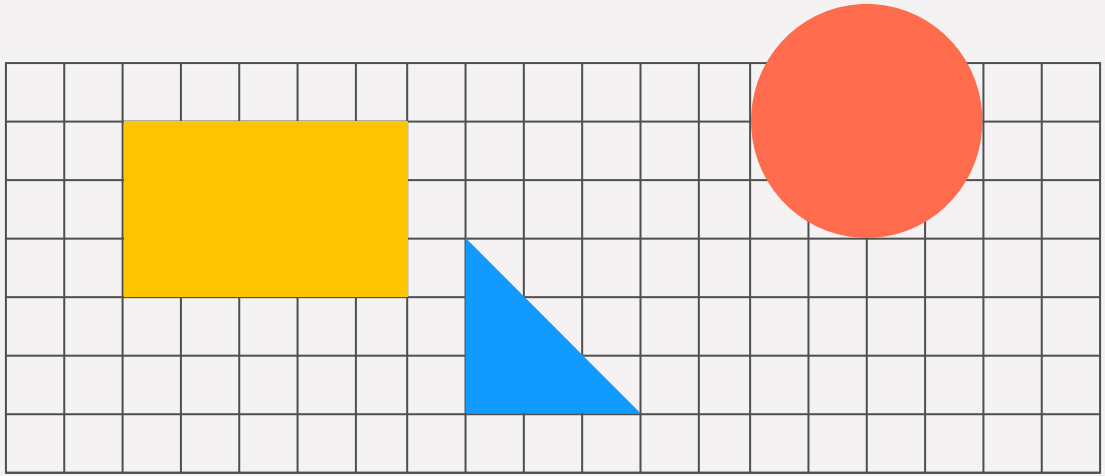
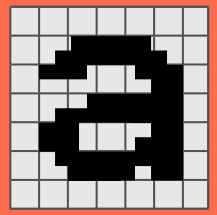
Structure Meets Behaviour!

Ali Abrishami



In This Lecture You Will...



Learn What DOM is

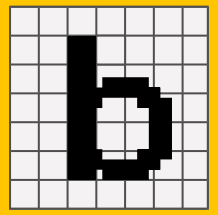
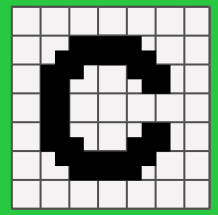


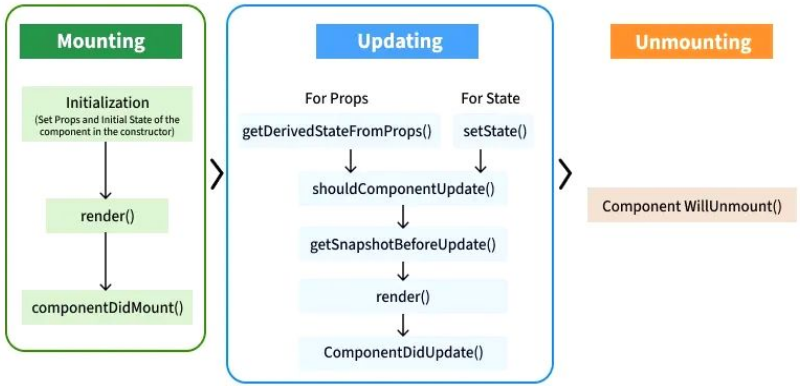
Figure out a way to make your pages interactive



Understand async programming usages in web

Component Lifecycle

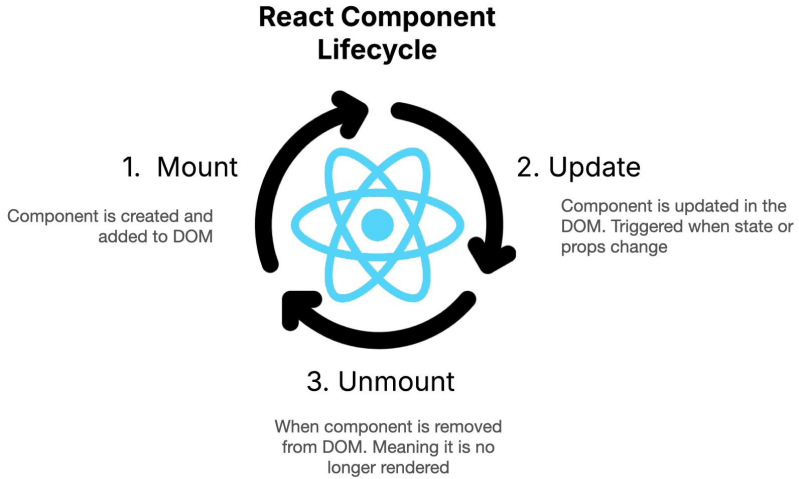
- **Created:** Component instance comes into existence with initial data.
- **Rendered:** A virtual representation is calculated from current state or inputs.
- **Committed:** Changes are applied to the UI in a consistent snapshot.
- **Active:** Component is visible and interacting with the outside world.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



Source: GeeksForGeeks

Component Lifecycle

- Components move through **mount**, **update**, and **unmount** phases during their lifetime.
- Rendering must remain pure and depend only on props and state.
- Side effects are managed with effects that run in response to dependency changes.
- Cleanup logic prevents memory leaks and stale subscriptions.



