React PART-3

React: *Introduction, Components - React Classes, Composing Components, passing data using Properties & Children, Dynamic Composition, React State - Initial State, Async State Initialization, Updating State, Event Handling, Stateless Components, Designing Components.*

Introduction to React

React is an open-source **JavaScript Library** used for building **User Interfaces (UIs)**, especially for **Single-Page Applications (SPAs)** where you need a fast and interactive user experience.

- Developed and maintained by **Meta** **(**formerly **Facebook)**.
- It allows developers to build **Reusable UI Components**.
- React handles the **View Layer** in an application (the "V" in MVC architecture).

History of React

Year	Milestone
2011	React was developed internally by Facebook.
2013	Open-sourced at JSConf US.
2015	React Native released for mobile development.
2016+	Wide adoption; major versions introduced features like Hooks, Concurrent Mode, etc.

Key Features

- **Component-Based Architecture**
  - UI is divided into small components.
  - Each component manages its own state and can be reused.

❖ Virtual DOM

- React doesn’t directly manipulate the real DOM.
- Instead, it uses a **Virtual DOM** (a lightweight copy of the real DOM).
- When something changes, React compares the old and new virtual DOM (**diffing**) and updates only what is necessary (**reconciliation**).

❖ JSX (JavaScript XML)

- React uses JSX syntax: a mix of HTML + JavaScript.
- Makes UI code easy to write and understand.

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

❖ Unidirectional Data Flow

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

❖ Declarative

- You describe **what** the UI should look like, not **how** to update it.
- React handles UI updates automatically when state changes.

❖ Strong Ecosystem

- Works with **React Router** for navigation, **Redux/Context API** for state management, and **Next.js** for server-side rendering.

React Architecture

- **Components**: Building blocks of React UI.
- **Props**: Input data passed to components.
- **State**: Internal data storage that changes over time.
- **Hooks**: Special functions (useState, useEffect, etc.) that let you use state and lifecycle features in functional components.
- **Virtual DOM**: Efficiently updates UI.

Types of Components

1. **Functional Components** (recommended)

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

2. **Class Components** (older way)

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

React Lifecycle

- **Mounting** → Component is created (constructor, render, componentDidMount)
- **Updating** → When state or props change (shouldComponentUpdate, componentDidUpdate)
- **Unmounting** → Component is removed (componentWillUnmount)

Hello World Example

```
import React from 'react';
import ReactDOM from 'react-dom';

function App()
{
  return <h1>Hello, React World!</h1>;
}

ReactDOM.render(<App />, document.getElementById('root'));
```

Advantages of React

- Fast and efficient (Virtual DOM).
- Component reusability.
- Large community and ecosystem.
- SEO-friendly with server-side rendering (Next.js).
- Easy integration with other libraries.

Disadvantages

- Learning curve for beginners (JSX, props, state, hooks).
- Needs additional libraries (routing, state management).
- Frequent updates may cause compatibility issues.

Real-World Applications of React

- Facebook, Instagram, WhatsApp Web
- Netflix, Uber, Airbnb
- Amazon, Flipkart (some parts)

In short:
React = **Fast, Efficient, Component-Based UI Library** that powers modern web apps

React Ecosystem

React by itself is focused only on the UI, but it's commonly used with:

Tool	Purpose
React Router	Handling routing/navigation
Redux, Zustand, Recoil	State management
Next.js	Framework for SSR, SSG, and hybrid apps
Axios / Fetch API	Making HTTP requests
Tailwind CSS / Styled Components	Styling

Use Cases

React is ideal for:

- **Single-page applications (SPAs)**
- **Dashboards**
- **E-commerce platforms**
- **Interactive forms**
- **Mobile apps** (via React Native)

Key Benefits

✅ Reusable components
✅ Huge community and ecosystem
✅ Rich developer tools (React DevTools)
✅ Easy testing and debugging
✅ Seamless integration with other libraries

Challenges

❌ Learning curve (especially for beginners with JSX, Hooks, state management)
❌ Needs external libraries for routing, state, etc.
❌ Frequent updates may require rewrites

Example: **A Simple React App**

```
import React from "react";
function App() {
  const name = "Kishore"; // you can change this to any name
  return (
    <div style={{ textAlign: "center", marginTop: "50px" }}>
      <h1>Welcome to React □</h1>
      <h2>Hello, {name}! □</h2>
      <p>This is your first React app.</p>
    </div>
  );
}
export default App;
```

Create and Run the React App in NodeJS

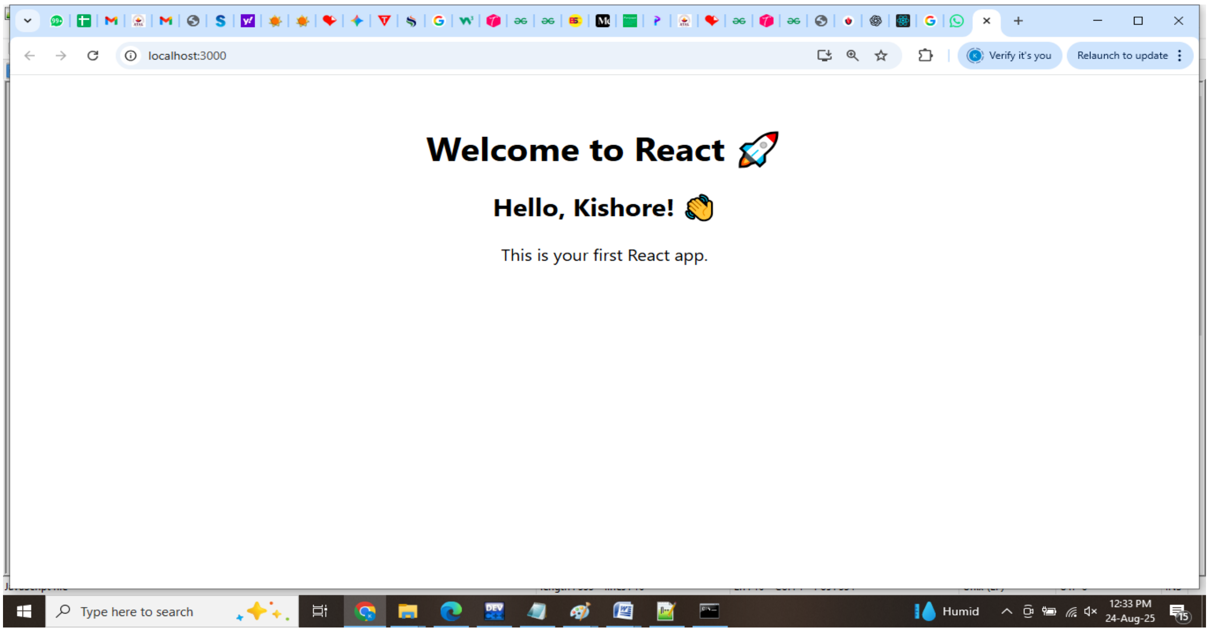
Step 1: Create React Project

```
npx create-react-app welcome-app
```

