---

1. React Basics

1. What is React? Why use it?

React is a **declarative, component-based JavaScript library** for building user interfaces. It's not a full-fledged framework like Angular or Vue, but a library focused on the **view layer** of an application. It was developed by Facebook and is widely used for building **single-page applications (SPAs)** and mobile apps (with React Native).

You'd use React because it makes building complex UIs manageable and efficient. Its component-based architecture promotes **reusability** and easier maintenance. The **Virtual DOM** ensures high performance by only updating the necessary parts of the real DOM. Plus, its **declarative approach** makes your code more predictable and easier to debug, as you just describe what the UI should look like for a given state, and React handles the rest.

2. Difference between React and other frameworks (Angular, Vue).

Feature	React	Angular	Vue
Type	Library	Framework	Framework
Architecture	Component-based, Unopinionated	Component-based, Opinionated (MVC)	Component-based, Progressive
Data Binding	One-way	Two-way	Two-way
DOM	Virtual DOM	Real DOM	Virtual DOM
Language	JavaScript/JSX	TypeScript	JavaScript
Learning Curve	Moderate	Steep	Gentle

Ecosystem	Large, requires third-party libraries	All-in-one solution	Growing, flexible

- **React** is a **library** and is **unopinionated**. It focuses on the view layer, so you often need to choose other libraries for routing (React Router) or state management (Redux).
- **Angular** is a **full-fledged framework** and is **opinionated**. It provides a complete solution with built-in features for routing, state management, and more. It heavily uses **TypeScript** and a more structured architecture.
- **Vue** is also a **framework**, but more **progressive**. You can use it for simple enhancements to a web page or for building complex SPAs. It's often considered easier to learn than React or Angular.

3. What are the key features of React?

- **Component-Based Architecture:** UIs are built from encapsulated, reusable components.
- **Declarative:** You describe the desired state of the UI, and React figures out how to get there.
- **Virtual DOM:** React uses an in-memory representation of the DOM to efficiently update the UI.
- **JSX:** A syntax extension for JavaScript that looks like HTML, making it easier to write UI code.
- **One-Way Data Binding:** Data flows from parent to child components via props, making the data flow predictable.
- **React Hooks:** Functions that let you use state and other React features in functional components.

4. What is JSX in React? Why use it?

JSX (JavaScript XML) is a syntax extension for JavaScript that allows you to write HTML-like code directly within your JavaScript files. It's not a new language; it's just a syntactic sugar that gets compiled into regular JavaScript by tools like Babel.

You use JSX because it makes writing and understanding UI code more intuitive and visually appealing. It allows you to define the structure of your UI (HTML) and its logic (JavaScript) in the same place.

Example:

JavaScript

```
// Without JSX
const element = React.createElement('h1', { className: 'greeting' }, 'Hello, world!');

// With JSX
```

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

5. What are components in React? Types (functional vs class).

A **component** is an independent, reusable piece of code that defines the structure and behavior of a part of the UI. Think of a UI as being composed of building blocks, and each block is a component (e.g., a button, a header, a user profile card).

Functional Components

These are simple JavaScript functions that accept props as an argument and return a React element. With the introduction of Hooks, they are the preferred way to write new components.

Example:

```
JavaScript

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

Class Components

These are ES6 classes that extend React.Component and have a render() method that returns a React element. They were traditionally used for components that needed to manage state or lifecycle methods.

Example:

```
JavaScript

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

6. What is the difference between Element and Component in React?

- **React Component:** This is the **template or blueprint**. It's a function or a class that can be used to create an instance of a UI element. It's a general concept, not a specific instance.
- **React Element:** This is an **object that represents the actual UI element** that will be rendered on the screen. It's the "what to render" instruction. When React "sees" a component, it creates a React element for it.

Example:

You define a Welcome Component (the blueprint). When you use <Welcome name="Sara" /> in your code, you're creating a Welcome Element (an instance of the blueprint).

7. What is Virtual DOM? How is it different from Real DOM?

The **Virtual DOM (VDOM)** is a lightweight, in-memory representation of the Real DOM. It's a JavaScript object that mirrors the structure of the browser's DOM. When a component's state changes, React first updates the VDOM. It then compares the new VDOM with the old one (a process called **diffing**) to figure out exactly which parts of the Real DOM need to be updated.

Difference from Real DOM

Feature	Virtual DOM	Real DOM
Type	JavaScript object	Browser's API
Speed	Fast	Slow
Updates	Updates a lightweight representation first, then only applies necessary changes to the Real DOM.	Updates the entire structure, which triggers re-rendering and reflow.
Purpose	Used by React for efficient UI updates	The actual structure of the web page
Cost	Cheap to create and update	Expensive to create and update

The Virtual DOM is so much faster because it avoids the costly process of directly manipulating the Real DOM, which triggers layout and painting.

8. Explain React’s one-way data binding.

One-way data binding means that data flows in a single direction in a React application: from parent to child components. A parent component passes data to its child components through **props**. The child component cannot directly modify the data it receives. If a child component needs to communicate back to the parent, it does so by calling a function that the parent passed down as a prop.

This approach makes the data flow predictable and easier to debug, as you always know where a piece of data originated from. It prevents unwanted side effects and makes the application more stable.

9. What are props in React?

Props (short for properties) are how you pass data from a parent component to a child component. They are a read-only object that a functional component receives as its first argument or a class component can access via `this.props`. They are **immutable**, meaning a child component cannot change the props it receives.

Example:

```
JavaScript

// Parent Component
const App = () => {
  return <Greeting name="Alice" />;
};

// Child Component
const Greeting = (props) => {
  return <h1>Hello, {props.name}</h1>; // props.name is "Alice"
};
```

10. What is state in React? Difference between state and props.

State is an object that holds data or properties that belong to a component and can change over time. When a component's state changes, React re-renders the component. State is **local** to the component and is meant to be managed and updated within that component.

Difference between State and Props

Feature	State	Props
Purpose	To manage component's own data that can change over time.	To pass data from a parent component to a child component.
Mutability	Mutable (can be changed using `useState` or `this.setState`).	Immutable (read-only, cannot be changed by the child).

Origin	Managed within the component itself.	Passed down from a parent component.
Use Case	Data that a component needs to keep track of internally (e.g., a counter, a form input value).	Configuration or data given to a component from the outside (e.g., a username, a title).

---

2. React Component Lifecycle

11. What are React lifecycle methods?

React component lifecycle methods are special methods that are automatically called at different stages of a component's existence. These stages include:

- **Mounting:** When an instance of a component is being created and inserted into the DOM.
- **Updating:** When a component is being re-rendered as a result of changes to props or state.
- **Unmounting:** When a component is being removed from the DOM.

These methods allow you to execute code at specific points in a component's life, such as fetching data (`componentDidMount`) or cleaning up resources (`componentWillUnmount`).

12. Difference between `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.

- `componentDidMount()`: Called immediately after a component is first **rendered and mounted** to the DOM. This is the ideal place for tasks that need a fully rendered component, such as:
    - Making API calls to fetch data.
    - Setting up subscriptions or timers.
    - Interacting with the DOM.
- `componentDidUpdate(prevProps, prevState)`: Called after a component **re-renders** due to changes in its props or state. It's used for:
    - Making network requests when props change (e.g., fetching new data when a user ID prop is updated).
    - Updating the DOM in response to a state change.
- `componentWillUnmount()`: Called right before a component is **unmounted and destroyed**. This is your last chance to perform any necessary cleanup, such as:
    - Canceling network requests.
    - Clearing timers or event listeners.
    - Unsubscribing from subscriptions.

13. What replaced lifecycle methods in functional components? (Hooks).

React Hooks replaced lifecycle methods in functional components. Instead of separate methods for different lifecycle stages, Hooks like `useEffect` combine this functionality. The `useEffect` Hook can handle all three stages: mounting, updating, and unmounting, depending on how you use it.

