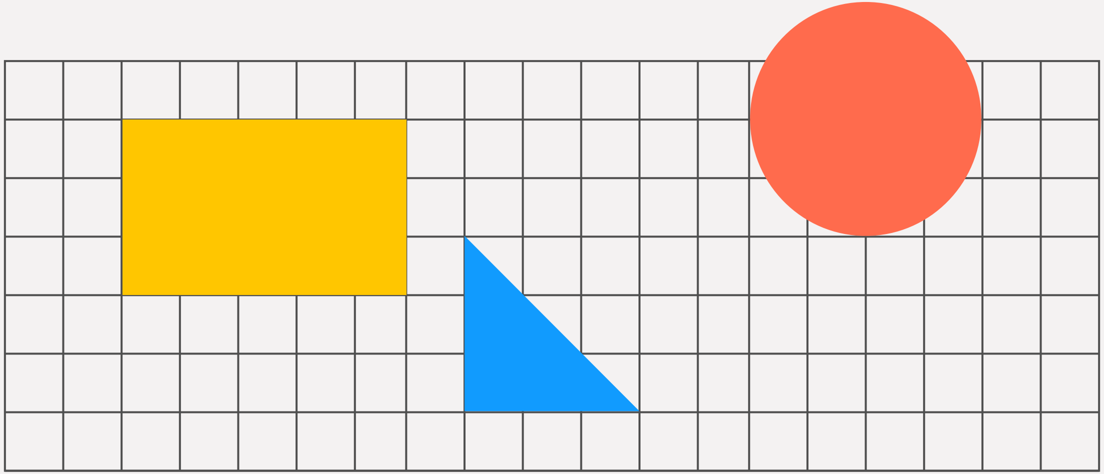
React: Act I

Ali Abrishami



In This Lecture You Will...

a

Learn What DOM is

b

Figure out a way to make your pages interactive

c

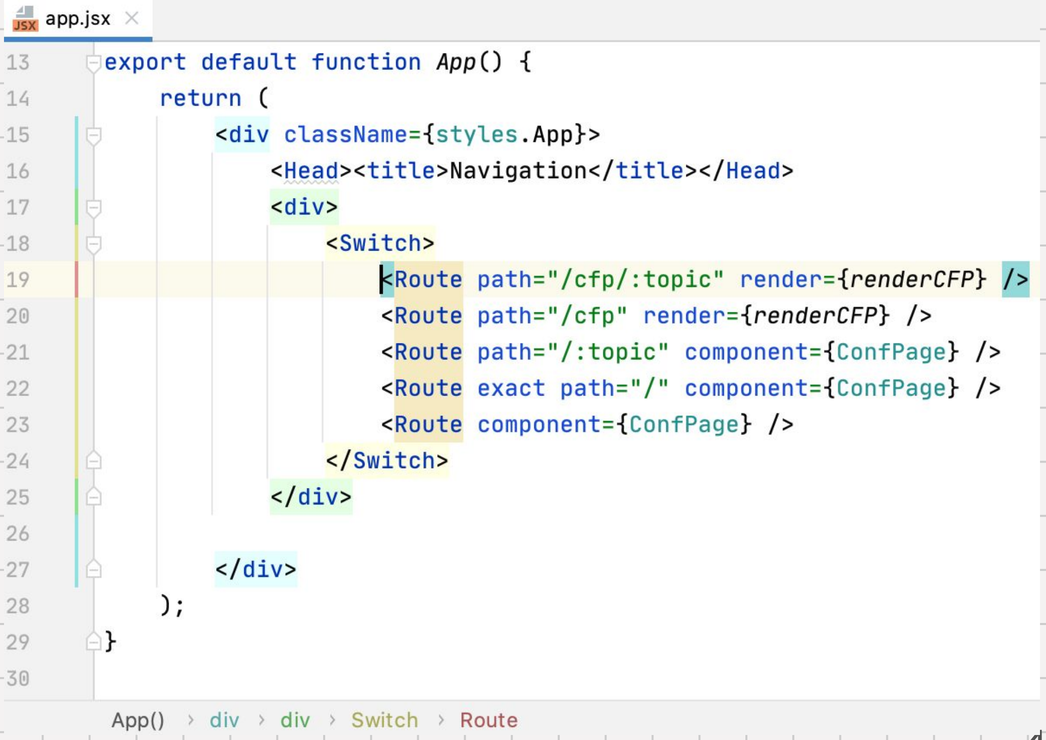
Understand async programming usages in web

