**Introduction to React.js**

**What is React.js? How is it different from other JavaScript frameworks and libraries?**

React.js is a popular JavaScript library created by Meta (Facebook) for building user interfaces, especially in single-page applications. It allows developers to create reusable components that manage their own state and update efficiently when data changes, thanks to its use of a Virtual DOM.

* **React vs Angular**: Angular is a full-fledged framework with built-in routing, forms, and dependency injection. React is more flexible and lightweight but requires third-party tools for some features.
* **React vs Vue**: Vue is also component-based and uses a virtual DOM, but it offers a more opinionated and simpler setup. React has a larger ecosystem and more flexibility.
* **React vs jQuery**: jQuery directly manipulates the DOM, while React uses a virtual DOM for performance. React is more suited for building complex UIs.

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

React is built on two key principles: the Virtual DOM and Component-Based Architecture.

**Virtual DOM**

React uses a Virtual DOM, which is a lightweight copy of the actual DOM. When something changes in a component, React updates the virtual DOM first, compares it with the previous version (a process called diffing), and then updates only the changed parts of the real DOM. This makes UI updates much faster and more efficient.

**Component-Based Architecture**

React encourages building UIs with **components**—small, reusable pieces of code that each represent a part of the UI (like a button, form, or entire page section). Components can manage their own **state** and receive data via **props**. This modular approach makes code more maintainable, scalable, and easy to reuse.

**What are the advantages of using React.js in web development?**

**Component-Based Architecture**React promotes reusable, self-contained components, making code more modular, maintainable, and easier to test.

**Virtual DOM for Better Performance**React updates only the parts of the DOM that change, thanks to its virtual DOM. This improves rendering speed and overall performance, especially in large applications.

**Fast and Efficient**With its optimized rendering and minimal reloading, React makes applications more responsive and smooth.

**Declarative Syntax (JSX)**JSX allows developers to write HTML-like code in JavaScript, making UI code easier to read and debug.

**Strong Ecosystem and Community**React has a huge community, lots of third-party libraries, and tools like Redux, React Router, and Next.js to support complex needs**.**

**SEO-Friendly with Server-Side Rendering**Libraries like Next.js make React apps more SEO-friendly by enabling server-side rendering.

**Easy to Learn and Use**Developers with basic JavaScript knowledge can quickly pick up React, especially since it focuses only on the view layer.

**Cross-Platform Development**  
With React Native, you can build native mobile apps using the same React principles, allowing for shared logic across platform

JSX (JavaScript XML)

**What is JSX in React.js? Why is it used?**

JSX (JavaScript XML) is a syntax extension used in React that allows you to write HTML-like code directly inside JavaScript. It makes it easier to create and visualize React components by blending markup and logic in a clear, concise way.

**JSX**

const element = <h1>Hello, world!</h1>;

**JS**

React.createElement('h1', null, 'Hello, world!');

JSX is not required in React, but it's widely used because:

* It improves **readability** and **maintainability**.
* You can **embed JavaScript expressions** inside JSX using {}.
* It helps you **visualize the UI** structure more clearly.
* Tools like Babel transform JSX into standard JavaScript for browsers to understand.

Overall, JSX makes writing React components more intuitive and speeds up the development process.

**How is JSX different from regular JavaScript? Can you write JavaScript inside JSX?**

**JSX** is different from regular JavaScript in that it allows you to write HTML-like code directly within JavaScript, making it easier to describe what the UI should look like.

While regular JavaScript uses functions like React.createElement() to build UI elements, JSX provides a more readable and concise syntax that gets compiled into those function calls behind the scenes.

const name = "Jane";

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

You can use variables, function calls, math operations, and ternary operators within {}, but not control flow statements like if or for. JSX blends the power of JavaScript with the structure of HTML for more dynamic and maintainable UI code.

