Module 9 - Introduction to React.js

Introduction to React.js

Question 1: What is React.js? How is it different from other JavaScript frameworks and libraries?

React.js is a popular **JavaScript library** developed by **Facebook** for building **user interfaces**, especially **single-page applications** (SPAs). It allows developers to create **reusable UI components** and manage dynamic views efficiently.

Key Features of React:

- **Component-Based Architecture**: UI is broken into small, reusable pieces.
- **Virtual DOM**: Improves performance by minimizing real DOM updates.
- **Declarative Syntax**: You describe *what* the UI should look like, and React takes care of *how* to render it.
- **Unidirectional Data Flow**: Data flows from parent to child components via props.
- **React Hooks**: Enable state and side-effects in functional components.
- **JSX Syntax**: Allows HTML-like code within JavaScript for clearer UI structure.

How React is Different from Other JS Frameworks/Libraries

Feature	React.js	Angular	Vue.js
Type	**Library**	Framework	Framework
Developer	Facebook	Google	Evan You (ex-Google)
Architecture	Component-based	Component + MVC pattern	Component-based
DOM	**Virtual DOM**	Real DOM with change detection	Virtual DOM
Data Binding	**One-way (unidirectional)**	Two-way	Two-way (with option for one-way)
Learning Curve	Moderate	Steep	Easy to Moderate
Size C Flexibi	Small C Flexible	Large C Opinionat	Lightweight C

lity		ed	Flexibl e
Use Case	UI layer of web apps	Full app with routing, services	UI layer or full apps

Question 2: Explain the core principles of React such as the virtual DOM and component- based architecture.

React is built on a few powerful and efficient principles that make it highly scalable and fast for building modern web applications. The **two most important concepts** are:

1. **Component-Based Architecture**

What It Means:

React breaks the UI into **independent, reusable components**—each responsible for rendering a small, specific part of the user interface.

Example:

You might have components like:

- <Header />
- <Navbar />
- <ProductCard />
- <Footer />

Each component can have its own:

- **HTML structure (via JSX)**

- **Styling**
- **Logic (state, events)**

2. **Virtual DOM (Document Object Model) What It Is:**

The **Virtual DOM** is a **lightweight copy** of the real DOM kept in memory. React uses it to **efficiently update the UI**.

Working **:**

1. When state or props change, React **creates a new virtual DOM.**
2. It **compares** the new virtual DOM to the previous one (**diffing**).
3. React then **calculates the minimal changes** needed.
4. It **updates only the changed parts** in the real DOM (not the entire page).

3. **Declarative UI**

- You describe **what** the UI should look like based on current data, not how to change it.

jsx
return <h1>{isLoggedIn ? 'Welcome!' : 'Please log in'}</h1>;

4. **Unidirectional Data Flow**

- Data flows **from parent to child** via **props**.
- Makes the app predictable and easier to debug.

Summary

Principle	Description
Component Architecture	UI is divided into reusable, isolated components
Principle	**Description**
Virtual DOM	Efficient UI updates by comparing a virtual copy to the real DOM
Declarative UI	Focus on **what** the UI should show, not **how** to update it
Unidirectional Data Flow	Predictable state and props flow from top to bottom

Question 3: What are the advantages of using React.js in web development?

React.js offers a powerful, efficient, and developer-friendly approach to building modern web applications. Here are the main benefits:

1. Fast Rendering with Virtual DOM

- React uses a Virtual DOM to update only the parts of the page that change.
- This makes UI updates fast and efficient, even in large, dynamic applications.

2. Component-Based Architecture
   - Build UIs using reusable, self-contained components.
   - Makes code more modular, easier to maintain, and scalable for large projects.
3. Declarative Syntax
   - You define what the UI should look like, and React handles the rendering.
   - Leads to clearer, more readable code and fewer bugs.
4. Unidirectional Data Flow
   - Data flows in one direction (parent → child), making the app logic predictable and easier to debug.
5. State Management with Hooks
   - With React Hooks (useState, useEffect, etc.), managing component state is simpler and cleaner.
   - Hooks allow you to write logic without converting to class components.
6. Large Ecosystem C Community
   - Rich ecosystem of tools like:
     - React Router for routing

- Redux or Context API for state management

    - Next.js for server-side rendering
- Huge community support and regular updates.

