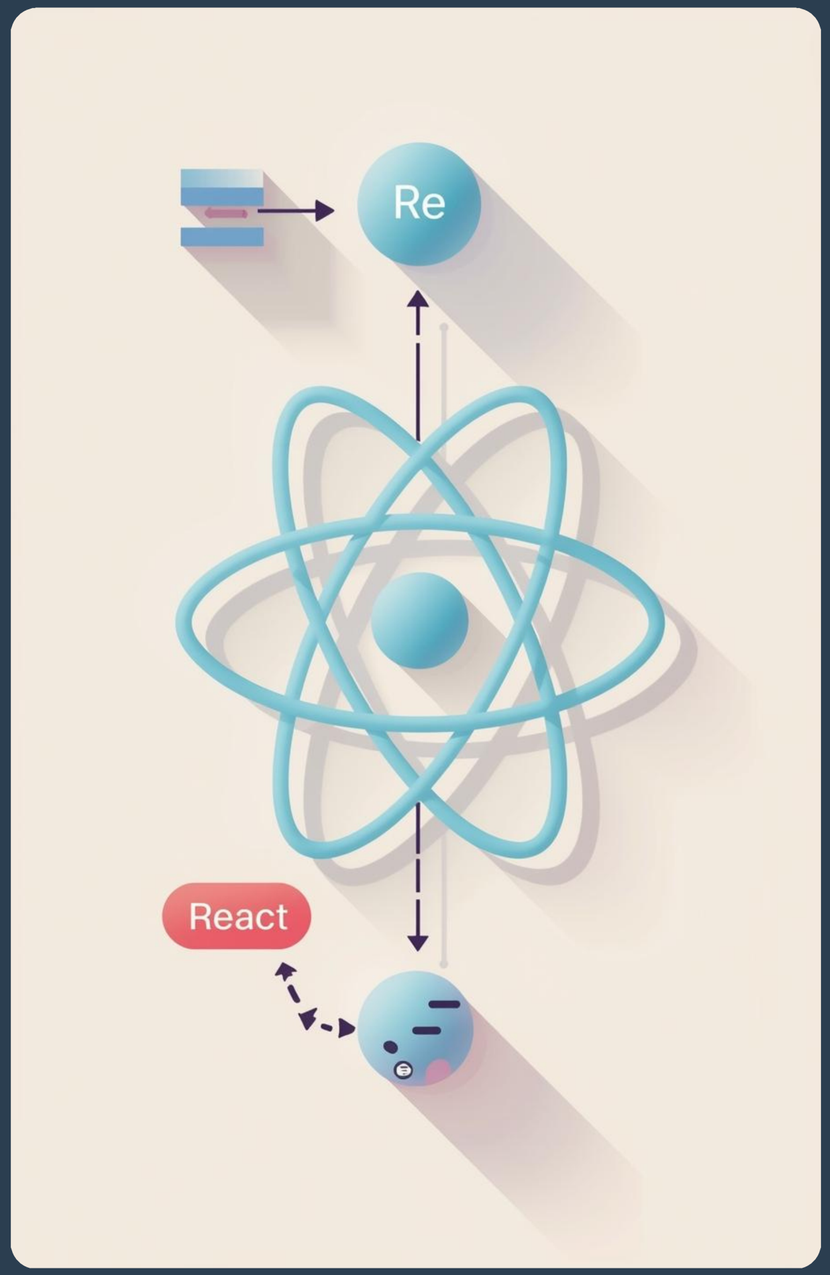
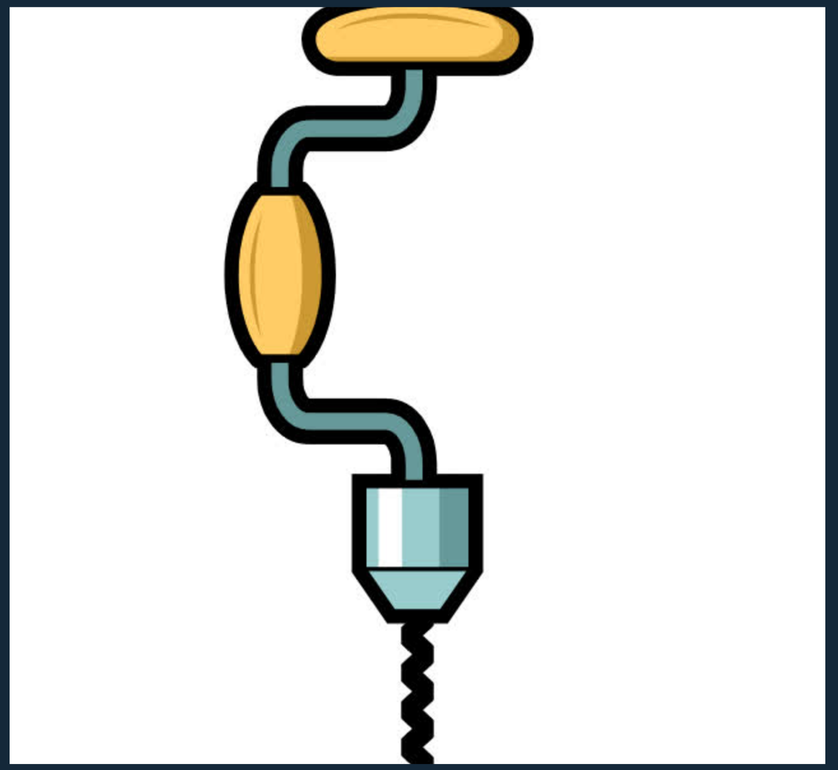
React Design Patterns

Clean Code made Easy!

