React.js

1. Introduction to React
2. Prerequisites
3. React Components
4. JSX (JavaScript XML)
5. Properties
6. State
7. Event Handling
8. Conditional Rendering
9. Lists & Keys
10. Forms
11. Lifecycle Methods
12. Advanced Hooks
13. Context API
14. Custom Hooks
15. Code Splitting & Lazy Loading
16. Error Boundaries
17. Portals
18. Controlled vs. Uncontrolled Components
19. Higher-Order Components (HOC)
20. Render Props
21. Reconciliation & Keys
22. Server-Side Rendering (SSR)
23. State Management Beyond React
24. Styling in React
25. Testing in React

1. Introduction to React

React.js is an **open-source JavaScript library** for building **fast, interactive, and reusable user interfaces**.
It focuses on building **component-based** frontends for single-page applications (SPAs).

Key Features

- **Component-Based Architecture** – Break UI into reusable parts.
- **Declarative UI** – You tell React *what* to show, React decides *how*.
- **Virtual DOM** – Updates only the parts of the UI that change.
- **One-Way Data Flow** – Data flows from parent to child.
- **JSX Syntax** – Write HTML-like code inside JavaScript**.**

2. Prerequisites

Before starting with React, you should know:

- HTML, CSS, JavaScript
- ES6+ features (let/const, arrow functions, destructuring)
- DOM concepts
- Basic npm usage

3. React Components

React apps are built with **components** — small, reusable pieces of UI.

Types of Components

1. **Functional Components** – Modern, simple, use **Hooks**.
2. **Class Components** – Older style, still supported.

Example – Functional Component

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

Example – Class Component

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

4. JSX (JavaScript XML)

JSX allows HTML-like syntax in JavaScript.

Example

```
const user = "Alice";
const element = <h1>Hello, {user}</h1>;
```

- Must have **one parent element**.
- Use {} for JavaScript expressions.

5. Properties

Props are **read-only** data passed from parent to child.

```
function Greet(props) {
  return <h1>Hello, {props.name}</h1>;
}
<Greet name="John" />
```

6. State

State stores data inside a component and can change over time.

Example with useState

```
import { useState } from 'react';
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </>
  );
```

7. Event Handling

Handling events is similar to handling events on DOM elements, but with some differences in syntax and behavior.

Key Points about Event Handling in React

- React events are named using **camelCase** instead of lowercase.
- You pass a **function** as the event handler, not a string.
- In React, you usually use **synthetic events** which wrap the native browser events for cross-browser compatibility.
- You often define event handler functions inside your component.
- When you want to pass arguments to an event handler, you use an arrow function or bind.

```
function ClickButton() {
  function handleClick() {
  alert("Button clicked!");
  }
  return <button onClick={handleClick}>Click Me</button>;
}
```

8. Conditional Rendering

Conditional rendering in React means displaying different UI elements or components based on some condition or state. It works like regular JavaScript conditions (if statements, ternary operators, logical &&, etc.) but inside React's JSX.

Why use Conditional Rendering?

- To show or hide components or elements dynamically
- To display loading spinners, error messages, or success messages based on state
- To render different components for different user roles or app states

```
function UserGreeting({ isLoggedIn }) {
  return isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please sign up.</h1>;
}
```

9. Lists & Keys

You often need to display multiple items—like a list of users, posts, or products. You do this by **rendering lists** using JavaScript's array methods, usually .map().

When rendering lists, **each element should have a unique key prop**. Keys help React identify which items changed, were added, or removed — improving performance and avoiding bugs.

How to Use Keys

- The key must be **unique among siblings**.
- Usually, a unique ID from the data is best.
- Avoid using array indexes as keys unless you have no better option.

```
const numbers = [1, 2, 3];
const listItems = numbers.map(num => <li key={num}>{num}</li>);
return <ul>{listItems}</ul>;
```

10. Forms

In React, forms are handled a bit differently than plain HTML forms because React controls the form data via state. This is called controlled components.