Step 2: cd welcome-app

Step 3: npm start

Output:



Welcome to React 🚀

Hello, Kishore! 👋

This is your first React app.

React Architecture Overview

React follows a **component-based architecture** with a **declarative UI** approach. It can be divided into three main parts:

1. **Components (UI Layer)**
   - Building blocks of the application (like Lego pieces).
   - Each component manages its own **state** and receives **props** from its parent.
   - Components are reusable and form a **tree structure** (component hierarchy).

```
App
├── Header
├── Sidebar
└── Content
    ├── Post
    └── Comment
```

2. **Virtual DOM & Reconciliation (Rendering Layer)**
   - React doesn't update the **real DOM** directly (which is slow).
   - Instead, it uses a **Virtual DOM** (a lightweight JS object representing the DOM).
   - When state/props change:
     1. React creates a new Virtual DOM.
     2. It compares it with the previous one (**Diffing algorithm**).
     3. Only the changed nodes are updated in the actual DOM (**Reconciliation**).

❖ This makes React **fast and efficient**.

Data Flow in React (Unidirectional Flow)

React follows a **unidirectional data flow**:

- **Parent → Child** via **props**.
- Child can't directly change parent's state (only via callback functions).
- This makes the app predictable and easier to debug.

For global state or cross-component data, we use:

- **Context API**
- **State management libraries** (Redux, Zustand, Recoil, etc.)

Typical React Architecture in Applications

A real-world React app usually follows this layered structure:

1. Presentation Layer (UI Components)

- Dumb/stateless components (just display data).
- Example: Button, Card, Navbar.

2. Container Layer (Stateful Components / Hooks)

- Smart components that manage state, API calls, business logic.
- Example: TodoListContainer fetching todos from API.

3. State Management Layer

- Local state (useState, useReducer)
- Global state (Context API, Redux, Zustand)

4. Service Layer

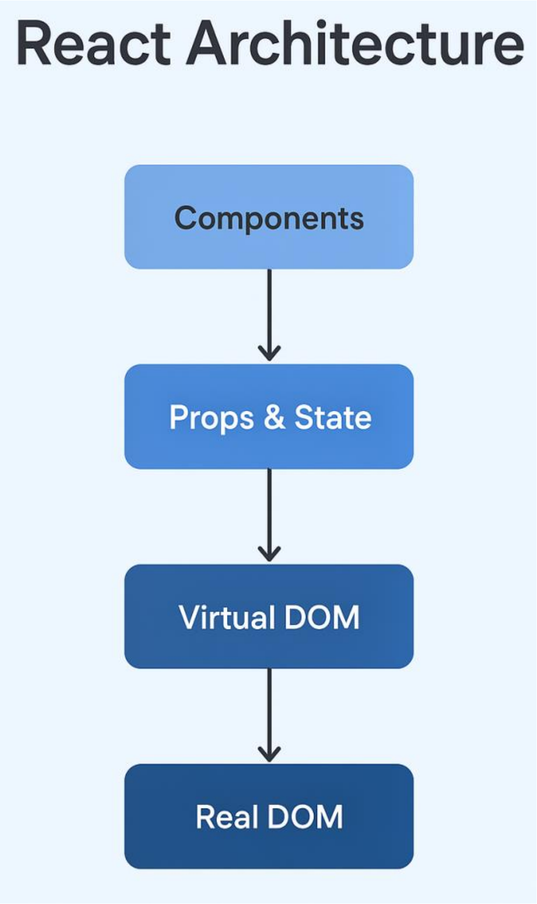
- Handles API requests, authentication, caching.
- Example: api.js with Axios calls.

5. Routing Layer

- Handled by **React Router** or frameworks like **Next.js**.
- Manages navigation between pages.

React Architecture Diagram

```
+--------------------------------------------------+
|                React Application                 |
+--------------------------------------------------+
|                Routing Layer                   |
|  (React Router / Next.js - Page Navigation)      |
+--------------------------------------------------+
|        State Management Layer (Redux, etc.)      |
+--------------------------------------------------+
|  Container Components (Logic, API calls,
hooks)  |
+--------------------------------------------------+
|     Presentational Components (UI & Layout)
|
+--------------------------------------------------+
|  Virtual DOM & React Fiber (Diffing +
Rendering)  |
+--------------------------------------------------+
|          Real DOM (Browser Rendering)            |
+--------------------------------------------------+
```



Key Architectural Features of React

- **Component-based UI** → Reusability.
- **Unidirectional Data Flow** → Predictability.
- **Virtual DOM + Fiber** → Performance optimization.
- **Declarative UI** → Easier to understand and maintain.
- **Pluggable ecosystem** → Routing, state, styling can be customized.

In short, React's architecture is:

Components (UI) → **Virtual DOM (diffing)** → **Fiber (reconciliation)** → **Real DOM (rendering)**

React Class Components

A **Class Component** is an ES6 JavaScript class that extends **React.**

- ❖ **Component**s comes with built-in features like **state** and **lifecycle methods**.

