**🔄 Building a React Counter Application**

**📋 1. Basic Implementation (Not Recommended)**

let count = 0; // Global variable approach

function Counter() {

function increment() {

count += 1;

renderApp();

}

function decrement() {

count -= 1;

renderApp();

}

return (

<div>

<h1>Counter: {count}</h1>

<button onClick={increment}>Increment</button>

<button onClick={decrement}>Decrement</button>

</div>

);

}

function renderApp() {

ReactDOM.createRoot(document.getElementById("root")).render(<Counter />);

}

**⚠️ Problems with this Approach:**

* **Global Variable Issues**
  + ❌ Lack of encapsulation
  + ❌ Shared state between components
  + ❌ Unpredictable behavior
* **Manual Rendering**
  + ❌ Inefficient full re-renders
  + ❌ Bypasses React's optimization
  + ❌ Additional complexity
* **Poor Debugging**
  + ❌ Can't track in React DevTools
  + ❌ Difficult to maintain
  + ❌ State resets on component remount

**🚀 2. Improved Implementation using useState**

function Counter() {

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

function increment() {

setCount(count + 1);

}

function decrement() {

setCount(count - 1);

}

return (

<div>

<h1>Counter: {count}</h1>

<button onClick={increment}>Increment</button>

<button onClick={decrement}>Decrement</button>

</div>

);

}

**💡 Understanding useState**

**Basic Syntax**

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

* state: Current value
* setState: Function to update state
* initialValue: Starting value

**State Updates**

setCount(count + 1); // Updates state and triggers re-render

**🔄 3. React's Reconciliation Process**

**🧩 How React Updates the DOM**

**Virtual DOM Creation**

* React maintains a virtual copy of the DOM
* Changes are first made to this virtual copy

**Diffing Process**

Previous Virtual DOM → Compare ← New Virtual DOM

↓

Identify Changes

↓

Update Real DOM

**Optimization**

* Only necessary changes are made
* Updates are batched together
* Minimal DOM manipulation

**📝 Example of Reconciliation**

// Initial Render

<div>Counter: 0</div>

// After increment

<div>Counter: 1</div>

// React only updates the text content

**⚖️ 4. Comparison: Global Variable vs useState**

| **Feature** | **Global Variable** | **useState** |
| --- | --- | --- |
| Encapsulation | ❌ No | ✅ Yes |
| Auto Re-render | ❌ Manual | ✅ Automatic |
| React DevTools | ❌ Not visible | ✅ Trackable |
| Performance | ❌ Poor | ✅ Optimized |
| Multiple Instances | ❌ Shared state | ✅ Independent state |

**🌟 5. Best Practices**

**🔍 State Updates**

// Good

function increment() {

setCount(count + 1);

}

// Better (for multiple updates)

function increment() {

setCount(prevCount => prevCount + 1);

}

**📦 Component Organization**

function Counter() {

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

// Group related functions

const handlers = {

increment: () => setCount(count + 1),

decrement: () => setCount(count - 1)

};

return (

<div>

<h1>Counter: {count}</h1>

<button onClick={handlers.increment}>+</button>

<button onClick={handlers.decrement}>-</button>

</div>

);

}

**🔢 Multiple State Variables**

function Counter() {

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

const [step, setStep] = useState(1);

return (

<div>

<h1>Counter: {count}</h1>

<input

type="number"

value={step}

onChange={e => setStep(Number(e.target.value))}

/>

<button onClick={() => setCount(count + step)}>

Add {step}

</button>

</div>

);

}

**💎 Key Takeaways**

* Always use React's state management (useState)
* Avoid global variables for component state
* Understand React's reconciliation process
* Let React handle DOM updates
* Keep components focused and organized
* Use the functional update form when new state depends on old state

🚀 **Remember**: React's state management is designed to make your applications more predictable and easier to debug!

**📊 Using useState with Different Data Types in React**

**🔤 1. String State**

**📝 Basic String State**

function StringExample() {

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

function updateName() {

setName("Jane");

}

return (

<div>

<h1>Name: {name}</h1>

<button onClick={updateName}>Change Name</button>

</div>

);

}

**⌨️ String State with Input**

function NameInput() {

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

return (

<div>

<input

value={name}

onChange={(e) => setName(e.target.value)}

placeholder="Enter name"

/>

<p>Hello, {name || "Guest"}!</p>

</div>

);

}

**📋 2. Array State**

**📊 Simple Array Management**

function ArrayExample() {

const [items, setItems] = React.useState(["Apple", "Banana"]);

function addItem() {

setItems([...items, "Orange"]); // Add new item

}

function removeItem(index) {

setItems(items.filter((\_, i) => i !== index)); // Remove item

}

return (

<div>

<h1>Items:</h1>

<ul>

{items.map((item, index) => (

<li key={index}>

{item}

<button onClick={() => removeItem(index)}>X</button>

</li>

))}

</ul>

<button onClick={addItem}>Add Orange</button>

</div>

);