The `useEffect` Hook

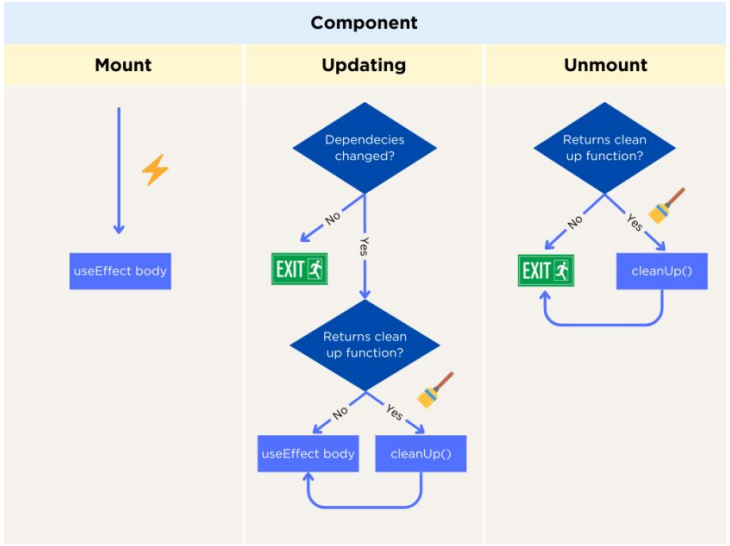
- `useEffect` runs after React renders, so it's for reacting to changes, not controlling rendering.
- It's designed to connect your component to **external things** like APIs, timers, etc.
- An effect can return a **cleanup function** to stop or undo what it started.
- The **dependency array** tells 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!

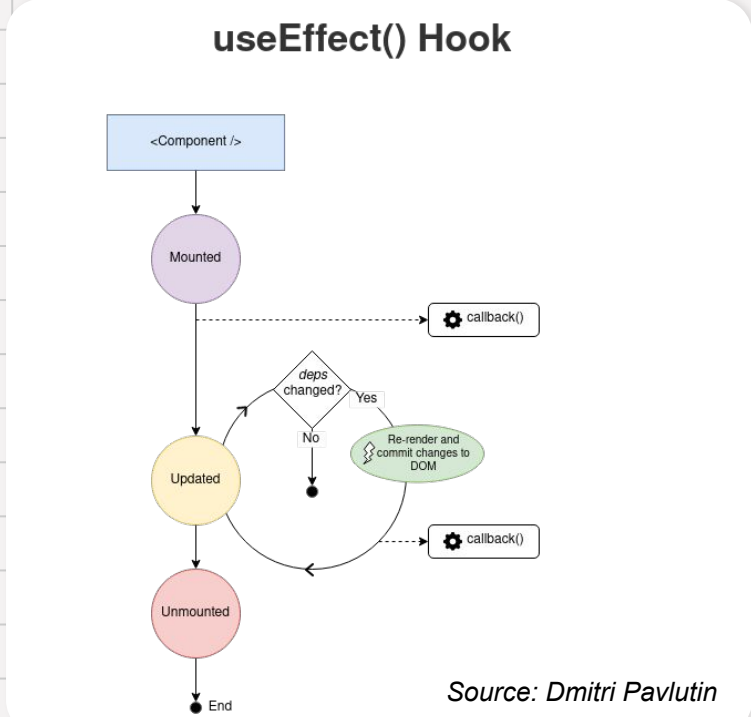
- Do not use `useEffect` to derive state from props, compute it during render!
- Calling `setState` blindly inside effects causes cascades.
- Missing dependencies produce stale closures (smoke testing is not real testing!)
- Effects may run multiple times in Strict Mode (fragile logic will expose itself!).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



Source: Babbel

Dependencies!