14. What is `shouldComponentUpdate()`? Why use it?

`shouldComponentUpdate(nextProps, nextState)` is a lifecycle method in class components that is called before a re-render. It receives the new props and state as arguments and **must return a boolean value** (`true` or `false`).

- If it returns `true` (the default), the component will re-render.
- If it returns `false`, the re-render process is skipped.

You use it to **optimize performance** by preventing unnecessary re-renders when you know that a change in props or state won't affect the component's output. In functional components, `React.memo` and `useMemo` serve a similar purpose.

15. What is error boundary in React?

An **Error Boundary** is a special type of React component that catches JavaScript errors anywhere in its child component tree, logs those errors, and displays a fallback UI instead of the component tree that crashed.

They are defined using a class component with either or both of the static lifecycle methods `static getDerivedStateFromError()` (for rendering a fallback UI) and `componentDidCatch()` (for logging error information). Error boundaries only catch errors in the render phase, lifecycle methods, and constructors of their children. They do not catch errors in event handlers or asynchronous code.

Example:

JavaScript

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

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

  componentDidCatch(error, errorInfo) {
    // You can also log the error to an error reporting service
    console.error("Error caught by Error Boundary:", error, errorInfo);
  }

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

```
    }
    return this.props.children;
  }
}
```

---

3. React Hooks

16. What are React Hooks? Why introduced?

React Hooks are functions that let you "hook into" React features like state and lifecycle methods from **functional components**. They were introduced in React 16.8 to address several limitations of class components:

- **Logic Reuse:** It was difficult to reuse stateful logic between components. Hooks allow you to extract and reuse this logic in custom hooks.
- **Complexity:** Class components with their multiple lifecycle methods (`componentDidMount`, `componentDidUpdate`, etc.) could become complex and hard to manage, especially when handling related logic in different methods.
- **`this` keyword:** The `this` keyword in class components can be confusing and lead to bugs. Hooks eliminate the need for `this`.

Hooks make functional components the preferred way to write React code, leading to more concise, readable, and maintainable code.

17. Difference between useState and useReducer.

- `useState`: A basic hook for managing **simple state**. It returns a state value and a function to update it. It's great for primitive values, objects, or arrays where the updates are straightforward.
- `useReducer`: A more powerful hook for managing **complex state logic**. It's an alternative to `useState` for when state updates depend on the previous state or when the state logic is complex. It works similarly to Redux, using a **reducer function** to handle state transitions.

When to use `useReducer` instead of `useState`:

- When state transitions involve multiple sub-values.
- When the next state depends on the previous state.
- When you have complex state logic spread across multiple event handlers.

18. What is useEffect? Difference between componentDidMount and useEffect.

`useEffect` is a hook that lets you perform **side effects** in functional components. A "side effect" is anything that affects something outside the component's scope, like:

- Fetching data from an API.
- Setting up a subscription.

- Manually changing the DOM.
- Setting timers.

`useEffect` accepts a function and an optional dependency array. The function is called after every render by default. The dependency array tells React when to re-run the effect.

Difference between `componentDidMount` and `useEffect`

`componentDidMount` is a class lifecycle method that **only runs once** after the initial render. `useEffect` is more versatile:

- **`useEffect` without a dependency array:** Runs on every render.
- **`useEffect` with an empty array `[]`:** Runs **only once** on mount, similar to `componentDidMount`.
- **`useEffect` with a dependency array `[propA, stateB]`:** Runs on mount and whenever any of the values in the array change.

19. What is useContext? Difference between Context API and Redux.

`useContext` is a hook that provides a way to consume a React Context. React's **Context API** is a feature that allows you to pass data through the component tree without having to pass props down manually at every level (a problem known as "prop drilling").

Difference between Context API and Redux

Feature	Context API	Redux
Purpose	Simple state management for data that's "global" to a tree of components.	Predictable state container for complex, large-scale applications.
Complexity	Simple, built-in.	More complex, requires boilerplate (actions, reducers, etc.).
Use Case	Theming, user authentication status, language settings.	Large-scale applications with frequent state updates and complex logic.

Middleware	No built-in concept.	Robust middleware support for handling side effects (Thunk, Saga).
Debugging	Can be difficult to trace state changes.	Excellent debugging tools (Redux DevTools) that show every state change.

Context API is not a replacement for Redux. Use Context for simple, infrequent state updates. Use Redux for complex applications where you need a centralized, predictable state container and advanced debugging tools.

20. What is useRef? Practical use cases.

`useRef` is a hook that returns a mutable ref object whose `.current` property is initialized to the passed argument. The returned object will persist for the full lifetime of the component. A ref is a generic container whose value can be set and accessed.

Practical Use Cases:

- **Accessing DOM elements:** The most common use case. You can get a direct reference to a DOM node.
- **Storing a mutable value:** You can use a ref to store any value that you want to persist across renders without causing a re-render when it changes. This is useful for storing things like a timer ID.

Example:

```
JavaScript

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

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

  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={focusInput}>Focus the input</button>
    </div>
  );
};
```

21. What is useMemo and useCallback? When to use them?

Both `useMemo` and `useCallback` are optimization hooks used for **memoization**. Memoization is a technique that stores the result of a function call and returns the cached result when the same inputs occur again.

- `useMemo`: **Memoizes a value.** It re-computes the value only when one of its dependencies has changed. It's used to avoid expensive calculations on every render.
- `useCallback`: **Memoizes a function.** It returns a memoized version of the callback function that only changes if one of its dependencies has changed. It's used to prevent child components from re-rendering unnecessarily due to a new function reference.

When to use them:

- `useMemo`: When you have a computationally **expensive calculation** that you don't want to re-run on every render.
- `useCallback`: When you're passing a function as a prop to a **child component that is memoized** (e.g., using `React.memo`). Without `useCallback`, the parent component would create a new function on every render, causing the child component to re-render even if its props haven't conceptually changed.

22. What is custom hook in React? Example.

A **custom hook** is a JavaScript function whose name starts with "use" and that can call other hooks. They are a powerful way to **extract and reuse stateful logic** from components.

Example: Let's create a custom hook `useFetch` to fetch data from an API.

JavaScript

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

const useFetch = (url) => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch(url);
        if (!response.ok) {
          throw new Error(`HTTP error! status: ${response.status}`);
        }
        const result = await response.json();
        setData(result);
      } catch (e) {
        setError(e);
      } finally {
```

```
        setLoading(false);
      }
    };
    fetchData();
  }, [url]);

  return { data, loading, error };
};
```

Now, any component can use this hook to easily fetch data:

```
JavaScript

const UserProfile = ({ userId }) => {
  const { data, loading, error } = useFetch(`https://api.example.com/users/${userId}`);

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error: {error.message}</div>;

  return <div>Welcome, {data.name}</div>;
};
```

23. Rules of hooks in React.

There are two main rules to follow when using Hooks:

1. **Only call Hooks at the top level:** Don't call Hooks inside loops, conditions, or nested functions. This ensures that Hooks are called in the same order on every render.
2. **Only call Hooks from React functions:** Call them from a functional component or a custom Hook. Don't call them from a regular JavaScript function.

These rules are essential for React to correctly manage the state and effects associated with a component.

24. Difference between controlled and uncontrolled components.

- **Controlled Components:** The form data is handled by the component's **state**. The value of an input element is always driven by the state, and every state update is managed by a handler function. This gives you full control over the form data.
- **Uncontrolled Components:** The form data is handled by the **DOM itself**. You use a `ref` to get the value directly from the DOM when you need it (e.g., on form submission). They are simpler to implement for simple use cases but offer less control.

Example (Controlled):

```
JavaScript
```

```
const ControlledInput = () => {
  const [name, setName] = useState('');
  return (
    <input
      type="text"
      value={name}
      onChange={e => setName(e.target.value)}
    />
  );
};
```

Example (Uncontrolled):