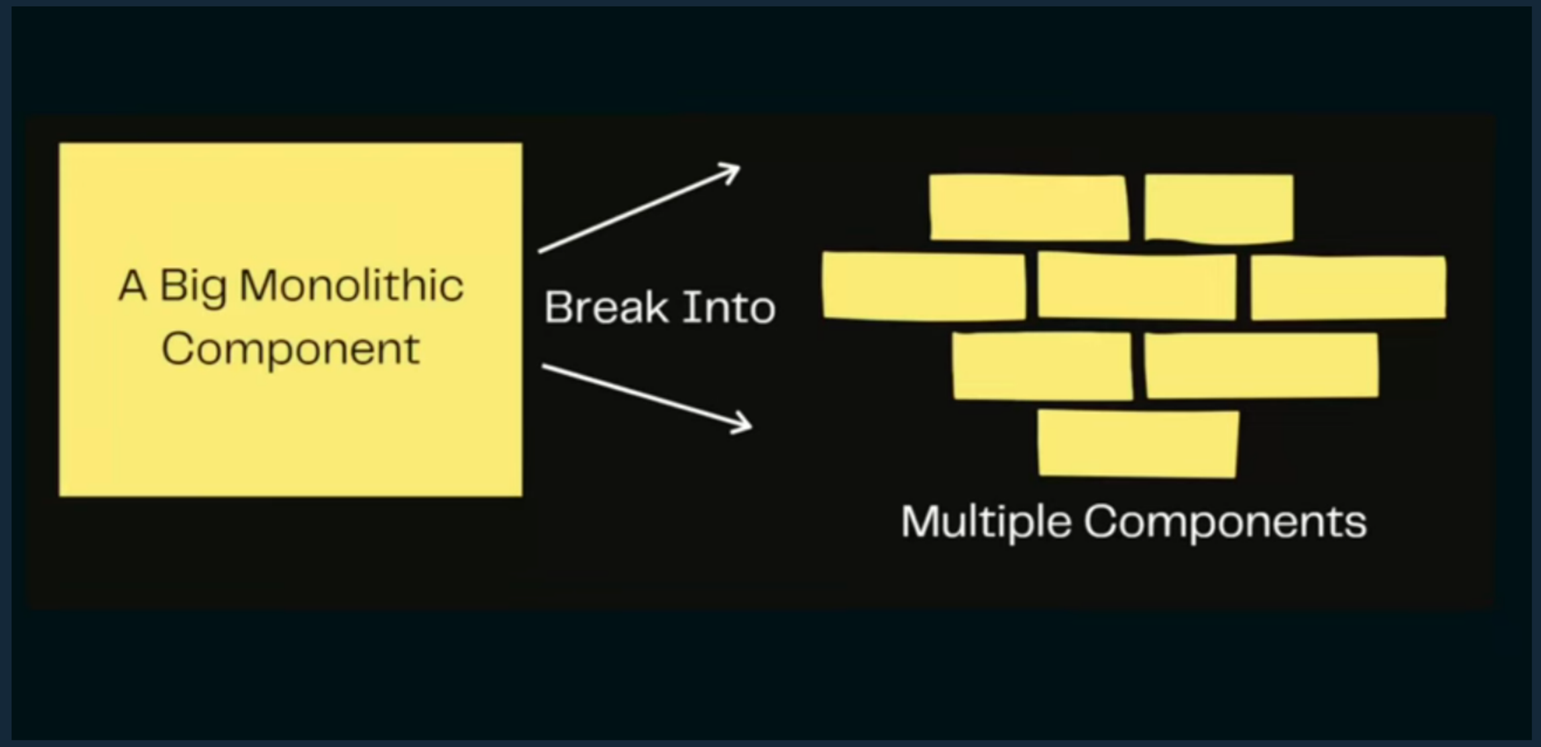


Introduction



What is React ?

React is Declarative and component based JavaScript library for building user interface!