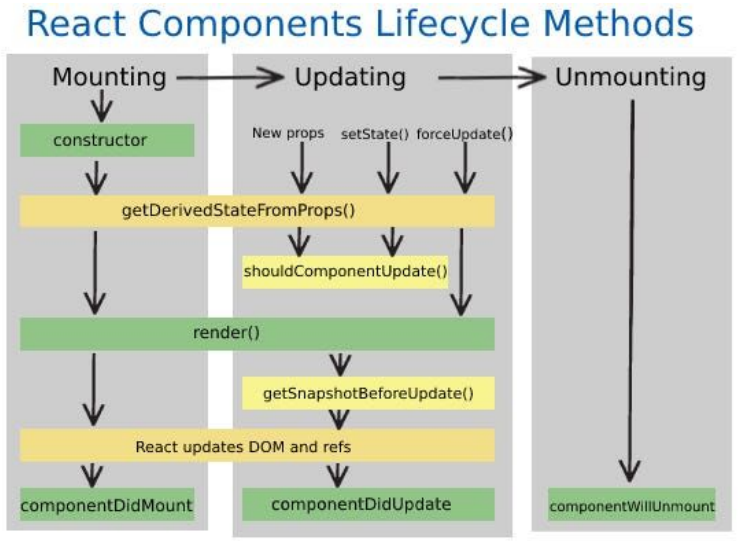
- **Dependencies** are values from render scope that the effect uses and reacts to.
- An empty array [ ] means “run once on mount, clean up on unmount.”
- Omitting the array means “run after every render” (rarely what you want).
- Changing any dependency triggers cleanup first, then re-runs the effect.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



Source: Dmitri Pavlutin

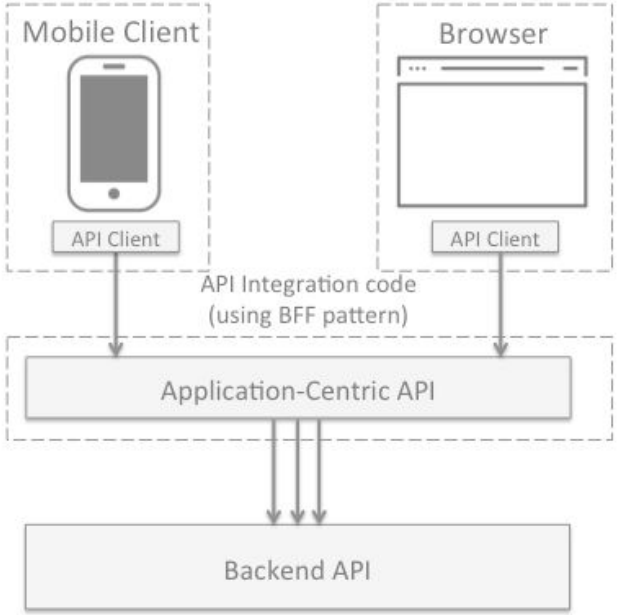
Conclusion of `useEffect`

- `useEffect` runs side effects after render and replaces multiple class lifecycle methods.
- Dependency arrays control when an effect executes and when it is skipped.
- Cleanup functions run before re-execution or unmount to avoid leaks and inconsistencies.
- Effects capture values from render, making dependency correctness critical.
- `useEffect` timing differs from `useLayoutEffect`, which runs synchronously before paint.



Fetching Some Data

- Frontend applications fetch data by outsourcing storage and logic to a backend.
- Data is requested asynchronously over HTTP APIs.
- Fetching typically occurs after initial render using effects.
- UI state reflects loading, success, and error phases.
- React focuses on rendering, not data persistence or transport.



Method I: fetch

- Use the browser's built-in `fetch` API for HTTP requests.
- Requires manual handling of errors and JSON parsing.
- Commonly combined with `useEffect` for data loading on mount.
- Cleanup or guards are needed to avoid state updates after unmount.
- No external dependencies, but more boilerplate code.

sample.tsx

```tsx
function Users() {
  const [users, setUsers] = React.useState([]);
  const [loading, setLoading] = React.useState(true);

  React.useEffect(() => {
    let ignore = false;

    async function load() {
      try {
        const res = await fetch("/api/users");
        if (!res.ok) throw new Error("Request failed");
        const data = await res.json();
        if (!ignore) setUsers(data);
      } catch (e) {
        console.error(e);
      } finally {
        if (!ignore) setLoading(false);
      }
    }

    load();
    return () => { ignore = true; };
  }, []);

  if (loading) return <p>Loading…</p>;
  return <pre>{JSON.stringify(users, null, 2)}</pre>;
}
```

Method II: Axios!

