect`?
5. What are some common pitfalls to avoid when using the `useEffect` hook?

Redux

What is Redux? Redux is a state management library often used with React to manage the application's state in a predictable and centralized way. Redux provides a global store that holds the application's state, making it easier to manage and debug complex applications.

Core Principles of Redux:

1. **Single Source of Truth:** The state of the entire application is stored in a single object called the store.
2. **State is Read-Only:** The only way to change the state is by dispatching an action.
3. **Changes are Made with Pure Functions:** Reducers are pure functions that specify how the state changes in response to actions.

Example:

```
import { createStore } from 'redux';

// Reducer function
const counterReducer = (state = { count: 0 }, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
```

```
};

// Create store
const store = createStore(counterReducer);

// Subscribe to store
store.subscribe(() => console.log(store.getState()));

// Dispatch actions
store.dispatch({ type: 'INCREMENT' }); // { count: 1 }
store.dispatch({ type: 'DECREMENT' }); // { count: 0 }
```

Redux Thunk

What is Redux Thunk? Redux Thunk is a middleware that allows you to write action creators that return a function instead of an action. This is especially useful for handling asynchronous logic, like fetching data from an API.

Example:

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import axios from 'axios';

// Action creator
const fetchData = () => {
  return async (dispatch) => {
    const response = await axios.get('https://api.example.com/data');
    dispatch({ type: 'FETCH_DATA_SUCCESS', payload: response.data });
  };
};

// Reducer
const dataReducer = (state = { data: [] }, action) => {
  switch (action.type) {
    case 'FETCH_DATA_SUCCESS':
      return { ...state, data: action.payload };
    default:
```

```
      return state;
  }
};

// Store
const store = createStore(dataReducer, applyMiddleware(thunk));

// Dispatch async action
store.dispatch(fetchData());
```

useReducer() Hook

What is useReducer()? useReducer is a React hook for managing state that is more complex than useState. It is similar to Redux reducers and is suitable for local component state.

Example:

```
import React, { useReducer } from 'react';

const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    default:
      return state;
  }
}


function Counter() {
  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 Counter;
```

Class Component

What is a Class Component?

Class components are one of the ways to define a React component using ES6 classes. They can manage their own state and lifecycle methods.

```
Example:
import React, { Component } from 'react';

class Welcome extends Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}

export default Welcome;
```

State with Class Components

Example of State in Class Components:

```
import React, { Component } from 'react';

class Counter extends Component {
  state = { count: 0 };

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

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}

export default Counter;
```

Lazy Loading

What is Lazy Loading? Lazy loading is a technique to load components or resources only when they are needed, which improves performance.

Example:

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

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

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
```

```
      <LazyComponent />
    </Suspense>
  );
}

export default App;
```

Error Boundaries

What are Error Boundaries?

Error boundaries catch JavaScript errors in child components and display a fallback UI instead of crashing the application.

Example:

```
import React, { Component } from 'react';

class ErrorBoundary extends Component {
  state = { hasError: false };

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}

export default ErrorBoundary;
```

React Profiler

What is the React Profiler?

The React Profiler is a tool to measure the performance of React applications by identifying slow components.

How to Use: Enable the React DevTools and go to the "Profiler" tab.

Example:

```
import React, { Profiler } from 'react';

function App() {
  const onRenderCallback = (
    id, phase, actualDuration, baseDuration, startTime, commitTime,
interactions
  ) => {
    console.log(`${id} rendered in ${actualDuration}ms`);
  };

  return (
    <Profiler id="MyComponent" onRender={onRenderCallback}>
      <MyComponent />
    </Profiler>
  );
}

export default App;
```

Functional Component vs Class Component

Differences:

Feature	Functional Component	Class Component
Syntax	Simple function	ES6 class
State Management	useState, useReducer	this.state and setState
Lifecycle Methods	useEffect	Lifecycle methods like componentDidMount
Performance	Faster and simpler	Slightly heavier

Example Comparison:

Functional Component::

```
function Greeting({ name }) {
  return <h1>Hello, {name}</h1>;
}
```

```
Class Component:
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Interview Questions

1. Redux

1. What is Redux, and why is it used in React applications?
2. Explain the three core principles of Redux.
3. What is the role of actions, reducers, and the store in Redux?
4. How do you integrate Redux with a React application?
5. What are the limitations of Redux, and when should you avoid using it?

2. Redux Thunk

1. What is Redux Thunk, and why is it used?
2. How does Redux Thunk help in handling asynchronous actions?
3. Explain the middleware concept in Redux with respect to Redux Thunk.
4. Write an example of using Redux Thunk to fetch data from an API.
5. What are the advantages and disadvantages of using Redux Thunk?

3. useReducer() Hook

1. What is the `useReducer` hook in React, and how does it differ from `useState`?
2. Explain the structure of a reducer function used with `useReducer`.
3. How would you implement a counter using the `useReducer` hook?
4. What are some scenarios where `useReducer` is preferred over `useState`?
5. Can you combine `useReducer` with `useContext`? If yes, how?

4. Class Component

1. What are class components in React, and how do they differ from functional components?
2. How do you define and use state in a class component?
3. What is the purpose of lifecycle methods in class components?
4. How do you handle events in class components?
5. Write a simple example of a class component with state and props.

5. State with Class Components

1. How do you initialize state in a React class component?
2. What is the difference between `this.state` and `this.setState()`?
3. How does `setState()` work asynchronously in class components?
4. How can you update state based on the previous state in a class component?
5. What are common pitfalls when working with state in class components?

6. Lazy Loading

1. What is lazy loading in React, and why is it used?
2. How do you implement lazy loading with React's `React.lazy()` and `Suspense`?

3. What are the performance benefits of lazy loading in a React application?
4. How do you handle errors in lazy-loaded components?
5. Write an example of lazy loading a component in a React application.

7. Error Boundaries

1. What are error boundaries in React, and what problem do they solve?
2. How do you create an error boundary using a class component?
3. What lifecycle method is used to catch errors in error boundaries?
4. Can error boundaries catch errors in event handlers or async code? Why or why not?
5. Write an example of implementing an error boundary in a React application.

8. React Profiler

1. What is the React Profiler, and how is it used?
2. How can you identify performance bottlenecks using the React Profiler?
3. What are the key metrics tracked by the React Profiler?
4. How do you enable the React Profiler in a React application?
5. What are some best practices for optimizing a React application using the Profiler?

9. Functional Component Vs Class Component

1. What are the key differences between functional and class components in React?
2. How has the introduction of hooks impacted the use of class components?
3. Which component type is preferred for new applications, and why?
4. Can functional components have lifecycle methods? How?
5. What are the advantages of functional components over class components?