**Discuss the importance of using curly braces {} in JSX expressions.**

In JSX, curly braces {} are essential because they allow you to embed JavaScript expressions directly into the markup. This feature makes JSX dynamic and enables developers to create interactive UIs.

Importance of Curly Braces in JSX:

1. Insert Dynamic Content  
   Curly braces let you include variables, function results, and other expressions inside JSX:
2. const user = "Hima";
3. <h1>Hello, {user}!</h1>
4. Conditional Rendering  
   You can display content conditionally using expressions like ternary operators:
5. {isLoggedIn ? <p>Welcome back!</p> : <p>Please log in.</p>}
6. Perform Calculations and Logic  
   Any JavaScript expression can be used to modify how data is displayed:
7. <p>Total: {price \* quantity}</p>
8. Call Functions in JSX  
   You can run functions and show their returned values:
9. <p>{formatDate(date)}</p>

Only expressions are allowed in {} — not full statements like if, for, or while

Curly braces make JSX powerful by allowing developers to combine JavaScript logic with HTML-like syntax, creating flexible and dynamic UI components.

Components (Functional & Class Components)

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

Components are the core building blocks of a React app. They are reusable pieces of UI that return JSX and can manage their own data using props and state.

Types of Components

1. Functional Components

* Defined as JavaScript functions
* Simpler and easier to write
* Use hooks like useState and useEffect for state and side effects

function Hello(props) {

return <h1>Hello, {props.name}</h1>;

}

2. Class Components

* Created using ES6 classes
* Use this.state and lifecycle methods like componentDidMount

class Hello extends React.Component {

render() {

return <h1>Hello, {this.props.name}</h1>;

}

}

Functional components are cleaner and preferred in modern React, while class components are older and more verbose.

**How do you pass data to a component using props?**

In React, you pass data to a component using props (short for "properties"). Props allow you to make components dynamic and reusable by supplying different values from a parent component.

Passing Props

To pass props, you add attributes to the component when you use it:

<Profile name="Alice" age={30} />

Here, name and age are props being passed to the Profile component.

Receiving Props

In a functional component, props are received as a parameter:

function Profile(props) {

return <p>{props.name} is {props.age} years old.</p>;

}

Or using destructuring for cleaner syntax:

function Profile({ name, age }) {

return <p>{name} is {age} years old.</p>;

}

In a class component, props are accessed using this.props:

class Profile extends React.Component {

render() {

return <p>{this.props.name} is {this.props.age} years old.</p>;

}

}

Key Points

* Props are read-only and passed from parent to child.
* They help make components flexible and reusable.
* You can pass any data type: strings, numbers, arrays, functions, or even other components.

**What is the role of render() in class components?**

In React class components, the render() method is essential. It defines what the component should display on the screen. Every class component must include a render() method, which returns JSX—the structure of the UI.

Key Functions of render():

1. Returns JSX:  
   The main job of render() is to return JSX that describes the component’s UI.
2. class Greeting extends React.Component {
3. render() {
4. return <h1>Hello, {this.props.name}!</h1>;
5. }
6. }
7. Automatic Execution:  
   React automatically calls render():
   * When the component first loads (mounts)
   * When there are changes in state or props
8. Pure Function:  
   render() should be a pure function, meaning it shouldn't modify state or cause side effects. It should only return what the UI looks like based on the current data.
9. Single Root Element:  
   It must return one root element or wrap multiple elements in a React fragment (<>...</>).

The render() method is a required part of class components in React. It’s responsible for returning JSX, which React uses to display the UI.

Props and State

**What are props in React.js? How are props different from state?**

Sure! Here's a medium-length explanation:

In **React.js**, **props** and **state** are two core ways to manage and pass data within components.

**Props**

**Props** (short for "properties") are **read-only** values passed from a **parent component to a child component**. They're used to customize or configure child components. For example, if you have a Greeting component that displays a name, you can pass the name as a prop:

function Greeting(props) {

return <h1>Hello, {props.name}!</h1>;

}

// Usage

<Greeting name="Alice" />

Props cannot be modified by the receiving component — they’re controlled by the parent.

