Day 01 — React Fundamentals

JSX → Elements • Virtual DOM • Reconciliation (Fiber) • Render & Commit • Lifecycle & Hooks • When to choose React • Ecosystem

> “ Goal: Build a solid mental model of **how React updates the UI efficiently** and when to use it. ”

Agenda

1. What is React & why it exists
2. JSX & React Elements
3. Virtual DOM
4. Reconciliation (Fiber) & Keys
5. Render & Commit phases + batching
6. Lifecycle & Hooks essentials
7. When to choose React
8. Ecosystem tour
9. Mini–hands-on exercises & quiz

Learning Outcomes

By the end of Day 01, you will:

- Explain how JSX becomes **React elements** and ultimately **DOM**.
- Describe **Virtual DOM**, **diffing**, and why React updates are efficient.
- Understand **Fiber** and the difference between **render** and **commit** phases.
- Use basic **hooks** (`useState` , `useEffect`) and reason about lifecycle.
- Identify scenarios where **React** is (and isn't) the right choice.

What is React?

- A **UI library** for building component-based interfaces.
- **Declarative**: Describe what the UI should look like; React figures out **how** to update it.
- **Predictable** updates via state → re-render → diff → commit.

Problem React solves

- Direct DOM manipulation is **imperative**, error-prone, and slow at scale.
- React abstracts the DOM behind **elements**, **Virtual DOM**, and **reconciliation**.

Imperative vs Declarative

Imperative (vanilla DOM)

```
const el = document.createElement("button");
el.textContent = "Clicked 0 times";
let count = 0;
el.addEventListener("click", () => {
  count++;
  el.textContent = `Clicked ${count} times`;
});
document.body.appendChild(el);
```

Declarative (React)

```
function Button() {
  const [count, setCount] = React.useState(0);
  return (
    <button onClick={() => setCount((c) => c + 1)}>
      Clicked {count} times
    </button>
  );
}
```

> “ In React, we re-describe the **UI for a given state**; React updates the DOM efficiently. ”

JSX & React Elements

- **JSX** = JavaScript XML, syntax sugar for describing UI.
- Transpiled by **Babel/TypeScript** to `React.createElement` calls.

JSX

```jsx
const el = <h1 className="title">Hello, React!</h1>;
```

Transpiled (conceptually)

```js
const el = React.createElement("h1", { className: "title" }, "Hello, React!");
```

React Element (plain object)

```
{
  type: 'h1',
  props: { className: 'title', children: 'Hello, React!' }
}
```

> Elements are **lightweight descriptions** of what to render, not real DOM nodes.

Rendering Basics (React 18)

```
import { createRoot } from "react-dom/client";
import App from "./App";

const container = document.getElementById("root");
const root = createRoot(container);
root.render(<App />);
```

- React builds a **tree of elements** from your components.
- The tree is compared to the previous one → **diff** → minimal DOM updates.

Virtual DOM

- A **lightweight JS representation** of the real DOM.
- React updates **Virtual DOM** first, then determines the **minimal set of real DOM changes**.

Why it’s fast

- JS operations are cheaper than touching the DOM.
- React batches DOM writes and reads to avoid layout thrashing.

Reconciliation & Fiber (High-level)

- **Reconciliation** = comparing the new element tree with the previous one (**diffing**).
- **Fiber** = React's internal architecture for splitting work into **units** that can be paused and resumed.
- Enables **responsive UIs** by prioritizing urgent updates and batching non-urgent ones.

Key heuristics

- Different `type` → **replace** node.
- Same `type` → **update** props and children.

Keys Matter

Bad (index as key)

```
{
  items.map((item, i) => <Row key={i} value={item} />);
}
```

Good (stable unique key)

```
{
  items.map((item) => <Row key={item.id} value={item} />);
}
```

> Using indices as keys can cause incorrect state association and

Render vs Commit Phases

- **Render phase**: Build the new element/fiber tree. Pure & can be paused.
- **Commit phase**: Apply changes to the **real DOM**; layout/paint happens here.
- React **batches** state updates to minimize commits.