Setup a React Project!

- Install Node.js (includes npm or use pnpm / yarn)
- Create app: `npm create vite@latest my-app`
- Move into project: `cd myapp`
- Install deps: `npm install` (or `pnpm i`)
- Start dev server: `npm run dev`
- Open `http://localhost:5173`

JS… But With an X

- A syntax extension for JavaScript
- Lets you write **HTML-like code** inside JS
- Makes UI structure more readable and **declarative**



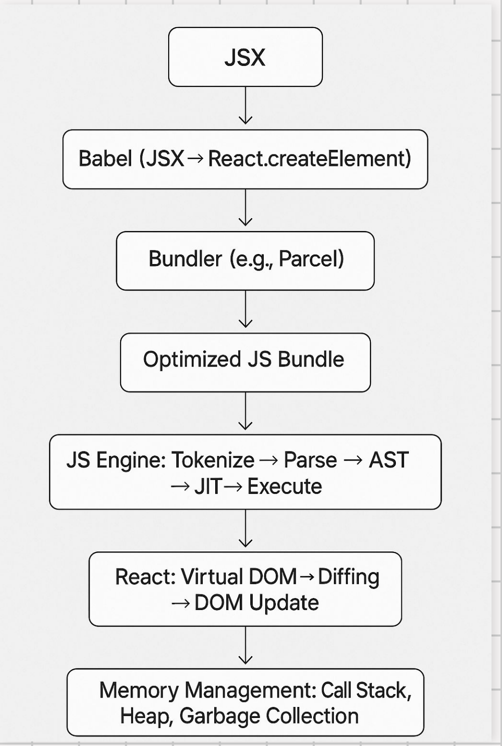
```
export default function App() {
    return (
        <div className={styles.App}>
            <Head><title>Navigation</title></Head>
            <div>
                <Switch>
                    <Route path="/cfp/:topic" render={renderCFP} />
                    <Route path="/cfp" render={renderCFP} />
                    <Route path="/:topic" component={ConfPage} />
                    <Route exact path="/" component={ConfPage} />
                    <Route component={ConfPage} />
                </Switch>
            </div>

        </div>
    );
}
```

JSX Syntax Rules & Expressions

- JSX looks like HTML, but it is JavaScript, so rules are stricter and explicit.
- You can embed any JavaScript expression inside `{}` (values, function calls, ternaries).
- Statements like `if`, `for`, or `while` are not allowed directly inside JSX.
- JSX must return a single root element (use fragments `<>...</>` when needed).
- Attribute names use camelCase (`className`, `onClick`, `htmlFor`).

Compile It With Babel!

- Babel is a **JavaScript compiler** used in React projects
- It allows developers to write modern JavaScript and JSX syntax
- Browsers cannot understand JSX directly
- Babel converts JSX into plain JS the browser can execute
- It also transpiles modern JS features (like ES6+) to older versions
- Ensures cross-browser compatibility and smoother development



JSX

Babel (JSX → React.createElement)

Bundler (e.g., Parcel)

Optimized JS Bundle

JS Engine: Tokenize → Parse → AST → JIT → Execute

React: Virtual DOM → Diffing → DOM Update

Memory Management: Call Stack, Heap, Garbage Collection

VDOM In depth

- React maintains a Virtual DOM tree in memory (a lightweight copy of the real DOM)
- When state or props change, React builds a new VDOM tree representing the updated UI.
- The Fiber reconciler compares the new and old trees node by node to find differences.
- Each node gets marked with an effect flag (`Placement`, `Update`, `Deletion`) during diffing.
- The commit phase applies only the required DOM operations (insert, update, remove).
- This results in minimal re-rendering, improving performance and consistency.