- Use Axios, a third-party HTTP client designed for promise-based requests that automatically parses JSON responses and normalizes error handling.
- Provides built-in support for request cancellation, interceptors, and global defaults.
- Integrates naturally with `useEffect` for lifecycle-driven data fetching.
- Introduces an external dependency in exchange for cleaner, more reusable request logic.

sample.tsx

```tsx
function Users() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const controller = new AbortController();

    axios.get("/api/users", { signal: controller.signal })
      .then((res) => setUsers(res.data))
      .catch((err) => {
        if (err.name !== "CanceledError") console.error(err);
      })
      .finally(() => setLoading(false));

    return () => controller.abort();
  }, []);

  if (loading) return <p>Loading…</p>;
  return <pre>{JSON.stringify(users, null, 2)}</pre>;
}
```

it's more concise and elegant!

Forms!

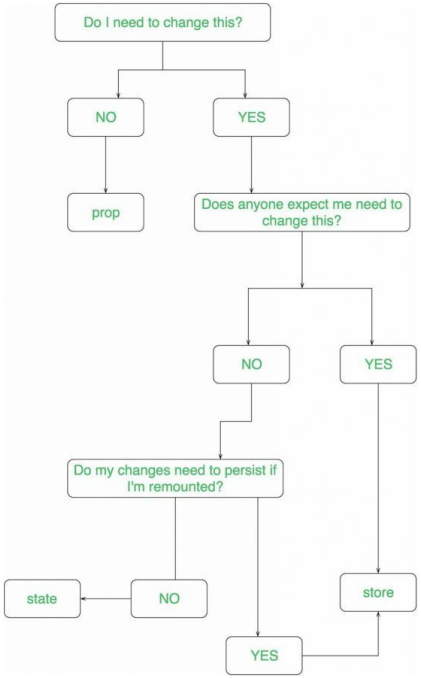
- Forms are controlled by React state to keep the UI and data in sync.
- **Controlled components** use `value` and `onChange` to manage inputs predictably.
- **Uncontrolled components** rely on refs for simpler, less state-heavy cases.
- Form submission is handled via event handlers rather than default browser behavior.
- Validation can be implemented synchronously in state or asynchronously with side effects.

sample.tsx

```tsx
function LoginForm() {
  const [email, setEmail] = React.useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log(email);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        placeholder="Email"
      />
      <button type="submit">Submit</button>
    </form>
  );
}
```

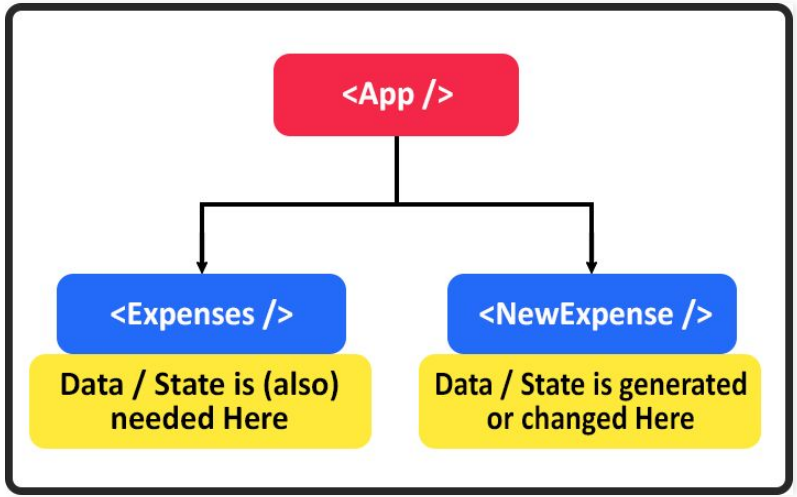
State Management Patterns

- State defines the source of truth for dynamic UI behavior.
- Poor state placement leads to prop drilling and duplication.
- Patterns help determine where state should live.
- Simpler patterns scale better before introducing libraries.
- React encourages explicit data flow over implicit sharing.



Pattern I: Lifting State Up

- State is moved to the closest common ancestor component.
- Child components receive data via props.
- Updates are centralized and predictable.
- Prevents state divergence between siblings.
- Increases parent responsibility but improves consistency.



Before...

sample.tsx

```tsx
function A(){
  const [v,setV]=React.useState(0);
  return <button onClick={()⇒setV(v+1)}>{v}</button>
}
function B(){
  const [v,setV]=React.useState(0);
  return <button onClick={()⇒setV(v+1)}>{v}</button>
}
export default function App(){return <><A/><B/></>}
```