Automatic batching (React 18)

```
setCount((c) => c + 1);
setText("hello");
// One re-render, one commit (batched)
```

Component Lifecycle & Hooks

Lifecycle moments

- **Mount** → component added to the DOM
- **Update** → state/props change triggers render
- **Unmount** → component removed

Hooks

- `useState` — local state
- `useEffect` — side effects (fetch, subscriptions, timers)
- Cleanup in effects runs on unmount or before re-running effect

```
function Clock() {
  const [now, setNow] = React.useState(() => new Date());
  React.useEffect(() => {
    const id = setInterval(() => setNow(new Date()), 1000);
    return () => clearInterval(id); // cleanup on unmount
  }, []);
  return <time>{now.toLocaleTimeString()}</time>;
}
```

When to Choose React

Great fit

- Interactive UIs, complex state flows, dashboards, SPAs/MPAs.
- Teams that value component reuse, strong ecosystem, TypeScript support.

Maybe not

- Mostly static sites with minimal interactivity (consider static site generators).
- Extremely simple pages where vanilla HTML/CSS/JS suffices.

Ecosystem Overview (Day 01 Teaser)

- **Rendering targets**: `react-dom` (web), React Native (mobile)
- **State**: Context, Redux, Zustand, Jotai (covered later)
- **Styling**: CSS Modules, Tailwind CSS, styled-components
- **Tooling**: Babel, Webpack, Vite, SWC (Day 08 deep dive)
- **Framework**: Next.js App Router (Days 09–11)
- **DevTools**: React DevTools, ESLint, Prettier

Hands-on #1 — Counter (JSX + State)

```
function Counter() {
  const [count, setCount] = React.useState(0);
  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount((c) => c + 1)}>+1</button>
      <button onClick={() => setCount(0)}>Reset</button>
    </div>
  );
}
```

Focus: JSX, state updates, rerender mental model.

Hands-on #2 — List & Keys

```
function PlayersList({ players }) {
  return (
    <ul>
      {players.map((p) => (
        <li key={p.id}>{p.name}</li>
      ))}
    </ul>
  );
}
```

Variation: Insert a new player at the top; observe why stable keys matter.

Demo: JSX → createElement → Element Object

```
const jsx = <h1 className="title">Hello</h1>;
```

```
// Conceptual output after transpilation
const el = React.createElement("h1", { className: "title" }, "Hello");
// Element is a plain object
// { type: 'h1', props: { className: 'title', children: 'Hello' } }
```

Takeaway: React works with **plain objects** first, not DOM nodes.

Common Pitfalls (Day 01)

- Mutating state directly instead of creating new values.
- Using array **index** as `key`.
- Side effects directly in the render body (use `useEffect`).
- Assuming state updates are synchronous (they’re not).

Quick Quiz

1. What does JSX compile to?
2. What's the purpose of the Virtual DOM?
3. When should you avoid using array index as a key?
4. Name the two high-level phases of an update.

(Answers: `React.createElement` *calls; stage DOM updates in JS for efficient diff/commit; when list items can reorder/insert/remove; render & commit)*

Discussion Prompts

- Where in your current codebase would React’s declarative model simplify logic?
- Which components could benefit most from stable keys and memoization?

Appendix — Batching & Transitions (Preview)

- React 18 **automatic batching** even across async boundaries.
- **Transitions** (e.g., `startTransition`) mark non-urgent updates to keep UI responsive.

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

function Search({ query, setQuery }) {
  function onInput(e) {
    const q = e.target.value;
    setQuery(q); // urgent (keeps input responsive)
    startTransition(() => {
      // non-urgent (e.g., filter big list)
      // setFilteredData(expensiveFilter(q));
    });
  }
```

Resources

- React Docs: *Main Concepts* & *Reconciliation*
- React DevTools (Chrome/Firefox)
- Babel REPL (to view JSX → JS transform)

Next Session (Day 02)

- **JavaScript advanced refresher**: ES6+, arrays, async/await, modules, closures.
- Prepare: Bring an example of imperative DOM code you've written recently.