7. SEO-Friendly (with tools like Next.js)
    - React by itself is a client-side library, but when paired with Next.js, it supports server-side rendering, which improves SEO and initial load time.
8. Cross-Platform Development
    - With React Native, you can build mobile apps using the same React principles and components.
9. Developer Tools
    - React Developer Tools (browser extension) helps with:
        - Inspecting components
        - Monitoring props and state
        - Debugging

efficiently

Summary Table

Advantage	Benefit
Virtual DOM	Fast UI updates
Component-Based Structure	Reusability and maintainability
Declarative UI	Simpler code and fewer bugs

Unidirectional Data Flow	Predictable and easier to debug
React Hooks	Manage state in functional components
Large Ecosystem	Tools and libraries for every need
SEO Support (with Next.js)	Better search engine visibility
React Native	Web + mobile development with the same logic
Dev Tools	Efficient debugging and inspection

2. JSX

(JavaScript XML)

Question 1: What is JSX in React.js? Why is it used?

JSX stands for JavaScript XML. It is a syntax extension for JavaScript that allows you to write HTML-like code inside JavaScript, which React then transforms into React elements.

Why Use JSX?
JSX makes it easier to:

- Visualize the UI structure
- Write cleaner, more readable code
- Combine markup and logic in the same place

Example:
Without JSX (using React.createElement):
jsx
Copy
Edit
React.createElement('h1',
null, 'Hello, world!'); With
JSX (simpler and clearer):
jsx
Copy
Edit
<h1>Hello, world!</h1>

This gets compiled into React.createElement() behind the scenes.

Benefits of JSX:

Feature	Why It’s Useful
HTML-like Syntax	Makes UI easier to understand
Tighter Integration	Logic + layout live together in one place
Compile-Time Error Checking	JSX helps catch syntax issues early
React Element Creation	Transforms into React.createElement() for performance

Question 2: How is JSX different from regular JavaScript? Can you write JavaScript inside JSX?

JSX and **regular JavaScript** are closely related, but there are a few key differences between them.

1. **Syntax:**

- **JSX** is an **HTML-like syntax** used within JavaScript code.
- **Regular JavaScript** is more straightforward and does not allow HTML-like tags directly inside it.

Example:
Jsx:
const element = <h1>Hello, World!</h1>;

javascript:
const element =
document.createElement('
h1'); element.textContent
= 'Hello, World!';

2. **Embedding Expressions in JSX:**
In JSX, you can **embed JavaScript expressions** inside {}.
3. **Attributes:**

- In JSX, **HTML attributes** are written in **camelCase**, while in regular JavaScript they

use the **hyphenated style**.

Yes, you can write JavaScript inside JSX using curly braces {}. This allows you to embed expressions, including variables, functions, and even conditional logic, directly within the JSX.
Examples:

1. Variables inside JSX:
2. JavaScript Functions inside JSX:
3. Conditional Rendering inside JSX:
4. Loops inside JSX (using map):

Feature	**JSX**	**Regular JavaScript**
Syntax	HTML-like syntax within JavaScript	Standard JavaScript syntax
Expressions	Can embed JavaScript expressions with {}	JavaScript expressions outside HTML
Attributes	Uses **camelCase** (e.g., className)	Uses **kebab-case** (e.g., class)
Event Handlers	Written like HTML attributes (e.g., onClick={handleClick})	JavaScript functions assigned to event listeners (addEventListener)

Question 3: Discuss the importance of using curly braces {} in JSX expressions.

In JSX, curly braces {} are used to embed JavaScript expressions inside the JSX markup. This is a key feature of JSX that allows you to integrate dynamic content and logic into your UI, making it interactive and flexible.

Why Use Curly Braces {}?

1. Embed JavaScript Expressions:
   Curly braces allow you to insert JavaScript expressions directly within JSX. This is crucial because JSX itself is an HTML-like syntax, but React needs a way to mix dynamic data with the static structure of the UI.
   - JavaScript Expressions include variables, functions, arithmetic operations, ternary operators, and more.

2. Dynamic Content:
   Curly braces allow you to render dynamic content based on the component's state or props.

3. Conditions and Logic in JSX:

You can use conditional rendering and expressions inside curly braces to control what content to display based on certain conditions.

4. Loops and Mapping:
   Curly braces also allow you to loop through data (using functions like map()) and render a list of elements dynamically.

