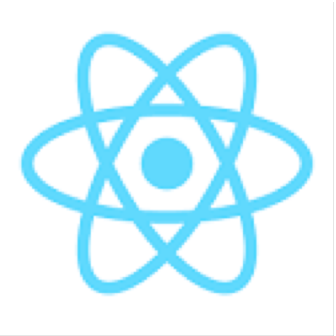


ReactJS.

Thinking in React

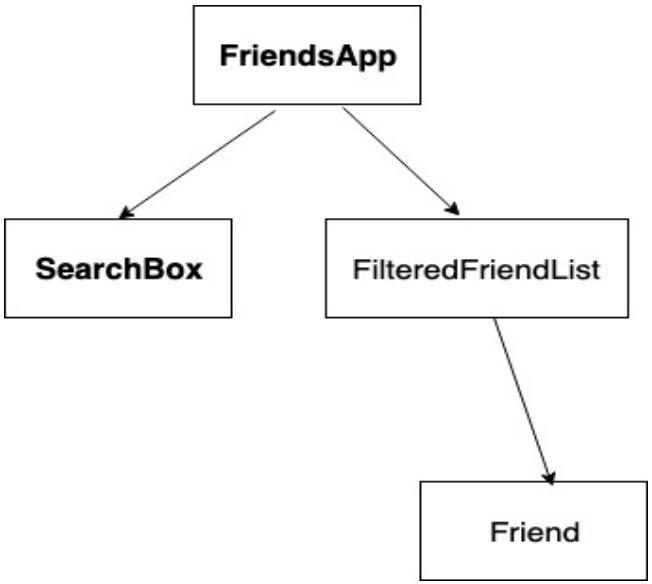
Developing a React web app

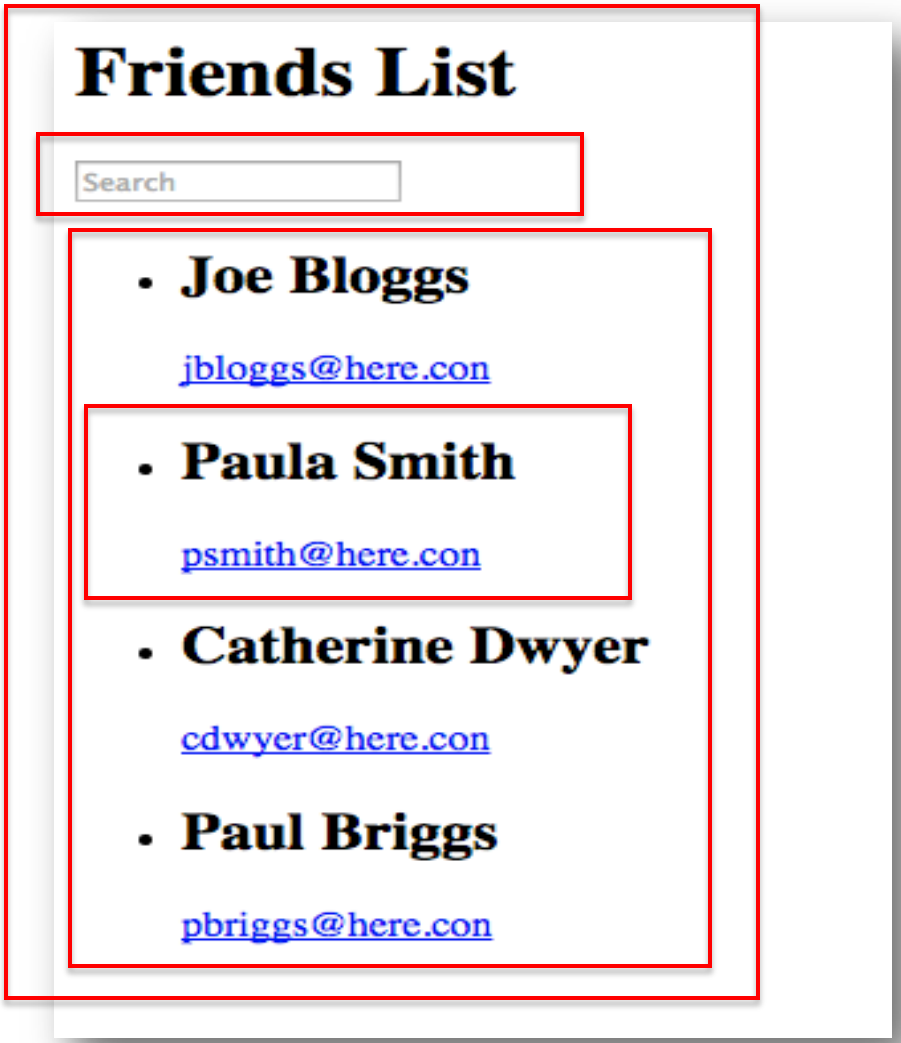
- **Step 1: Break the UI into a** component hierarchy**.**
- **Stap 2: Build a** static version **of the app.**
- **Step 3: Identify the** minimal **representation of** UI state**.**
- **Step 4: Identify where your state should live.**
- **Step 5: Add inverse data flow, if required.**

Starting point.