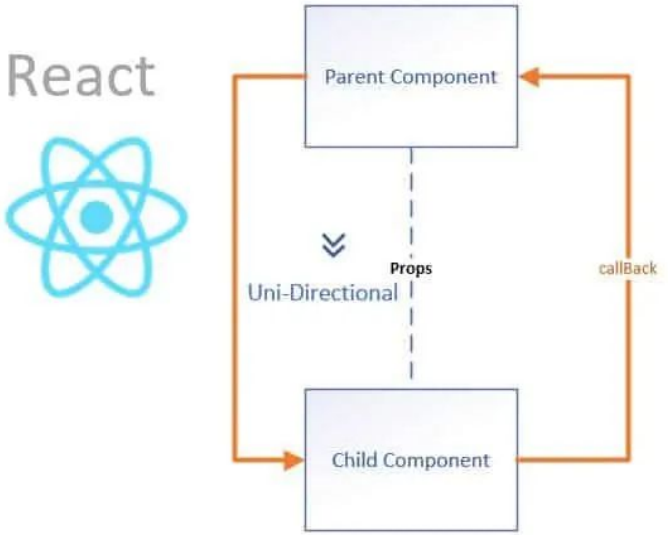
AFTER!

sample.tsx

```tsx
function A({v,onInc}){return <button onClick={onInc}>{v}</button>}
function B({v,onInc}){return <button onClick={onInc}>{v}</button>}
export default function App(){
  const [v,setV]=React.useState(0);
  const inc=()⇒setV(x⇒x+1);
  return <><A v={v} onInc={inc}/><B v={v} onInc={inc}/></>
}
```

Pattern II: Callbacks as Props

- Parent components pass functions to children for state updates.
- Children communicate intent without owning state.
- Enables unidirectional data flow.
- Keeps business logic out of presentational components.
- Can lead to prop drilling in deep trees.



Before...

sample.tsx

```tsx
function Child(){
  return <button onClick={()⇒{
    /* can't update parent */
  }}>Add</button>
}
export default function App(){
  const [n,setN]=React.useState(0);
  return <><p>{n}</p><Child/></>
}
```

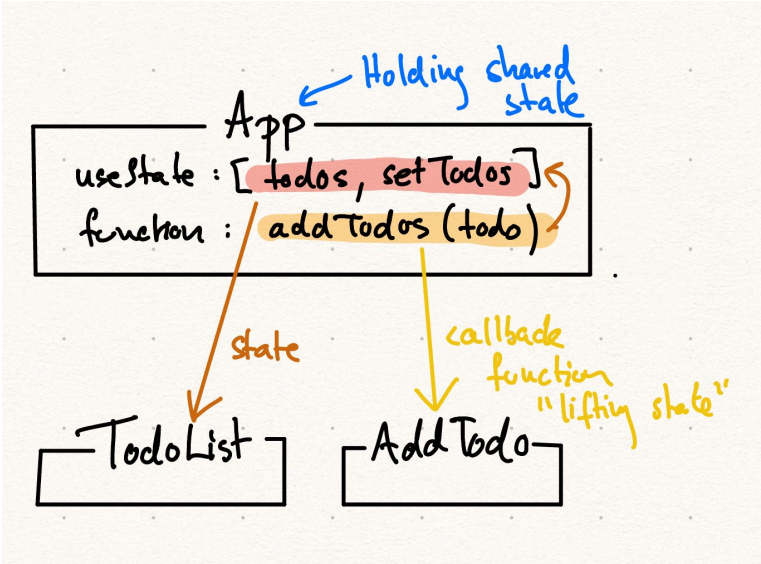
AFTER!

sample.tsx

```tsx
function Child({onAdd}){
  return <button onClick={onAdd}>Add</button>
}
export default function App(){
  const [n,setN]=React.useState(0);
  return <><p>{n}</p><Child onAdd={()⇒setN(x⇒x+1)}/></>
}
```

Pattern III: Simple Shared State Patterns

- State is shared via common parents or wrapper components.
- Context can reduce prop drilling for global-like data.
- Suitable for themes, auth state, or configuration.
- Avoids premature use of state libraries.
- Complexity grows if overused or poorly scoped.



Before...

sample.tsx

```tsx
function Deep({count}){return <p>{count}</p>}
function Mid({count}){return <Deep count={count}/>}
export default function App(){
  const [count]=React.useState(1);
  return <Mid count={count}/>
}
```

AFTER!

sample.tsx

```tsx
const Ctx=React.createContext(0);
function Deep(){
  const count=React.useContext(Ctx);
  return <p>{count}</p>
}
function Mid(){return <Deep/>}
export default function App(){
  const [count]=React.useState(1);
  return <Ctx.Provider value={count}><Mid/></Ctx.Provider>
}
```

Context API

- Context API provides a way to share state without prop drilling.
- Data is supplied via a `Provider` and consumed by descendant components.
- Best suited for global or app-wide concerns like theme or auth state, (updates trigger re-renders in all consuming components.)
- `useContext` allows function components to directly consume context values without wrappers or render props.

sample.tsx

```tsx
const ThemeCtx = createContext("light");

function Button() {
  const theme = useContext(ThemeCtx);
  return <button className={theme}>Click</button>;
}

export default function App() {
  return (
    <ThemeCtx.Provider value="dark">
      <Button />
    </ThemeCtx.Provider>
  );
}
```

No more prop drilling!