**State**

**State** is a data structure that belongs to a component itself and is used to **track changes over time**. Unlike props, state is **mutable** and can be updated using hooks like useState in functional components:

import { useState } from "react";

function Counter() {

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

return (

<>

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

<button onClick={() => setCount(count + 1)}>Increment</button>

</>

);

}

When state changes, the component re-renders with the new state.

**Key Differences**

| **Feature** | **Props** | **State** |
| --- | --- | --- |
| Scope | Passed from parent | Managed within the component |
| Mutability | Immutable | Mutable |
| Purpose | Configures components | Handles dynamic behavior |

**props** are for passing data *into* a component, and **state** is for managing data *within* a component.

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

In **React**, **state** is a special object used to manage **dynamic data** within a component. Unlike props, which are passed from parent to child and are read-only, **state is local and can be changed** by the component itself.

In **functional components**, state is managed using the useState hook:

import { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0); // count is state, setCount updates it

return (

<div>

<p>You clicked {count} times</p>

<button onClick={() => setCount(count + 1)}>Click me</button>

</div>

);

}

In this example, count is a piece of state, and setCount is the function that updates it. When the button is clicked, the state updates, and the component **re-renders automatically** to show the new count.

State is useful for:

* Tracking user input
* Managing UI interactions (like modals, toggles)
* Storing temporary values like counters or form data

**Key Points:**

* State is **local** to the component.
* It is **mutable** (can be updated).
* Updating state triggers a **re-render** of the component.

State is essential for building **interactive and dynamic React apps**.

**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 and trigger a re-render. You shouldn’t update state directly (e.g., this.state.count = 1) because React won’t know it needs to update the UI.

🔧 How it works:

* It merges the new state with the existing state.
* It tells React to re-render the component with updated data.

Ex

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

If the new state depends on the old state, use a function:

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

this.setState() keeps your UI in sync with your data.

Handling Events in React

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

**Event Handling in React vs. Vanilla JS**

In **vanilla JavaScript**, you use addEventListener to attach events:

button.addEventListener("click", () => alert("Clicked!"));

In **React**, you handle events directly in JSX using **camelCase** event names and functions:

<button onClick={() => alert("Clicked!")}>Click me</button>

**What Are Synthetic Events?**

React wraps native events in **Synthetic Events** to provide **cross-browser consistency**. These behave like regular events (event.target, event.preventDefault()), but work the same way across all browsers.

function handleSubmit(e) {

e.preventDefault(); // Synthetic event

}

**Summary:**

React simplifies and standardizes event handling using **synthetic events**, making it more reliable and easier to use than vanilla JS.

**What are some common event handlers in React.js? Provide examples of onClick, onChange, and onSubmit.**

**Common React Event Handlers**

**✅ onClick – Handles clicks:**

<button onClick={() => alert("Clicked!")}>Click Me</button>

**✅ onChange – Tracks input changes:**

<input type="text" onChange={(e) => console.log(e.target.value)} />

**✅ onSubmit – Handles form submission:**

<form onSubmit={(e) => { e.preventDefault(); alert("Submitted!"); }}>

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

</form>

These use **React's Synthetic Events**, which behave like native events but work consistently across browsers.

**Why do you need to bind event handlers in class components?**

**Why Bind Event Handlers in Class Components?**

**In React class components, you need to bind event handlers so that this refers to the component instance.**

**Without binding, this inside the handler is undefined when the method is used as a callback.**

**Example Fix:**

**Bind in the constructor:**

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

**Or use an arrow function:**

**handleClick = () => {**

**// 'this' is automatically bound**

**};**

**Binding ensures this works correctly inside event handlers in class components.**

Conditional Rendering

**What is conditional rendering in React? How can you conditionally render elements in a React component?**

Sure! Here's a **medium-short** explanation:

**🔹 What is Conditional Rendering in React?**

**Conditional rendering** means displaying elements or components **based on a condition** (like state or props). It lets you control what gets shown in the UI.