Controlled Components

- Form inputs like <input>, <textarea>, and <select> maintain their state in React state.
- The displayed value of the input is always driven by React state.
- On every change, you update the React state to keep the input and the state in sync.

```
function MyForm() {
  const [name, setName] = useState("");
  function handleSubmit(e) {
  e.preventDefault();
    alert("Submitted: " + name);
```

```
  }
  return (
    <form onSubmit={handleSubmit}>
      <input value={name} onChange={e => setName(e.target.value)} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

11. Lifecycle Methods

1. Mounting Phase

Called when the component is created and inserted into the DOM.

Key Methods:

- constructor(props)
  Initialize state and bind event handlers. Runs before mounting.
- static getDerivedStateFromProps(props, state)
  Rarely used; lets the component update state based on props before render.
- render()
  Returns the JSX to display.
- componentDidMount()
  Runs after the component is mounted. Good place to fetch data, start timers, set up subscriptions.

2. Updating Phase

Runs when props or state change, causing the component to re-render.

Key Methods:

- static getDerivedStateFromProps(props, state)
  Like in mounting, updates state based on prop changes.
- shouldComponentUpdate(nextProps, nextState)
  Return true or false to control if the component should re-render. Useful for optimization.
- render()
  Re-renders the UI.
- getSnapshotBeforeUpdate(prevProps, prevState)
  Captures information (like scroll position) before DOM updates.

- componentDidUpdate(prevProps, prevState, snapshot)
  Runs after the update is flushed to the DOM. Good for network requests based on prop/state changes.

3. Unmounting Phase

Runs when the component is removed from the DOM.

Key Method:

- componentWillUnmount()
  Clean up things like timers, subscriptions, event listeners.

Class Components

- componentDidMount() – after first render
- componentDidUpdate() – after update
- componentWillUnmount() – before removal

Functional Components (Hooks)

```
useEffect(() => {
console.log("Mounted");
  return () => console.log("Unmounted");
}, []);
```

12. Advanced Hooks

While useState and useEffect are the most commonly used hooks, React offers several powerful hooks for advanced patterns and performance optimization.

1. useReducer

- Alternative to useState for complex state logic.
- Similar to Redux reducer pattern.
- Useful when state depends on previous state or multiple sub-values.

2. useMemo

- Memoizes expensive calculations to avoid recomputing on every render.
- Accepts a function and dependency array.
- Only recalculates value if dependencies change.

3. useCallback

- Returns a memoized version of a callback function.
- Useful to prevent unnecessary re-renders of child components receiving functions as props.

4. useRef

- Returns a mutable object whose .current property persists between renders.
- Used to access DOM elements or store mutable values without causing re-renders.

5. useImperativeHandle

- Customizes the instance value that is exposed when using ref in parent components.
- Mainly used with forwardRef.

6. useLayoutEffect

- Similar to useEffect but fires synchronously after all DOM mutations.
- Useful for reading layout from the DOM and synchronously re-rendering.

7. useDebugValue

- Used to display a label for custom hooks in React DevTools for debugging.

13. Context API

The Context API allows you to **share data across components** without passing props manually at every level (known as "prop drilling").

It's great for global data like:

- User authentication info
- Theme settings (dark/light mode)
- Language preferences
- Any data needed by many components

How Context Works

1. **Create a Context object**
2. **Provide** the context value at a top-level component using a Provider
3. **Consume** the context value in any nested component

Why Use Context API?

- Avoids **prop drilling** where you pass props through many levels.

- Makes global state or settings accessible anywhere in the component tree.
- Cleaner and easier to maintain.

```
const ThemeContext = React.createContext();
function App() {
  return (
    <ThemeContext.Provider  value="dark">
      <Toolbar />
    </ThemeContext.Provider>
  );
}
```

14. Custom Hooks

Custom Hooks are **JavaScript functions** that start with the prefix use and allow you to **reuse stateful logic** across multiple components. They help keep your code clean and DRY (Don't Repeat Yourself).

Why Use Custom Hooks?

- To share logic like fetching data, managing forms, or subscribing to events across components.
- To abstract complex logic out of components.
- To improve code readability and maintainability.

Rules for Custom Hooks

- Must start with use prefix (e.g., useFetch, useForm).
- Can call other hooks like useState, useEffect, etc.
- Should follow the **Rules of Hooks** (only call hooks at the top level, only from React functions).

```
function useCounter(initialValue = 0) {
  const [count, setCount] = useState(initialValue);
  const increment = () => setCount(count + 1);
```

```
  return { count, increment };
}
```

15. Code Splitting & Lazy Loading

Code splitting is a technique to split your app's bundle into smaller chunks that can be loaded on demand. This improves the initial load time and performance by loading only the code needed for the current screen.

Why Use Code Splitting?

- Large apps can have big JavaScript bundles.
- Loading everything upfront slows down the initial page load.
- Code splitting loads parts of the app only when needed, reducing load times.
- Improves user experience by delivering faster initial rendering.

How to Do Code Splitting in React?

React supports code splitting via dynamic import() and the built-in React.lazy and Suspense APIs.

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
<Suspense fallback={<div>Loading...</div>}>
  <OtherComponent />
</Suspense>
```

16. Error Boundaries

Error Boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed.

Why Use Error Boundaries?

- Prevent your whole React app from crashing due to an error in one part.
- Show a user-friendly error message or fallback UI.
- Log errors for debugging or reporting.