Defining a Class Component

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

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

- Must extend React.Component (or Component if imported).
- Must include a render() method, which returns JSX.
- Props are accessed with this.props.

Usage:

```
<Welcome name="Alice" />
```

Key Features of Class Components

1. Props

- Passed from parent to child.
- Accessed with this.props.

```
class Greeting extends Component {
  render() {
    return <h2>Hi, {this.props.user}</h2>;
  }
}
```

2. State

- Local data storage, mutable only inside the component.
- Declared as an object inside the constructor (older style) or directly as a class property.

Example:

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

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

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

- this.setState() is the only way to update state.
- State updates are asynchronous (batched).

3. Lifecycle Methods

Class components come with special methods that run at different phases:

- ***Mounting (when component is created)***
  - constructor() – initialize state/props.
  - componentDidMount() – called once after render, good for API calls.
- ***Updating (when state/props change)***
  - componentDidUpdate(prevProps, prevState) – runs after re-render.
- ***Unmounting (when component is removed)***
  - componentWillUnmount() – cleanup (timers, subscriptions, listeners).

Example:

```
class Timer extends Component {
  state = { seconds: 0 };

  componentDidMount() {
    this.interval = setInterval(() => {
      this.setState({ seconds: this.state.seconds + 1 });
    }, 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return <p>Timer: {this.state.seconds} sec</p>;
  }
}
```

Class Components vs Functional Components

Feature	Class Components	Functional Components (Hooks)
Syntax	ES6 Class	Function (JSX return)
State Management	this.state + setState()	useState, useReducer
Lifecycle Methods	componentDidMount, etc.	useEffect
Code Simplicity	More verbose	Cleaner & concise
React Recommendation	Legacy	Preferred (Hooks since React 16.8)

When to Use Class Components?

- Legacy projects that still rely on them.
- Understanding React’s evolution (important for interviews!).
- Most new apps prefer **functional components + hooks**.

Composing Components in React

Composition means **building complex UIs from smaller, reusable components**.

- ❖ Instead of writing one big component, we split the UI into **modular pieces** and combine them.
- ❖ Think of it like **Lego blocks**: small blocks → combined → big structure.

Basic Example of Composition

```
function Avatar(props) {
  return <img src={props.src} alt="avatar" width="50" />;
}
function UserInfo(props) {
  return (
    <div>
      <Avatar src={props.user.avatar} />
      <h3>{props.user.name}</h3>
    </div>
  );
}
```

```
function Comment(props) {
  return (
    <div>
      <UserInfo user={props.user} />
      <p>{props.text}</p>
    </div>
  );
}

export default function App() {
  return (
    <Comment
      user={{ name: "Alice", avatar: "/alice.png" }}
      text="This is a comment"
    />
  );
}
```

Here:

- Avatar → reused inside UserInfo
- UserInfo → reused inside Comment
- App → combines everything

Ways of Composing Components

1. Containment (Children Prop)

Some components don't know their children ahead of time.

We use **props.children**.

```
function Card(props) {
  return <div className="card">{props.children}</div>;
}
function App() {
  return (
    <Card>
      <h2>Title</h2>
      <p>This is inside the card.</p>
    </Card>
  );
}
```

❖ Anything between <Card> ... </Card> becomes props.children.

2. Specialization (Passing Props)

Components can be customized by passing props.

```
function Button({ label, type }) {
  return <button className={`btn ${type}`}>{label}</button>;
}

function App() {
  return (
    <div>
      <Button label="Save" type="primary" />
      <Button label="Cancel" type="secondary" />
    </div>
  );
}
```

❖ Here both buttons **share the same base component**, just with different props.

3. Component Composition (Nesting)

Nest components to build larger structures.

```
function Header() {
  return <header>My Website</header>;
}
function Footer() {
  return <footer>© 2025</footer>;
}
function Layout({ children }) {
  return (
    <>
      <Header />
      <main>{children}</main>
      <Footer />
    </>
  );
}
```

```
function App() {
  return (
    <Layout>
      <h1>Welcome Home!</h1>
      <p>This is the homepage content.</p>
    </Layout>
  );
}
```

Why Composition is Better than Inheritance

React favors **composition over inheritance** because:

- More **flexible and reusable**.
- Components can be nested and specialized easily.
- Avoids rigid inheritance hierarchies.

Summary

- **Composition** = combining small components to build bigger ones.
- Methods:
  - **Containment** (props.children)
  - **Specialization** (custom props)
  - **Nesting** (layouts, pages, etc.)
- Encourages **reusability, modularity, and clarity**.

Passing Data Using Properties & Children

1. Passing Data with Props (Properties)

Props are like **function arguments** — they allow parent components to send data to child components.

Example:

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

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

- App passes name="Alice" as a prop.
- Welcome receives it as props.name.
- Props are **read-only** → child cannot modify them.

❖ Use **props** when data flows **parent** → **child**.

2. Passing Data with Children (Containment)

Sometimes we don’t know ahead of time what content a component should display.
That’s where **props.children** comes in.

Example: Card Component

```
function Card(props) {
  return <div className="card">{props.children}</div>;
}
function App() {
  return (
    <Card>
      <h2>Title</h2>
      <p>This content is passed as children.</p>
    </Card>
  );
}
```

- Whatever is placed **between <Card> ... </Card>** is automatically passed as props.children.
- This makes Card reusable for any type of content.

❖ Use **children** when you want **flexible nested content**.

Combining Props & Children

You can use both **props** and **children** together for maximum flexibility.

```
function AlertBox({ type, children }) {
  return <div className={`alert ${type}`}>{children}</div>;
}

function App() {
  return (
    <div>
      <AlertBox type="success">Data saved successfully!</AlertBox>
      <AlertBox type="error">Something went wrong.</AlertBox>
    </div>
  );
}
```

- type → configures styling (via props).
- children → provides flexible content inside the alert.