Class Activity Time!

What issues are caused by overusing the Context API?

How do other frameworks manage global states?

Redux!

- Redux is a predictable state management library for React applications.
- Application state is stored in a single global store.
- State updates occur through dispatched actions and pure reducers.
- Enables centralized debugging and time-travel inspection.
- Best suited for large applications with complex shared state.

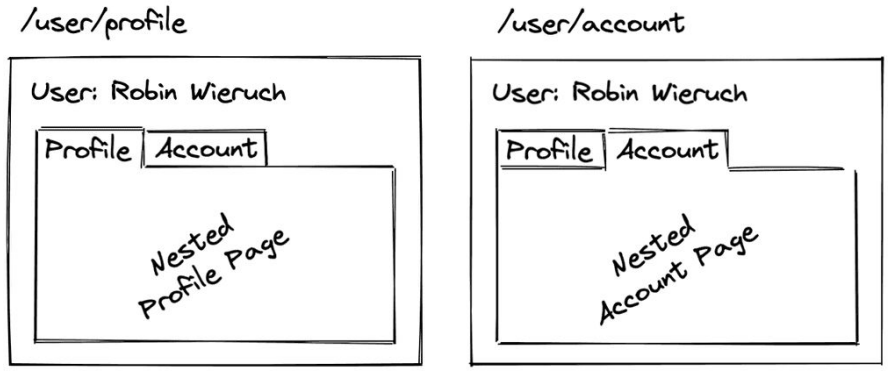
sample.tsx

```tsx
import { createStore } from "redux";

const initialState = { count: 0 };

function reducer(state = initialState, action) {
  if (action.type === "inc") {
    return { count: state.count + 1 };
  }
  return state;
}

const store = createStore(reducer);

store.dispatch({ type: "inc" });
console.log(store.getState());
```

State management made easy!

React Router

- React Router enables client-side routing in single-page applications.
- It maps URLs to components without full page reloads.
- Routing is declarative and defined in the component tree.
- Navigation updates the URL and re-renders matching routes.
- Browser history is managed via the History API.



react-router-dom

- Provided by the `react-router-dom` package for web applications.
- Core components include `BrowserRouter`, `Routes`, and `Route`.
- Hooks like `useParams`, `useNavigate`, and `useLocation` simplify routing logic.
- Supports nested routes and dynamic URL segments.
- Enables route-based code splitting and layout composition.

Example

sample.tsx

```tsx
import { BrowserRouter, Routes, Route, Link } from "react-router-dom";

function Home() {
  return <h2>Home</h2>;
}

function User() {
  return <h2>User Page</h2>;
}

export default function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link>
        <Link to="/user">User</Link>
      </nav>

      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/user" element={<User />} />
      </Routes>
    </BrowserRouter>
  );
}
```

Project Structure

- Components folder structure organizes UI into small, focused, reusable units.
- Pages represent route-level components and compose multiple UI components.
- Reusable UI components live in `/components` and avoid route-specific logic.
- Custom logic and shared behavior are extracted into `/hooks`.
- API calls and external integrations are centralized in `/services`.
- Static files such as images, icons, and fonts are stored in `/assets`.

Styling in React

- Styles can be applied using CSS, CSS Modules, or CSS-in-JS solutions.
- Styling is scoped to components to avoid global conflicts.
- Dynamic styles are derived from state and props.
- Separation of structure and presentation remains flexible.
- Styling choices impact maintainability and scalability.

TailwindCSS

- TailwindCSS is a utility-first CSS framework.
- Styles are applied directly in JSX via predefined class names.
- Eliminates the need for custom CSS files for most layouts.
- Enables consistent design through a shared utility system.
- Encourages rapid UI development with minimal context switching.

Class Activity Time!