```
JavaScript
```

```
const UncontrolledInput = () => {
  const inputRef = useRef(null);
  const handleSubmit = () => {
    alert(`Value: ${inputRef.current.value}`);
  };
  return (
    <>
      <input type="text" ref={inputRef} />
      <button onClick={handleSubmit}>Submit</button>
    </>
  );
};
```

25. How to optimize re-renders in React using hooks?

You can optimize re-renders by preventing components from re-rendering when their props or state haven't changed. Hooks provide several ways to do this:

- **`useMemo`:** Memoize expensive calculations so they don't re-run on every render.
- **`useCallback`:** Memoize callback functions passed to child components to prevent unnecessary re-renders of those children.
- **`React.memo`:** A higher-order component that memoizes a component, preventing it from re-rendering if its props are the same as the last render.
- **Use `useState` wisely:** Batch state updates and avoid updating state with the same value.
- **Custom hooks:** Extract complex state logic into custom hooks to isolate re-renders.

---

4. State Management

26. What are different ways of managing state in React?

There are several ways to manage state in a React application, ranging from simple to complex:

- **Local State:** Using `useState` or `useReducer` to manage state that is local to a single component.
- **Lifting State Up:** Moving state from a child component to its closest common ancestor so that it can be shared between multiple children.
- **Context API:** Using `createContext` and `useContext` to provide state to a whole subtree of components without prop drilling.
- **State Management Libraries:** Using external libraries like **Redux**, **Zustand**, **Jotai**, or **Recoil** for large-scale, complex applications that require a centralized, predictable state.

27. Difference between lifting state up and Context API.

- **Lifting State Up:** This is a simple pattern for sharing state between a few sibling components. You move the state to their shared parent. It's a great approach for **local, simple state sharing**, but it can lead to "prop drilling" if the components are deeply nested.
- **Context API:** This is designed to solve the prop drilling problem. It allows you to create a "global" state that can be consumed by any component within a certain part of the component tree, regardless of how deep it is. It's best for **infrequently updated state** that is needed by many components, like user authentication or theming.

28. What is Redux? Why use it?

Redux is a predictable state container for JavaScript apps. It's a standalone library but is most often used with React. Redux follows a strict **unidirectional data flow** with three core principles:

1. **Single source of truth:** The state of your entire application is stored in a single object tree within a single **store**.
2. **State is read-only:** The only way to change the state is by emitting an **action**, an object describing what happened.
3. **Changes are made with pure functions:** To specify how the state tree is transformed by actions, you write pure functions called **reducers**.

You use Redux to manage complex, shared state in large applications. It provides a structured way to handle state changes, making it easier to debug and test.

29. Difference between Redux and Context API.

- **Redux:**
    - **Use case:** Complex, large-scale applications with a lot of state and frequent updates.
    - **Performance:** Can be more performant for complex state trees due to selective updates.

    - **Debugging:** Excellent dev tools (Redux DevTools) that allow you to replay actions and inspect state changes.
    - **Boilerplate:** Requires more setup (actions, reducers, etc.).
- **Context API:**
    - **Use case:** Simple, low-frequency state updates (e.g., theme, user info).
    - **Performance:** Can cause unnecessary re-renders for all consuming components when the context value changes.
    - **Debugging:** Less structured, harder to trace state changes.
    - **Boilerplate:** Minimal setup, built into React.

30. What are Redux Thunk and Redux Saga?

Both are middleware for Redux used to handle **asynchronous actions**. Redux's reducers must be pure functions, so they can't handle side effects like API calls. Middleware sits between the dispatching of an action and the reducer, allowing you to intercept and handle these side effects.

- **Redux Thunk:** The simpler of the two. It's a small middleware that lets you write **action creators** that return a function instead of an action object. This function receives `dispatch` and `getState` as arguments, allowing you to dispatch multiple actions and perform logic. It's great for simple async logic.
- **Redux Saga:** A more complex middleware that uses **ES6 Generators** to handle side effects. It treats async actions as background tasks that can be started, stopped, and canceled. It's more powerful and provides better testability and control over complex async flows.

31. Difference between synchronous and asynchronous actions in Redux.

- **Synchronous Actions:** Actions that are immediately dispatched and processed by the reducer. The state is updated right away. For example, a user clicking a button to increment a counter. The `dispatch` call happens, the reducer receives the action, and the state is updated instantly.
- **Asynchronous Actions:** Actions that involve a delay, typically from a side effect like an API call. For example, fetching data from a server. The action is dispatched, but the reducer doesn't process it until the API call is complete. This requires middleware (like Redux Thunk or Saga) to handle the intermediate states (e.g., `FETCH_REQUEST`, `FETCH_SUCCESS`, `FETCH_FAILURE`).

32. What is Recoil in React?

Recoil is a state management library for React, created by Facebook. It's designed to be more **flexible and "React-like"** than Redux. It uses a graph-based approach to state management, where state is defined in small, reusable units called **atoms** and derived state is defined by **selectors**.

Recoil is great for:

- **Predictable state management** without the Redux boilerplate.
- **Concurrent Mode support** from the ground up.
- **Performance optimizations** as it only re-renders components that subscribe to the state that changed.

33. Difference between Redux Toolkit and traditional Redux.

- **Traditional Redux:** Required a lot of boilerplate code (actions, action creators, reducers, constants, `combineReducers`, etc.). This made it cumbersome to set up and manage.
- **Redux Toolkit (RTK):** The official, opinionated, batteries-included toolset for efficient Redux development. It simplifies the Redux workflow and reduces boilerplate by:
    - `configureStore`: Simplifies store setup.
    - `createSlice`: Combines actions and reducers into a single "slice" of state.
    - `createAsyncThunk`: Handles async actions out of the box.

RTK is the recommended way to write Redux code today and makes Redux a much more pleasant experience.

34. What is Zustand / Jotai in React?

Zustand and **Jotai** are modern, minimalist, and developer-friendly state management libraries for React. They are often referred to as "Flux-less" or "Redux-less" libraries because they avoid the complex patterns of traditional Redux.

- **Zustand:** A small, fast, and scalable state management library. It uses a **hook-based API** and is known for its simplicity and lack of boilerplate. It's like `useState` on steroids.
- **Jotai:** A primitive-based state management library. It's even more minimalist than Zustand, providing a single concept: the **atom**. Atoms are the smallest unit of state, and you build your state from these atoms. It's great for fine-grained state updates.

35. When should you choose Redux vs Context API?

- **Choose Redux when:**
    - Your application has a large, complex, and shared state that is frequently updated.
    - You need robust debugging tools to track every state change.
    - You need to handle complex asynchronous logic and side effects with middleware.
    - Your team is comfortable with the Redux pattern and boilerplate.
- **Choose Context API when:**
    - You are building a small-to-medium-sized application.
    - The state is not updated frequently (e.g., user authentication, theme).
    - The state is only needed by a small, isolated part of the component tree.
    - You want to avoid the complexity and boilerplate of a full-fledged state management library.

---

5. React Router

36. What is React Router? Why use it?

React Router is a standard library for routing in React. It allows you to create **declarative routing** in your single-page applications (SPAs). It keeps the UI in sync with the URL, allowing you to have different views for different URLs without a full page refresh.

You use it to:

- Build multi-page applications using only a single HTML file.
- Navigate between different pages or components.
- Pass data via URL parameters.
- Handle authentication-based routing.
- Keep your application's state and UI consistent with the browser's URL.

37. Difference between BrowserRouter, HashRouter, and MemoryRouter.

These are different router implementations provided by React Router.

- **`BrowserRouter`:** Uses the HTML5 history API (`pushState`, `replaceState`, etc.) to keep the UI in sync with the URL. It's the **most common and recommended** router for modern web applications. The URLs look clean (e.g., `www.example.com/about`). It requires server-side configuration to handle direct URL access.
- **`HashRouter`:** Uses the `hash` portion of the URL (e.g., `www.example.com/#/about`) to keep the UI in sync. It doesn't require any server-side configuration because the hash part is not sent to the server. It's often used for static sites or when server-side configuration is not possible. The URLs look less clean.
- **`MemoryRouter`:** Stores the URL history in memory. It does not read or write to the URL address bar. This is primarily used for **testing** and in non-browser environments like React Native.

38. Difference between `<Link>` and `<NavLink>`.

- **`<Link>`:** A component used for basic navigation. It renders an `<a>` tag and changes the URL without a full page refresh. It's used for simple navigation from one page to another.
- **`<NavLink>`:** A specialized version of `<Link>`. It's used for navigation and for automatically adding styling attributes (like an `active` class) to the rendered element when the URL it links to is the current URL. This is perfect for creating navigation menus where you want to highlight the active link.

Example:

```
JavaScript


// NavLink with active class
<NavLink to="/dashboard" activeClassName="active">Dashboard</NavLink>
```

39. What are route parameters in React Router?

Route parameters are dynamic parts of a URL that are used to capture a value. They are denoted by a colon (:) followed by the parameter name in the route path. This is a common way to pass data to a component.

Example:

- **Route Path:** `/users/:id`
- **URL:** `/users/123`
- **Route Parameter:** `id` will have the value `123`.

You can access these parameters in your component using the `useParams` hook.