Summary

- **Props** → Used for passing data/config **from parent to child**.
- **Children** → Used for **nesting dynamic content** inside a component.
- Together, they make React components **reusable and composable**.

Dynamic Composition

Dynamic Composition means **assembling components at runtime based on data, conditions, or user interaction**, instead of hardcoding component structures.

- It’s about making UIs **data-driven and flexible**.

1. Rendering Components Dynamically

We can render different components based on **props, state, or logic**.

```
function AdminPanel() {
  return <h2>Welcome, Admin</h2>;
}

function UserPanel() {
  return <h2>Welcome, User</h2>;
}

function Dashboard({ role }) {
  return role === "admin" ? <AdminPanel /> : <UserPanel />;
}

export default function App() {
  return <Dashboard role="admin" />;
}
```

- Composition happens dynamically depending on role.

2. Composing via Lists of Data

Dynamic composition is common when **mapping over arrays** to build UI.

```
function TodoItem({ task }) {
  return <li>{task}</li>;
}

function TodoList({ todos }) {
  return (
    <ul>
      {todos.map((t, i) => (
        <TodoItem key={i} task={t} />
      ))}
    </ul>
  );
}
export default function App() {
  const tasks = ["Learn React", "Build App", "Deploy"];
  return <TodoList todos={tasks} />;
}
```

- ❖ Here, the **data drives the component composition**.

3. Composition with Children (Flexible Layouts)

We can build **container components** that dynamically wrap children.

```
function Layout({ header, footer, children }) {
  return (
    <div>
      <header>{header}</header>
      <main>{children}</main>
      <footer>{footer}</footer>
    </div>
  );
}
export default function App() {
  return (
    <Layout
      header={<h1>My Site</h1>}
      footer={<small>© 2025</small>}
    >
      <p>This is dynamic page content</p>
    </Layout>
  );
}
```

- ❖ Components (header, footer, children) are **plugged in dynamically**.

4. Higher-Order & Dynamic Wrappers

We can wrap components dynamically to enhance behavior.

```
function withLogger(Component) {
  return function Wrapped(props) {
    console.log("Rendering:", Component.name);
    return <Component {...props} />;
  };
}

function Button(props) {
  return <button>{props.label}</button>;
}

const LoggedButton = withLogger(Button);

export default function App() {
  return <LoggedButton label="Click Me" />;
}
```

❖ The **composition is dynamic** — any component can be wrapped with new behavior.

Benefits of Dynamic Composition

✓ Reusable & flexible UIs
✓ Data-driven rendering
✓ Cleaner separation of logic & presentation
✓ Easier to scale large apps

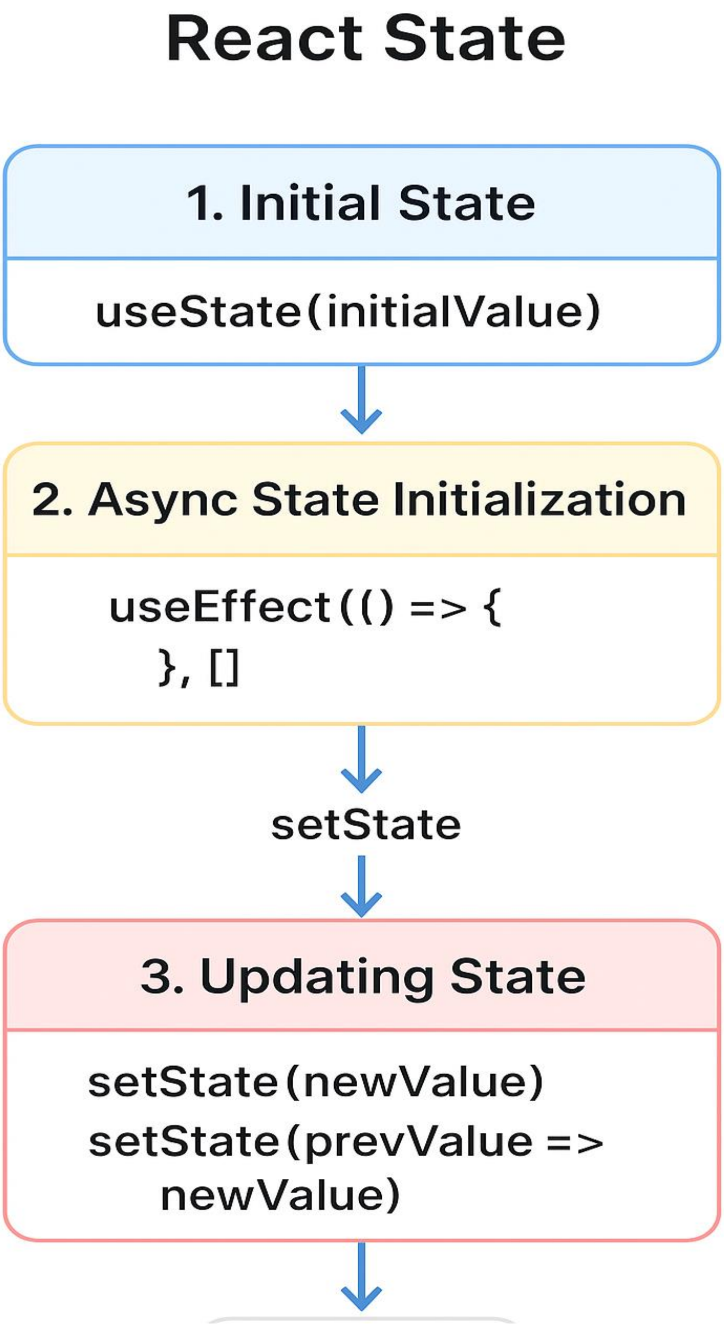
Summary

- **Static composition** → Predefined (fixed nesting of components).
- **Dynamic composition** → Components **assembled based on props, data, or conditions**.
- Achieved via:
    - Conditional rendering
    - Mapping data into components
    - Flexible container components (props.children)
    - Higher-Order Components (HOCs) or hooks

React State

What is State in React?