The Algorithm

- The legacy React docs note that a full tree diff is **$O(n^3)$** (far too **slow** for **real UIs**)
- React introduced heuristics to achieve **O(n)** diffing using simple rules.
- Rule 1: Different element types always produce **new subtrees**.
- Rule 2: Developers use key props so React can track **moved** or **reused** elements.
- Modern React (**Fiber**) made reconciliation **incremental** and **interruptible**, improving responsiveness.
- React now uses **priority lanes**, **effect lists**, and **bailout logic** for faster and minimal DOM updates.

An Example

- Consider a component tree with 3 levels: root → parent → child nodes.
- When a depth-2 node’s props or state change, React rebuilds that subtree from that node downward.
- The Virtual DOM **compares** the new depth-2 node with its previous version.
- If the type is the same, React **reuses** the node and diffs its children.
- Only the affected subtree (<u>depth-2 and its descendants</u>) is **re-rendered** and **reconciled**.
- The commit phase then applies the minimal DOM changes for that subtree, the rest of the tree stays **untouched**.

The French Fast!

- Vite is a next-generation **frontend build tool** created by Evan You (Vue’s creator).
- Designed to make frontend development extremely **fast** and **modern**.
- Uses native ES modules during development instead of bundling everything.
- Leverages Rollup for optimized production builds.
- Offers **instant hot module replacement** (HMR) for real-time code updates.
- Supports frameworks like React, Vue, Svelte, Preact, and plain JS out of the box.

Use Vite

- **Create a project** using `npm create-vite@latest my-app`
- Select your framework (e.g., React) when prompted.
- Move into your project: `cd my-app`
- **Install dependencies**: `npm install`
- **Start the dev server**: `npm run dev` → opens http://localhost:5173.
- **Build for production**: `npm run build`, then preview using `npm run preview`.

Component-based Dev.

- A modular architecture where the UI is divided into small, self-contained units.
- Each component handles its own logic, structure, and styling.
- Encourages **reusability**: one component can appear in multiple places.
- Promotes **maintainability**: updates stay local to a specific component.
- Supports **composition**: components combine to form larger, complex interfaces.
- Improves **collaboration**: multiple developers can work independently on different parts.

React Components

- React implements component-based development as its **core** principle.
- Components are independent, reusable building blocks of the UI.
- They accept **props as input** and **return JSX** describing what to render.
- React efficiently updates and re-renders components when data changes.
- Two main types: **Function-based** (modern) and **Class-based** (legacy).
- Enable declarative UI design, focusing on what to show, not how to update it.

Styling in React

1. Plain CSS: Import .css files and apply styles with `className`.

2. Global Styles: Put all styles in index.css and use across components.

3. Inline Styles: Use `style={{ ... }}` with a JS object directly in JSX.
   - Uses camelCase (backgroundColor, not background-color)
   - Values are strings or numbers

4. Tailwind CSS: Utility-first CSS with quick class-based styling.

5. Material UI: Prebuilt styled React components with theming.

6. Motion: Add animations using Framer Motion or similar libraries.

JS Expressions in JSX

- Curly braces `{}` embed JavaScript directly inside JSX.
- Example: `<h1>{user.name}</h1>` renders variable values.
- Expressions allowed:
  `{x + y}`, `{isLoggedIn ? 'Hi' : 'Login'}`.
- Only expressions, no statements (e.g., no `if`, `for`, etc.).