- **At the start of the development process we have:**
  1. **A mock-up of the UI.**
  2. **(Optionally) A JSON representation of the web API data model.**

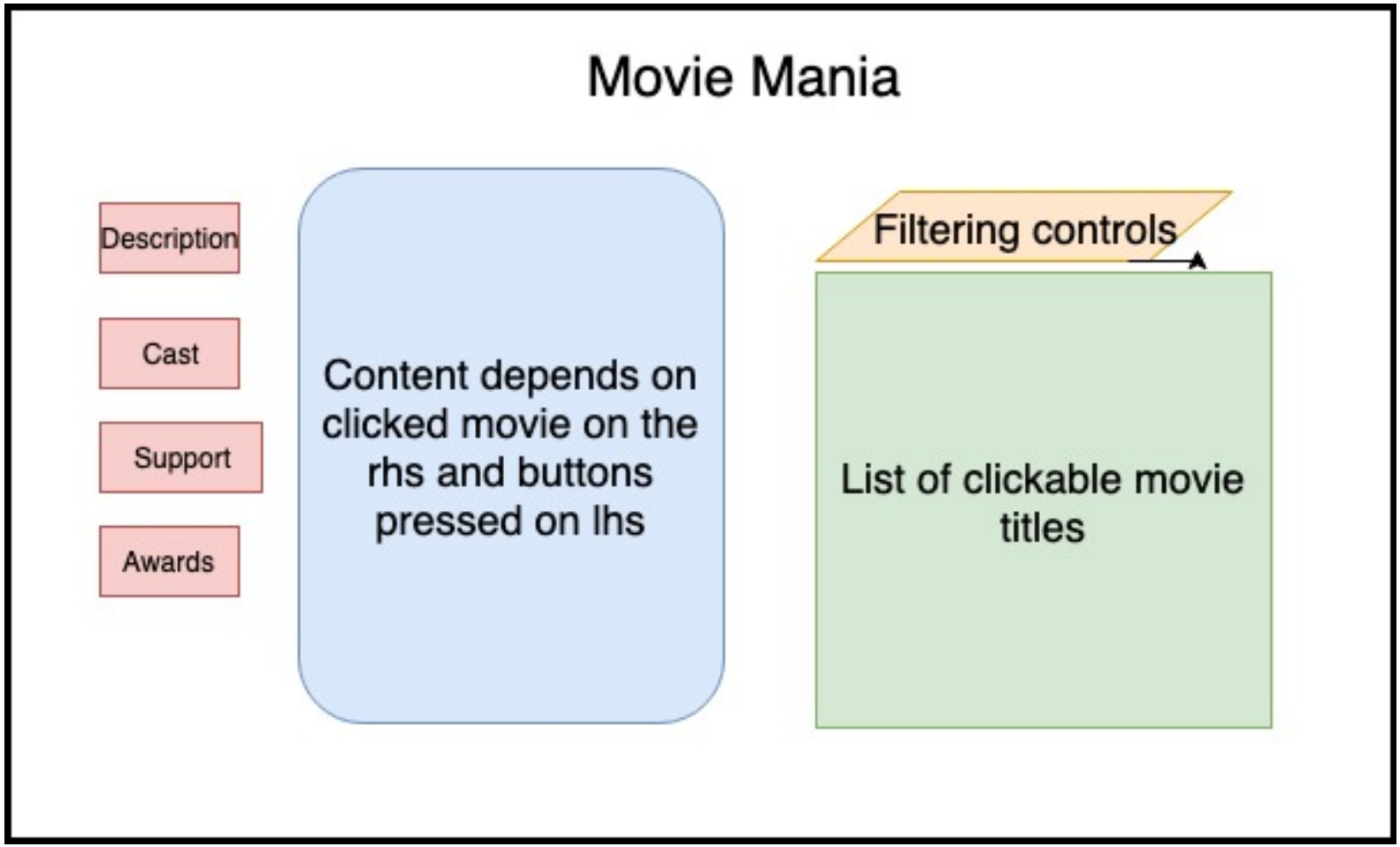
Step 1: Break the UI into a *component hierarchy*.