5. Embedding Function Calls:
   You can also call functions inside the curly braces to return values that will be rendered dynamically.

Components
(Functional s Class Components)

Que 1. What are components in React? Explain the difference between functional components and class components.

Ans: Components are the building blocks of a React application. They let you split the UI into independent, reusable pieces that can be managed separately. Each component defines a part of the UI and can maintain its own state, accept inputs (called props), and render UI elements.

Types of Components in React

React has two main types of components:

1. **Functional Components**
2. **Class Components**

---

1. **Functional Components**

- These are **JavaScript functions** that return JSX (a

syntax extension for rendering UI).

- **Stateless** in older React, but with **React Hooks** (like useState, useEffect), they can now have state and side effects.
- Preferred for most modern development due to simplicity and better performance.

2. **Class Components**

- Use **ES6 classes** and extend from React.Component.
- Must use a **constructor** for setting state and this keyword for referencing class members.
- Used more before React Hooks were introduced.
- **Key Differences**

Feature	Functional Components	Class Components
Syntax	JavaScript function	ES6 class
State Management	`useState` (Hooks)	`this.state` and `setState`
Lifecycle Methods	`useEffect` and other hooks	`componentDidMount`, etc.
`this` keyword	Not required	Required
Code Length	Shorter and cleaner	More verbose

Performance	Slightly better	Slightly heavier
Feature	**Functional Components**	**Class Components**
Modern Usage	Highly recommended (Hooks)	Older approach

Que 2. How do you pass data to a component using props?

In React, **props** (short for "properties") are used to pass data from a **parent component** to a **child component**. Props are **read-only** and cannot be modified by the child.

Working

1. **Pass props in the parent component**
2. **Access props in the child component using parameters (props or destructuring)**

Example

1. Parent Component (App.js)

```
import React

from 'react';
```

```
import Greeting
from './Greeting';
function App() {
 return (
  <div>
   <Greeting name="Alice" />
   <Greeting name="Bob" />
  </div>
 );
}
export default App;
```

2. Child Component (Greeting.js)

```
import React from 'react';
// Using
destructuring
function
Greeting({ name
}) {
 return <h2>Hello, {name}!</h2>;
}
```

```
export default Greeting;
```

Que 3. What is the role of render() in class components?

In React class components, the render() method is required and plays a central role. It defines what the UI should look like for that component.

Key Points:

- The render() method returns JSX (or null) that tells React what to display on the screen.
- It is called automatically whenever the component's state or props change.
- It should be a pure function—meaning it should not modify the component's state or interact with external systems (like APIs).

Props and State

Question 1: What are props in React.js? How are props different from state?

Props (short for "properties") in React are a way to pass data from a parent component to a child component. They are like function arguments in JavaScript and are used to customize or configure components.

Key Features of Props:

- Read-only (immutable inside the child)
- Passed from parent to child
- Can be of any data type (string, number, object, array, function, etc.)
- Accessed in components using props or destructuring

Props vs State in React

Feature	Props	State
Definition	Data passed from parent to child	Internal data managed by the component
Mutability	**Immutable** (read-only)	**Mutable** (can be updated)
Source	Provided by parent	Defined and updated

	component	within the component
Usage	Customize child components	Manage component behavior/data over time
Component Type	Used in **both** functional and class components	Mainly used in components that require interactivity
Example	`<Greeting name="John" />`	`this.setState({ count: 1 })` or `useState(1)`

Question 2: Explain the concept of state in React and how it is used to manage component data.

State in React is a built-in object used to **store dynamic data** within a component. Unlike props, which are passed **from parent to child**, state is **managed within the component itself** and can **change over time**, causing the component to **re-render** and update the UI.

State is used to:

- Track user input

- Handle UI updates (e.g., show/hide elements)
- Store data that changes (like counters, form inputs, API responses)

State in Functional Components (with Hooks)

React uses the useState() hook to manage state in functional components.

State in Class Components

In class components, state is defined as an object and updated with this.setState().

Props vs State (Quick Recap)

Feature	Props	State
Source	Passed from parent	Defined inside the component
Mutability	Immutable	Mutable (can be updated)
Usage	Pass static or dynamic data	Store and manage dynamic data

Question 3: Why is this.setState() used in class components, and how does it work?