How Error Boundaries Work

- Only **class components** can be error boundaries (as of now).
- They implement either or both lifecycle methods:
    - static getDerivedStateFromError(error) — update state to show fallback UI.
    - componentDidCatch(error, info) — perform side effects like logging.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }
  static getDerivedStateFromError() {
    return { hasError: true };
  }
  render() {
    return this.state.hasError ? <h1>Error occurred</h1> : this.props.children;
  }
}
```

17. Portals

React Portals provide a way to render children into a DOM node that exists **outside the DOM hierarchy** of the parent component.

Why Use Portals?

- To render modals, tooltips, dropdowns, or overlays that visually break out of the parent container.
- To avoid CSS issues like overflow: hidden or stacking context problems.
- To keep your React component hierarchy but place UI elements elsewhere in the DOM.

How Portals Work

Normally, React renders components inside their parent DOM node. Portals allow you to render children into a different DOM node.

ReactDOM.createPortal(<Modal />, document.getElementById('modal-root'));

18. Controlled vs. Uncontrolled Components

Controlled Components

- React **controls** the input's value via component state.
- Input's value is always driven by React state (value prop).
- You update state with onChange handler.
- React is the "single source of truth" for the form data.

Advantages

- Easier to validate input, enforce formatting, or conditionally enable/disable buttons.
- Full control over the input state.
- Works well with React's declarative nature.

Example:

```
import React, { useState } from 'react';
function ControlledInput() {
  const [name, setName] = useState('');
  function handleChange(event) {
  setName(event.target.value);
  }
  function handleSubmit(event) {
    event.preventDefault();
    alert('Submitted name: ' + name);
  }
  return (
    <form onSubmit={handleSubmit}>
      <input type="text" value={name} onChange={handleChange} />
      <button type="submit">Submit</button>
```

```
      </form>
    );
  }
```

Uncontrolled Components

- Input maintains its own internal state in the DOM.
- React does **not** control the input value.
- You access the input value using **Refs** (React.createRef or useRef).
- Closer to traditional HTML form behavior.

Advantages

- Easier to integrate with non-React code or libraries.
- Less code for simple use cases.
- Good when you don't need to respond to every keystroke.

Example:

```
import React, { useRef } from 'react';
function UncontrolledInput() {
  const inputRef = useRef(null);
  function handleSubmit(event) {
  event.preventDefault();
    alert('Submitted name: ' + inputRef.current.value);
  }
  return (
    <form onSubmit={handleSubmit}>
      <input type="text" ref={inputRef} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

19. Higher-Order Components (HOC)

A **Higher-Order Component** is a **function** that takes a component and returns a new enhanced component.

It's a pattern used for **reusing component logic**.

Why Use HOCs?

- Share common functionality (e.g., logging, theming, authentication) between components.
- Keep components clean and focused on UI.
- Avoid duplicating code.

Basic Syntax

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

20. Render Props

A **Render Prop** is a technique for sharing code between React components using a **prop whose value is a function**. This function returns React elements and allows components to share logic while keeping UI flexible.

Why Use Render Props?

- Share common behavior or stateful logic between components.
- Allow components to decide how the UI looks.
- More flexible than HOCs in some cases.

21. Reconciliation & Keys

Reconciliation is the process React uses to **update the DOM efficiently** when your app's state or props change.

How Reconciliation Works

- When React components update, React creates a new **virtual DOM tree**.
- React **compares** the new virtual DOM with the previous one (called **diffing**).
- React figures out the **minimal set of changes** needed to update the real DOM.
- It then updates only those parts of the DOM, making UI updates fast

Why is Reconciliation Important?

- Directly manipulating the DOM is slow.
- Updating the entire DOM tree on every change is inefficient.
- React's reconciliation optimizes performance by minimizing DOM operations.

Keys

Keys are special string or number attributes you add to elements in lists to help React identify which items have changed, are added, or removed.

Why Keys Are Important

- Keys help React **match elements** between renders.
- This helps React **preserve component state** and avoid unnecessary re-renders.
- Without keys, React uses the index by default, which can cause issues if list items reorder or change.

How Keys Work with Reconciliation

- When reconciling lists, React uses keys to match old and new elements.
- If keys are stable and unique, React can **efficiently update only the changed elements**.
- If keys are missing or not unique, React may unnecessarily re-create or reorder elements, hurting performance or causing bugs.

22. Server-Side Rendering (SSR)

Server-Side Rendering (SSR) means rendering your React components **on the server** into HTML before sending the page to the client's browser.

Why Use SSR?

- **Faster initial page load:** The browser receives fully rendered HTML, so users see content sooner.
- **Better SEO:** Search engines can crawl the fully rendered HTML easily.
- **Improved performance on slow devices:** Less work on client devices.
- Enables **social media previews** because metadata is in HTML.

How SSR Works in React

1. Server runs React code to render components to HTML using ReactDOMServer.renderToString() or renderToStaticMarkup().

2. The server sends the generated HTML to the client.
3. On the client side, React **hydrates** the HTML — attaches event listeners and makes it interactive without re-rendering everything.
4. After hydration, React behaves like a normal SPA (Single Page Application).