e-results

Search a student by name

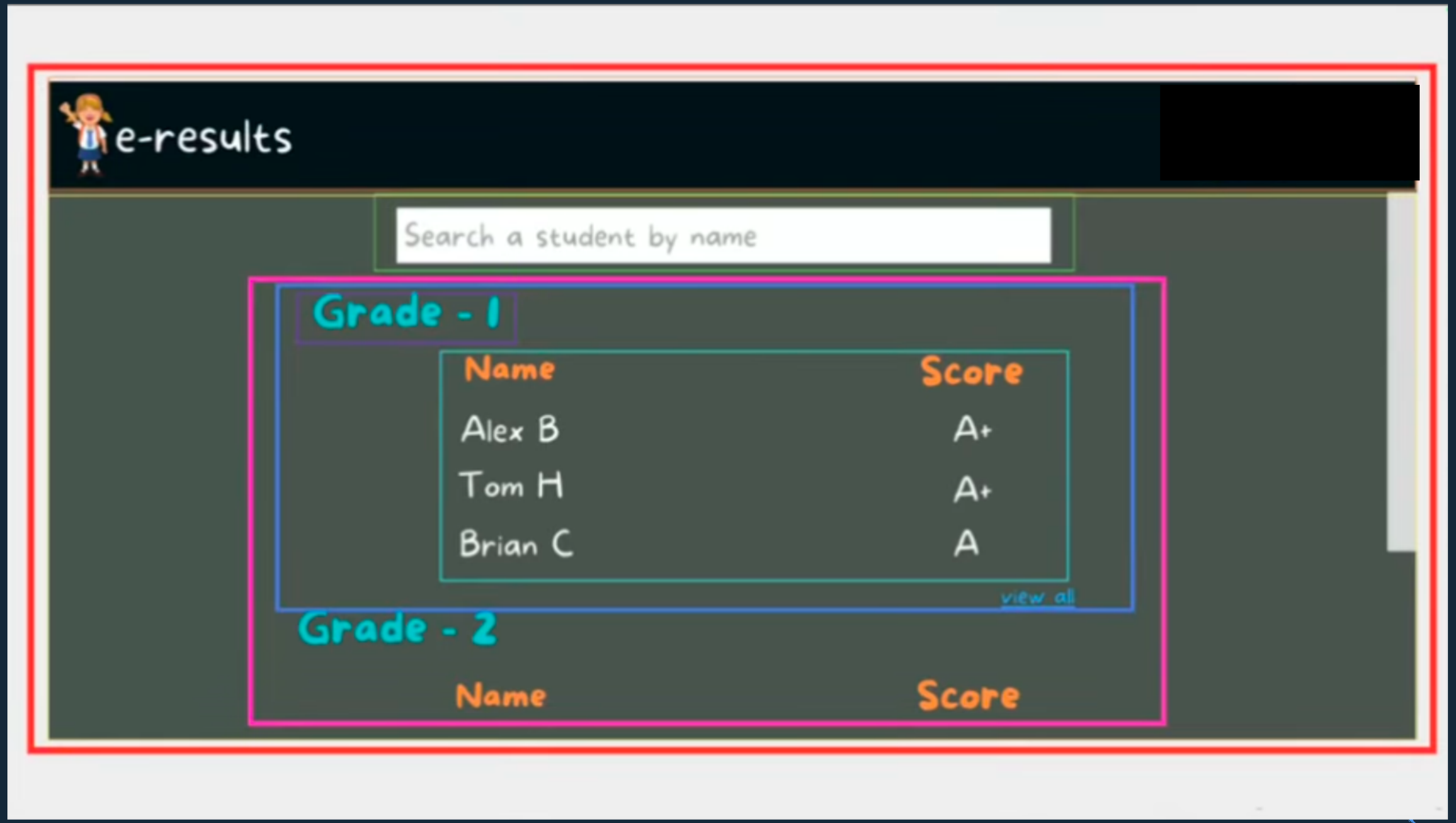
Grade - 1

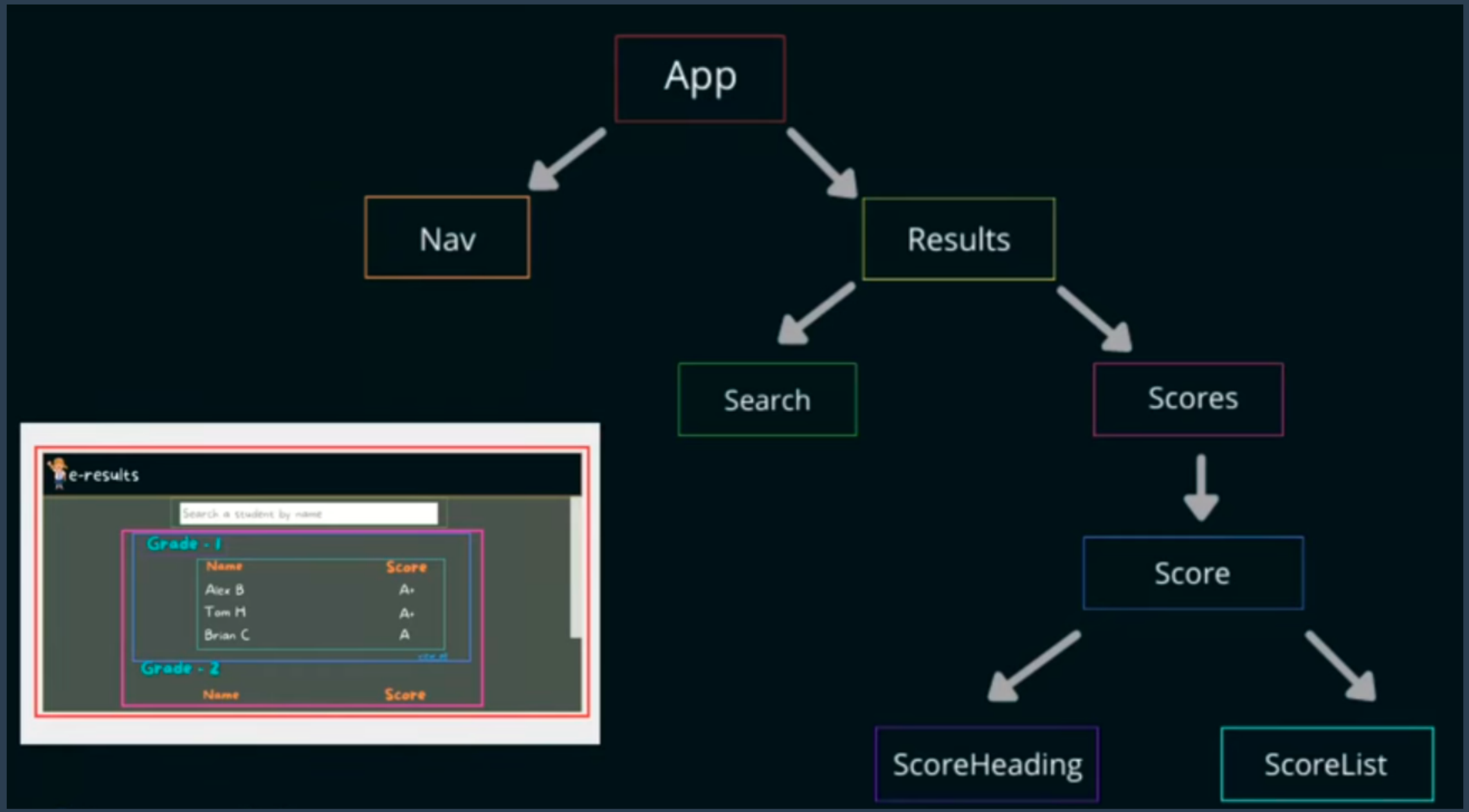
Name	Score
Alex B	A+
Tom H	A+
Brian C	A

view all

Grade - 2

Name	Score


e-results
Search a student by name
Grade - 1
Name
Score
Alex B
A+
Tom H
A+
Brian C
A
view all
Grade - 2
Name
Score


App
Nav
Results
Search
Scores
Score
ScoreHeading
ScoreList
e-results
Search a student by name
Grade - 1
Name
Score
Alex B
A+
Tom H
A+
Brian C
A
Grade - 2
Name
Score

