React: The Component Revolution

Building UIs That Scale

From DOM Manipulation Chaos to Component Harmony

How Sarah finally conquered complex user interfaces

Previously in Sarah's Journey...

JavaScript: Freed her from low-level C programming

TypeScript: Saved her from runtime errors

But now: TaskMaster has grown to 50+ pages!

New Problems:

- DOM manipulation everywhere
- State scattered across files
- Duplicate code for similar UI elements
- Changes require updating multiple files

What is the state?

> State scattered across files

- **State = Memory inside a component**
- It remembers data that changes over time
- When state changes → React **re-renders** the UI

Example

```
<p id="count">Count: 0</p>
<button id="btn">Increase</button>

<script>
  let count = 0;
  const countEl = document.getElementById("count");
  const btn = document.getElementById("btn");

  btn.addEventListener("click", () => {
    count++;
    countEl.textContent = "Count: " + count;
  });
</script>
```

- Works fine: just one state (count variable) and a matching UI (`Count: 0` with id = "count").
- But what if you have 50 pages, each with changing data with states and UI?

The Breaking Point

Even worse, Sarah should manage all kinds of change and corresponding updates.

```typescript
// Sarah's current TypeScript code
function updateTaskList() {
    const list = document.getElementById('taskList');
    list.innerHTML = '';

    tasks.forEach(task => {
        const li = createTaskElement(task);
        list.appendChild(li);
    });

    updateCounter();
    updateProgressBar();
    updateChart();
    refreshNotifications();
    syncWithServer();
    // ... 20 more update functions
}
```

What is React?

> React is the core UI library in a large ecosystem for building interactive, component-based applications through:
>
> - **Components**: Reusable UI building blocks
> - **Declarative**: Describe what UI should look like
> - **Virtual DOM**: Efficient updates
> - **One-way data flow**: Predictable state management

Key Insight: UI as a function of state

```
UI = f(state)
```

Core Concept: Components

Think of UI as LEGO blocks

```jsx
// A component is just a function that returns UI
function Welcome() {
    return <h1>Hello, React!</h1>;
}

// Use it like an HTML tag
<Welcome />
```

This is JSX: JavaScript + XML (HTML) syntax

- Looks like HTML but it's JavaScript
- Gets compiled to React.createElement() calls

Sarah's First Component

Before React (Imperative):

In this code, she tries to create an

- with a text and button for each task.

```
function createTaskItem(task) {
    const li = document.createElement('li');
    li.className = task.completed ? 'completed' : '';

    const text = document.createElement('span');
    text.textContent = task.text;

    const button = document.createElement('button');
    button.textContent = 'Delete';
    button.onclick = () => deleteTask(task.id);

    li.appendChild(text);
    li.appendChild(button);
    return li;
}
```

This is how this JavaScript function is used to create each task list.

```
const list = document.getElementById('task-list');
tasks.forEach(task => list.appendChild(createTaskItem(task)));
```

This approach works — but becomes painful when:

- You need to update or **re-render** items
- You want to react to state changes
- You must synchronize DOM and data manually

She has to constantly writing createElement, appendChild, and innerHTML.

After React (Declarative):

```
function TaskItem({ task, onDelete }) {
    return (
        <li className={task.completed ? 'completed' : ''}>
            <span>{task.text}</span>
            <button onClick={() => onDelete(task.id)}>
                Delete
            </button>
        </li>
    );
}
```

It is used as if it is an HTML tag.

```
<TaskItem key={task.id} task={task} onDelete={handleDelete} />
```

Sarah: "I just describe what I want, not how to build it!"

Props: Component Communication

Props (properties) are **how** we pass data to components:

```
interface TaskItemProps {
    task: Task;
    onToggle: (id: number) => void;
    onDelete: (id: number) => void;
}

function TaskItem({ task, onToggle, onDelete }: TaskItemProps) {
    return (
        <li>
            <input
                type="checkbox"
                checked={task.completed}
                onChange={() => onToggle(task.id)}
            />
            <span>{task.text}</span>
            <button onClick={() => onDelete(task.id)}>×</button>
        </li>
    );
}
```

State: Component Memory

State (variable) is data that changes over time:

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

function Counter() {
    // Declare state variable
    const [count, setCount] = useState(0);

    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={() => setCount(count + 1)}>
                Increment
            </button>
        </div>
    );
}
```

Key: When state changes, React re-renders automatically!

The useState Hook

Instead of managing the state (count) manually, we use the `useState` function to get both the state (count) and the method to change the state (setCount).