Example:

```
import express from 'express';
import React from 'react';
import ReactDOMServer from 'react-dom/server';
import App from './App';
const app = express();
app.get('*', (req, res) => {
  const appHtml = ReactDOMServer.renderToString(<App />);
  const html = `
    <!DOCTYPE html>
    <html>
      <head><title>My SSR React App</title></head>
      <body>
        <div id="root">${appHtml}</div>
        <script src="/bundle.js"></script> <!-- Your bundled React app -->
      </body>
    </html>
  `;
  res.send(html);
});
app.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

23. State Management Beyond React

While React's useState and useReducer work great for local component state, many apps need to manage **global or complex state** that spans many components.

Popular State Management Solutions Beyond React

1. Redux

- One of the most popular state management libraries.
- Central **store** holds the app state.
- State is immutable and updated via **actions** and **reducers**.
- Works well with middleware like redux-thunk or redux-saga for async logic.
- Has DevTools for debugging.

Usage: Great for large, complex apps needing strict state management.

2. MobX

- Reactive state management library.
- Uses **observable state** and **computed values**.
- Less boilerplate than Redux, more flexible.
- Automatically tracks dependencies and updates components.

Usage: Suitable for apps favoring simplicity and automatic updates.

3. Recoil

- State management library developed by Facebook.
- Works seamlessly with React hooks.
- Supports atoms (units of state) and selectors (derived state).
- Enables fine-grained updates.

Usage: Good for React apps wanting modern, hook-friendly global state.

4. Zustand

- Minimalistic and fast state management.
- Uses hooks API with simple and flexible API.
- No boilerplate, easy to learn.

5. Context API with useReducer

- React’s built-in solution for moderate global state.
- Combine useReducer with Context for centralized logic.
- Avoid overusing, as it can cause performance issues with large state.

6. Apollo Client (for GraphQL)

- Manages both remote and local state.
- Handles caching, queries, mutations.
- Integrates state management with server data fetching.

24. Styling in React

React doesn’t impose a specific way to style components, so you can choose from multiple techniques depending on your needs and project size.

Common Styling Approaches

1. CSS Stylesheets (Global CSS)

- Traditional .css files linked globally.
- Styles apply globally unless scoped.
- Simple to use but can cause naming conflicts.

2. CSS Modules

- CSS files scoped locally by default.
- Avoids global namespace collisions.
- Classes imported as objects.

3. Inline Styles

- Style as an object directly on elements via the style prop.
- Useful for dynamic styles.
- No support for pseudo-classes or media queries.

4. Styled Components (CSS-in-JS)

- Use libraries like styled-components.
- Write actual CSS syntax inside JavaScript.
- Supports theming, dynamic styling, scoped styles.

- Requires installing a library.

5. Other CSS-in-JS Libraries

- Emotion
- JSS
- Stitches
- Linaria

They offer similar benefits with slightly different APIs.

6. Utility-First CSS Frameworks

- Tailwind CSS is popular for utility classes.
- Use predefined classes in JSX.

25. Testing in React

Testing ensures your React components work correctly, remain bug-free, and maintain quality as your app grows.

Common Types of Testing in React

1. **Unit Testing**
   Tests individual components or functions in isolation.
2. **Integration Testing**
   Tests how multiple components work together.
3. **End-to-End (E2E) Testing**
   Tests the entire app flow in a real browser environment (user perspective).

Popular Testing Tools for React

- **Jest**
  - Facebook's testing framework, included by default with Create React App.
  - Supports mocking, assertions, snapshot testing, and code coverage.
- **React Testing Library (RTL)**
  - Encourages testing components the way users interact with them.
  - Focuses on accessibility and best practices.
- **Enzyme** (less popular now)
  - Provided utilities for shallow, mount, and static rendering.

    - React Testing Library is preferred now.
- **Cypress / Playwright / Puppeteer**
    - Tools for E2E testing with browser automation.

Example:

```
import React from 'react';

function Button({ onClick, children }) {

  return <button onClick={onClick}>{children}</button>;

}

export default Button;
```

Summary:

React.js is an **open-source JavaScript library** developed by Facebook for building **user interfaces (UIs)**, especially for **single-page applications (SPAs)**.
It allows developers to create **reusable UI components** and efficiently update the DOM when data changes.

- **Component-Based** – Breaks UI into small, reusable pieces.
- **Virtual DOM** – Updates only what's necessary, improving performance.
- **Declarative Syntax** – You describe what the UI should look like, and React handles updates.
- **JSX (JavaScript XML)** – Lets you write HTML-like code inside JavaScript.
- **Unidirectional Data Flow** – Data flows from parent to child components, making debugging easier.
- **Cross-Platform** – Can be used with **React Native** for mobile apps.