React Design Patterns

Container-Presenter Pattern

Compound Components Pattern

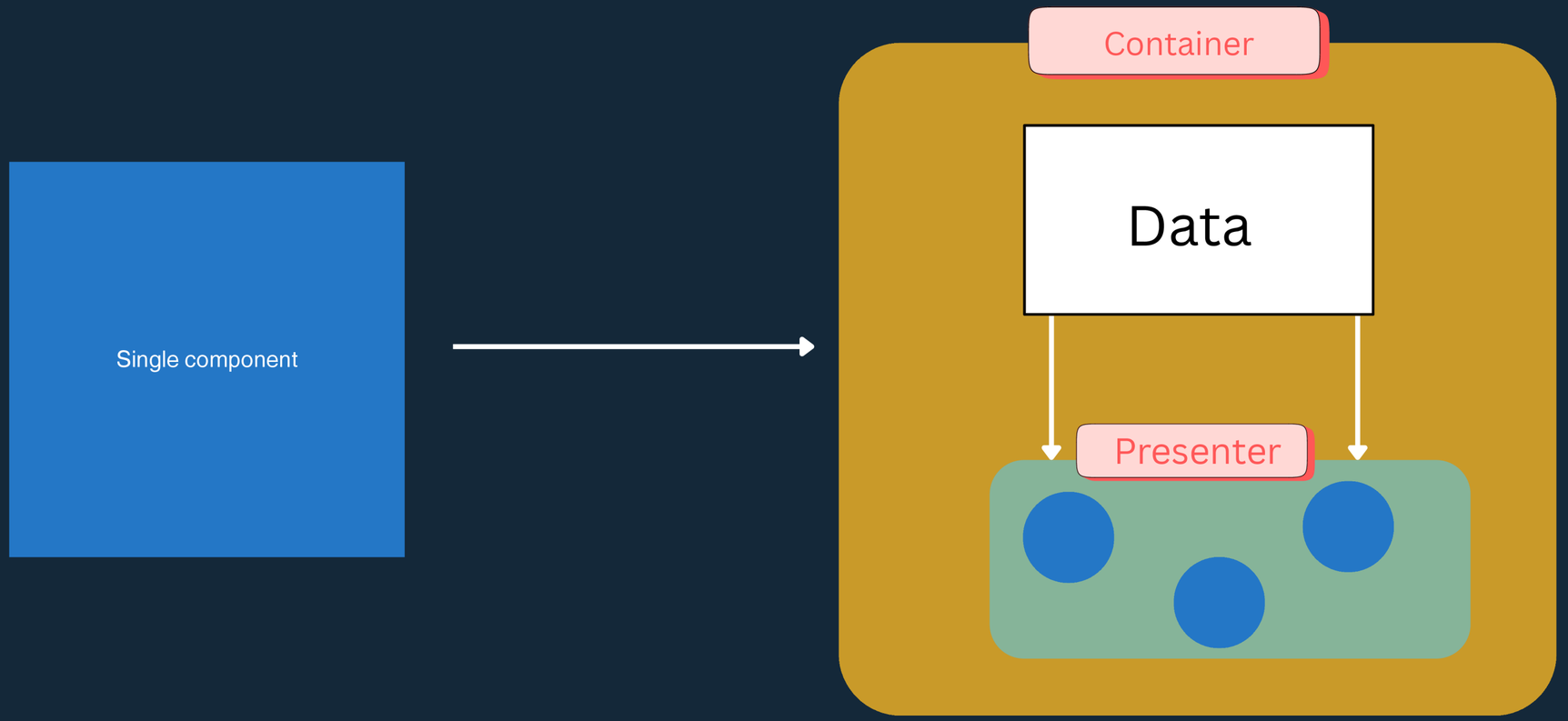
Custom Hooks Pattern

Provider Pattern (Context API)

Container-Presenter Pattern

- How to identify code smells
- How to implement pattern
- Real-World Use Case
- Potential Pitfalls

Container-Presenter Pattern



Use Cases

Use it for data-heavy components, especially when:

- Fetching data from APIs
- Examples: user dashboards, product catalog pages, analytics reports, real-time data displays

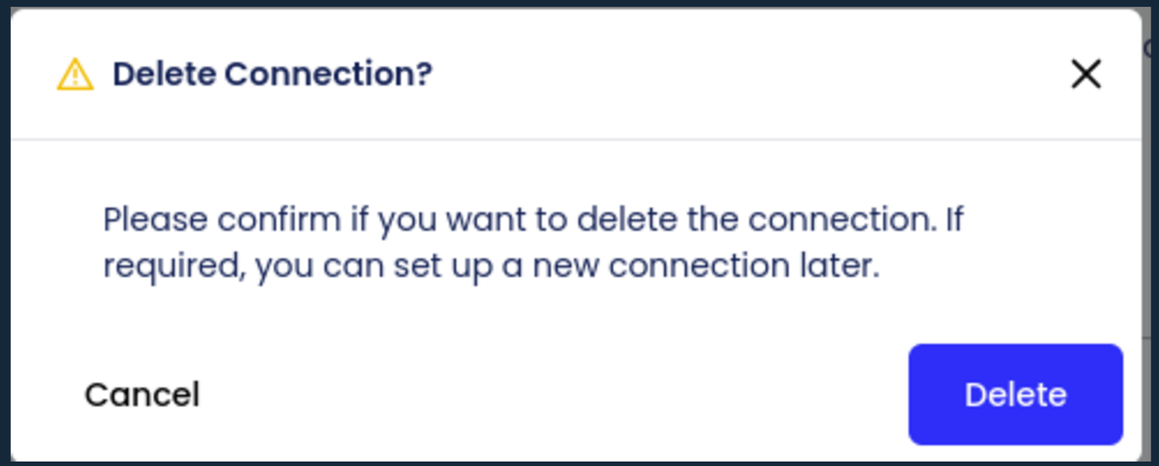
Pitfall

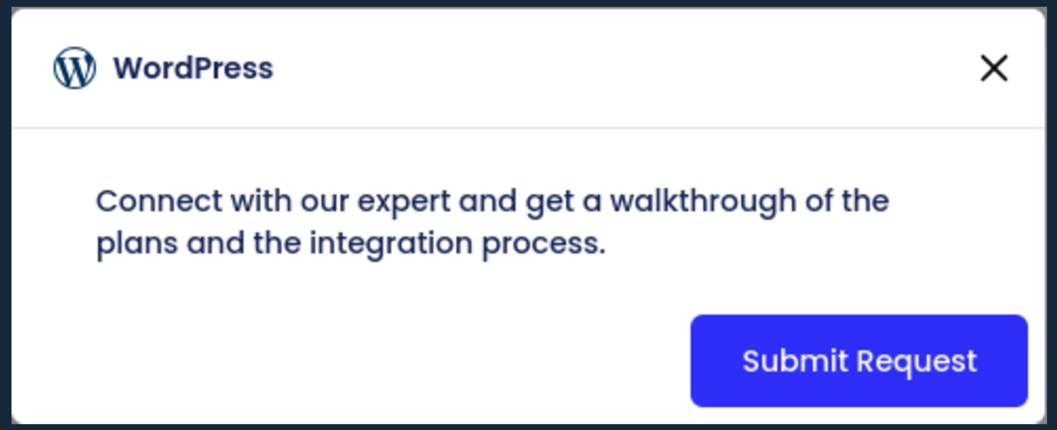
- **Don’t overengineer simple components**
  - Avoid splitting tiny, static pieces (e.g., a spinner or a simple error message) into container + presenter.
- **Watch for prop drilling explosion**
  - If presenters pass many props (10–15+) down multiple layers, the pattern becomes a burden.
  - Excessive prop chains reduce readability and increase maintenance cost.
- **Switch patterns when needed**
  - If prop drilling or complex state arises, consider alternatives: Context, Reducer or other state-management patterns.
  - Use container–presenter as one tool — not the only tool.
- **Rule of thumb**
  - Use the pattern for clear separation of data vs UI (data-heavy or form-heavy cases).
  - Re-evaluate when it creates new problems; evolve the architecture gradually.