Lists in JSX

- Render arrays with `{array.map(...)}`: `{items.map(i => <p>{i}</p>)}`
- Each element needs a unique key: `{u => <li key={u.id}>{u.name}</li>}`
- Return JSX or components from `.map()`:
  `{posts.map(p => <PostCard post={p} />)}`
- Avoid using index as key unless list never changes.
- Combine with fragments:
  `{list.map(i => <><h3>{i.title}</h3><p>{i.text}</p></>)}`

Event handling

- Use camelCase events: `<button onClick={handleClick}>OK</button>`
- Pass a function, not a call: `onClick={doSomething}` not `onClick={doSomething()}`
- Get event object: `onClick={(e) => console.log(e.target)}`
- Update state on input: `onChange={(e) => setValue(e.target.value)}`
- Handle forms: `<form onSubmit={handleSubmit}>...</form>`

States? What States?

- **State:** component **memory** that **stores dynamic data.**
- Allows React components to remember values between renders.
- Changing state triggers automatic UI updates.
- Managed using `useState()` hook → `[value, setValue]`.
- Local to the component, but can be passed to children as props.

State in React

- Use `useState()` hook to store component data
  `const [count, setCount] = useState(0)`
- Changing state re-renders the component **automatically**.
- Always update using the setter: `setCount(count + 1)`
- State is local to each component by default.
- Can pass state and setters as props to child components.

Class Activity Time

If two sibling components need the same data, where can that state live so both can access it?

What happens when state is stored in a child but the parent needs to react to its changes?

Which component should own state: the one that updates it, the one that reads it, or the closest common ancestor? Why?

Hook

- **Hook**: reusable function that adds behavior to code.
- Can be called at **specific points** in a program.
- Often used to **share logic** across different parts.
- Doesn’t run on its own, **invoked** by **other** code.
- Examples: event hooks, lifecycle hooks, logging hooks.

React Hooks

- React hooks: special functions to use state and features in functional components.
- Examples: `useState`, `useEffect`, `useRef`.
- Allow stateful logic without writing class components.
- Follow rules of hooks: only call at **top level**, in React functions.
- Enable code reuse and side-effect management cleanly.

`setState` Hook

- Comes from `useState()`: `const [value, setValue] = useState(0)`
- Updates the state and triggers a re-render.
- Asynchronous: multiple updates may batch together.
- Use a callback form for dependent updates: `setValue(v => v + 1)`
- **Never** modify state directly: always use the **setter**.

React Props

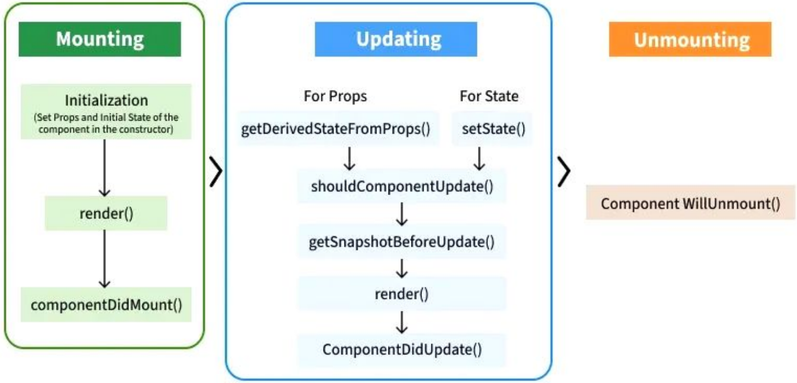
- **Props**: inputs passed from parent to child components.
- Used to configure or customize components.
- Read-only: child components can’t modify them.
- Passed like attributes: `<UserCard name="Darth Vader" age={49} />`
- Access inside component: `function UserCard({ name, age }) { ... }`

Component Composition

- Building UIs by combining smaller components together.
- Promotes reusability and clean separation of concerns.
- Components can nest inside others: `<Layout><Header /></Layout>`
- Use props and children to pass content: `<Card>{content}</Card>`
- Encourages modular, maintainable React architecture.