}

**📝 Array of Objects**

function TodoList() {

const [todos, setTodos] = React.useState([

{ id: 1, text: "Learn React", done: false },

{ id: 2, text: "Build App", done: false }

]);

function toggleTodo(id) {

setTodos(todos.map(todo =>

todo.id === id

? {...todo, done: !todo.done}

: todo

));

}

return (

<ul>

{todos.map(todo => (

<li key={todo.id}

style={{textDecoration: todo.done ? 'line-through' : 'none'}}>

<input

type="checkbox"

checked={todo.done}

onChange={() => toggleTodo(todo.id)}

/>

{todo.text}

</li>

))}

</ul>

);

}

**📦 3. Object State**

**🧩 Basic Object State**

function ObjectExample() {

const [user, setUser] = React.useState({

name: "John",

age: 30

});

function updateAge() {

setUser({

...user, // Spread existing properties

age: user.age + 1 // Update specific property

});

}

return (

<div>

<h1>Name: {user.name}</h1>

<h2>Age: {user.age}</h2>

<button onClick={updateAge}>Increase Age</button>

</div>

);

}

**🔄 Nested Object State**

function ProfileManager() {

const [profile, setProfile] = React.useState({

personal: {

name: "John",

age: 30

},

settings: {

darkMode: false,

notifications: true

}

});

function toggleDarkMode() {

setProfile({

...profile,

settings: {

...profile.settings,

darkMode: !profile.settings.darkMode

}

});

}

return (

<div>

<h2>{profile.personal.name}'s Profile</h2>

<button onClick={toggleDarkMode}>

{profile.settings.darkMode ? 'Light Mode' : 'Dark Mode'}

</button>

</div>

);

}

**✅ 4. Boolean State**

**🔄 Toggle State**

function BooleanExample() {

const [isLoggedIn, setIsLoggedIn] = React.useState(false);

return (

<div>

<h1>{isLoggedIn ? "Welcome back!" : "Please log in"}</h1>

<button onClick={() => setIsLoggedIn(!isLoggedIn)}>

{isLoggedIn ? "Log Out" : "Log In"}

</button>

</div>

);

}

**🌟 Best Practices**

**🛡️ 1. Immutable Updates**

// ❌ Wrong - Mutating state directly

const [user, setUser] = useState({name: "John"});

user.name = "Jane"; // Don't do this!

// ✅ Correct - Creating new state object

setUser({...user, name: "Jane"});

**🔄 2. Functional Updates**

// When new state depends on old state

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

// Better than setCount(count + 1)

setCount(prevCount => prevCount + 1);

**📊 3. Complex State Updates**

const [items, setItems] = useState([]);

// Adding item to array

function addItem(newItem) {

setItems(prevItems => [...prevItems, newItem]);

}

// Removing item from array

function removeItem(id) {

setItems(prevItems => prevItems.filter(item => item.id !== id));

}

// Updating item in array

function updateItem(id, updates) {

setItems(prevItems => prevItems.map(item =>

item.id === id ? {...item, ...updates} : item

));

}

**💎 Key Points to Remember**

* Always use setState function to update state
* Never modify state directly
* Create new objects/arrays instead of mutating existing ones
* Use the spread operator (...) for copying objects and arrays
* Use functional updates when new state depends on previous state
* Keep state as simple as possible
* Split complex state into multiple useState calls if needed

🚀 React's useState hook is powerful and flexible, capable of handling any data type you need to manage in your components!

**📝 Building a React Todo Application**

**🔧 Basic Todo App Implementation**

function TodoApp() {

const [currentTodo, setCurrentTodo] = React.useState("");

const [todoList, setTodoList] = React.useState([]);

function handleAddTask() {

if (currentTodo.trim() === "") return;

setTodoList([...todoList, currentTodo]);

setCurrentTodo("");

}

function handleDeleteTask(indexToDelete) {

const updatedList = todoList.filter((\_, index) => index !== indexToDelete);

setTodoList(updatedList);

}

return (

<div>

<h1>To-Do List</h1>

<div>

<input

type="text"

placeholder="Enter a task"

value={currentTodo}

onChange={(e) => setCurrentTodo(e.target.value)}

/>

<button onClick={handleAddTask}>Add Task</button>

</div>

<ul>

{todoList.map((todo, index) => (

<li key={index}>

{todo}

<button onClick={() => handleDeleteTask(index)}>Delete</button>

</li>

))}

</ul>

</div>

);

}

**🧩 Understanding the Components**

**📊 1. State Management**

// Input field state

const [currentTodo, setCurrentTodo] = React.useState("");

// Todo list state

const [todoList, setTodoList] = React.useState([]);

**➕ 2. Adding Tasks**

function handleAddTask() {

// Validation

if (currentTodo.trim() === "") return;

// Add new task

setTodoList([...todoList, currentTodo]);

// Clear input

setCurrentTodo("");

}

**❌ 3. Deleting Tasks**

function handleDeleteTask(indexToDelete) {

const updatedList = todoList.filter((\_, index) => index !== indexToDelete);

setTodoList(updatedList);

}