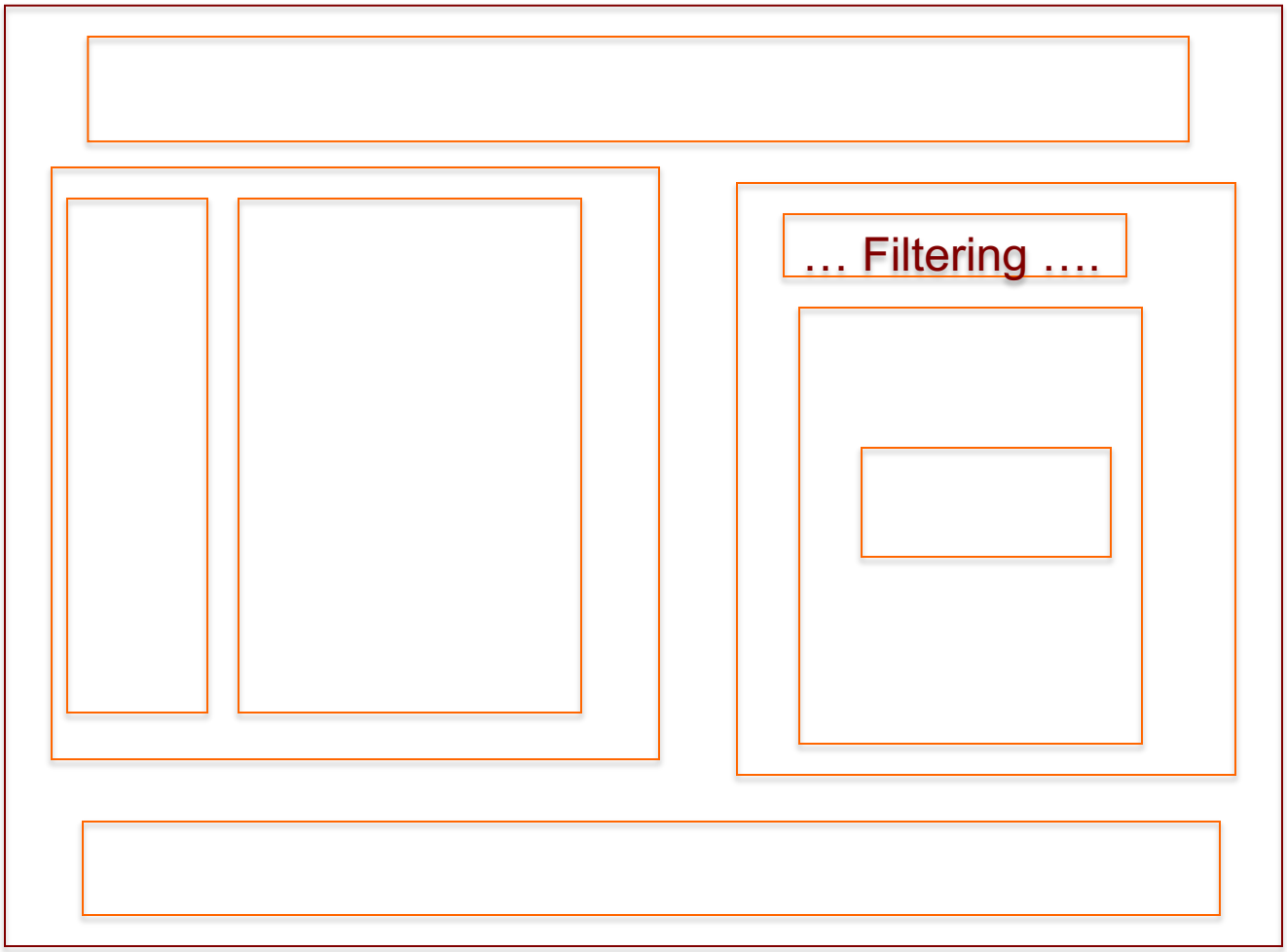


- **Possibly use the data model as a guide.**
  - **The UI and data models tend to adhere to the same information architecture.**