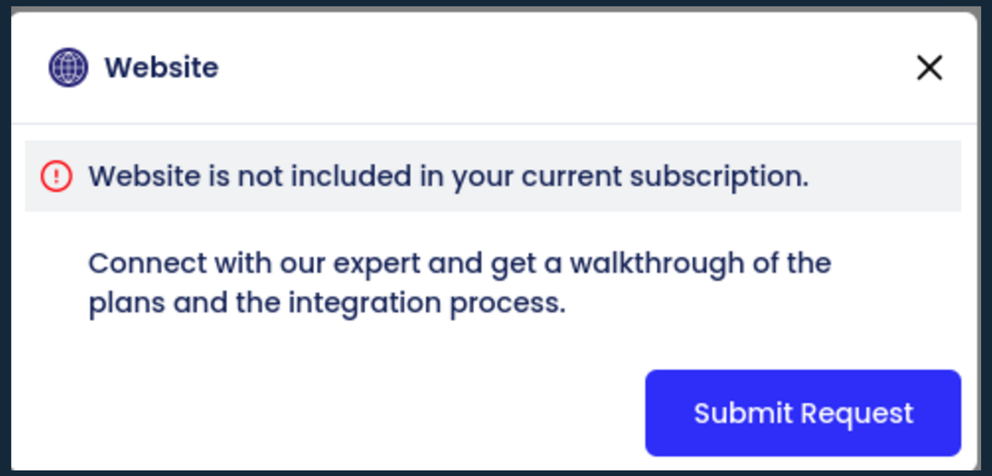
Compound components Pattern

- Bloated Components from prop soup
- How to implement pattern
- Examples
- Use Cases
- Potential Pitfalls

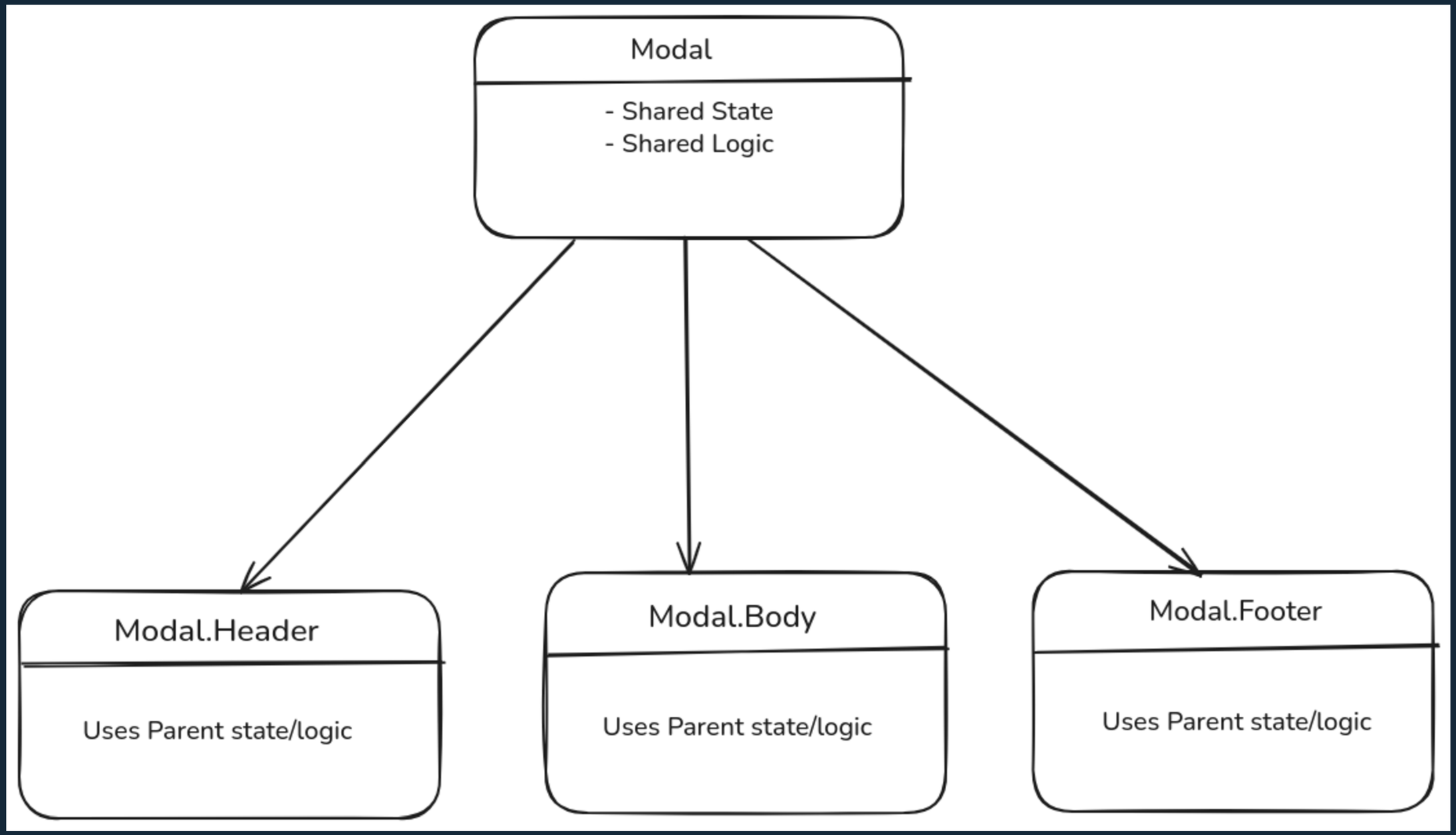
Compound components Pattern







Compound components Pattern


Modal
- Shared State
- Shared Logic
Modal.Header
Uses Parent state/logic
Modal.Body
Uses Parent state/logic
Modal.Footer
Uses Parent state/logic

Use Cases

- Build components where **layout, structure, and nesting matter.**
- Ideal for UI elements like **modals, accordions, tabs, dropdowns, tables**, etc.
- Works well when components have **multiple sub-parts** (header, body, row, item).
- Helps keep components **isolated, flexible, maintainable, and reusable.**

Pitfall

- Don’t use it for very **simple components** (e.g., basic button, spinner, icon).
- **Don’t overuse the pattern**
  - Not every component needs to be a compound component.
  - Apply it only when child structure and nesting truly matter.
- **Avoid re-exporting subcomponents separately**
  - Export only the main component (e.g., Modal).
  - Keep Modal.Header, Modal.Body inside the parent export to prevent misuse.
  - Prevents using ModalHeader or ModalFooter without the actual Modal, which can break future updates.