Compare: Styling with Traditional CSS vs. TailwindCSS

sample.tsx

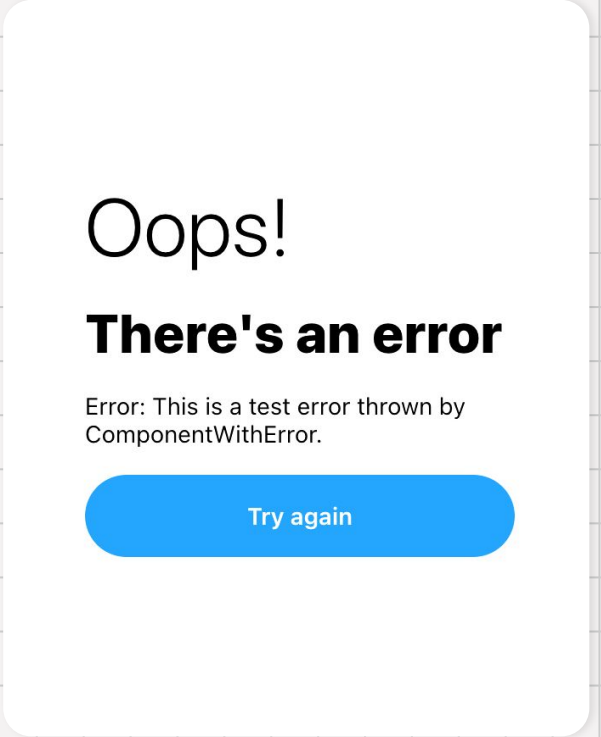
```tsx
export default function Button() {
  return <button className="btn">Click</button>;
}

/* CSS (same file for illustration) */
const styles = `
.btn {
  padding: 8px 12px;
  background: #2563eb;
  color: white;
}
`;
```

sample.tsx

```tsx
export default function Button() {
  return (
    <button className="px-3 py-2 bg-blue-600 text-white">
      Click
    </button>
  );
}
```

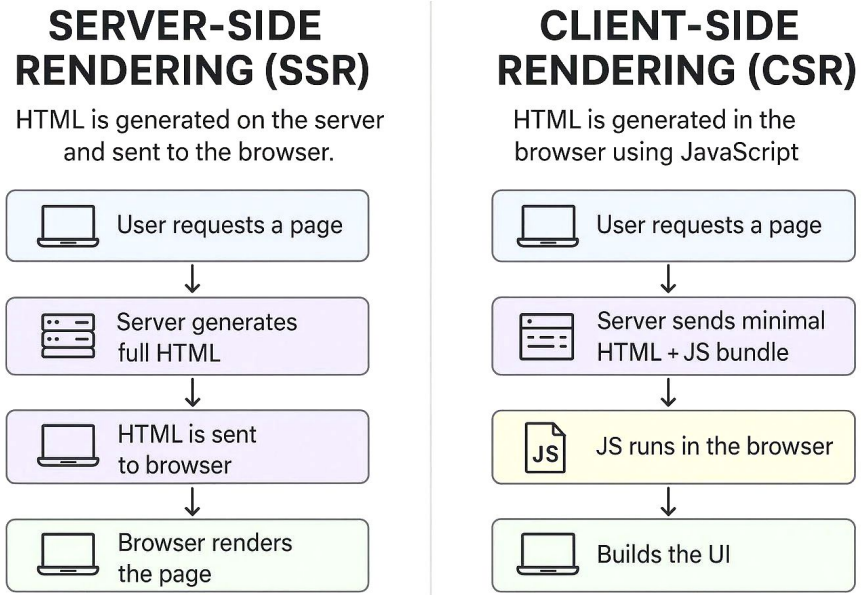
Error Handling

- Errors can occur during rendering, data fetching, or user interactions.
- Runtime errors should be anticipated and handled gracefully.
- Error boundaries catch render-time errors in the component tree.
- Async errors are handled explicitly in effects and event handlers.
- Proper error handling improves resilience and user experience.



What is SSR?

- Rendering occurs on the server instead of exclusively in the browser.
- The server sends fully rendered HTML to the client.
- Improves initial load performance and perceived speed.
- Enhances SEO by exposing content to crawlers immediately.
- Requires hydration to attach React interactivity on the client.



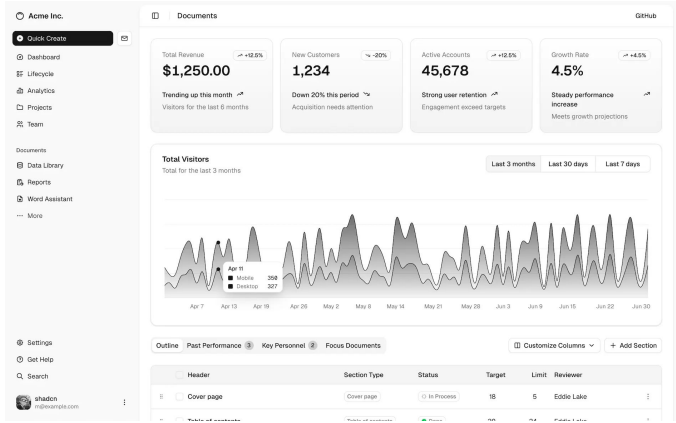
Source: Reddit

Next.js

- Next.js is a React framework with built-in SSR support.
- Enables multiple rendering strategies: SSR, SSG, and CSR.
- File-based routing simplifies application structure.
- Provides built-in optimizations for performance and SEO.
- Commonly used for production-grade React applications.