- **State** = local, mutable data managed inside a component.
- Unlike **props** (external & read-only), **state** is internal and can change over time.
- When **state changes**, React automatically **re-renders** the component.



1. Initial State

There are multiple ways to set initial state depending on whether you use **functional** or **class components**.

(a) Functional Components → useState

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

function Counter() {
  const [count, setCount] = useState(0); // initial state = 0
  return <p>Count: {count}</p>;
}
```

- useState(initialValue) defines the initial value.

(b) Lazy Initialization (for expensive operations)

Instead of passing a value directly, you can pass a function → React will call it only **once** (on mount).

```
const [user, setUser] = useState(() => {
  console.log("Initial calculation runs only once");
  return { name: "Alice", age: 25 };
});
```

- This is useful for **expensive computations** or fetching from localStorage.

(c) Class Components → this.state

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 }; // initial state
  }
  render() {
    return <p>Count: {this.state.count}</p>;
  }
}
```

2. Async State Initialization

Important: useState itself does **not support async initialization** (since it expects a synchronous initial value).

But there are patterns to handle async cases:

(a) Initialize state as null / loading, then update with useEffect

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

function UserProfile() {
  const [user, setUser] = useState(null); // initial empty state

  useEffect(() => {
    async function fetchData() {
      const res = await fetch("/api/user");
      const data = await res.json();
      setUser(data); // async update
    }
    fetchData();
  }, []);

  if (!user) return <p>Loading...</p>;
  return <h2>Hello, {user.name}</h2>;
}
```

- ❖ Best practice for fetching async data → initialize with null/default value, then update.

(b) Lazy Initializer + Sync Storage

If your initial value comes from localStorage or a cache, use **lazy init** so it runs only once:

```
const [theme, setTheme] = useState(() => localStorage.getItem("theme") || "light");
```

3. Updating State

(a) Functional Components → setState from useState

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

// Simple update
setCount(count + 1);

// Functional update (recommended when new state depends on old state)
setCount(prev => prev + 1);
```

- ❖ React **batches state updates** for performance.
- ❖ Always use the **functional form** if the next state depends on the previous one.

(b) Class Components → this.setState()

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

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

- ❖ Directly modifying state (e.g., this.state.count++) will **not trigger re-render**. Always use setState.

(c) Partial State Updates (Class Components only)

- In classes, setState() **merges** the new state with existing state.
- In hooks, setState **replaces** the state value (so for objects, spread operator is needed).

```
// ❌ Wrong: overwrites entire object
setUser({ name: "Alice" });

// ✅ Correct: preserve previous fields
setUser(prev => ({ ...prev, name: "Alice" }));
```

Key Points about State Updates

1. **Asynchronous Updates** → React may batch updates for performance.
2. setCount(count + 1);
3. setCount(count + 1);
4. // This won't give +2 immediately due to batching

   ✅Use functional updates:

   setCount(prev => prev + 1);
   setCount(prev => prev + 1); // Correct → +2

5. **State is isolated per component** → updating one component's state does not affect others.
6. **Re-renders only happen when state changes** → if you set the same value, React won't re-render.

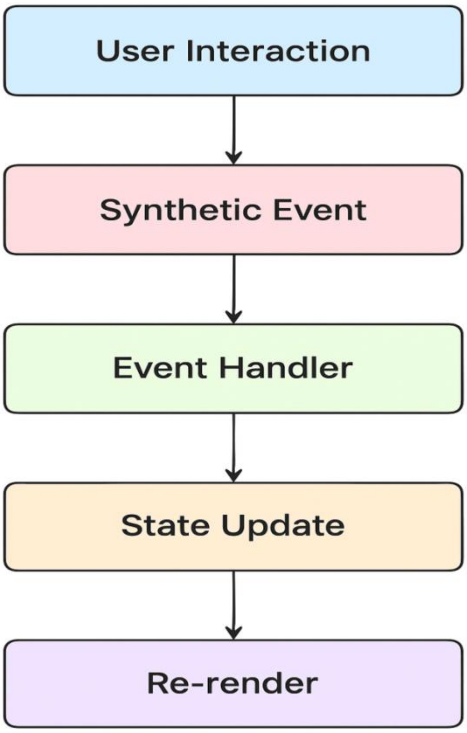
Summary

- **Initial State** → set via useState(initialValue) or this.state = {}.
- **Async Initialization** → not directly supported, but handled via useEffect or lazy initialization.
- **Updating State**:
  - Always use setState / setCount (never mutate directly).
  - Prefer **functional updates** if new state depends on old state.
  - In **class components**, updates merge; in **hooks**, updates replace (so use spread).

Event Handling in React

Event handling in React is the process of **capturing user interactions** (clicks, typing, submitting forms, etc.) and executing a function in response.

- ❖ React's event system is a **synthetic wrapper** around the browser's native events, making them work consistently across browsers.



1. Event Handling in Functional Components

(a) Basic Example

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

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

- ❖ In React, event names are written in **camelCase** (e.g., onClick, not onclick).
- ❖ The event handler is usually a function reference, **not a string** (like in plain HTML).

(b) Inline Event Handler

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

❖ Inline functions are fine for simple actions, but **avoid for performance-critical code** (because a new function is created on every render).