Custom Hooks

- Why Hooks?
- What is React Hooks?
- Built-in hooks overview
- Rules of hooks
- Custom hooks patterns
- Examples
- Use-Case
- Pitfalls

Why Hooks ?

- Before React 16.8, functional components had no state or side effects
- Logic sharing relied on HOCs and Render Props
- These caused prop drilling, wrapper nesting, and debugging difficulty
- Developer experience became complex and messy
- React introduced Hooks to manage state, side effects, and context in functional components
- Hooks allow clean, reusable logic without wrappers
- Components stay focused on UI + business logic while reusable hooks handle
  - Data fetching
  - Async operations
  - Theming
  - UI behaviors
  - State management
- Hooks simplified code, improved reusability, and provided a better developer experience

What is Hook ?

- A Hook is a special function that lets you hook into React features:
  - State management
  - Side effects (API calls, timers)
  - Context
- Components often do generic tasks:
  - API calls, loading state, error handling, showing data
  - These tasks are reusable across components
- Instead of duplicating logic, use a hook:
  - Manages state, effects, and reusable logic
  - Reduces component complexity
- Hooks vs Utility Functions:
  - Like utilities, but React-aware
  - Integrates with React’s state, side effects, and context
- **Benefit**: Clean, reusable, maintainable components

Map of Hooks

State Management

- useState
- useReducer
- useSyncExternalStore

Context Hooks

- useContext

Transition Hooks

- useTransition
- useDeferredValue

Ref Hooks

- useRef
- useImperativeHandle

Random Hooks

- useDebugValue
- useId

Performance Hooks

- useMemo
- useCallback

Effect Hooks

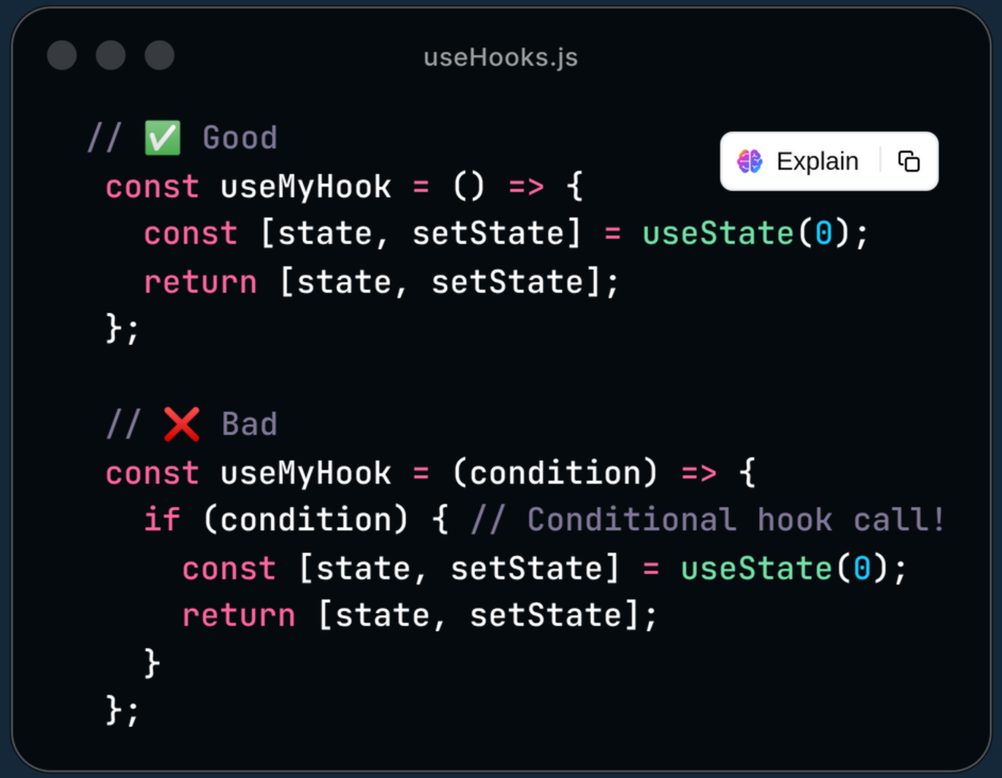
- useEffect
- useLayoutEffect
- useInsertionEffect

React 19 Hooks

- useFormStatus
- useFormState
- useOptimistic
- use

Rules of Hooks ?

- **Rule 1**: Only Call Hooks at the Top Level
  - Never call hooks inside:
    - Loops
    - Conditions (if/else)
    - Nested functions



Rules of Hooks ?

- **Rule 2**: Call Hooks Only from React Functions
  - Hooks can be called from:
    - React functional components
    - Other hooks
  - Not allowed in plain JavaScript functions or utilities
- Why These Rules Exist
  - React relies on the exact order of hooks during each render.
  - React stores hooks in a linked list inside each component’s fiber node
  - Following the rules ensures state and effects are tracked correctly

Custom Hooks

- A custom hook is a JavaScript function whose name starts with use.
- Can call other hooks inside it.
- Think of it like Lego blocks: packaging small interactive logic pieces to build new functionality.
- Purpose:
  - Separate reusable logic from components
  - Reuse the logic across multiple components
- Key Points:
  - Custom hooks return values to the consuming component
  - Unlike components, no JSX — focus on functionality / business logic
  - Components handle UI / presentation, hooks handle logic / state
-

Use Cases

- useLocalStorage
- useDebounce
- useFetch
- useScrollPosition
- useOnlineStatus
- useClipboard
- useInterval

Pitfall

1. Overengineering / Premature Abstraction

- Issue: You might create a hook for logic that is only used once.
- Effect: Adds unnecessary complexity and reduces readability.
- Solution: Only extract to a custom hook if the logic is reused in multiple components.