ShadCN

- shadcn/ui is a collection of reusable, accessible React components.
- Components are copied into the project, not installed as a dependency.
- Built on top of Radix UI and TailwindCSS.
- Fully customizable and owned by the application codebase.
  `
- Encourages consistency without locking into a component library.



READ MORE!

TanStack Ecosystem

- TanStack is a collection of headless, framework-agnostic UI/data libraries.
- Libraries are designed to be composable, typed, and production-focused.
- TanStack tools target common app infrastructure problems at scale.
- Many packages support React plus other frameworks.
- TanStack Query is the most common entry point for React apps.

Source: GitHub

TanStack Query

- TanStack Query manages remote data that can change outside your UI.
- It standardizes loading, error, caching, and refetching behavior.
- It reduces boilerplate compared to manual `useEffect` & `useState`.
- It prevents duplicate requests through request deduplication.
- It improves perceived performance via caching and background updates.

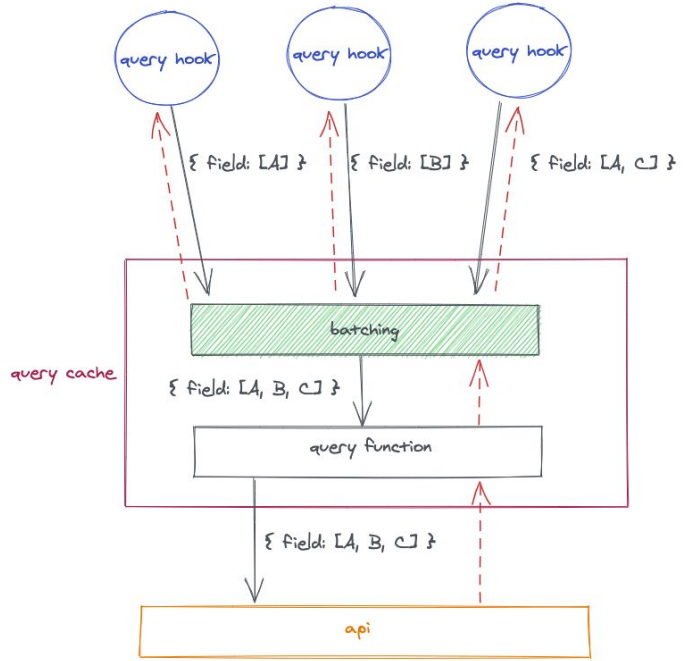
sample.tsx

```tsx
import { useQuery } from "@tanstack/react-query";

function Users() {
  const { data, isLoading } = useQuery({
    queryKey: ["users"],
    queryFn: () => fetch("/api/users").then(r => r.json())
  });

  if (isLoading) return <p>Loading…</p>;
  return <pre>{JSON.stringify(data, null, 2)}</pre>;
}
```

Notice how it solves many of your concerns at once!

Queries in TanStack

- Queries fetch data and store it in a cache keyed by a queryKey.
- Components subscribe to cached results and re-render on updates.
- Refetching can happen automatically on focus, reconnect, or intervals.
- Stale data policies control when cached data should be refreshed.
- Query keys enable scoping, sharing, and invalidation strategies.



Mutations

- Mutations handle create/update/delete operations against the server.
- Success handlers typically invalidate related queries to refresh cached data.
- Optimistic updates can update the UI before the server responds.
- Rollbacks handle failures to keep UI and cache consistent.
- Mutation lifecycles help coordinate side effects cleanly.

sample.tsx

```tsx
import { useMutation, useQueryClient } from "@tanstack/react-query";

function AddUser() {
  const qc = useQueryClient();

  const mutation = useMutation({
    mutationFn: (user) =>
      fetch("/api/users", {
        method: "POST",
        body: JSON.stringify(user),
      }),
    onSuccess: () => qc.invalidateQueries(["users"]),
  });

  return <button onClick={() => mutation.mutate({ name: "New" })}>Add</button>;
}
```

Mutations happen just when you need them!

TanStack Best Practices

- Treat TanStack Query as the default for API state, not UI state.
- Normalize query keys and keep them consistent across the app.
- Prefer invalidation over manual refetch wiring for correctness.
- Use select/transform patterns to keep components minimal.
- Combine with lightweight client state (Context/Zustand) when needed.