some golden React rules

Project Structure & Architecture (1–10)

Organize code by feature, not by file type

- Use a clear folder structure (`components/`, `hooks/`, `utils/`, `pages/`, `services/`)
- Keep components small and reusable
- Follow Separation of Concerns — logic and UI should be separate
- Use a state management pattern early (e.g., Context, Redux, Zustand)
- Keep global state minimal — prefer local state
- Use barrel files (`index.js`) to simplify imports
- Avoid deeply nested folders
- Use layout components for page structure reuse
- Use types with TypeScript or JSDoc if not using TS

Follow Separation of Concerns(SoC) — logic and UI should be separate

It means separating different responsibilities of your component —

Concern	Description
Logic	State handling, API calls, data processing
Presentation	UI markup, styling, displaying data

Instead of mixing logic and UI in one big chunk, you split them up— usually by creating:

- *Custom hooks* to handle logic
- *Presentational components* to handle rendering

Example Without Separation (Mixed UI + Logic):

```
function UserList() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  useEffect(() => {
    fetch("/api/users")
      .then((res) => res.json())
      .then((data) => {
        setUsers(data);
        setLoading(false);
      });
```

```
  }, []);
  if (loading) return <p>Loading ...</p>;
  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

Everything (API + state + UI) is **mixed in one component** — works fine, but hard to scale.

Example With Separation of Concerns

`useUsers.js` (Logic)

```
import { useEffect, useState } from "react";
export function useUsers() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  useEffect(() => {
    fetch("/api/users")
      .then((res) => res.json())
      .then((data) => {
        setUsers(data);
        setLoading(false);
      });
  }, []);

  return { users, loading };
}
```

`UserList.jsx` (UI)

```
import { useUsers } from "./useUsers";
function UserList() {
  const { users, loading } = useUsers();
  if (loading) return <p>Loading ...</p>;
  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

- Logic is encapsulated in `useUsers()`
- UI is clean and focused
- Easy to reuse `useUsers()` anywhere

Rule: Keep global state minimal — prefer local state
You should only use global state (like Redux, Context, Zustand, etc.) when:

- Multiple components across your app need to share the same data
- The data is truly global (e.g., user auth, theme, language)
  For everything else, keep the state local to the component using `useState`, `useReducer`, etc.
  Why prefer local state?

Why prefer local state?

Local State	× Global State
Easy to debug	Can become hard to trace
Easy to maintain	Can grow messy over time
Encapsulated in a small area	Shared everywhere = risk of bugs
Fewer re-renders (more optimized)	Might re-render whole app

Example:
Don't do this:

```
// Global context just for toggling one button 🙄
const ToggleContext = createContext();
```

Do this instead:

```
function ToggleButton() {
  const [on, setOn] = useState(false);
  return <button onClick={() => setOn(!on)}>{on ? "ON" : "OFF"}</button>;
}
```

What is a Barrel File?

A barrel file is just an `index.js` (or `index.ts`) inside a folder that re-exports everything in that folder — so you can simplify your imports.

```
import Header from './components/Header/Header';
import Footer from './components/Footer/Footer';
import Sidebar from './components/Sidebar/Sidebar';
```

Do this with a barrel:
`components/index.js`

```
export { default as Header } from './Header/Header';
export { default as Footer } from './Footer/Footer';
export { default as Sidebar } from './Sidebar/Sidebar';
```

Now you can import like this:
`import { Header, Footer, Sidebar } from './components';`

Use layout components for page structure reuse

What is a layout component?

A layout component is a wrapper that holds the common structure of your pages — like headers, sidebars, footers, etc.
Instead of repeating the same structure in every page, you create one layout and reuse it across your app.
Without layout component (bad practice):

```
function HomePage() {
  return (
    <>
      <Header />
      <Sidebar />
      <main>Welcome to the Home Page</main>
      <Footer />
    </>
  );
}

function AboutPage() {
  return (
    <>
      <Header />
      <Sidebar />
      <main>About us</main>
      <Footer />
    </>
  );
```

```
}
```

With a layout component:

```
export default function Layout({ children }) {
  return (
    <>
      <Header />
      <Sidebar />
      <main>{children}</main>
      <Footer />
    </>
  );
}
```

`HomePage.jsx`

```
import Layout from './Layout';
function HomePage() {
  return (
    <Layout>
      <h1>Welcome to the Home Page</h1>
    </Layout>
  );
}
```

`AboutPage.jsx`

```
import Layout from './Layout';
function AboutPage() {
  return (
    <Layout>
      <h1>About us</h1>
    </Layout>
  );
}
```

Components Best Practices (11–20)

- Components should be **pure** — avoid side effects in render
- Use **React.memo** for expensive or stable child components
- Use **props** for input, callbacks for output

- Avoid prop drilling — use context or composition when needed
- Name components and props clearly and consistently
- Keep **controlled inputs** (always use `value` + `onChange`)
- Break components down before they hit 100 lines
- Clean up **useEffect** properly (avoid memory leaks)
- Use **PropTypes** or TypeScript to enforce types
- Keep UI components **stateless** where possible

Components should be pure — avoid side effects in render

A pure component:

- Always renders the same output for the same props and state
- Does not cause side effects during rendering
  It's like a math function: same input → same output, every time.