2. Overuse of State / Effects

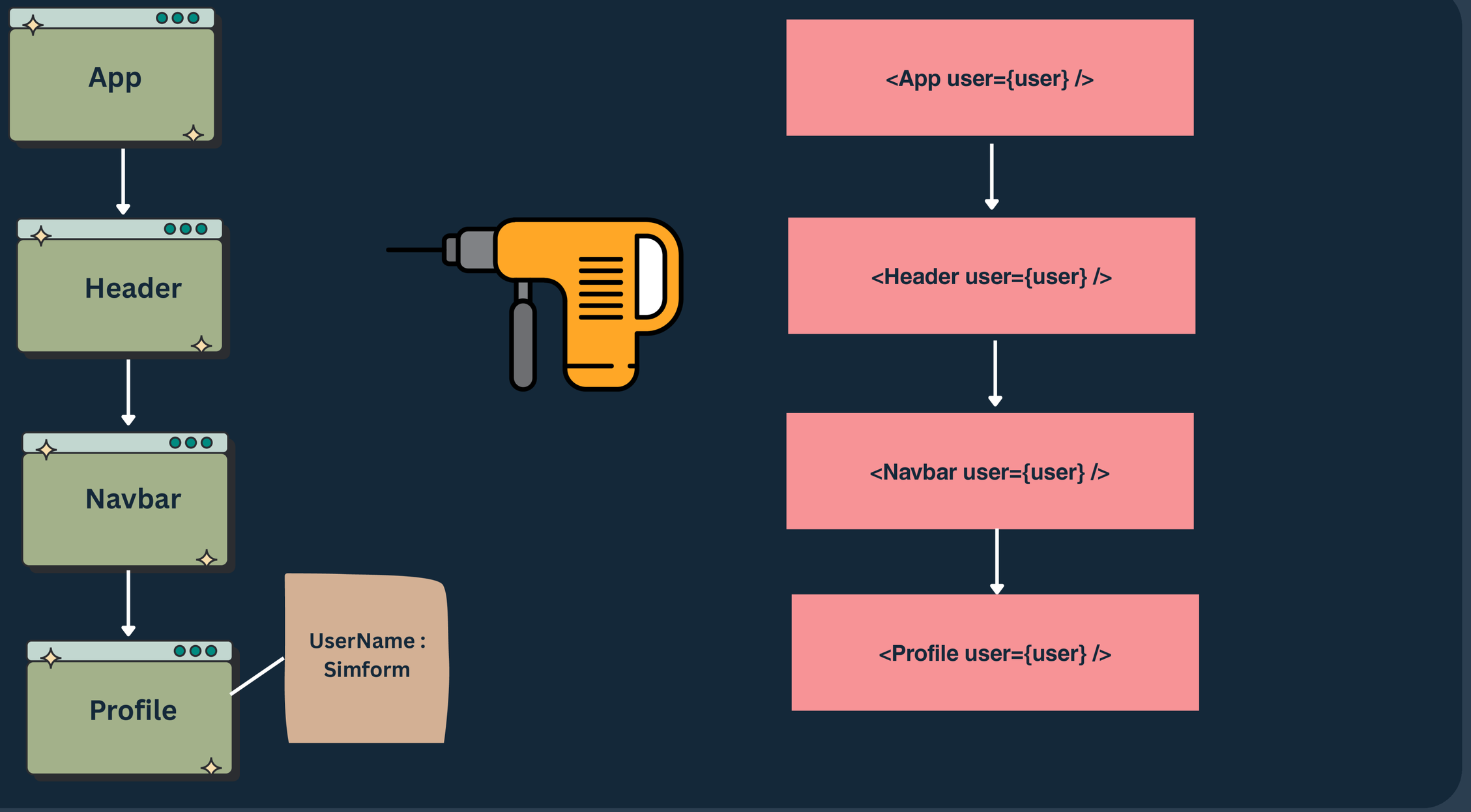
- Issue: Adding unnecessary useState or useEffect in a custom hook.
- Effect: Performance overhead and potential for redundant renders.
- Solution: Only include essential state and side-effects, keep the hook lightweight.

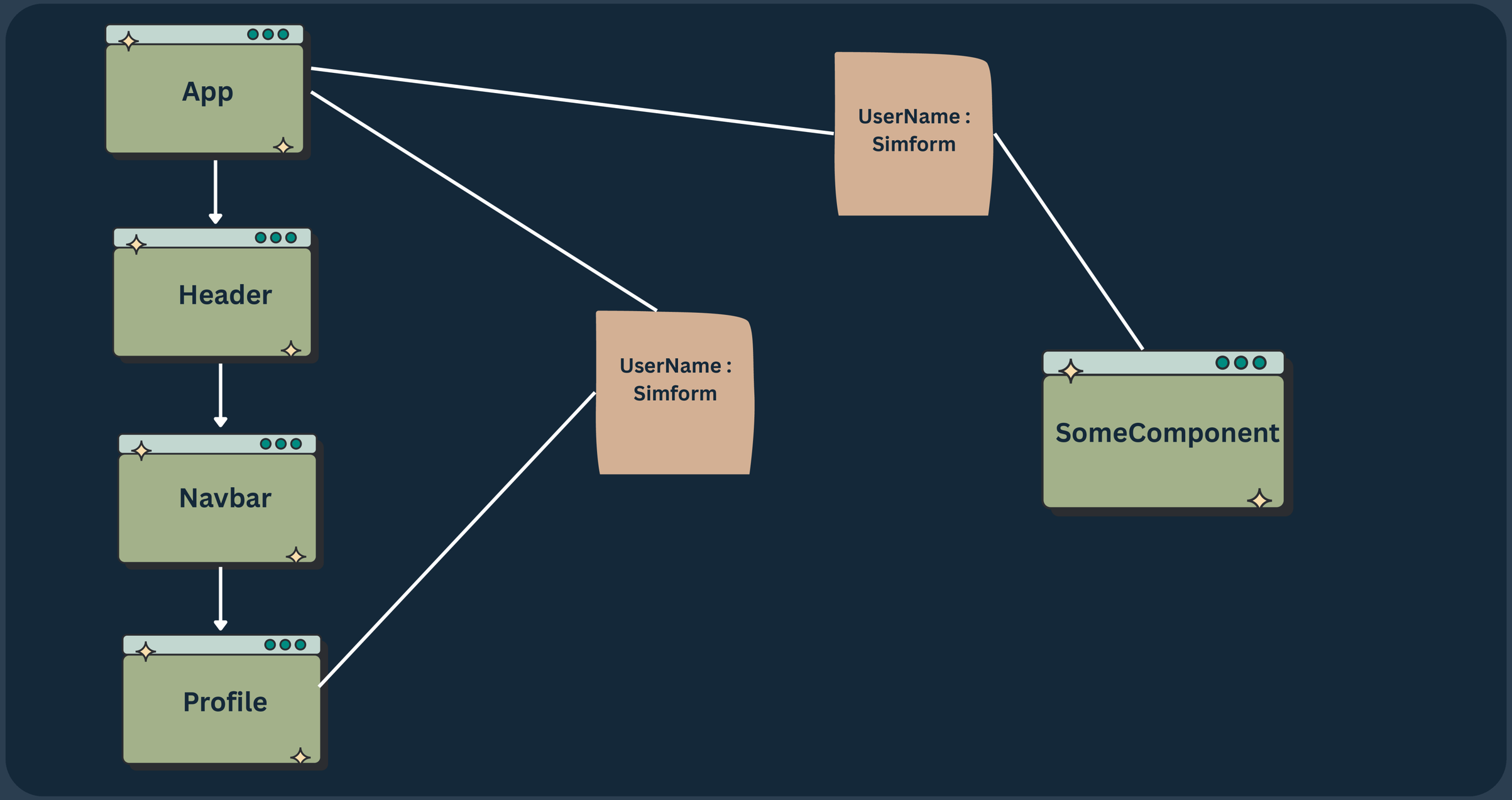
3. Tight Coupling to a Specific Component