```
JavaScript

import { useParams } from 'react-router-dom';

const UserProfile = () => {
  const { id } = useParams();
  return <h1>User ID: {id}</h1>;
};
```

40. Difference between `useNavigate` and `useHistory`.

- `useNavigate` (React Router v6): The new hook for programmatic navigation. It returns a function that you can call to navigate to a new URL. It's the recommended way to navigate in modern React Router.
- `useHistory` (React Router v5): The old hook for programmatic navigation. It returned a `history` object with methods like `push` and `replace`. It's deprecated in v6 and should be avoided in new projects.

Example (v6):

```
JavaScript

import { useNavigate } from 'react-router-dom';

const MyComponent = () => {
  const navigate = useNavigate();
  const handleClick = () => {
    navigate('/dashboard');
  };
  return <button onClick={handleClick}>Go to Dashboard</button>;
};
```

41. How to implement nested routes?

Nested routes are routes that are rendered within another component's route. This is useful for creating layouts where parts of the UI change while other parts stay the same (e.g., a dashboard with a sidebar).

In React Router v6, you use a nested `<Route>` component inside another `<Route>`'s `element`. The parent route should have a `*` or a trailing slash to indicate it's a "catch-all" for its children.

Example:

```
JavaScript

// Parent Route
<Route path="/dashboard" element={<DashboardLayout />}>
  // Nested Routes
  <Route index element={<DashboardHome />} /> // Index route for /dashboard
  <Route path="profile" element={<UserProfile />} />
  <Route path="settings" element={<DashboardSettings />} />
</Route>
```

42. Difference between `Switch` (v5) and `Routes` (v6).

- **`Switch` (v5):** A component that rendered the **first** child `<Route>` that matched the location. It was important for preventing multiple components from rendering for the same URL.
- **`Routes` (v6):** The new component that replaced `Switch`. It works similarly, but it's smarter. It renders the **best** match for the location instead of just the first. It's a more powerful and intuitive way to manage routes.

43. What is lazy loading routes?

Lazy loading (or code splitting) is a technique that splits your code into separate bundles. Instead of loading your entire application's code at once, it only loads the code needed for a specific page or component when it's requested.

In React, you use **`React.lazy()`** and **`<Suspense>`** to implement lazy loading. `React.lazy()` lets you render a dynamic import as a regular component, and `<Suspense>` shows a fallback UI (like a loading spinner) while the component is being loaded.

Example:

```
JavaScript

import React, { Suspense, lazy } from 'react';
import { BrowserRouter, Routes, Route } from 'react-router-dom';

const About = lazy(() => import('./About'));
const Home = lazy(() => import('./Home'));
```

```
const App = () => (
  <BrowserRouter>
    <Suspense fallback={<div>Loading...</div>}>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </Suspense>
  </BrowserRouter>
);
```

44. How to handle protected routes (authentication-based routing)?

A protected route is a route that can only be accessed by an authenticated user. You can implement this using a wrapper component that checks for authentication before rendering the actual component.

Example:

```
JavaScript

import { useAuth } from './auth-context';
import { Navigate, Outlet } from 'react-router-dom';

const ProtectedRoute = () => {
  const { isAuthenticated } = useAuth(); // Custom hook to check auth status

  if (!isAuthenticated) {
    return <Navigate to="/login" replace />;
  }

  return <Outlet />; // Renders the child route
};

// In your router configuration:
<Route element={<ProtectedRoute />}>
  <Route path="/dashboard" element={<Dashboard />} />
</Route>
```

45. How to redirect in React Router?

In React Router v6, you use the `<Navigate>` component for redirection. It's a declarative way to navigate, as it renders a redirection instead of the component itself.

Example:

```
JavaScript

import { Navigate } from 'react-router-dom';
```

```
const MyComponent = ({ isUserLoggedIn }) => {
  if (!isUserLoggedIn) {
    return <Navigate to="/login" />;
  }
  return <div>Welcome!</div>;
};
```

You can also use the `useNavigate` hook for programmatic redirection, as shown in a previous question.

---

6. React Performance

46. What causes re-renders in React?

A component **re-renders** when its state or props change. Here are the main causes:

- **State change:** When you call a state updater function (e.g., `setState` or `setCount`).
- **Props change:** When a parent component re-renders, it passes new props to its children, causing them to re-render.
- **Context change:** When a value in a Context Provider changes, all consuming components re-render.
- **`forceUpdate()`:** A method in class components that forces a re-render. It should be avoided.
- **`useReducer` dispatch:** Dispatching an action to a reducer causes a re-render.

47. How to prevent unnecessary re-renders?

Preventing unnecessary re-renders is a key part of React performance optimization. You can do this by:

- **`React.memo`:** A HOC that memoizes a component. It prevents the component from re-rendering if its props have not changed.
- **`useMemo`:** Memoize the result of an expensive calculation.
- **`useCallback`:** Memoize functions passed as props to children components.
- **`shouldComponentUpdate`:** A lifecycle method for class components that lets you control when a component re-renders.
- **State Colocation:** Keep state as close as possible to where it's needed to avoid re-rendering large parts of the component tree.
- **Key Prop:** Use unique and stable keys for list items to prevent the re-creation of components.

48. Difference between React.memo and useMemo.

- **`React.memo`:** A **higher-order component (HOC)** that wraps a **component**. It tells React to skip rendering the component if its props are the same. It's for **preventing re-renders of the entire component**.

- **`useMemo`:** A **hook** that memoizes a **value**. It re-computes the value only when one of its dependencies changes. It's for **preventing expensive calculations** inside a component.

Example:

```
JavaScript

// React.memo to prevent re-render of the entire component
const MyComponent = React.memo((props) => {
  // ...
});