What *not* to do (impure component):

```
function MyComponent({ name }) {
  fetch(`/api/hello?name=${name}`); // ❌ side effect inside render

  return <div>Hello {name}</div>;
}
```

This is **impure** because:

- It makes an API call on every render
- React re-renders often, and now you get multiple unwanted calls

Pure version using `useEffect`:

```
function MyComponent({ name }) {
  useEffect(() => {
    fetch(`/api/hello?name=${name}`);
  }, [name]); // ✅ safe and controlled side effect

  return <div>Hello {name}</div>;
}
```

A Pure Component:

```
function Greeting({ name })
{   return <h1>Hello, {name}!</h1>; }
```

This:
Takes in props

- Returns the same output
- Has no side effects
- Is *pure, predictable, and safe*
  Modifying state directly
  Mistake:

```
function Counter() {
  const [count, setCount] = useState(0);
  const increase = () => {
    count += 1; // ❌ Direct mutation — won't trigger re-render
    console.log(count);
  };
  return <button onClick={increase}>Increment</button>;
}
```

Why it's wrong:

- Directly modifying state (`count += 1`) does not trigger a re-render
- React won't know the state changed, so your UI won't update
  Correct way:

```
const increase = () => {
  setCount(prev => prev + 1); // Triggers a re-render safely
};
```

2. Calling side effects inside render
   Mistake:

```
function Profile({ username }) {
  localStorage.setItem("lastUser", username); // ❌ Side effect in render
  return <p>Welcome back, {username}!</p>;
}
```

Why it's wrong:

- Every time the component re-renders, it writes to `localStorage`
- React's render should be **pure**, but this is doing a side effect
- Can lead to performance issues, infinite loops, or inconsistent state
  Correct way (with `useEffect`):

```
useEffect(() => {
  localStorage.setItem("lastUser", username);
}, [username]);
```

3. Using non-deterministic values in render (`Math.random()`, `Date.now()`)
   Mistake:

```
function RandomKeyButton() {
  return <button key={Math.random()}>Click</button>;
}
//One more
function Timestamp() {
  return <div>{Date.now()}</div>; // ❌ Always changing on re-render
}
```

- Random or time-based values make your component output unpredictable
- Can break key stability, reconciliation, or cause infinite re-renders
  Correct way:
  Use `useMemo`, `useEffect`, or store it in state if needed:

```
const randomKey = useMemo(() => Math.random(), []);
return <button key={randomKey}>Click</button>;
// code
const [timestamp] = useState(() => Date.now());
return <div>{timestamp}</div>;
```

Relying on global variables that can change unexpectedly
Mistake:

```
let theme = "dark";
function Header() {
  return <h1 className={theme === "dark" ? "dark-header" : "light-
header"}>Hi!</h1>;
}
```

Then later in another file:

```
theme = "light";
```

Why it's wrong:

- Global mutable state breaks encapsulation

- You can't track where state changes are coming from
- React won't detect changes unless you re-render manually

```
const ThemeContext = createContext();
function App() {
  const [theme, setTheme] = useState("dark");
  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      <Header />
    </ThemeContext.Provider>
  );
}
```

Now `Header` can consume theme in a predictable, reactive way:

```
const { theme } = useContext(ThemeContext);
```

Use props for input, callbacks for output

Input via `props` (parent passes label)

```
function MyButton({ label }) {
  return <button>{label}</button>;
}
```

Output via callback (parent handles click)

```
function MyButton({ onClick }) {
  return <button onClick={onClick}>Click me</button>;
}
```

Full example (Parent controls everything)

```
function ParentComponent() {
  const handleClick = () => {
    alert('Button clicked!');
  };
  return <MyButton label="Submit" onClick={handleClick} />;
}
function MyButton({ label, onClick }) {
  return <button onClick={onClick}>{label}</button>;
}
```

Rule: **Use props for input, callbacks for output**
In React, components communicate *downward via props (input)* and *upward via callbacks (output).*

What it means:

In React, components communicate **downward via props** (input) and **upward via callbacks** (output)

🔽 Input (Parent → Child)	🔼 Output (Child → Parent)
Props	Callback functions
Data goes down	Events/data go up

Button component

Input via `props` (parent passes label)

```
function MyButton({ label }) {
  return <button>{label}</button>;
}
```

Output via callback (parent handles click)

```
function MyButton({ onClick }) {
  return <button onClick={onClick}>Click me</button>;
}
```

Full example (Parent controls everything)

```
function ParentComponent() {
  const handleClick = () => {
    alert('Button clicked!');
  };
  return <MyButton label="Submit" onClick={handleClick} />;
}
function MyButton({ label, onClick }) {
  return <button onClick={onClick}>{label}</button>;
}
```

× Anti-pattern to avoid:
Mutating props or managing output directly inside child:

```
function MyButton({ label }) {
  label = "Forced value"; // ❌ Don't modify props
  return <button>{label}</button>;
}
//
function MyButton() {
  // ❌ Internal state that parent can't access
  const [clicked, setClicked] = useState(false);
  return <button onClick={() => setClicked(true)}>Click</button>;
}
```

What does “cleaning up useEffect” mean?