Component Lifecycle

- Created: Component instance comes into existence with initial data.
- Rendered: A virtual representation is calculated from current state or inputs.
- Committed: Changes are applied to the UI in a consistent snapshot.
- Active: Component is visible and interacting with the outside world.
- Disposed: Component is removed and all associated resources are released.

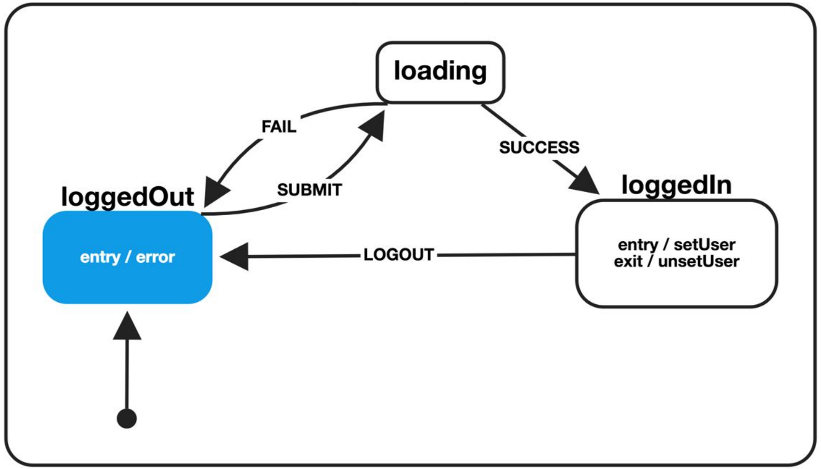


Source: GeeksForGeeks

Class Activity Time

You are given the state machine of a login page/section:

1. What components do you need for the following page?
2. Discuss the lifecycle of `LoginButton`.
3. Does it make sense to implement all the said components from scratch? Is there a better approach?



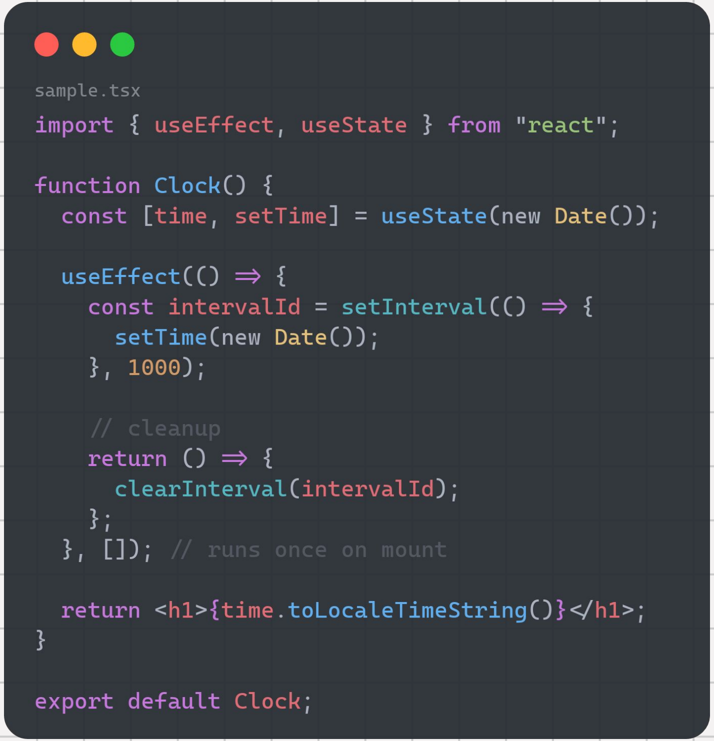
Source: CSS Tricks

Debug Your React Code

- Use React DevTools. inspect components, props, and state live.
- `console.log()` wisely. log variables, state, and props in key spots.
- Check error boundaries. wrap components to catch rendering errors.
- Verify component tree: ensure correct data flow via props & state.
- Watch re-renders. use `React.memo`, `useCallback`, and `useEffect` deps carefully.