// useMemo to prevent expensive calculation
const calculatedValue = useMemo(() => {
  return expensiveCalculation(a, b);
}, [a, b]);
```

49. What is code splitting in React? (React.lazy, Suspense).

Code splitting is a technique that splits your JavaScript bundle into smaller, more manageable chunks. This allows the browser to download only the code needed for the current page, which significantly improves initial load time.

- **`React.lazy()`:** A function that lets you render a dynamic import as a regular component. It takes a function that returns a `Promise` that resolves to a module with a default export containing a React component.
- **`<Suspense>`:** A component that lets you specify a fallback UI (like a loading spinner) to display while the lazy-loaded component is being loaded.

This combination is a built-in way to implement code splitting in React.

50. How does React handle reconciliation?

Reconciliation is the process by which React updates the UI to match the component's state. When a component's state or props change, React builds a new tree of React elements. It then uses a **diffing algorithm** to compare this new tree with the previous one. This comparison identifies the minimal set of changes needed to update the Real DOM.

The process is:

1. A component's state or props change.
2. A new Virtual DOM tree is created.
3. React's diffing algorithm compares the new VDOM with the old VDOM.
4. A list of "diffs" or changes is created.
5. React's rendering engine updates the Real DOM with only the necessary changes.

This process is what makes React so efficient, as it avoids re-rendering the entire DOM.

51. What is key prop in React lists? Why is it important?

The `key` prop is a special string attribute you must include when creating lists of elements. React uses the key to uniquely identify each list item.

Why is it important?

When a list is updated (e.g., an item is added, removed, or reordered), React uses the key to identify which specific component has changed. Without a key, React would just re-render the entire list, which is inefficient. With a key, React can:

- Efficiently update the DOM.
- Move or remove components without re-creating them.
- Preserve the state of components (e.g., a form input's value).

Rule: The `key` must be a **stable, unique identifier** for each item. Don't use the index as a key if the list can be reordered, as this can lead to unpredictable behavior and performance issues.

52. Difference between controlled vs uncontrolled inputs in forms.

This is a repetition of question 24, but in the context of forms.

- **Controlled Inputs:** Form inputs whose value is controlled by React's state. The value is set and updated via `useState` and an `onChange` handler. This gives you full control over the form data and validation.
- **Uncontrolled Inputs:** Form inputs whose value is managed by the DOM itself. You use a `ref` to get the value when you need it. They are simpler for simple forms but offer less control.

53. How to improve performance in large React apps?

- **Code Splitting:** Use `React.lazy` and `Suspense` to split your app into smaller bundles.
- **Memoization:** Use `React.memo`, `useMemo`, and `useCallback` to prevent unnecessary re-renders and expensive calculations.
- **State Colocation:** Keep state close to the components that use it to limit the re-render scope.
- **Virtualization/Windowing:** Use libraries like `react-window` or `react-virtualized` to render only the visible items in a long list.
- **Keys:** Use stable, unique keys for list items.
- **Server-Side Rendering (SSR) / Static Site Generation (SSG):** Render the initial HTML on the server to improve first-paint time and SEO.
- **Use the React DevTools Profiler:** Identify performance bottlenecks by profiling your components.
- **Avoid Inline Functions/Objects:** In a component's render method, creating a new function or object on every render can break memoization.

54. What is windowing in React? (react-window, react-virtualized).

Windowing (or list virtualization) is a technique used to improve the performance of rendering long lists. Instead of rendering all the items in a list, it only renders the ones that are currently visible within the viewport (the "window"). As the user scrolls, the library dynamically renders and re-uses the component instances for the items that come into view.

- **`react-virtualized`:** A more feature-rich library that provides components for various types of virtualized lists, tables, and grids.
- **`react-window`:** A smaller, simpler, and faster library. It has fewer features but is more performant for simple virtualized lists. It is the recommended choice unless you need the advanced features of `react-virtualized`.

55. What is hydration in React?

Hydration is the process by which a static, server-rendered HTML page (generated by SSR) is turned into a fully interactive React application on the client-side. The initial HTML is rendered on the server, which is good for performance and SEO. When the JavaScript bundle is downloaded and executed in the browser, React "hydrates" the static HTML by attaching event listeners and making it interactive without re-rendering the entire page.

---

7. React Advanced Concepts

56. What is higher-order component (HOC) in React? Example.

A **Higher-Order Component (HOC)** is an advanced technique for reusing component logic. A HOC is a function that takes a component as an argument and returns a **new, enhanced component**. They are a common pattern in older React codebases (before Hooks) for sharing logic.

Example: A HOC that adds a `user` prop to a component.

JavaScript

```
// HOC
const withUser = (WrappedComponent) => {
  return (props) => {
    const user = { name: "John Doe" }; // Example user data
    return <WrappedComponent {...props} user={user} />;
  };
};

// Component to be enhanced
const Profile = (props) => {
  return <h1>Hello, {props.user.name}</h1>;
};

// The enhanced component
const ProfileWithUser = withUser(Profile);
```

57. Difference between HOC and render props.

Both HOCs and render props are patterns for sharing component logic.

- **HOC (Higher-Order Component):** A function that **takes a component and returns a new component**. It's a "wrapper" pattern.
    - **Pros:** Easy to use once created, clean separation of concerns.
    - **Cons:** Can lead to "wrapper hell" (multiple layers of HOCs), can hide original props, name collisions.
- **Render Props:** A pattern where a component passes a function (a "render prop") to its children via a prop. The child component can then call this function with the shared data to render a part of the UI.
    - **Pros:** Avoids "wrapper hell", more flexible.
    - **Cons:** Can be less readable due to deeply nested JSX, can break memoization.

With the introduction of Hooks, both patterns have become less common as Hooks provide a more idiomatic way to reuse logic.

58. What is Context API in React?

This is a repetition of question 19, but in more detail.

React's **Context API** provides a way to pass data through the component tree without manually passing props down at every level. It consists of three parts:

- **`React.createContext()`:** Creates a Context object.
- **`<Context.Provider>`:** A component that provides a value to all its descendant components.
- **`useContext(Context)`:** A hook that allows a component to consume the value from the nearest `Provider` above it in the tree.

It's a solution for "prop drilling" and is best for simple, application-wide data that doesn't change frequently, such as theme, user authentication, or language settings.

59. What is React Fiber architecture?

React Fiber is a complete rewrite of React's core reconciliation algorithm. It was introduced in React 16 to improve performance and enable new features like concurrency. Before Fiber, React's rendering process was synchronous and couldn't be interrupted. A long render could block the main thread and make the UI feel unresponsive.

Fiber's key features:

- **Incremental Rendering:** It can break the rendering work into smaller chunks and spread them out over multiple frames.
- **Pause and Resume:** It can pause, abort, and resume rendering work.
- **Prioritization:** It can assign different priorities to different types of updates (e.g., high-priority user input, low-priority background work).

This architecture is the foundation for React's new concurrent features and allows for a smoother, more responsive user experience.

60. Difference between React 16, 17, 18 (concurrent features, Suspense, etc.).

- **React 16 (Fiber):** The major rewrite that introduced the Fiber architecture, `componentDidCatch` (error boundaries), fragments, and portals. It also laid the groundwork for Hooks and concurrent features.
- **React 17:** A "stepping stone" release. It didn't introduce major new features but made it easier to adopt future versions of React. The main change was how event handlers were attached to the DOM, moving from the document level to the root, which improved compatibility and security.
- **React 18:** The first version to fully introduce **concurrent features** and the new `createRoot` API.
    - **`createRoot`:** The new way to render React apps, which enables concurrent mode.
    - **Automatic Batching:** Automatically batches multiple state updates into a single re-render.
    - **`startTransition` and `useTransition`:** Hooks for marking state updates as "transitions" that can be interrupted.
    - **`<Suspense>`:** Now fully supported on the server for SSR and streaming HTML.

61. What is Concurrent Mode in React 18?

Concurrent Mode (now just **Concurrency**) is a set of new capabilities in React that allows it to prepare multiple UI versions at the same time. This is achieved by making rendering "interruptible."

The key benefits are:

- **Improved User Experience:** It can prevent slow rendering from blocking user input, making the UI feel more responsive.
- **Interruptible Rendering:** It can pause a long-running render to handle a higher-priority update (e.g., a user typing in a search box).

It is not a "mode" you turn on, but rather a set of underlying features enabled by the new `createRoot` API.

62. What are transitions in React 18 (startTransition, useTransition)?

Transitions are a new concept in React 18 that distinguish between urgent updates and less urgent, "transition" updates.

- **Urgent Updates:** Like typing in a text field, a click, etc. They should feel instantaneous.
- **Transition Updates:** Like a filter change, a data fetch, or a navigation. They can be delayed without the UI feeling sluggish.

- `useTransition()`: A hook that returns a `isPending` boolean to tell you if a transition is in progress and a `startTransition` function.
- `startTransition(callback)`: A function that lets you wrap a state update. React will treat this update as a low-priority transition, which can be interrupted if a more urgent update comes in.

This helps you keep the UI responsive during slow data fetches or complex calculations.

63. What is server-side rendering (SSR) in React?

Server-side rendering (SSR) is the process of rendering a React component to HTML on the server and sending it to the client. The browser receives the fully formed HTML and can display it immediately, without waiting for the JavaScript bundle to load. Once the JavaScript loads, React takes over and makes the page interactive (this is the **hydration** process).

Benefits:

- **Improved Performance:** Faster initial load time, better perceived performance.
- **SEO:** Search engine crawlers can easily index the content since it's already in HTML.
- **Accessibility:** Content is available even if JavaScript is disabled.

Popular frameworks like **Next.js** and **Gatsby** handle SSR for you.

64. What is static site generation (SSG)? Difference from SSR.

Static Site Generation (SSG) is a build-time process where all the HTML pages of a website are generated in advance and served as static files. The content is pre-rendered at build time.

Difference from SSR:

| Feature | Server-Side Rendering (SSR) | Static Site Generation (SSG) |

| :--- | :--- | :--- |

| Generation Time | On-the-fly, at request time. | At build time. |

| Use Case | Data that changes frequently (e.g., a live stock ticker, a social media feed). | Data that changes infrequently (e.g., a blog post, a documentation page). |

| Performance | Fast initial load. | Extremely fast, as files are pre-built and can be served from a CDN. |

| Hosting | Requires a server. | Can be hosted on a static file server or a CDN. |

SSG is typically faster and cheaper to host than SSR. Frameworks like Next.js and Gatsby support both.

65. Difference between Next.js and Create React App.

- **Create React App (CRA):** A tool for creating a **single-page application (SPA)**. It sets up a client-side rendering environment with a pre-configured build pipeline (Webpack, Babel). It's great for learning React and building small-to-medium-sized client-side apps.
- **Next.js:** A **framework** for building server-rendered React applications. It's a full-stack solution with built-in features like **server-side rendering (SSR)**, **static site generation (SSG)**, API routes, and a file-system-based router. It's the go-to choice for production-grade, performant, and SEO-friendly React apps.

CRA is a simple tool for getting started, while Next.js is a full-featured framework for building production-ready, performant web applications.

---

8. Testing in React

66. How to test React components? (Jest, React Testing Library).

The standard tools for testing React components are **Jest** and **React Testing Library**.

- **Jest:** A JavaScript testing framework created by Facebook. It's the **test runner** and provides the assertion library (`expect`) and mocking functionality.
- **React Testing Library (RTL):** A library that provides a set of utilities for testing React components. Its philosophy is to test components in a way that resembles how users would use them. It encourages testing from the user's perspective, focusing on what the user sees and interacts with, rather than on the component's internal implementation details.

67. Difference between unit testing and integration testing in React.

- **Unit Testing:** Testing individual, isolated units of code (e.g., a single function, a component in isolation). The goal is to ensure each unit works correctly on its own. You might test that a `sum` function returns the correct result or that a component renders the correct initial state.
- **Integration Testing:** Testing how different parts of an application work together. For a React app, this means testing how components interact with each other, with state management, or with an API. The goal is to ensure that the different "units" are integrated correctly.

React Testing Library is great for integration testing because it encourages you to test components by rendering them and interacting with them as a user would.

68. How to mock API calls in React tests?

You can mock API calls in React tests using **Jest's mocking capabilities**. You can mock the `fetch` API or any other library you use (e.g., `axios`).

Example (Mocking `fetch` with Jest):