**✅ Using if statements:**

if (isLoggedIn) {

return <h1>Welcome back!</h1>;

} else {

return <h1>Please log in.</h1>;

}

**✅ Using ternary operator:**

<p>{isLoggedIn ? "Welcome!" : "Please sign in."}</p>

**✅ Using && (for one condition):**

{isLoggedIn && <button>Logout</button>}

React lets you **conditionally render** elements using JavaScript logic like if, ? :, and &&, giving you flexible control over what appears in the UI.

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

React uses standard JavaScript syntax to conditionally render elements inside JSX. Here are the common ways:

**if-else Statement**

Used **outside** JSX, typically before the return statement.

if (isLoggedIn) {

return <h1>Welcome!</h1>;

} else {

return <h1>Please log in.</h1>;

}

**Ternary Operator (? :)**

Used **inside** JSX for inline conditions.

<h1>{isLoggedIn ? "Welcome!" : "Login"}</h1>

**Logical AND (&&)**

Renders something **only if** the condition is true.

{isLoggedIn && <button>Logout</button>}

Use if-else for complex logic, the ternary operator for inline conditions, and && for simple one-way checks in JSX.

Lists and Keys

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

To render a list in React, you use **Array.map()** to loop through the items and return JSX for each one:

const fruits = ["Apple", "Banana", "Orange"];

return (

<ul>

{fruits.map((fruit, index) => (

<li key={index}>{fruit}</li>

))}

</ul>

);

**Why Use Keys?**

**Keys** help React **identify which items changed, added, or removed**. This makes updates more efficient and prevents bugs.

* Keys should be **unique** and **stable** (like an ID).
* Avoid using array indexes if the list can change order.

<li key={fruit.id}>{fruit.name}</li>

Use .map() to render lists, and always include a unique key to help React optimize rendering and maintain UI stability.

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

**Keys** are special props used when rendering lists in React. They help React **identify each item** and track changes like additions, deletions, or reordering.

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

**What If You Don’t Use Unique Keys?**

* React may **re-render incorrectly** or **mix up items**.
* Performance may suffer.
* UI bugs can occur (e.g., incorrect input values or animations).

Always use **unique and stable keys** (like IDs). Without them, React can’t reliably track list items, leading to bugs and inefficient updates.

Forms in React

**How do you handle forms in React? Explain the concept of controlled components.**

**Handling Forms in React**

In React, forms are handled by linking **input elements to state**, so you can control and respond to user input.

**Controlled Components**

A **controlled component** is a form element (like <input>) whose value is **controlled by React state**.

function MyForm() {

const [name, setName] = useState("");

const handleChange = (e) => setName(e.target.value);

return (

<form>

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

<p>Your name is: {name}</p>

</form>

);

}

**Why Use Controlled Components?**

* Keeps form data in sync with component state
* Makes it easier to validate, reset, or manipulate inputs

Controlled components link form inputs to state, giving you full control over user input and making form handling more predictable.

**What is the difference between controlled and uncontrolled components in React?**

**Controlled Components** are form elements whose values are controlled by React state.

const [input, setInput] = useState("");

<input value={input} onChange={(e) => setInput(e.target.value)} />

* React manages the input value
* Easier to validate and handle data

**Uncontrolled Components** use the DOM to manage state, often with ref.

const inputRef = useRef();

<input ref={inputRef} />

* React does not control the value
* Useful for simple or non-interactive forms

Controlled = React handles input state

Uncontrolled = DOM handles input state  
Controlled components are preferred for more dynamic and interactive forms.