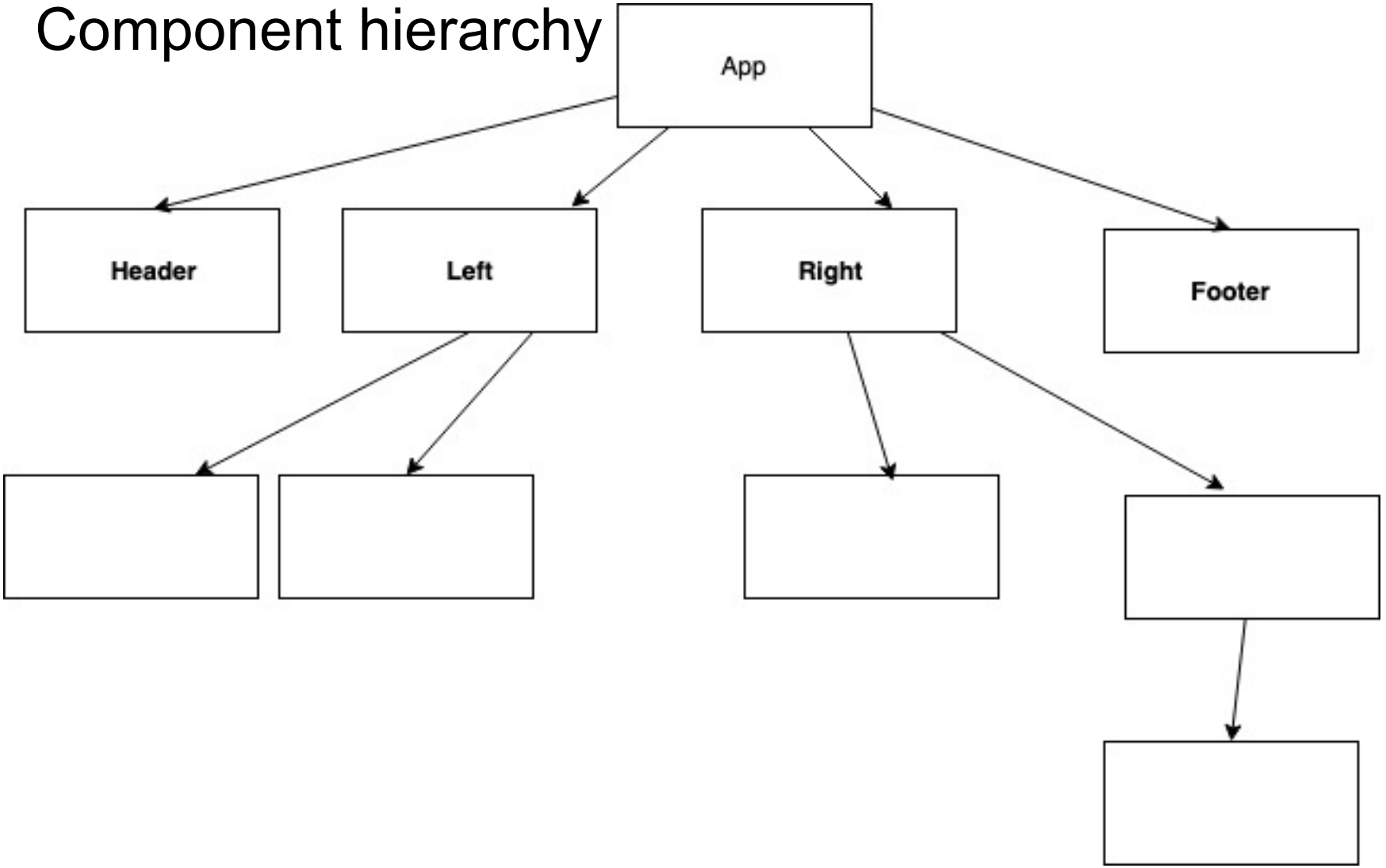
Sample App – Mock UI

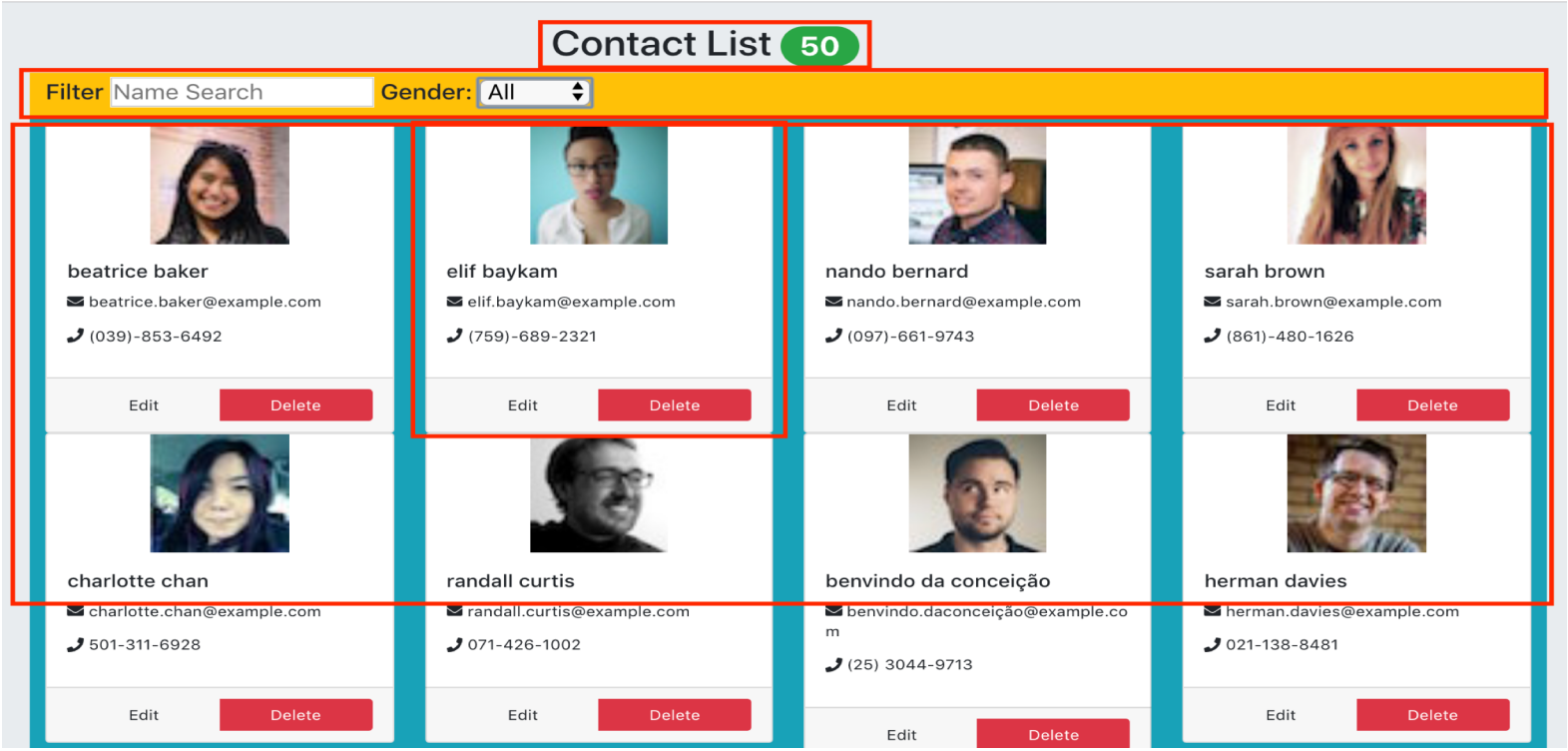


Sample App – Component breakdown

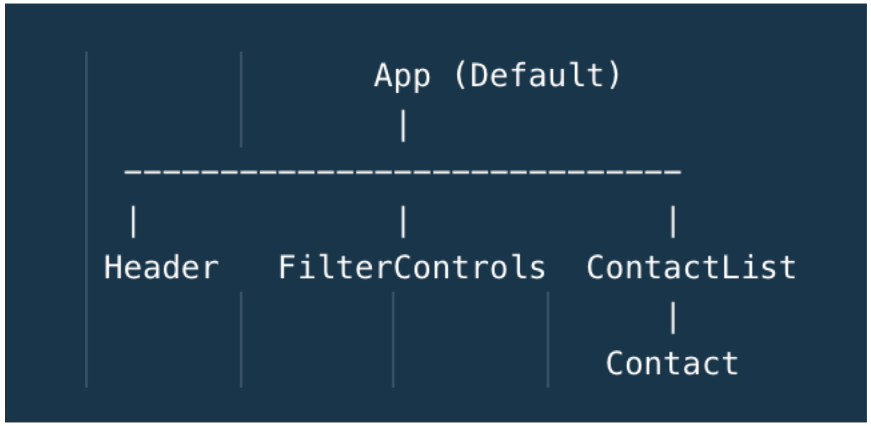


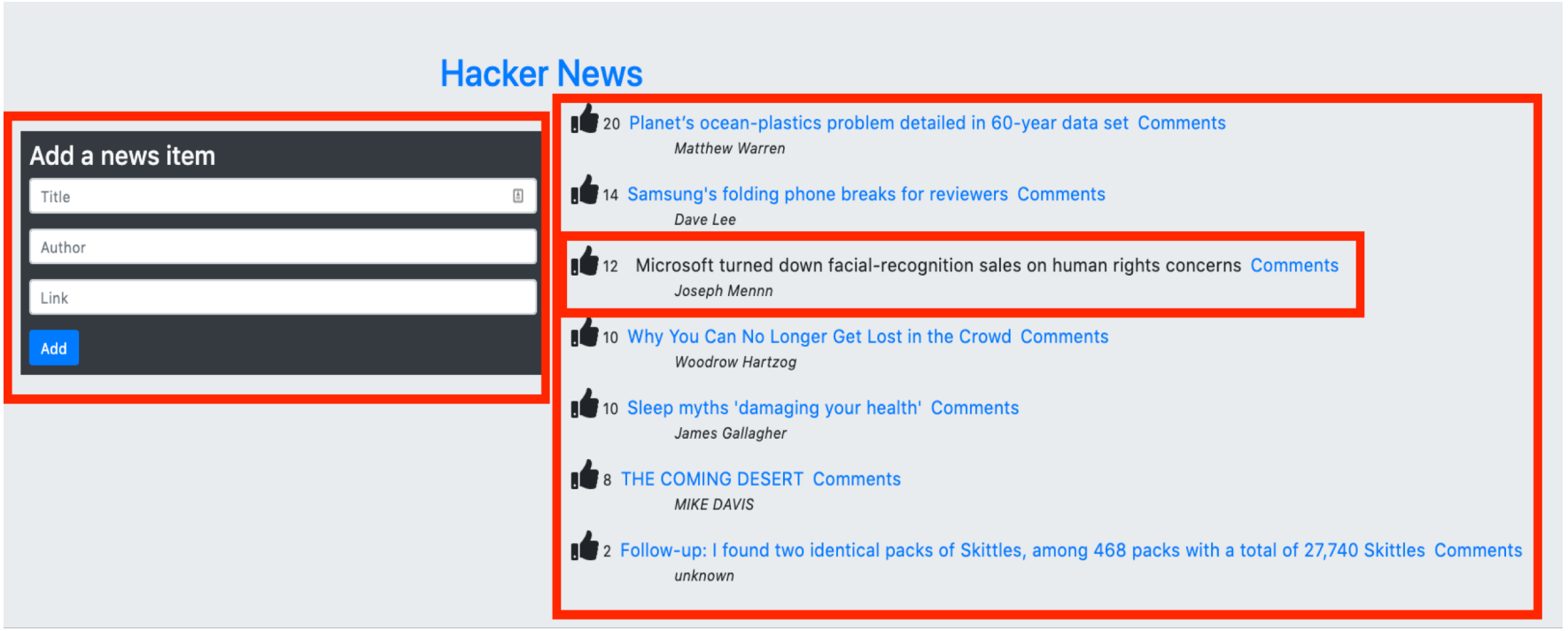
Sample App - Component hierarchy





Contact List App





Hacker News App

Step 1: Break the UI into a component hierarchy.

- **Additional crtieria for devising component breakdown:**
    1. **The** single responsibility principle.
    2. **If it's doing too much, break it up.**
    3. **If it has too much code, break it up.**

- **[Same principles when dealing with Object Oriented design.]**

Stap 2: Build a static version.

- **Use Storybook to build component library.**
  - **Helps determine component** prop requirements**. ******