```
const [count, setCount] = useState(0); // default value is 0

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

This is another example:

```
function TodoInput({ onAdd }) {
    const [text, setText] = useState('');

    const handleSubmit = (e) => {
        e.preventDefault();
        if (text.trim()) {
            onAdd(text);
            setText(''); // Clear input
        }
    };

    return (
        <form onSubmit={handleSubmit}>
            <input
                value={text}
                onChange={(e) => setText(e.target.value)}
                placeholder="What needs to be done?"
            />
            <button type="submit">Add</button>
        </form>
    );
}
```

Component Composition

Build complex UIs from simple components:

```
function TodoApp() {
    const [todos, setTodos] = useState<Task[]>([]);

    const addTodo = (text: string) => {
        const newTodo = { id: Date.now(), text, completed: false };
        setTodos([...todos, newTodo]);
    };

    return (
        <div className="todo-app">
            <h1>My Tasks</h1>
            <TodoInput onAdd={addTodo} />
            <TodoList todos={todos} />
            <TodoStats todos={todos} />
        </div>
    );
}
```

One-Way Data Flow

Data flows down, events flow up

```
App (state: todos)
  ├── TodoInput (prop: onAdd)
  │     └── Calls onAdd when form submits
  ├── TodoList (prop: todos)
  │     └── TodoItem (props: todo, onToggle, onDelete)
  └── TodoStats (prop: todos)
```

This makes the app predictable and easy to debug!

Virtual DOM Magic

Problem with direct DOM manipulation:

```
// Every small change touches the real DOM
element.innerHTML = newContent;  // Expensive!
element.style.color = 'red';     // Triggers repaint
element.classList.add('active'); // Triggers reflow
```

React's solution: Virtual DOM

1. React creates a virtual representation
2. When state changes, creates new virtual DOM
3. Compares (diffs) old vs new
4. Updates only what changed

Sarah Discovers useEffect

Problem: "When I refresh the page, my todos disappear! I need to save todos to localStorage!"

- React keeps data in **memory** (state): But once you refresh the page — **it’s gone**.
- We need a **persistent place** to save data → that’s where `localStorage` comes in.

What is `localStorage`?

- Built-in browser storage
- Saves data as **key-value pairs**
- Persists even after reload or browser close
  (until user clears it manually)

```
localStorage.setItem('name', 'Sarah');
localStorage.getItem('name'); // "Sarah"
```

Why do we need `useEffect`?

React has two worlds:

1. Rendering – What the screen looks like (pure, predictable)
2. Side Effects – Things that happen outside React (fetching, saving, subscribing)

useEffect handles the second world — anything that touches the outside world.

When React renders:

- It draws your UI based on state + props
- It should **not** directly read/write browser APIs, network, storage, etc.

Why?

- Because React may re-render many times
- It must stay pure (no side effects during render).

The `useEffect` as the Solution

> useEffect = “Do something after render”

It tells React:

"Once you’ve updated the screen, now run this code."

Example:

```
useEffect(() => {
  console.log("App rendered!");
});
```

This runs after React finishes painting the UI — not during rendering.

```
function TodoApp() {
    const [todos, setTodos] = useState<Task[]>([]);

    // Run side effect after render
    useEffect(() => {
        localStorage.setItem('todos', JSON.stringify(todos));
    }, [todos]); // Only run when todos change

    // Load from localStorage on mount
    useEffect(() => {
        const saved = localStorage.getItem('todos');
        if (saved) {
            setTodos(JSON.parse(saved));
        }
    }, []); // Empty array = run once

    // ... rest of component
}
```

1. useState([])

- Creates an empty todo list in memory

2. useEffect(..., [])

- Runs once when component mounts
- Loads todos from localStorage (if any)
- Updates state with saved data

3. useEffect(..., [todos])

- Runs every time todos changes
- Saves the latest list to localStorage

Think of useEffect as React's “after paint” hook; React paints the screen — then you can safely do side work (fetch, log, save, etc.).

Sarah's Revelation

Sarah: "It's not just less code—it's better organized code!"

> **Before React**:
>
> - 500+ lines of DOM manipulation
> - State scattered everywhere
> - Hard to add features
> - Bugs from inconsistent updates
>
> **After React**:
>
> - 200 lines of declarative components
> - Centralized state management
> - Easy to add/modify features
> - Predictable updates

Vanilla JS/TS:

```
// Manual DOM updates
function updateUI() {
    element.innerHTML = '';
    data.forEach(item => {
        const el = createElement(item);
        element.appendChild(el);
    });
}

// Call everywhere data changes
updateUI();
```

React:

```
// Automatic updates
function List({ data }) {
    return (
        <ul>
            {data.map(item =>
                <Item key={item.id} {...item} />
            )}
        </ul>
    );
}
// React handles updates!
```

React Ecosystem

Sarah discovers a whole ecosystem:

- **React Router**: Client-side routing
- **Redux/Zustand**: Advanced state management
- **React Query**: Server state management
- **Styled Components**: CSS-in-JS
- **Next.js**: Full-stack React framework
- **React Native**: Mobile apps

```
// Example: React Router
<Routes>
    <Route path="/" element={<Home />} />
    <Route path="/grades" element={<GradeCalculator />} />
    <Route path="/tasks" element={<TaskManager />} />
</Routes>
```

Summary: The Complete Journey

```
C Language (Low-level hell)
    ↓ JavaScript (High-level freedom)
        ↓ TypeScript (Type safety)
            ↓ React (Component architecture)
                ↓ Happy, productive developer!
```

Each technology solved specific problems:

- **JavaScript**: Escaped manual memory management
- **TypeScript**: Caught errors before runtime
- **React**: Tamed UI complexity

Remember...

> Think in Components
>
> Let React Handle the DOM
>
> State Drives Everything