2. Event Handling in Class Components

```
class Button extends React.Component {
  handleClick() {
    alert("Class Button Clicked!");
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

❖ If you use this.handleClick directly, you might face **binding issues** with this.

✓ Solution → bind the method in constructor:

```
constructor(props) {
  super(props);
  this.handleClick = this.handleClick.bind(this);
}
```

✓ Or use class property syntax (arrow function auto-binds):

```
handleClick = () => {
  alert("Arrow function binding!");
};
```

3. Passing Arguments to Event Handlers

```
function Button({ message }) {
  function handleClick(msg) {
    alert(msg);
  }
  return <button onClick={() => handleClick(message)}>Click Me</button>;
}
```

❖ Use an arrow function inside onClick to pass parameters.

4. Synthetic Events in React

React uses a **SyntheticEvent object**, which is a cross-browser wrapper.

- It has the same interface as native events (event.target, event.preventDefault(), etc.), but it's pooled for performance.

Example:

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

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
}
```

5. Commonly Used React Events

- **Mouse Events** → onClick, onDoubleClick, onMouseEnter, onMouseLeave
- **Keyboard Events** → onKeyDown, onKeyPress, onKeyUp
- **Form Events** → onChange, onSubmit, onInput
- **Focus Events** → onFocus, onBlur
- **Clipboard Events** → onCopy, onPaste, onCut

6. Event Handling Best Practices

1. Use **function references**, not strings:

   ```
   <button onClick={handleClick}>Correct</button>
   ```

2. Prefer **functional components + hooks** over class components.
3. Use **event delegation cautiously**: React already optimizes events at the root using its synthetic event system.
4. When updating state in an event handler, use the **functional form** if state depends on the previous value:

```
setCount(prev => prev + 1);
```

Summary

- React events are **camelCase** and use **functions, not strings**.
- Functional components use useState and event handlers directly.
- Class components need binding (this.handleClick = this.handleClick.bind(this)) unless using arrow functions.
- Synthetic events provide a **cross-browser consistent interface**.
- Always use event.preventDefault() instead of returning false (like in plain HTML).

Stateless Components

Stateless Components are components that do **not manage their own state** internally.

- ❖ They receive **data via props** from their parent component and render UI accordingly.
- ❖ Typically written as **functional components** (using functions instead of classes).

1. Characteristics of Stateless Components

1. **No internal state (useState / this.state)**
2. **Rely on props** for data
3. **Easier to test** since they are pure functions
4. **Faster rendering** (no state updates → fewer re-renders)
5. Ideal for **UI rendering, presentational logic**

2. Example of Stateless Component

(a) Functional Stateless Component

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

// Usage
<Greeting name="Alice" />
```

- ❖ Here, Greeting has no state. It simply uses props to display data.

(b) With Arrow Function

```
const Greeting = ({ name }) => <h1>Hello, {name}!</h1>;
```

3. When to Use Stateless Components?

- For **presentational UI** (buttons, labels, headers, cards).
- When component **does not need to track user input** or local changes.
- To **separate concerns** (container components handle logic, stateless components handle UI).

4. Stateless vs Stateful Components

Feature	Stateless Component	Stateful Component
State Management	✗ No internal state	✓ Has useState / this.state
Data Source	Props (external)	State + Props
Purpose	UI rendering / presentation	Logic, state management, UI
Complexity	Simple, easy to test	More complex
Performance	Faster	Can be slower (due to updates)

5. Best Practices for Stateless Components

1. Keep them **pure** (same input → same output).
2. Avoid unnecessary logic inside them.
3. Use destructuring for cleaner props handling:

```
const UserCard = ({ name, age }) => (
  <div>
    <h2>{name}</h2>
    <p>Age: {age}</p>
  </div>
);
```

4. Use them for **reusability** (headers, footers, buttons).

Summary

- **Stateless components** = no local state, just props.
- Best for **presentational UI**.
- Easier to test, faster, and more maintainable.
- Complements **stateful container components**, which manage logic & state.

Component Design

Component Design in React is about **structuring UI into reusable, modular pieces**.

❖ Good design ensures:

- **Reusability** – Write once, use in multiple places.
- **Maintainability** – Easier to modify and debug.
- **Readability** – Clear separation of concerns.
- **Scalability** – Supports large, complex apps.

---

1. Principles of Good Component Design

1. **Single Responsibility Principle (SRP)**
   - Each component should **do one thing well**.
   - Example: A Button component should only render a button, not handle authentication logic.
2. **Reusability**
   - Build components that can be reused with different data using **props** and **children**.
3. **Separation of Concerns**
   - Split **presentational (UI)** and **container (logic)** components.
   - Example:
     - UserList → handles data fetching (stateful).
     - UserCard → displays each user (stateless).
4. **Composition over Inheritance**
   - React encourages building complex UIs by **composing smaller components** instead of using class inheritance.

2. Types of Components

1. **Presentational (Stateless)**
   - Focused on UI (markup & styles).
   - Receive data via props.

```
const UserCard = ({ name, age }) => (
          <div>
            <h2>{name}</h2>
            <p>Age: {age}</p>
          </div>
        );
```

2. **Container (Stateful)**
   - Handle logic (state, API calls).
   - Pass data to presentational components.

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

  useEffect(() => {
    fetch("/api/users")
      .then(res => res.json())
      .then(data => setUsers(data));
  }, []);

return (
       <div>
         {users.map(user => (
           <UserCard key={user.id} name={user.name} age={user.age} />
         ))}
       </div>
     );
   }
```

3. Component Composition Patterns

Children as content

```
function Card({ children }) {
  return <div className="card">{children}</div>;
}

<Card>
  <h2>Title</h2>
  <p>Some content inside</p>
</Card>
```

- **Props for customization**

```
<Button variant="primary" label="Save" />
<Button variant="secondary" label="Cancel" />
```

- **Higher-Order Components (HOC) / Hooks**

  For reusing logic (e.g., authentication, fetching).
- **Render Props**

```
<DataProvider render={(data) => <Chart data={data} />} />
```