  - **Start with 'leaf' components, and work up the hierarchy, e.g. Contact → ContactList**
  - **Consider** multiple stories **for a component, e.g. prop boundry values, default value.**

- **Using a sample data set, render the UI** but ignore all interactivity.

- **Components should only have a** render **method.**
  - **No lifecycle methods, event handlers or state, yet.**

- Design Principle**: Decouple structure from interactivity, initially.**
- **"Lots of typing but little thinking."**

Step 3: Identify the *minimal* (but complete) representation of **UI** *state*.

- **Try to keep as many components as possible stateless.**
  - **Stateless components simply render props.**
- **Follow the DRY principle (Don't Repeat Yourself).**
- **Common app pattern – A stateful component computes the props for its subordinates based on current state, domain data and/or its own props.**

Step 3: Identify UI state.

- **What shouldn't go in State?**
  1. **Domain Data - data retrieved from a Web API/service.**
     - **Temporarily stored on the client-side, but not as UI state.**
  2. **Computed data, e.g. subset of matching friends**
     - **Avoids keeping computed state in sync with user interaction; Just re-compute it when necessary.**
  3. **Copies of props: Props are the 'source of truth'.**
     - **Unless props' previous value(s) effects rendering.**
  4. **React components; State should always be JSON-serializable.**

Step 3: Identify UI state.