The `useEffect` Hook

- `useEffect` runs after React renders, so it’s for reacting to changes, not controlling rendering.
- It’s designed to connect your component to external things like APIs, timers, etc.
- An effect can return a cleanup function to stop or undo what it started.
- The dependency array tells React when the effect should run; accuracy here matters.
- Keeping render logic pure makes components easier to reason about and test..

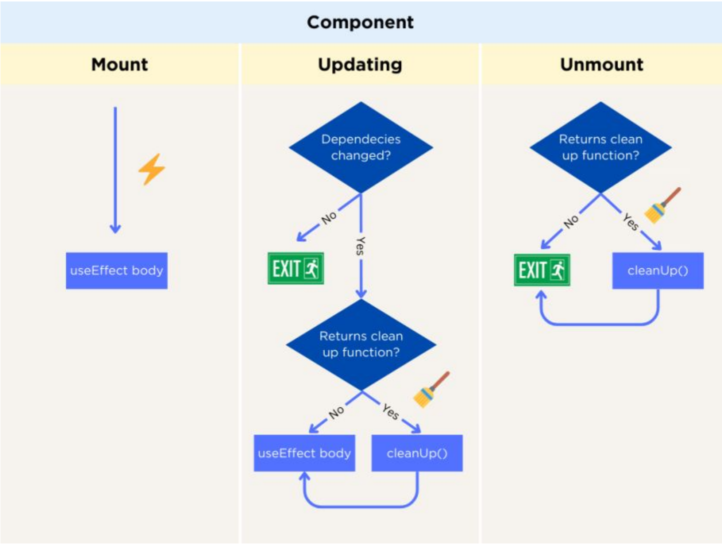


sample.tsx

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

function Clock() {
  const [time, setTime] = useState(new Date());

  useEffect(() => {
    const intervalId = setInterval(() => {
      setTime(new Date());
    }, 1000);

    // cleanup
    return () => {
      clearInterval(intervalId);
    };
  }, []); // runs once on mount

  return <h1>{time.toLocaleTimeString()}</h1>;
}

export default Clock;
```

useEffect Rules!

- Do not use useEffect to derive state from props, compute it during render!
- Calling setState blindly inside effects causes cascades.
- Missing dependencies produce stale closures (smoke testing is not real testing!)
- Effects may run multiple times in Strict Mode. (fragile logic will expose itself!)
- If removing an effect changes nothing, it never belonged there.

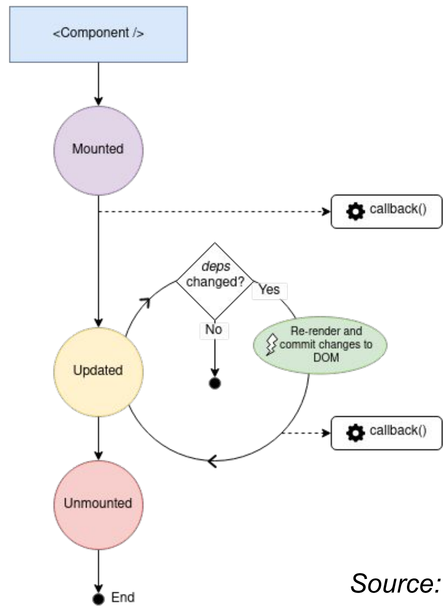


Source: Babbel

Dependencies!

- Dependencies are values from render scope that the effect uses and reacts to.
- An empty array [ ] means “run once on mount, clean up on unmount.”
- Omitting the array means “run after every render” (rarely what you want).
- Changing any dependency triggers cleanup first, then re-runs the effect.
- Incorrect or missing dependencies cause stale data and hard-to-debug bugs.

useEffect() Hook



Source: Dmitri Pavlutin