In React class components, this.setState() is used to update the component's state. It's the only correct way to change state and trigger a re-render of the component with the new data.

How It Works

- State should never be modified directly (e.g., this.state.count = 1).
- this.setState() schedules an update to the component's state and tells React to re-render the component.
- React merges the new state with the existing state object

Handling Events in React

Here's a clear and concise breakdown of your questions on React event handling:

Question 1: How are events handled in React compared to vanilla JavaScript? Explain the concept of synthetic events.

React vs Vanilla JS Event Handling:

Aspect	Vanilla JavaScript	React
Syntax	element.addEventListener('click', handler)	<button onClick={handler}>Click<

Aspect	Vanilla JavaScript	React
		/button>
Event Binding	Done manually	Handled declaratively through JSX
Context (this)	Must manage manually	Must bind manually in class components
Clean up	Remove manually if needed	Handled by React during unmounting

Synthetic Events in React:

- React uses a wrapper around native events called **SyntheticEvent**.
- It normalizes events so that they behave consistently across all browsers.
- Synthetic events wrap around the browser's native events and provide the same interface.

Example:

```
function handleClick(event) {
console.log(event.type); // SyntheticEvent
}

<button onClick={handleClick}>Click Me</button>
```

Question 2: What are some common event handlers in

React.js? Provide examples of onClick, onChange, and onSubmit.

1. onClick

Triggered when an element is clicked.

```
function handleClick() {
 alert("Button clicked!");
}

<button onClick={handleClick}>Click Me</button>
```

2. onChange

Triggered when input value changes.

```
function handleChange(event) {
console.log("Input value:", event.target.value);
}

<input type="text" onChange={handleChange} />
```

3. onSubmit

Triggered when a form is submitted.

```
function handleSubmit(event) {
 event.preventDefault();
 alert("Form submitted!");
}

<form onSubmit={handleSubmit}>
```

<input type="text" />

<button type="submit">Submit</button>

</form>

Question 3: Why do you need to bind event handlers in class components?

Reason for Binding:

In JavaScript class methods, this is not bound by default. In React **class components**, if you reference this inside an event handler, it will be undefined unless you bind it.

Solution:

1.**Bind in constructor:**

class MyComponent extends React.Component {

constructor(props) {

super(props);

this.handleClick = this.handleClick.bind(this);

}

handleClick() {

console.log("Clicked!", this);

}

render() {

return <button onClick={this.handleClick}>Click

Me</button>;

}

}

2.**Use class fields (ES6+):**

```
class MyComponent extends React.Component {
  handleClick = () => {
    console.log("Clicked!", this);
  };

  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

Conditional Rendering

Question 1: What is conditional rendering in React? How can you conditionally render elements in a React component?

Definition:

Conditional rendering in React means displaying different UI elements or components **based on certain conditions** (e.g., state, props, etc.).

React renders elements like functions: it evaluates expressions in JSX and shows the result in the UI.

How to Conditionally Render:

You can use JavaScript control structures like:

- if-else
- ternary operator
- && (logical AND)
- switch
- Conditional functions

Example:

```
function Greeting(props) {
const isLoggedIn = props.isLoggedIn;
if (isLoggedIn) {
return <h1>Welcome back!</h1>;
}
return <h1>Please log in.</h1>;
}
```

Question 2: Explain how if-else, ternary operators, and && (logical AND) are used in JSX for conditional rendering.

1. if-else Statement

Use it outside the return statement (not directly inside JSX).

```
function Message(props) {
let message;
if (props.isLoggedIn) {
```

```
message = <h1>Welcome back!</h1>;
} else {
message = <h1>Please sign up.</h1>;
}
return message;
}
```

2. Ternary Operator

Use it **inside JSX** for inline conditional rendering.

```
function UserStatus(props) {
return (
<div>
{props.isLoggedIn ? <p>You are logged in.</p> :
<p>You are not logged in.</p>}
</div>
);
}
```

3. && (Logical AND) Operator

Use when you want to render something **only when a condition is true**.