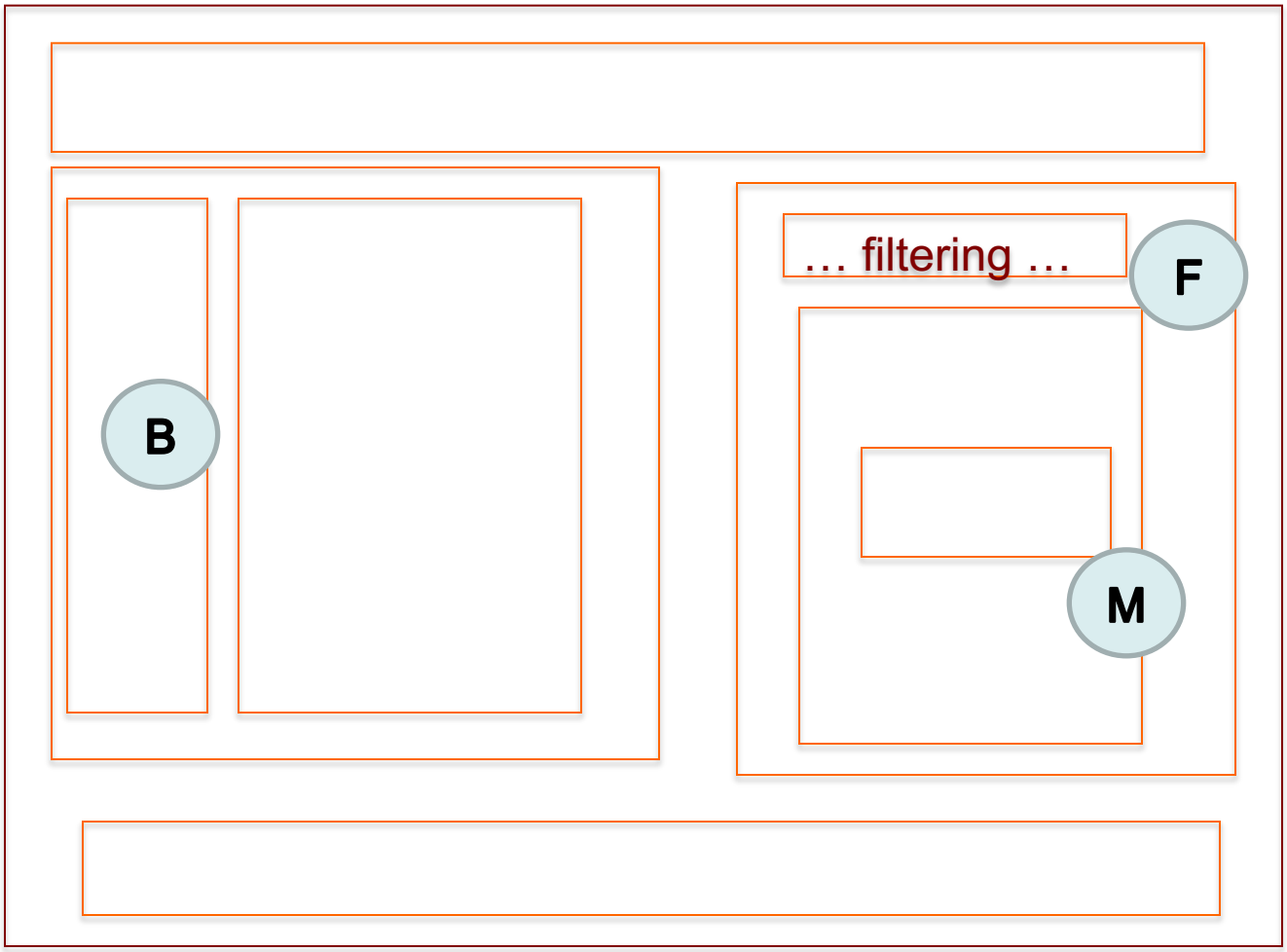
- **What should go in state?**
  - **User dialogues selections – check box, menu, radio button. input text fields.**
  - **Data that an event handlers changes, e.g. counter, text box value.**

- **How to identify state:**
  1. **Identify all of the places where data appears in the UI.**
  2. **For each one, ask a set of questions:**
     I. **Is it passed in via props? If so, probably isn't state.**
     II. **Is it modifiable? If not, probably isn't state.**
     III. **Can you compute it based on any other state or props? If so, it's not state.**

Example: Filtered Friends app

- **Think of all of the places where data appears in the UI:**
  1. **Full List of friends.**
     - **Supplied from web API → Not state.**
  2. **Search text.**
     - **Modifiable, User input → State.**
  3. **Filtered list of friends.**
     - **Computed → Not State.**
  4. **Friend details (name, etc)**
     - **Passed in as props, Not modifiable → Not state**

