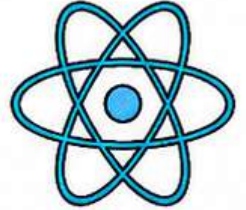


Chapter 4: React State & Events

4.1 State Introduction

- * **State** is a built-in React object that allows a component to create and manage its own data.
- * Unlike props, state is **private** and controlled within the component.
- * When the state changes, the component automatically **re-renders** to reflect the updated data.
- * State is **mutable**, meaning it can change over time based on user interaction or other actions.



Key Points :

- ✓ State is used to store data that changes over time.
- ✓ It is managed within the component.
- ✓ Updating state causes the component to re-render.
- ✓ Each component can have its own state.



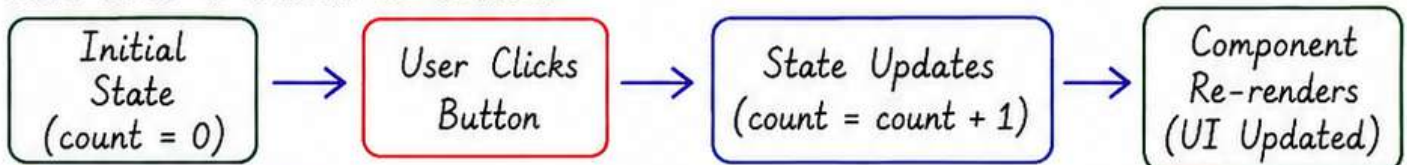
How State Works in React ?

- * React **re-renders** the component whenever the state changes.
- * The updated state values are shown in the UI.
- * State makes components **dynamic** and **interactive**.



Example

Consider a counter component that increases a number each time a button is clicked.



Key Takeaway :



State allows a component to **manage its data** and update the UI **dynamically** in response to changes.



4.2 useState()

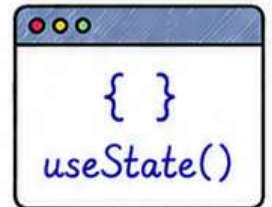
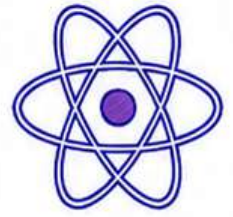
- * `useState()` is a Hook provided by React to add state to functional components.
- * It returns an array with **two** values.
- * The first value is the **current state**.
- * The second value is a function that allows us to update the state.

Syntax :

```
const [state, setState] = useState(initialValue);
```

Key Points :

- ✓ `useState()` can be used multiple times in a component.
- ✓ The state value can be of any data type.
- ✓ The initial value is used only during the first render.



4.3 Updating State

- * State can be **updated** using the function returned by `useState()`.
- * When the state is updated, React **re-renders** the component.
- * We should **never** update the state directly.
- * Always use the `setState` function to update the state.

Example :

```
import { useState } from "react";
function Counter() {
  const [count, setCount] = useState(0); // initial state
  const increase = () => {
    setCount(count + 1); // updating state
  };
  return (
    <button onClick={increase}>Clicked {count} times</button>
  );
}
```



Key Takeaway :



`useState()` lets us **add** state to components and **update** it using the provided function, triggering **UI updates**.



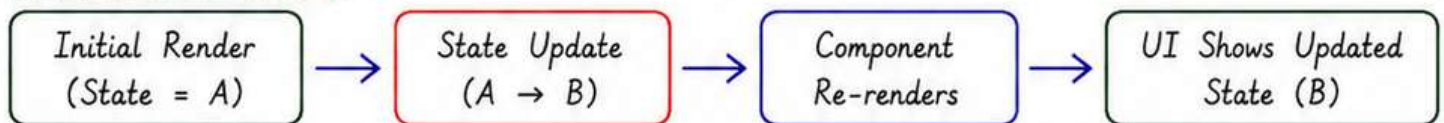
4.4 State and Re-rendering

- * When a state is updated, React automatically **re-renders** the component.
- * Re-rendering means the component function is called again and the **updated UI** is shown.
- * This ensures that the user always sees the **latest state**.
- * Re-rendering only happens when the state actually **changes**.

Key Points :

- ✓ Updating state causes the component to re-render.
- ✓ React compares the old and new state.
- ✓ If the state is same, no re-render happens.
- ✓ Re-rendering keeps the UI in sync with the state.

Re-rendering Flow :



4.5 Multiple States

- * A component can have **multiple state variables**.
- * Each piece of state is **independent** of the others.
- * This helps in managing different data separately.
- * Each state is updated using its **own setter function**.

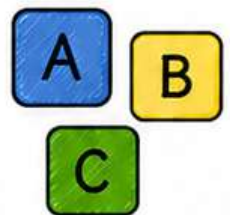
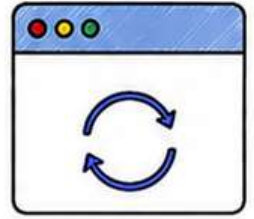
Example :

```
import { useState } from "react";  
function UserForm() {  
  const [name, setName] = useState(""); // first state  
  const [email, setEmail] = useState(""); // second state  
  const [age, setAge] = useState(0); // third state  
  ...  
}
```

Key Takeaway :



Multiple states allow us to manage **different data** in a component and update them **independently**.