```
function Notifications(props) {
return (
<div>
{props.hasMessages && <p>You have new
```

```
messages!</p>}
</div>
);
}
```

- If hasMessages is true, the <p> is shown.
- If hasMessages is false, React renders null.

Question 1: How do you render a list of items in React? Why is it important to use keys when rendering lists?

Rendering a List in React:

You render a list by **mapping** over an array and returning JSX for each item.

Example:

```
function ItemList() {
  const items = ['Apple', 'Banana', 'Cherry'];

  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ul>
  );
}
```

Why Are Keys Important?

- **Keys help React identify which items have changed, been added, or removed.**
- They allow React to **optimize re-rendering** of lists for better performance.
- Without keys, React has to re-render the entire list unnecessarily.

Question 2: What are keys in React, and what happens if you do not provide a unique key?

What Are Keys?

- **Keys are unique identifiers** assigned to list elements.
- They should be **stable, predictable, and unique** (e.g., IDs from a database, not array indexes unless

there's no alternative).

What Happens Without Unique Keys?

If you don’t provide keys or use non-unique ones:

- React may **mix up components** during re-renders.
- It can lead to **incorrect UI behavior** such as:
  - Losing component state.
  - Wrong elements being updated or removed.
- React will show a **warning** in the console:

"Each child in a list should have a unique 'key' prop."

Bad Example (using index as key):

```
{items.map((item, index) => (
  <li key={index}>{item}</li>
))}
```

- This may cause problems if items are reordered or deleted.

Good Example (using unique ID):

```
const items = [
  { id: 101, name: 'Apple' },
  { id: 102, name: 'Banana' },
];

{items.map(item => (
  <li key={item.id}>{item.name}</li>
))}
```

Forms in React

Question 1: How do you handle forms in React? Explain the concept of controlled components.

Handling Forms in React:

In React, form handling involves:

1. Storing form input values in **component state** (using useState or this.state).
2. Updating state with an **onChange** event handler.
3. Handling form submission with **onSubmit**.

Controlled Components:

A **controlled component** is a form element (e.g., input, textarea, select) whose **value is controlled by React state**.

Example:

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


function MyForm() {
  const [name, setName] = useState('');


  const handleChange = (event) => {
    setName(event.target.value); // Update state
  };


  const handleSubmit = (event) => {
```

```
    event.preventDefault();      // Prevent page reload
    alert('Submitted: ' + name);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" value={name}
onChange={handleChange} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

- The input's value is **bound to the state** name.
- It updates using onChange, making it a **controlled component**.

Question 2: What is the difference between controlled and uncontrolled components in React?

Feature	**Controlled Component**	**Uncontrolled Component**
Control source	React state (useState, this.state)	DOM (uses ref to access values)
Data handling	React controls input value	Browser DOM controls input value
Best for	Dynamic validation, instant feedback,	Simple forms, quick setup, third-party

Feature	Controlled Component form logic	Uncontrolled Component integrations
onChange required?	✅ Yes	❌ No
Example input	<input value={state} onChange={...} />	<input ref={inputRef} />

Uncontrolled Component Example:

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

function UncontrolledForm() {
  const inputRef = useRef();

  const handleSubmit = (event) => {
    event.preventDefault();
    alert('Submitted: ' + inputRef.current.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" ref={inputRef} />
      <button type="submit">Submit</button>
    </form>
```

```
  );
}
```

Lifecycle Methods (Class Components)

Question 1: What are lifecycle methods in React class components? Describe the phases of a component's lifecycle.

What Are Lifecycle Methods?

Lifecycle methods are **special functions** in React **class components** that are called at **different stages** of a component's existence:

- When it is **created**
- When it is **updated**
- When it is **destroyed**

Phases of a Component's Lifecycle:

Phase	Description
Mounting	Component is created and inserted into the DOM
Updating	Component is re-rendered due to state/props changes
Unmounting	Component is removed from the DOM

Lifecycle Methods by Phase:

1. **Mounting (Component is being added to the DOM):**
    - constructor()

- static getDerivedStateFromProps()
- render()
- componentDidMount()

2. **Updating (State or props have changed):**
   - static getDerivedStateFromProps()
   - shouldComponentUpdate()
   - render()
   - getSnapshotBeforeUpdate()
   - componentDidUpdate()
3. **Unmounting (Component is being removed):**
   - componentWillUnmount()

Question 2: Explain the purpose of componentDidMount(), componentDidUpdate(), and componentWillUnmount().