4. Designing Component Hierarchy

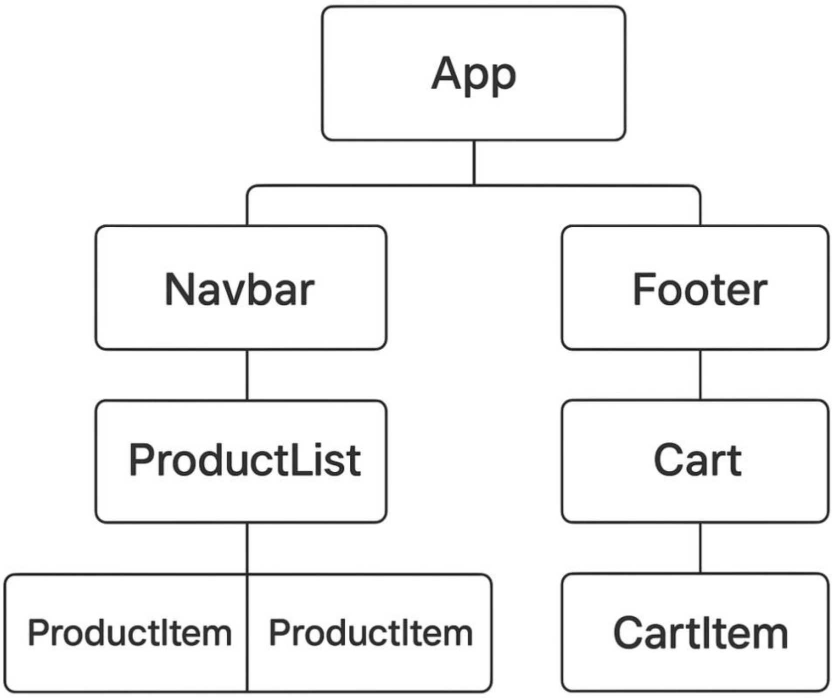
When designing a UI:

1. Break UI into a **component tree**.

   Example: Shopping Cart
2. App

```
├── Navbar
├── ProductList
│   ├── ProductItem
│   └── ProductItem
├── Cart
│   ├── CartItem
└── Footer
```

3. Identify which components need **state**.
   - If state is shared → lift state up to the nearest common ancestor.
4. Pass data **down via props**, and actions **up via callbacks**.



5. Best Practices

- Keep components **small & focused**.
- Prefer **functional components + hooks**.
- Use **prop-types** or **TypeScript** for type checking.
- Co-locate files (e.g., each component has its own folder with .js, .css).
- Avoid **deep prop drilling** → use Context API or state managers (Redux, Zustand).
- Use **composition** instead of making "God Components".

Summary

- Break UI into **small, reusable components**.
- Use **stateless (UI)** + **stateful (logic)** separation.
- Favor **composition** over inheritance.
- Lift state up when multiple children need it.
- Apply **best practices** for scalability and maintainability.

Shopping Cart in React

A **Shopping Cart** is a classic use case to demonstrate **state management, props passing, and component composition**.

1. Component Hierarchy

```
App
├── Navbar
├── ProductList
│   ├── ProductItem
├── Cart
│   ├── CartItem
└── Footer
```

- **App** → Parent container, holds state (products, cart items).
- **Navbar** → Displays navigation + cart count.
- **ProductList** → Displays all available products.
- **ProductItem** → Individual product with "Add to Cart" button.
- **Cart** → Shows added items.
- **CartItem** → Represents one item in the cart.
- **Footer** → Static footer.

2. State Management

- **App** (root) holds **state**:
  - products → available products.
  - cart → list of items in the cart.
- **State is passed down** as **props** to children.
- **Callbacks (functions)** are passed to children so they can **update state** in the parent.

Create and Run the React App in NodeJS

Step 1: Create React Project

```
npx create-react-app shopping-cart-app
```

Step 2: cd shopping-cart-app

Step 3: npm start

3. Code

App.js

```
import React, { useState } from "react";
import Navbar from "./Navbar";
import ProductList from "./ProductList";
import Cart from "./Cart";
import Footer from "./Footer";

function App() {
  const [products] = useState([
    { id: 1, name: "Laptop", price: 1000 },
    { id: 2, name: "Phone", price: 500 },
    { id: 3, name: "Headphones", price: 100 },
  ]);

  const [cart, setCart] = useState([]);
  const addToCart = (product) => {
    setCart([...cart, product]);
  };

  const removeFromCart = (index) => {
    const newCart = [...cart];
    newCart.splice(index, 1);
    setCart(newCart);
  };
  return (
    <div>
      <Navbar cartCount={cart.length} />
      <ProductList products={products} addToCart={addToCart} />
      <Cart cart={cart} removeFromCart={removeFromCart} />
      <Footer />
    </div>
  );
}
export default App;
```

Navbar.js

```
function Navbar({ cartCount }) {
  return (
    <nav style={{ background: "#eee", padding: "10px" }}>

 <h1> 🛒 Online Shopping Cart App Using React</h1>

<div>Cart: {cartCount} items</div>
    </nav>
  );
}
export default Navbar;
```

ProductItem.js

```
function ProductItem({ product, addToCart }) {
  return (
    <div style={{ marginBottom: "10px" }}>
      <span>
        {product.name} - ${product.price}
      </span>
      <button onClick={() => addToCart(product)} style={{ marginLeft: "10px" }}>
        Add to Cart
      </button>
    </div>
  );
}
export default ProductItem;
```

ProductList.js

```
import ProductItem from "./ProductItem";
function ProductList({ products, addToCart }) {
  return (
    <div>
      <h2>Products</h2>
      {products.map((p) => (
        <ProductItem key={p.id} product={p} addToCart={addToCart} />
      ))}
    </div>
  );
}
export default ProductList;
```