When you use `useEffect`, sometimes you **start something** that continues running in the background — like:

- A timer (`setTimeout`, `setInterval`)
- An event listener (`window.addEventListener`)
- A subscription (e.g. WebSocket, Firebase)
- A fetch that updates state after the component unmount
  If you don’t stop or "clean up" these when the component unmounts or updates, it can cause **memory leaks** or **unexpected behavior**.
  How to Clean Up

```
useEffect(() => {
  // Start something
  const id = setInterval(() => {
    console.log("running...");
  }, 1000);
  return () => {
    clearInterval(id);
    console.log("cleaned up");
  };
}, []);
```

WebSocket / Firebase / RxJS Subscriptions

```
useEffect(() => {
  const socket = new WebSocket("wss://example.com");
  socket.onmessage = (msg) => console.log("Received:", msg);
  return () => socket.close(); // ✅ clean up connection
}, []);
```

What happens if you don’t clean up?

- Memory leaks
- State updates on unmounted components (React warning)
- Performance drops
- Multiple listeners stacking up

TL;DR Cleanup Rule

```
useEffect(() => {
  // Set up logic here (e.g. listener, timer, subscription)
  return () => {
    // Clean up here
  };
}, [dependencies]);
```

State & Logic Management (21–30)

- Use `useState` for UI state, `useReducer` for complex logic
- Extract custom logic into **custom hooks**
- Memoize expensive computations using `useMemo`
- Memoize stable functions with `useCallback`
- Avoid unnecessary re-renders — monitor performance
- Always understand why a component re-renders
- Prefer **lifting state up** to the closest common ancestor
- Avoid excessive state nesting — flatten it
- Use `immer` or `useReducer` for immutable updates
- Do not mutate state directly (`state.value = x` ×)

Prefer lifting state up to the closest common ancestor

Prefer lifting state up to the closest common ancestor
so that *shared state* lives where *all components that need it can access and control it.*

What does "Lifting State Up" mean?
Imagine you have two sibling components that need to **share the same state**.
Instead of managing state in one of the siblings, move it up to their common parent — that's called lifting state up.

Siblings need to share selected value

```
function Parent() {
  return (
    <>
      <InputComponent />
      <DisplayComponent />
    </>
```

```
  );
}
// InputComponent stores its own state
function InputComponent() {
  const [text, setText] = useState('');
  return <input value={text} onChange={(e) => setText(e.target.value)} />;
}
// DisplayComponent can't access that text
function DisplayComponent() {
  return <p>Can't see the input here ❌</p>;
}
```

Lifted Up:

```
function Parent() {
  const [text, setText] = useState('');
  return (
    <>
      <InputComponent text={text} setText={setText} />
      <DisplayComponent text={text} />
    </>
  );
}
function InputComponent({ text, setText }) {
  return <input value={text} onChange={(e) => setText(e.target.value)} />;
}
function DisplayComponent({ text }) {
  return <p>You typed: {text}</p>;
}
```

Avoid excessive state nesting — flatten it

one of the most overlooked but powerful tips in writing maintainable React code
Avoid excessive state nesting — flatten it.
Nested state structures look natural at first...
but they're harder to update, harder to track, and lead to more bugs.

Bad: Deeply Nested State

```
const [user, setUser] = useState({
  profile: {
    name: '',
    age: 0,
  },
  settings: {
    theme: 'dark',
    notifications: true,
```

```
  }
});
```

Now if you want to update just the `name`, you need to do:

```
setUser(prev => ({
  ...prev,
  profile: {
    ...prev.profile,
    name: 'John',
  },
}));
```

Better: Flattened State

```
const [name, setName] = useState('');
const [age, setAge] = useState(0);
const [theme, setTheme] = useState('dark');
const [notifications, setNotifications] = useState(true);
```

API & Side Effects (31–40)

- Use custom hooks for API calls (`useFetch`, `useApi`)
- Cancel API calls on unmount with `AbortController`
- Use libraries like React Query, SWR for data fetching
- Always show loading and error states
- Retry failed requests intelligently (exponential backoff)
- Handle all edge cases (empty, loading, failed, etc.)
- Avoid calling APIs directly inside components
- Use `.env` for base URLs, API keys
- Securely handle sensitive data
- Mock APIs for local dev & testing

Testing & Debugging (41–45)

Write unit tests for logic-heavy components/hooks

- Use tools like Jest, React Testing Library
- Use React DevTools, Profiler for performance
- Use `console.log` smartly during development, remove before prod
- Clean up unused variables, dead code, comments

Build & Deployment (46–50)

- Use **ESLint**, **Prettier**, and a code formatter
- Enable **strict mode** in React
- Split bundles for performance (lazy load)
- Avoid unnecessary 3rd-party dependencies
- Optimize for **performance**, **accessibility**, and **SEO**