- **1. componentDidMount()**

- Called **once** after the component is mounted (rendered to the DOM).
- Use it for:
  - Fetching data from APIs
  - Adding event listeners
  - Setting up timers

Example:

```
componentDidMount() {
  console.log('Component mounted');
```

```
  fetch('/api/data')
    .then(response => response.json())
    .then(data => this.setState({ data }));
}
```

- **2. componentDidUpdate(prevProps, prevState)**
  - Called **after an update** (state or props change).
  - Use it to:
    - React to state/prop changes
    - Fetch new data if props have changed

Example:

```
componentDidUpdate(prevProps) {
  if (this.props.id !== prevProps.id) {
    // Fetch new data when prop `id` changes
    this.fetchData(this.props.id);
  }
}
```

- **3. componentWillUnmount()**
  - Called **right before the component is removed** from the DOM.
  - Use it to:
    - Clean up event listeners
    - Clear intervals/timeouts

- Cancel API calls

Example:

```
componentWillUnmount() {
  console.log('Component will unmount');
  clearInterval(this.timerID); // Clean up a timer
}
```

Hooks (useState, useEffect, useReducer, useMemo, useRef, useCallback)

Question 1: What are React Hooks? How do useState() and useEffect() hooks work in functional components?

- **What Are React Hooks?**

Hooks are **functions** that let you use **React features like state and lifecycle methods** in **functional components**, without writing class components.

React introduced Hooks in **version 16.8**.

- **useState()**

. Allows you to add **state** to functional components.

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

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

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

- **useEffect()**
  - Allows you to perform **side effects** like data fetching, DOM updates, and timers.
  - Acts like **componentDidMount**, **componentDidUpdate**, and **componentWillUnmount** combined.

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

function Timer() {
  useEffect(() => {
    const timer = setInterval(() => {
      console.log("Tick...");
    }, 1000);

    return () => clearInterval(timer); // cleanup like componentWillUnmount
```

```
  }, []); // Empty dependency array = run only once (on mount)
}
```

✅ **Question 2: What problems did hooks solve in React development? Why are hooks considered important?**

🔹 **Problems Hooks Solved:**

1. **No logic reuse in class components**
   → Custom hooks allow reuse of logic across components.
2. **Too much boilerplate in class components**
   → Hooks simplify code with fewer lines.
3. **Confusing this context**
   → Hooks don't require this at all.
4. **Split logic across lifecycle methods**
   → useEffect combines related logic in one place.

✅ **Why Hooks Matter:**

- Make **functional components powerful** and **stateful**.
- Enable **better separation of concerns** with custom hooks.
- Improve **readability, reuse, and testability**.

✅ **Question 3: What is useReducer? How is it used in React apps?**

- **useReducer() is an alternative to useState() for managing complex state logic, like:**
  - Multiple related state values
  - State transitions based on actions

```
import { 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 (
    <>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
```

```
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
    </>
  );
}
```

✅ Question 4: What is the purpose of useCallback and useMemo Hooks?

Hook	Purpose
useCallback(fn, deps)	Returns a **memoized version of a callback function**, so it's not re-created on every render.
useMemo(() => value, deps)	Returns a **memoized value** to avoid recalculating expensive operations.

- **useCallback Example:**

```
const memoizedCallback = useCallback(() => {
  doSomething();
}, [dependencies]);
```

- **useMemo Example:**

```
const expensiveValue = useMemo(() => computeHeavy(data), [data]);
```

✅ Question 5: What's the Difference between useCallback & useMemo?

Feature	useCallback	useMemo
Returns	A **memoized function**	A **memoized value**
Use for	Preventing function re-creation	Avoiding expensive recalculations
Syntax	useCallback(fn, deps)	useMemo(() => value, deps)

✅ Question 6: What is useRef? How does it work in a React app?

- 🔹 **Purpose:**
  - useRef() creates a **mutable reference** that persists across renders.
  - Commonly used to:
    - Access DOM elements
    - Store values that don't cause re-renders

- 🔹 **Example - Accessing DOM:**

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

function InputFocus() {
  const inputRef = useRef();

  useEffect(() => {
    inputRef.current.focus(); // focus input on mount
```

```
  }, []);

  return <input ref={inputRef} />;
}
```

- **Example - Storing Value:**

```
const renderCount = useRef(0);
renderCount.current += 1;
```

Note: Updating useRef.current does **not** cause a re-render.

Routing in React (React Router)

Question 1: What is React Router? How does it handle routing in single-page applications?