Lifecycle Methods (Class Components)

**What are lifecycle methods in React class components? Describe the phases of a component’s lifecycle.**

**Lifecycle methods** are special functions in React **class components** that run at specific points during a component’s life—like when it's **created**, **updated**, or **removed** from the DOM.

**Lifecycle Phases:**

1. **Mounting** (Component is added to the DOM):
   * constructor()
   * componentDidMount()
2. **Updating** (Props or state change):
   * shouldComponentUpdate()
   * componentDidUpdate()
3. **Unmounting** (Component is removed from the DOM):
   * componentWillUnmount()

Lifecycle methods help you **run code at specific stages** of a class component’s life—for setup, updates, or cleanup.

**Explain the purpose of componentDidMount(), componentDidUpdate(), and componentWillUnmount().**

**componentDidMount()**

* Runs **once** after the component is added to the DOM.
* Great for **fetching data**, setting up **timers**, or initializing **subscriptions**.

componentDidMount() {

console.log("Component mounted");

}

**componentDidUpdate(prevProps, prevState)**

* Runs **after every update** (when props or state change).
* Useful for **reacting to changes**, like updating the DOM or making API calls.

componentDidUpdate(prevProps) {

if (prevProps.id !== this.props.id) {

// Fetch new data

}

}

**componentWillUnmount()**

* Runs **right before** the component is removed from the DOM.
* Used for **cleanup** like removing timers or unsubscribing from events.

componentWillUnmount() {

clearInterval(this.timer);

}

These methods let you **hook into the component’s lifecycle** to perform actions when it mounts, updates, or unmounts.

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

**What are React hooks? How do useState() and useEffect() hooks work in functional components?**

**Hooks** let you use features like **state** and **lifecycle methods** in **functional components**, without needing classes.

**useState()**

* Adds **state** to a function component.
* Returns a value and a function to update it.

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

**useEffect()**

* Handles **side effects** (e.g., fetching data, timers).
* Runs after render. Can also run on updates or unmount.

useEffect(() => {

// effect code

return () => {

// cleanup code

};

}, []);

useState() manages state, and useEffect() handles side effects—both are key hooks in React functional components.

**What problems did hooks solve in React development? Why are hooks considered an important addition to React?**

Before hooks, managing state and side effects in React required class components, which were often:

* Hard to reuse logic across components
* Messy with large lifecycle methods
* Confusing this binding

Why Are Hooks Important?

Hooks allow:

* State and side effects in functional components
* Cleaner, reusable logic with custom hooks
* Simpler and more readable code

Hooks made it easier to write cleaner, reusable, and more functional React code—solving many limitations of class components.

**What is useReducer ? How we use in react app?**

useReducer is a React hook used to manage complex state logic in functional components. It's an alternative to useState, especially when state updates depend on previous state or involve multiple sub-values.

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;

}

}

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

Then use it in JSX:

<button onClick={() => dispatch({ type: "increment" })}>+</button>

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

useReducer is useful for managing more complex state logic in React apps. It works like Redux but is built into React.

**What is the purpose of useCallback & useMemo Hooks?**

**What’s the Difference between the useCallback & useMemo Hooks?**

**What is useRef ? How to work in react app?**

Routing in React (React Router)

**What is React Router? How does it handle routing in single-page applications?**

**Explain the difference between BrowserRouter, Route, Link, and Switch components in React Router.**

React – JSON-server and Firebase Real Time Database

**What do you mean by RESTful web services?**

**What is Json-Server? How we use in React ?**

**How do you fetch data from a Json-server API in React? Explain the role of fetch() or axios() in making API requests.**

**What is Firebase? What features does Firebase offer?**

**Discuss the importance of handling errors and loading states when working with APIs in React**

Context API

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

**Explain how createContext() and useContext() are used in React for sharing state.**

State Management (Redux, Redux-Toolkit or Recoil)

**What is Redux, and why is it used in React applications? Explain the core concepts of actions, reducers, and the store.**

**How does Recoil simplify state management in React compared to Redux?**