CartItem.js

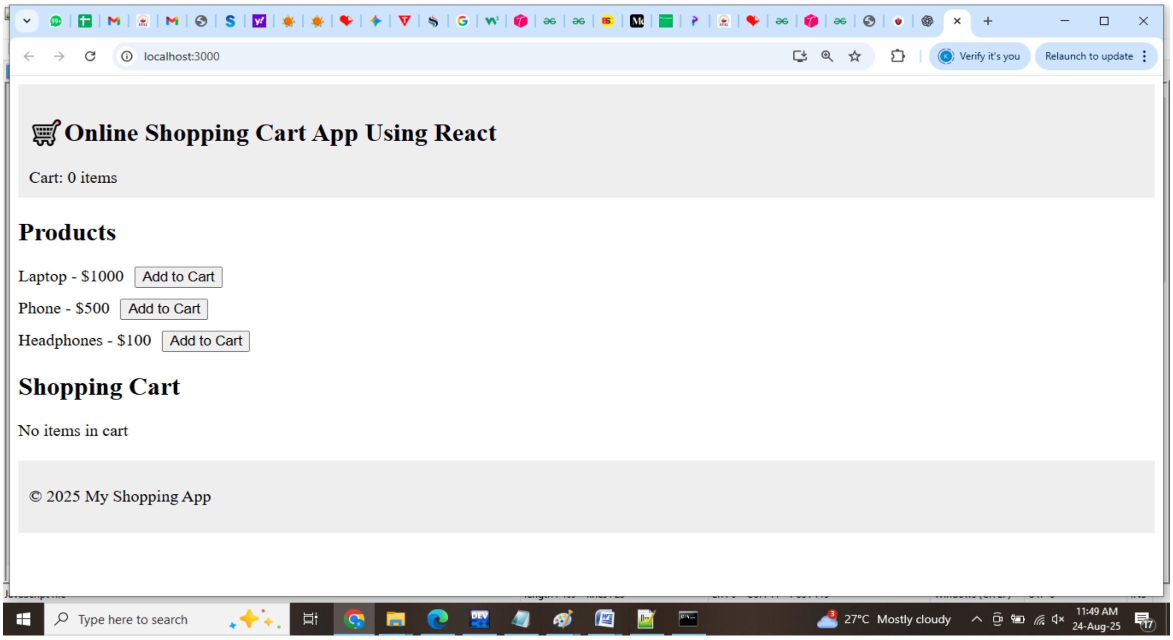
```
function CartItem({ item, index, removeFromCart }) {
  return (
    <div style={{ marginBottom: "10px" }}>
      {item.name} - ${item.price}
      <button
        onClick={() => removeFromCart(index)}
        style={{ marginLeft: "10px", color: "red" }}
      >
        Remove
      </button>
    </div>
  );
}
export default CartItem;
```

Cart.js

```
import CartItem from "./CartItem";

function Cart({ cart, removeFromCart }) {
  return (
    <div>
      <h2>Shopping Cart</h2>
      {cart.length === 0 ? (
        <p>No items in cart</p>
      ) : (
        cart.map((item, index) => (
          <CartItem
            key={index}
            item={item}
            index={index}
            removeFromCart={removeFromCart}
          />
        ))
      )}
    </div>
  );
}
export default Cart;
```

Output:



🛒 **Online Shopping Cart App Using React**

Cart: 0 items

Products

Laptop - $1000 Add to Cart

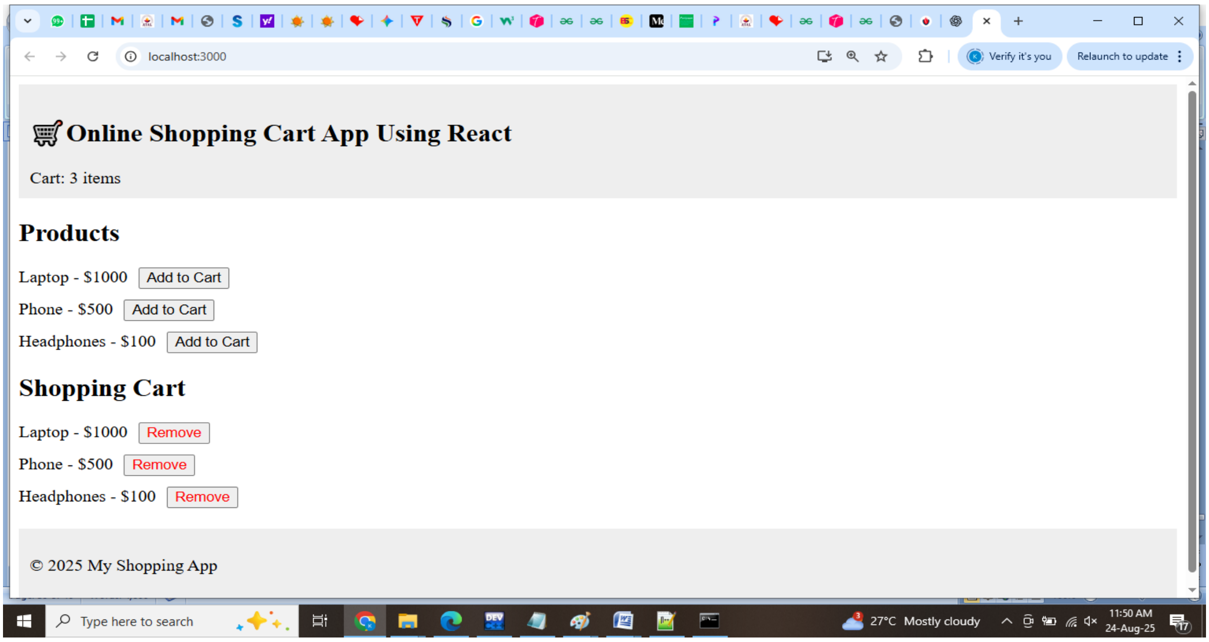
Phone - $500 Add to Cart

Headphones - $100 Add to Cart

Shopping Cart

No items in cart

© 2025 My Shopping App



🛒 **Online Shopping Cart App Using React**

Cart: 3 items

Products

Laptop - $1000 Add to Cart

Phone - $500 Add to Cart

Headphones - $100 Add to Cart

Shopping Cart

Laptop - $1000 Remove

Phone - $500 Remove

Headphones - $100 Remove

© 2025 My Shopping App

4. Key Concepts

✅ **Props** – Data flows from parent (App) → child (Navbar, ProductList, Cart).
✅ **State** – Cart items tracked in App.
✅ **Event Handling** – "Add to Cart" button updates state.
✅ **Composition** – Smaller components (ProductItem, CartItem) build the larger UI.
✅ **Unidirectional Data Flow** – Data flows **down**, actions (callbacks) flow **up**.