- **What is React Router?**
  - **React Router** is a standard routing library for React applications.
  - It enables **navigation** between components **without refreshing the page**.
  - It's used to create **multi-page experiences in single-page applications (SPAs)**.
- **How Routing Works in SPAs:**

In a typical SPA:

- The app loads a **single HTML file** (index.html).
- React Router updates the **URL in the browser** without reloading the page.
- It determines **which component to render** based on the URL using **JavaScript**, not server-based routing.

Example:

```
import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';
import Home from './Home';
import About from './About';

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </Router>
  );
}
```

Question 2: Explain the difference between BrowserRouter, Route, Link, and Switch components in React Router.

Component	Purpose
BrowserRouter	Wraps the app and enables React Router to use the **HTML5 history API** for clean URLs.
Route	Defines a route/path and tells React

Component	Purpose
	which component to render when that path matches.
Link	Replaces <a> tags. Used to **navigate between routes** without a full page reload.
Switch (v5)	Renders **only the first <Route> that matches**. Replaced by Routes in **React Router v6**.

BrowserRouter Example:

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


<BrowserRouter>
  {/* Your Routes go here */}
</BrowserRouter>
```

Route Example (React Router v6):

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

Link Example:

```
<Link to="/about">Go to About Page</Link>
```

- **Note:** Unlike <a href>, <Link to> prevents full page reloads and preserves SPA behavior.

Switch Example (React Router v5):

```
import { Switch, Route } from 'react-router-dom';
```

```
<Switch>
  <Route path="/" exact component={Home} />
  <Route path="/about" component={About} />
</Switch>
```

React - JSON-server and Firebase Real Time Database

✅ Question 1: What do you mean by RESTful Web Services?

🔹 **RESTful Web Services:**

REST stands for **Representational State Transfer**. It's a set of rules and conventions for building **web services** that can be accessed via **HTTP**.

🔹 **Key Features:**

- Operate over **HTTP methods**: GET, POST, PUT, DELETE
- Use **URLs to represent resources**
- **Stateless** (each request is independent)
- Typically return data in **JSON** format

📌 **Example:**

- GET /users → Get all users
- GET /users/1 → Get user with ID 1
- POST /users → Create a new user

✅ Question 2: What is JSON-Server? How is it used in React?

🔹 **JSON Server:**

A lightweight **fake REST API server** that runs on a local machine. It allows you to mock backend APIs with just a **db.json** file.

🔹 **Why Use It in React:**

- Great for **frontend development** when real backend is not available.
- Simulates **CRUD operations** for local testing.

📦 **Installation & Usage:**

```
npm install -g json-server
json-server --watch db.json --port 3001
```

✅ **Question 3: How do you fetch data from a JSON-Server API in React? Explain the role of fetch() or axios().**

🔹 **1. Using fetch():**

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

function App() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch('http://localhost:3001/users')
      .then(response => response.json())
      .then(data => setUsers(data));
```

```
  }, []);

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

- **2. Using axios() (a popular HTTP client):**

```
npm install axios
import axios from 'axios';

useEffect(() => {
  axios.get('http://localhost:3001/users')
    .then(response => setUsers(response.data))
    .catch(error => console.error(error));
}, []);
```

Role of fetch/axios:

- **fetch()** is built-in JavaScript for HTTP requests.
- **axios** simplifies syntax and automatically parses JSON, handles timeouts, interceptors, etc.

Question 4: What is Firebase? What features does Firebase offer?

🔹 **Firebase:**

Firebase is a **Backend-as-a-Service (BaaS)** by Google.

🔹 **Core Features:**

Category	Features
Auth	Email/password, Google, Facebook login
Database	Firestore (NoSQL), Realtime Database
Storage	Upload images/files to cloud storage
Hosting	Host your React apps easily
Functions	Serverless cloud functions (backend logic)
Notifications	Push notifications, analytics

Why Use Firebase:

- Scalable, fast, secure
- Easy to integrate with React
- No need to manage a backend

Question 5: Importance of Handling Errors & Loading States in React When Using APIs

🔹 **Why It's Important:**

- **User experience**: Feedback during loading and error states improves usability.
- **Stability**: Prevents crashes if API fails.
- **Debugging**: Easier to trace issues.

📘 **Example with fetch:**