```
JavaScript
```

```
// In your test file:
global.fetch = jest.fn(() =>
  Promise.resolve({
    json: () => Promise.resolve({ data: 'mocked data' }),
    ok: true,
  })
);

// Your component that calls the API
const MyComponent = () => { /* ... */ };

// In your test, you can now render MyComponent
// and it will use the mocked fetch function.
```

You can also use dedicated mocking libraries like `msw` **(Mock Service Worker)**, which intercepts network requests at the service worker level and provides a more realistic mocking environment.

69. Difference between shallow rendering and full rendering.

- **Shallow Rendering:** Renders only the component itself, without rendering its children components. It's a way to test a component in isolation, without worrying about the behavior of its children. This was a feature of the **Enzyme** testing library.
- **Full Rendering (Mounting):** Renders the component and all of its children, as they would be rendered in a real browser. This is the default behavior of **React Testing Library** and is closer to how a user would interact with the component, making it great for integration tests.

The React team and community now favor full rendering because it promotes testing from the user's perspective.

70. What is Enzyme? Why use React Testing Library instead?

Enzyme is a JavaScript testing utility for React created by Airbnb. It provides utilities for rendering components, manipulating them, and traversing their output. It was the de facto standard for a long time.

Why use React Testing Library instead?

- **Philosophy:** RTL's main principle is to test components like a user would. It discourages testing implementation details (like a component's internal state) and focuses on the UI and user interactions.
- **Better Practices:** It promotes writing tests that are more robust and less likely to break from refactoring. If you change a component's implementation but the UI and behavior remain the same, your RTL tests should still pass.
- **Official Recommendation:** The React team now officially recommends RTL.

71. What are snapshot tests in React?

Snapshot tests are a type of test that captures the rendered output of a component and saves it as a text file (a "snapshot"). The next time you run the test, Jest compares the new rendered output with the saved snapshot. If they don't match, the test fails.

Use cases:

- Ensuring that the UI doesn't change unexpectedly.
- Catching accidental styling or structural changes.

Drawbacks:

- Can lead to "snapshot spamming" where developers just update the snapshot instead of fixing the underlying issue.
- They don't test behavior, only the rendered output.

They are a useful tool but should be used in conjunction with other types of tests.

72. How to test hooks in React?

You don't test hooks in isolation. You test the **component that uses the hook**. Since hooks manage state and side effects, you test that the component's output or behavior is correct after interacting with the hook.[1]

Libraries like `@testing-library/react-hooks` (or its replacement `@testing-library/react` since Hooks are now part of regular component testing) provide a wa[2]y to test custom hooks in isolation by creating a test component that uses the hook.

Example:

```
JavaScript

// Test a custom hook
import { renderHook, act } from '@testing-library/react-hooks';
import useCounter from './useCounter';