4.6 Objects in State

- * We can store **objects** in state to manage related data.
- * When updating an object, we must create a **new object**.
- * React **does not merge** the old and new object automatically.
- * Always **spread the previous state** and update only the required fields.

Example :

```
import { useState } from "react";
function Profile() {
  const [user, setUser] = useState({ name: "Alice", age: 22 });
  const updateName = () => {
    setUser({ ...user, name: "Bob" }); // create new object
  };
  ...
}
```

Key Points :

- ✓ Objects help in storing related data together.
- ✓ Always create a new object while updating.
- ✓ Use the spread operator (...) to keep existing data.
- ✓ Update only the fields that are needed.

Key Takeaway :



Objects in state should be updated **immutably** by creating a new object and modifying only the required fields.



4.7 Arrays in State

- * Arrays can be stored in state to manage **lists of items**.
- * When updating an array, create a **new array**.
- * Use spread operator or array methods like **map**, **filter**, **slice**, etc.
- * **Never mutate** the existing array directly.

Example :

```
import { useState } from "react";
function List() {
  const [items, setItems] = useState(["Apple", "Banana"]);
  const addItem = () => {
    setItems([...items, "Orange"]); // create new array
  };
  const removeFirst = () => {
    setItems(items.slice(1)); // remove first item
  };
  ...
}
```

Key Points :

- ✓ Arrays help in storing collections of data.
- ✓ Always create a new array while updating.
- ✓ Use spread operator or array methods to modify arrays.
- ✓ Avoid direct mutation methods like **push**, **pop**, **shift**, **unshift**.

Key Takeaway :



Arrays in state should be updated **immutably** to ensure proper re-rendering and predictable behavior.



4.8 State Immutability

- * State in React should be treated as **immutable**.
- * We should **never modify** the state directly.
- * Always create a **new copy** of the state and update it.
- * Immutability helps React to **detect changes** and re-render the component.



Wrong vs Right Way :

❌ Wrong Way (Mutating State)	✅ Right Way (Immutable Update)
<pre>// Directly modifying state (Wrong) user.name = "Bob"; setUser(user);</pre>	<pre>// Creating new object (Correct) setUser({ ...user, name: "Bob" });</pre>



Key Points :

- ✓ State should be updated immutably.
- ✓ Never change the existing state value directly.
- ✓ Use spread operator (...), map(), filter(), etc., to create new values.
- ✓ Immutability makes the application more predictable and bug-free.



Key Takeaway :



Always treat state as **read-only** and update it **immutably** to ensure correct re-rendering and better performance.



4.9 Event Handling

- * Events are **user actions** like click, change, submit, etc.
- * React provides a simple way to handle events using **event handlers**.
- * We pass a function to the event and it will be executed when the event occurs.
- * Event names are written in **camelCase** in React (onClick, onChange, etc.).



Common Events :

Event Name	Description
onClick	Triggers when an element is clicked
onChange	Triggers when the value of an input changes
onSubmit	Triggers when a form is submitted
onMouseOver	Triggers when the mouse moves over an element
onKeyDown	Triggers when a key is pressed

Key Points :

- ✓ Use arrow functions or named functions as handlers.
- ✓ Pass the function without calling it (use **onClick={handleClick}**).
- ✓ We can pass parameters to event handlers if needed.
- ✓ Events make React components interactive and dynamic.

Example :

```
function Button() {
  const handleClick = () => {
    alert("Button Clicked!");
  };

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



Key Takeaway :



Event handling allows us to respond to **user actions** and build **interactive web applications** using React.



4.10 Forms in React

- * Forms are used to **collect input** from the user.
- * React makes it **easy to handle form data**.
- * We can handle form submissions and input changes using **state**.
- * React supports various form elements like **input**, **textarea**, **select**, **checkbox**, **radio buttons**, etc.

Example:

```
function SimpleForm() {  
  return (  
    <form>  
      <input type="text" placeholder="Enter your name" />  
      <textarea placeholder="Enter your message"></textarea>  
      <button type="submit">Submit</button>  
    </form>  
  );  
}
```

Key Points :

- ✓ Forms help in collecting user data.
- ✓ React handles form elements using state and events.
- ✓ We can handle input changes and form submissions easily.

Key Takeaway :



Forms in React allow us to **collect**, **manage** and **submit** user data efficiently.



4.11 Controlled Components

- * In controlled components, form data is handled by **React state**.
- * The **input value** is controlled by the **state**.
- * Every **change** in the input **updates the state**.
- * Controlled components are the recommended approach in React.

Example:

```
import { useState } from "react";  
function ControlledInput() {  
  const [name, setName] = useState("");  
  const handleChange = (e) => {  
    setName(e.target.value);  
  };  
  return (  
    <div>  
      <input type="text" value={name} onChange={handleChange} />  
      <p>Your Name: {name}</p>  
    </div>  
  );  
}
```

Key Points :

- ✓ Controlled components store form data in **state**.
- ✓ The value of the input is set from state.
- ✓ **onChange** updates the state whenever the user types.
- ✓ React has full control over the form data.

Key Takeaway :



Controlled components keep form data in **React state**, making it easy to validate, submit and manage user inputs.



4.12 Uncontrolled Components

- * In uncontrolled components, form data is handled by the **DOM**.
- * We use **refs** to get the input values.
- * The input value is **not** controlled by React **state**.
- * This approach is less common but useful in some cases.

Example :

```
import { useRef } from "react";
function UncontrolledForm() {
  const nameRef = useRef();
  const handleSubmit = (e) => {
    e.preventDefault();
    alert("Name: " + nameRef.current.value);
  };
  return (
    <form onSubmit={handleSubmit}>
      <input type="text" ref={nameRef} placeholder="Enter your name" />
      <button type="submit">Submit</button>
    </form>
  );
}
```

A browser window titled "Uncontrolled Form" with a text input field containing the placeholder "Enter your name" and a "Submit" button.

Here, the input value is taken from the DOM using **ref**.



Key Points :

- ✓ Uncontrolled components use **refs** to access form values.
- ✓ The input value is managed by the **DOM**, not by React **state**.
- ✓ Use **defaultValue** instead of **value**.
- ✓ Useful for integrating with non-React code or libraries.

Key Takeaway :



Uncontrolled components are **simple** to use but give **less control** compared to controlled components.



4.13 Form Validation

- * Validation ensures the user enters **correct data**.
- * We can validate on **change**, on **blur** or on **submit**.
- * React makes it easy to show **validation messages**.
- * We store **validation state** and display **errors** conditionally.

Example :

```
import { useState } from "react";
function ValidateForm() {
  const [name, setName] = useState("");
  const [error, setError] = useState("");
  const handleSubmit = (e) => {
    e.preventDefault();
    if (name.trim() === "") {
      setError("Name is required");
    } else {
      setError("");
      alert("Form Submitted Successfully!");
    }
  };
  return (
    <form onSubmit={handleSubmit}>
      <input type="text" value={name} onChange={(e) => setName(e.target.value)} />
      {error} && <p style={{color: "red"}}>{error}</p>
      <button type="submit">Submit</button>
    </form>
  );
}
```

A browser window titled "Form Validation" with a text input field containing the placeholder "Enter your name". Below the input field, the text "Name is required" is displayed in red. A "Submit" button is at the bottom.

Validation Types :

- On Change
- On Blur
- On Submit



Key Points :

- ✓ Always validate user input to improve user experience.
- ✓ Store **errors** in **state** and show messages conditionally.
- ✓ Validate on change, blur or submit based on requirements.
- ✓ Prevent form submission if validation fails.

Key Takeaway :



Form validation helps in collecting **accurate data** and provides **better feedback** to users.

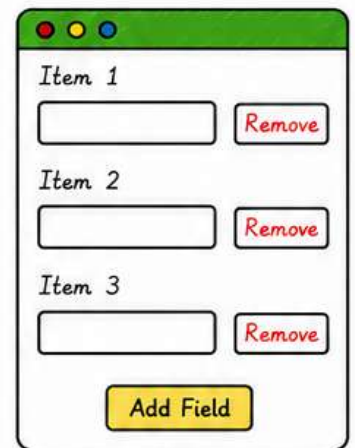


4.14 Dynamic Forms

- * Dynamic forms allow users to **add** or **remove** fields on the fly.
- * Useful for collecting lists of items like **multiple addresses**, **phones**, **skills**, etc.
- * We can manage dynamic fields using **arrays in state**.
- * React re-renders the form whenever the state changes.

Example :

```
import { useState } from "react";
function DynamicForm() {
  const [items, setItems] = useState([{ name: "" }]);
  const handleChange = (index, value) => {
    const newItems = [...items];
    newItems[index].name = value;
    setItems(newItems);
  };
  const addField = () => {
    setItems([...items, { name: "" }]); // add new field
  };
  const removeField = (index) => {
    const newItems = items.filter((_, i) => i !== index);
    setItems(newItems); // remove field
  };
  return (
    <div>
      {items.map((item, index) => (
        <div key={index} style={{ marginBottom: "10px" }}>
          <input type="text"
            value={item.name}
            onChange={(e) => handleChange(index, e.target.value)}
            placeholder={`Item ${index + 1}`} />
          {items.length > 1 && (
            <button type="button" onClick={() => removeField(index)}>Remove</button>
          )}
        </div>
      )}
      <button type="button" onClick={addField}>Add Field</button>
    </div>
  );
}
```



New fields can be added or removed dynamically.



Key Points :

- ✓ Dynamic forms are created based on user actions.
- ✓ Use **arrays in state** to manage multiple form fields.
- ✓ Add new fields using **spread operator**.
- ✓ Remove fields using **filter()**.
- ✓ React automatically updates the UI on state change.

Flow Diagram :



Key Takeaway :



Dynamic forms in React provide flexibility to collect multiple values. Managing them with **arrays in state** makes the form powerful and user-friendly.