```
function App() {
```

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

  useEffect(() => {
    fetch('http://localhost:3001/users')
      .then(response => {
        if (!response.ok) throw new Error('Network error');
        return response.json();
      })
      .then(data => setData(data))
      .catch(error => setError(error.message))
      .finally(() => setLoading(false));
  }, []);

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

  return <ul>{data.map(u => <li
key={u.id}>{u.name}</li>)}</ul>;
}
```

- **Best Practices:**
  - Use loading to show spinners/skeletons
  - Use try/catch or .catch() to handle failures

- Show fallback UI or retry options

Context API

✅ **Question 1: What is the Context API in React? How is it used to manage global state across multiple components?**

🔹 **What is the Context API?**

The **Context API** is a feature in React that allows you to **share state or values globally** across the component tree without passing props manually at every level.

🔹 **Why use it?**

- To **avoid prop drilling** (passing props down multiple levels).
- Useful for managing app-wide data like:
  - Auth state
  - Theme
  - Language
  - User info

🔹 **Key Concepts:**

1. createContext() — Create a context.
2. Provider — Makes data available to child components.
3. useContext() — Allows access to context data in any component.

✅ Question 2: How are createContext() and useContext() used in React for sharing state?

🔹 Step-by-step Example:

1. Create Context

```
import { createContext } from 'react';


const ThemeContext = createContext(); // default value optional
```

2. Provide Context

Wrap a part of the component tree with ThemeContext.Provider.

```
import React, { useState } from 'react';
import { ThemeContext } from './ThemeContext';


function App() {
  const [theme, setTheme] = useState('light');


  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}
```

3. Consume Context with useContext()

```
import { useContext } from 'react';
import { ThemeContext } from './ThemeContext';

function Toolbar() {
  const { theme, setTheme } =
useContext(ThemeContext);

  return (
    <>
      <p>Current theme: {theme}</p>
      <button onClick={() => setTheme(theme === 'light' ?
'dark' : 'light')}>
        Toggle Theme
      </button>
    </>
  );
}
```

State Management (Redux, Redux-Toolkit or Recoil)

What is Redux, and why is it used in React applications?

- **What is Redux?**

Redux is a **predictable state container** for JavaScript apps. It helps you manage and centralize application state in a single store.

- **Why Use Redux in React?**

- To manage **complex global state**.
- Ensures **consistent state updates** across components.
- Makes **debugging easier** using tools like Redux DevTools.
- Encourages **unidirectional data flow**.

Core Concepts of Redux

Concept	Description
Store	A centralized **object** that holds the app's state.
Action	A plain JS object that **describes what happened**.
Reducer	A pure function that **takes current state and action**, returns new state.

Example:

1. Action

```
const increment = { type: 'INCREMENT' };
```

2. Reducer

```
function counterReducer(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
```

```
    default:
      return state;
  }
}
```

3. Store

```
import { createStore } from 'redux';
const store = createStore(counterReducer);

store.dispatch({ type: 'INCREMENT' });
console.log(store.getState()); // 1
```

Redux With React:

You typically use the react-redux bindings with:

- Provider – makes store available to React
- useSelector() – reads state
- useDispatch() – sends actions

Question 2: How does Recoil simplify state management in React compared to Redux?

- **What is Recoil?**

Recoil is a **state management library developed by Facebook** that integrates deeply with React and uses **hooks** to manage shared state.

Redux vs Recoil: Comparison

Feature	Redux	Recoil
Setup	Requires boilerplate: actions, reducers, store	Minimal setup — uses atom and hooks
Learning Curve	Steep (especially with middleware like thunk)	Easy — follows React's functional style
Data Structure	Centralized single store	Decentralized atoms for shared state
Boilerplate	More (actions, reducers)	Less (direct state management with atoms/selectors)
React Integration	External (react-redux) binding needed	Built-in hook-based, native React approach
Async Handling	Middleware required (thunk/saga)	Easy with selectors (async capable)

Recoil Core Concepts:

Concept	Description
Atom	A piece of state (like useState) that can be shared
Selector	A derived/computed state (like useMemo)

Recoil Example:

```
import { atom, useRecoilState } from 'recoil';

const counterAtom = atom({
  key: 'counter', // unique ID
  default: 0,     // default state
});

function Counter() {
  const [count, setCount] =
useRecoilState(counterAtom);

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