test('should increment counter', () => {
  const { result } = renderHook(() => useCounter());
  act(() => {
    result.current.increment();
  });
  expect(result.current.count).toBe(1);
});
```

73. What is end-to-end testing in React? (Cypress, Playwright).

End-to-end (E2E) testing is a testing method that verifies the entire application flow from start to finish, simulating a real user's behavior. It tests the application as a black box, from the user's perspective, across all its layers (UI, API, database).

Popular E2E testing frameworks for React:

- **Cypress:** A fast, reliable E2E testing framework. It runs directly in the browser and provides a great developer experience with real-time reload and a time-travel debugger.
- **Playwright:** Developed by Microsoft, it's a powerful tool for E2E testing and automation. It supports multiple browsers and provides a simple API for interacting with the web page.

74. Difference between TDD and BDD in React context.

- **Test-Driven Development (TDD):** A development methodology where you **write the test first**, then write the minimum code required to pass that test, and then refactor the code. It focuses on the technical implementation and ensures that every piece of code is covered by a test.
- **Behavior-Driven Development (BDD):** An extension of TDD. It focuses on **writing tests that describe the application's behavior** from the user's perspective. It uses a more human-readable language (often in a "Given-When-Then" format) and is a collaboration tool between developers and non-technical stakeholders.

For React, RTL's user-centric approach aligns well with BDD.

75. How to test async code in React?

When testing async code (e.g., API calls, timers), you need to handle the asynchronous nature of the code.

- **`async/await`:** The most common way. You write an `async` test function and `await` the result of the asynchronous operation.
- **Jest's `done()` callback:** For older test runners, you can pass a `done` callback to the test function and call it when the async operation is complete.
- **`act()`:** A utility from React Testing Library that ensures all state updates and renders are flushed before assertions are made. You should wrap any code that causes state updates (e.g., a button click, an API call) in `act()`.

---

9. Security & Deployment

76. How to secure React apps from XSS?

Cross-Site Scripting (XSS) is a type of attack where a malicious script is injected into a web page and executed by other users. React is relatively secure against XSS by default because it **escapes all strings** before rendering them to the DOM.

However, you can still be vulnerable if you use:

- `dangerouslySetInnerHTML`: This prop allows you to insert raw HTML. Only use it with sanitized HTML from a trusted source.
- Unsanitized data from props or state.
- Links with `javascript:` protocol from untrusted sources.

Prevention:

- **Avoid `dangerouslySetInnerHTML`** unless you're sure of the source.
- Use a library like `dompurify` to sanitize HTML.
- Validate and sanitize all user input.
- Ensure that any dynamic URLs are not from an untrusted source.

77. What is CSRF and how to prevent it in React apps?

Cross-Site Request Forgery (CSRF) is an attack that forces an authenticated user to submit a malicious request to a web application.

Prevention:

- **CSRF Tokens:** The most common defense. The server generates a unique, unpredictable token, includes it in a hidden field or a header in the form, and verifies it on submission. The token ensures the request came from your application and not an attacker's.
- **SameSite Cookies:** Set the `SameSite` attribute of your cookies to `Strict` or `Lax`. This prevents the browser from sending the cookie with cross-site requests.
- **Using `fetch` API and not relying on cookies.**

78. How to handle authentication in React (JWT, cookies, sessions)?

Authentication in a React SPA is generally handled by storing some form of token on the client-side after the user logs in.

- **JSON Web Tokens (JWT):** The most popular method. The server returns a JWT after a successful login. The React app stores the token (in local storage or a cookie) and includes it in the `Authorization` header of subsequent API requests.
- **Cookies:** Can be used to store a session ID or a JWT. Cookies with the `HttpOnly` flag are more secure against XSS attacks, as they cannot be accessed by JavaScript.
- **Sessions:** The server-side stores session information, and the client receives a session ID in a cookie. This is less common in SPAs as it requires a stateful server, but it can be more secure as the session data isn't on the client.

The general flow is:

1. User enters credentials.
2. React app sends a request to the server.
3. Server validates and returns a token.
4. React app stores the token and uses it for all future authenticated requests.

79. How to deploy React app to production (Netlify, Vercel, AWS, Docker)?

- **Netlify/Vercel:** These are the simplest options for deploying a React SPA. They are **serverless platforms** that automatically detect your project and set up the build process and hosting. Just push your code to a Git repository, and they handle the rest. Great for SPAs and Next.js projects.

- **AWS S3 + CloudFront:** A very common and scalable approach. You build your app (`npm run build`), upload the static files to an S3 bucket, and use CloudFront as a CDN to serve them.
- **Docker:** You can containerize your React app and its server environment using Docker. This allows you to deploy to any platform that supports Docker, providing consistency and portability.

80. Difference between development and production build in React.

- **Development Build (`npm start`):**
    - **Unoptimized:** The code is not minified or compressed.
    - **Includes Dev Tools:** Contains extra code for debugging, warnings, and React DevTools.
    - **Hot Reloading:** Supports hot module replacement for a fast development loop.
    - **Larger Bundle Size:** The build is larger and slower.
- **Production Build (`npm run build`):**
    - **Optimized:** The code is minified, compressed, and tree-shaken.
    - **Smaller Bundle Size:** All development-only code is removed.
    - **Static Files:** Creates a `build` folder with static HTML, CSS, and JS files.
    - **Faster:** The bundle is optimized for speed and performance.

81. What is tree shaking in React builds?

Tree shaking (or "dead code elimination") is a build process optimization where unused code is removed from the final JavaScript bundle. It's a key feature of modern bundlers like Webpack.

Example: If you import a library but only use one function from it, tree shaking will remove all the other unused functions from the final bundle, resulting in a smaller file size.

82. How to optimize bundle size in React?

- **Code Splitting:** Lazy load components and routes.
- **Tree Shaking:** Ensure your bundler is configured for tree shaking.
- **Analyze Your Bundle:** Use a tool like Webpack Bundle Analyzer to see what's in your bundle and identify large libraries.
- **Use smaller libraries:** Choose lightweight alternatives to large libraries.
- **Image Optimization:** Compress and resize images. Use formats like WebP.
- **Gzip/Brotli Compression:** Configure your server to serve compressed files.

83. What is source map in React build?

A **source map** is a file that maps the minified, production-ready code back to the original source code. When a user encounters an error in production, the browser can use the source map to show you the exact line number and file in your original code, making debugging much easier.

Source maps are typically generated alongside the production build and are not served to the user unless the browser's DevTools are open.

84. Difference between client-side rendering (CSR) and server-side rendering (SSR).

- **Client-Side Rendering (CSR):**
  - The browser downloads a minimal HTML file and the JavaScript bundle.
  - The browser then uses JavaScript to build the DOM and render the content.
  - **Pros:** Fast subsequent page loads.
  - **Cons:** Slower initial load time, bad for SEO, requires JavaScript.
  - **Example:** A typical `create-react-app` project.
- **Server-Side Rendering (SSR):**
  - The server generates the full HTML for the page and sends it to the browser.
  - The browser displays the content immediately.
  - The browser then downloads the JavaScript to make the page interactive (hydration).
  - **Pros:** Fast initial load, great for SEO.
  - **Cons:** Slower for dynamic, frequently updated content; requires a server.
  - **Example:** A Next.js or Gatsby project.

85. What is hydration attack in React?

A **hydration attack** is a security vulnerability that can occur in a server-side rendered (SSR) application. It happens when there is a mismatch between the HTML generated by the server and the virtual DOM tree generated by the client after hydration.

If an attacker can inject malicious code into the server-rendered HTML, and the client-side JavaScript then "hydrates" over it, the malicious script could be executed.

Prevention:

- Ensure the data used for server-side rendering is properly sanitized.
- Never use `dangerouslySetInnerHTML` with unsanitized data, on either the server or the client.
- Be mindful of how you handle dynamic data.

---

10. Real-world & System Design

86. How does React handle large-scale applications?

Building a large-scale React application requires careful architecture and design. Key strategies include:

- **Component Structure:** Organizing components into a clear hierarchy (e.g., `pages`, `components`, `layouts`).
- **State Management:** Using a centralized state management solution like Redux, Recoil, or Zustand for shared state.

- **Performance Optimization:** Implementing code splitting, memoization, and virtualization.
- **Modularization:** Breaking the application into smaller, self-contained modules or **microfrontends**.
- **Testing:** Having a robust testing strategy (unit, integration, E2E) to ensure stability.
- **Monorepo:** Using a monorepo (with tools like Lerna or Turborepo) to manage multiple related projects in a single repository.

87. Difference between monolithic and microfrontend architecture in React.

- **Monolithic Architecture:** A single, large, tightly coupled codebase. All parts of the application are in one repository and are deployed together. This is the traditional way of building applications.
    - **Pros:** Simple to set up and deploy.
    - **Cons:** Hard to scale, difficult to manage for large teams, a change in one part can affect the entire app.