- Issue: Hook relies too much on a specific component structure or props.
- Effect: Reduces reusability.
- Solution: Design hooks to be flexible and independent of specific components.

Provider Pattern

- What is Provider Pattern?
- What Problem does it solve ?
- What is Context ?
- Examples
- Multiple Providers and contexts
- Use Cases
- Potential Pitfalls


App
Header
Navbar
Profile
UserName :
Simform
<App user={user} />
<Header user={user} />
<Navbar user={user} />
<Profile user={user} />


App
Header
Navbar
Profile
UserName : Simform
UserName : Simform
SomeComponent

Provider & Context

- **Context:**
  - Defines a boundary in the component tree
  - Makes data or functionality available within that boundary
  - Example: user info, theme, authentication, branding
- **Provider:**
  - The mechanism to supply data inside the context boundary
  - Wraps components that need access to the context
- **Key Concept:**
  - Context = “where” the data is available
  - Provider = “how” the data is provided

Provider & Context

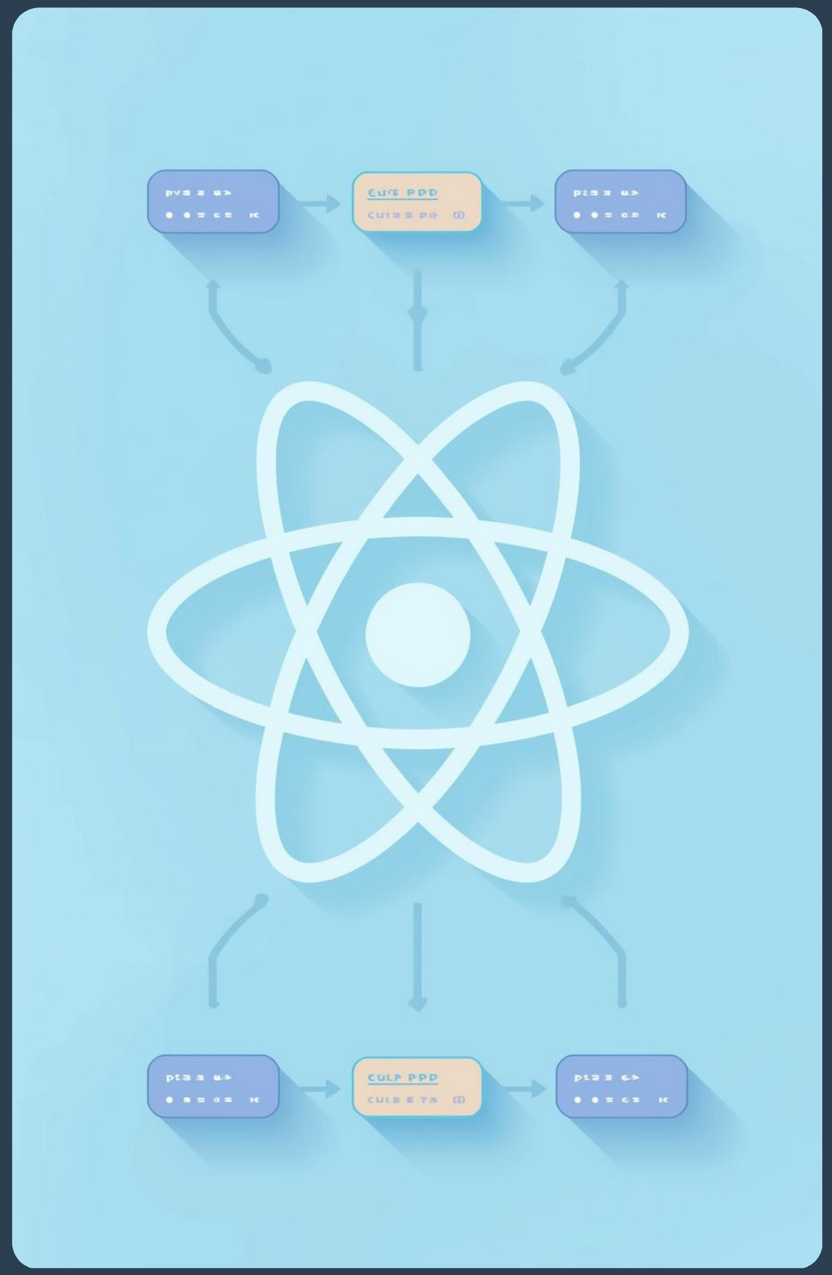
- Steps
  - Create the Context
  - Create Provider
  - Wrap the Component Hierarchy with Provider
  - Create a Hook to make the context available
  - Use the data from the context whenever needed

Use Cases

- **Avoid Prop Drilling:** Share data across deeply nested components.
- **Global State Sharing:** Theme, language, auth user, permissions.
- **App-Wide Services:** API client, analytics, logger, feature toggles.
- **Cross-Cutting UI:** Modals, toasts, layout config, navigation state.
- **Low-Change Shared Data:** Environment config, roles, app settings.

Pitfalls

- Re-render Cascade: All consumers re-render when value changes.
- Overuse of Context: Putting everything in context leads to bloat.
- Hard to Debug: Updates are less obvious than props changes.
- Mixed Responsibilities: Unrelated states inside one context.
- Heavy Setup: Doing async or expensive work inside providers.



- Controlled vs Uncontrolled Components
- State Reducer Pattern
- Observer / Pub-Sub Pattern
- Performance Pattern
- Slot Pattern
- Hook Factory / strategy Pattern
- Facade Pattern
- Error Boundary Pattern
- Suspense Pattern

Thank you!