- **Microfrontend Architecture:** An architecture where a large web application is composed of multiple independent applications (microfrontends). Each microfrontend is developed, deployed, and managed autonomously.
    - **Pros:** Allows for independent teams, technology agnostic, scalable.
    - **Cons:** More complex to set up, requires coordination between teams.

88. What are microfrontends in React?

Microfrontends are a front-end architectural style that extends the concept of microservices to the browser. The idea is to break down a large, monolithic front-end into smaller, more manageable applications that can be developed and deployed independently.

Example: A complex e-commerce site could be broken down into a `user profile` microfrontend, a `shopping cart` microfrontend, and a `product catalog` microfrontend. Each team can work on their own microfrontend, using their preferred technology, and the main application shell combines them.

89. How to implement internationalization (i18n) in React?

Internationalization (i18n) is the process of making an application adaptable to different languages and regions.

Steps to implement i18n in React:

1. **Choose a library:** Libraries like `react-i18next` or `react-intl` are great for this.
2. **Define translations:** Store translated strings in JSON or JavaScript files.
3. **Use a `Provider`:** Wrap your app in a provider component that makes the translations available to all components.
4. **Use Hooks/HOCs:** Use the library's hooks (e.g., `useTranslation` from `react-i18next`) to access the translated strings in your components.

90. How to implement dark mode in React?

There are several ways to implement a dark mode in React:

1. **CSS Variables:** The simplest and most modern way. Define color variables in your CSS, and use a root class (e.g., `.light` and `.dark`) to switch the values.
2. **React Context API:** Create a context to store the theme (`'light'` or `'dark'`). Use a provider at the top level to manage the theme state and a hook (`useContext`) to access the theme in child components.
3. **State Management Libraries:** Use Redux, Zustand, etc., for managing the theme state if it's part of a larger, global state.

The most common approach is a combination of Context API and CSS variables.

91. How to handle real-time data in React (WebSockets, SSE)?

Real-time data refers to data that is delivered as soon as it's available, without the client having to constantly poll for updates.

- **WebSockets:** A full-duplex, bidirectional communication protocol over a single TCP connection. It's the standard for real-time applications like chat apps and online games. You can use libraries like `socket.io` or the native WebSocket API.
- **Server-Sent Events (SSE):** A unidirectional protocol where the server pushes updates to the client. It's simpler than WebSockets and is great for scenarios where the client only needs to receive data from the server (e.g., live stock updates).
- **Polling:** The simplest but least efficient method. The client sends a request to the server at regular intervals to check for new data.

92. Difference between REST API and GraphQL in React context.

Feature	REST API	GraphQL
Data Fetching	Over-fetching (gets more data than needed) and under-fetching (needs multiple requests).	Precise fetching (gets exactly the data you ask for in one request).
Endpoints	Multiple endpoints for different resources (e.g., `/users`, `/posts`).	A single endpoint (e.g., `/graphql`).
Flexibility	Less flexible. The server dictates the data structure.	More flexible. The client dictates the data structure.

Tooling	Simple clients like `fetch` or `axios`.	Requires more powerful clients like Apollo or Relay.
Use Case	Most web applications.	Complex applications with nested data, mobile apps where bandwidth is a concern.

GraphQL is often a better choice for complex React applications that need to fetch data from multiple sources in a single request.

93. What is Apollo Client in React?

Apollo Client is a comprehensive state management library for GraphQL. It's a popular choice for managing data in a React application that uses a GraphQL API.

Key features:

- **Declarative Data Fetching:** You declare the data you need in your components, and Apollo handles the fetching.
- **Caching:** It has a powerful in-memory cache that automatically updates your UI when data changes.
- **State Management:** It can be used for both remote data and local state management, acting as a replacement for Redux.
- **Optimistic UI:** It supports optimistic updates, which make the UI feel faster by updating it immediately and reverting it if the server-side operation fails.

94. How to implement optimistic UI updates in React?

Optimistic UI is a user experience pattern where the UI is updated immediately in response to a user action, even before the server confirms the action was successful. This makes the application feel more responsive. If the server request fails, the UI is rolled back to its previous state.

Example: When a user "likes" a post, the UI immediately shows the new "like" count. A network request is sent in the background. If the request fails, the "like" count is decremented.

You can implement this with:

- **Apollo Client:** Has built-in support for optimistic updates.
- **React Query / SWR:** These data-fetching libraries also provide optimistic update features.
- **Manually:** By using a state management library to update the state immediately and then rolling it back on a failed network request.

95. How to handle forms in React (Formik, React Hook Form)?

While you can handle forms with plain React state, it can become cumbersome for complex forms. Libraries simplify the process.

- **Formik:** A popular library for building forms. It simplifies state management, validation, and submission. It uses a **controlled component** approach.
- **React Hook Form (RHF):** A library that uses a **ref-based, uncontrolled component** approach. It's known for being lightweight and performant because it avoids unnecessary re-renders.

React Hook Form is often the preferred choice for new projects due to its performance benefits and simpler API.

96. Difference between controlled and uncontrolled form libraries.

- **Controlled Libraries (e.g., Formik):**
    - The form state is stored in a React state hook or a state management library.
    - Every keystroke triggers a state update and a re-render.
    - **Pros:** Full control over form data, easy to add validation on every keystroke.
    - **Cons:** Can lead to performance issues with large forms due to frequent re-renders.
- **Uncontrolled Libraries (e.g., React Hook Form):**
    - The form state is managed by the DOM via refs.
    - The library only gets the form data on submission.
    - **Pros:** Excellent performance, as it avoids re-renders on every keystroke.
    - **Cons:** Less control over input values in real-time.

97. How to handle accessibility (a11y) in React apps?

Accessibility (a11y) means making your application usable by people with disabilities.

- **Use Semantic HTML:** Use semantic tags like `<button>`, `<nav>`, `<main>`, etc., instead of generic `div`s.
- **ARIA Attributes:** Use `aria-*` attributes to provide additional context to screen readers. For example, `aria-label`, `aria-describedby`.
- **Keyboard Navigation:** Ensure that all interactive elements are reachable and usable with a keyboard. Use the `tabindex` attribute wisely.
- **Focus Management:** Use `useRef` and `useEffect` to manage focus, especially in modal dialogs or after a user action.
- **Color Contrast:** Ensure a good color contrast ratio for users with low vision.
- **Linting:** Use ESLint plugins like `eslint-plugin-jsx-a11y` to catch accessibility issues.

98. What are React design patterns (Container-Presenter, Compound, HOC)?

React design patterns are common solutions to recurring problems in React development.

- **Container-Presenter (or Smart-Dumb Components):**
    - **Container Component:** Handles the logic (data fetching, state management). It's "smart."

    - **Presenter (or Dumb) Component:** Receives data and callbacks as props and handles the rendering. It's "dumb" and reusable.
    - **With Hooks, this pattern is less common** as components can handle both logic and rendering.
- **Compound Components:** A pattern where components work together implicitly by sharing state through a parent component's context.
    - **Example:** A `Select` component that uses an `Option` component as a child. The `Select` component manages the state and provides context for its children.
- **Higher-Order Component (HOC):** As discussed earlier, a function that takes a component and returns a new, enhanced component.

99. How to migrate a React class component app to hooks?

1. **Start with the smallest components:** Migrate simple, presentational components first.
2. **Replace `this.state` with `useState`:** Use `useState` for simple state.
3. **Replace `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` with `useEffect`:**
    - `componentDidMount`: Use `useEffect` with an empty dependency array (`[]`).
    - `componentDidUpdate`: Use `useEffect` with a dependency array containing the props/state that cause the update.
    - `componentWillUnmount`: Return a cleanup function from `useEffect`.
4. **Replace `this.context` with `useContext`:** Consume a context using the `useContext` hook.
5. **Replace HOCs and render props with custom hooks:** Refactor shared logic into custom hooks.
6. **Test thoroughly:** Ensure the refactored components behave identically.

100. What are common performance bottlenecks in React and how to fix them?

- **Unnecessary Re-renders:** The most common bottleneck.
    - **Fix:** Use `React.memo`, `useMemo`, and `useCallback`.
- **Large Bundle Size:** Slows down initial page load.
    - **Fix:** Code splitting, tree shaking, and using smaller libraries.
- **Deep Component Trees:** Can make debugging difficult and lead to performance issues if not memoized.
    - **Fix:** Flatten the tree where possible, use state colocation, and consider using Context API.
- **Expensive Calculations:** A component's render method or body is doing a lot of work.
    - **Fix:** Move the calculation to a `useMemo` hook.
- **Long Lists:** Rendering thousands of items at once.
    - **Fix:** Use virtualization/windowing libraries like `react-window`.
- **Prop Drilling:** Passing props through many layers of components.
    - **Fix:** Use Context API or a state management library.