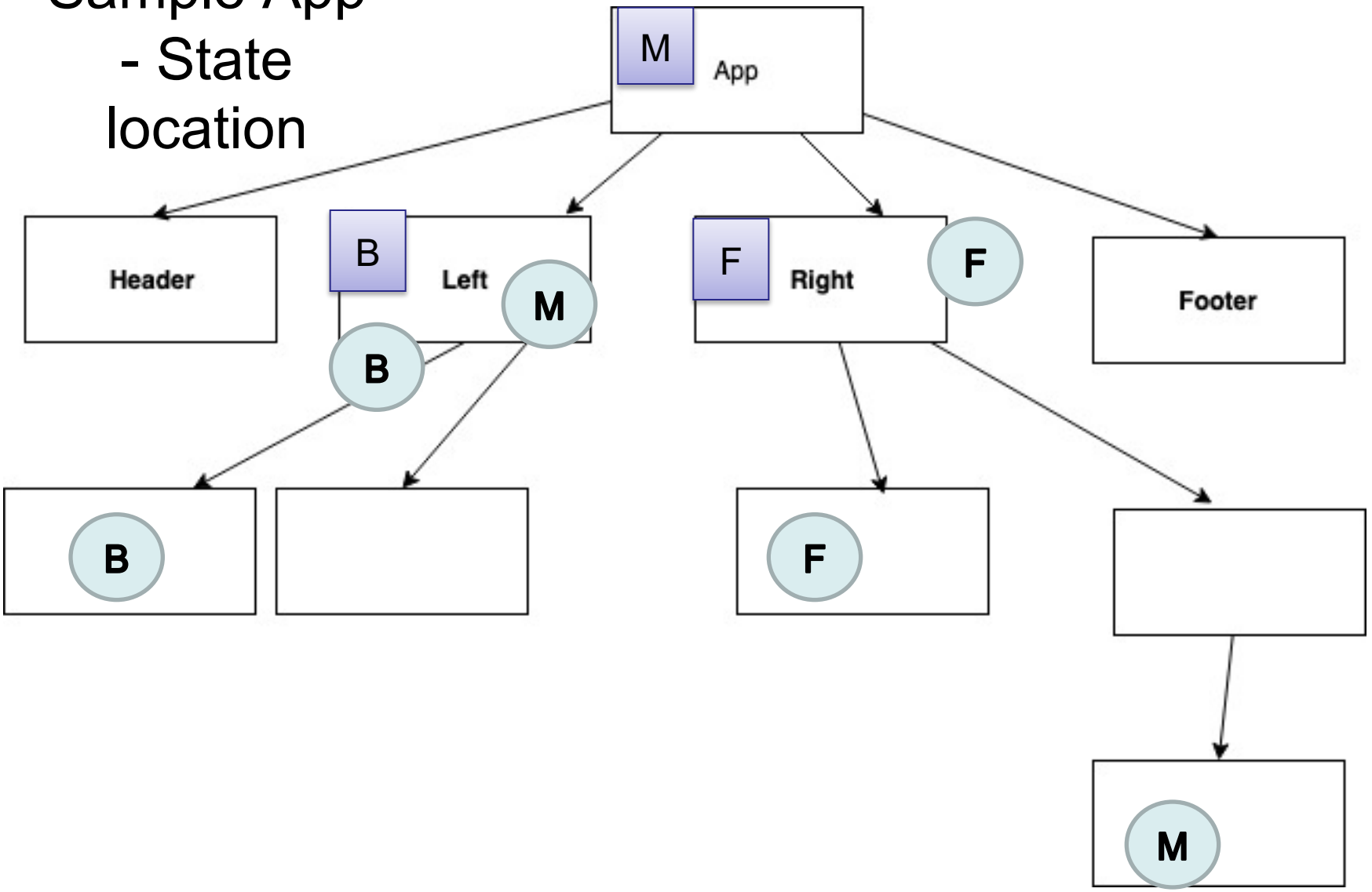
Sample App - Identify UI state



Step 4: Identify where state should live.

- **For each piece of UI state, go through this process:**
  1. **Identify every component that renders something based on its value.**
  2. **From 1 above, identify the ‘common’ ancestor component.**
  3. **[If there is no obvious candidate, create a new ancestor component.]**
  4. **Add state initialization code to selected ancestor component**

Sample App - State location



Sample: Filtered Friends app

- **Only 1 state variable (searchText).**
- **Design A – 1-way data flow design.**
  - FriendsApp **component needs to display the text & use it to compute the filtered list of friends.**
  - **No other component uses this state.**

- **Design B – Inverse Data Flow design.**
  - FriendsApp **component needs it to compute the filtered list of friends.**
  - SearchBox **component needs to display it in the text field.**

  → FriendsApp **is a 'common' component**

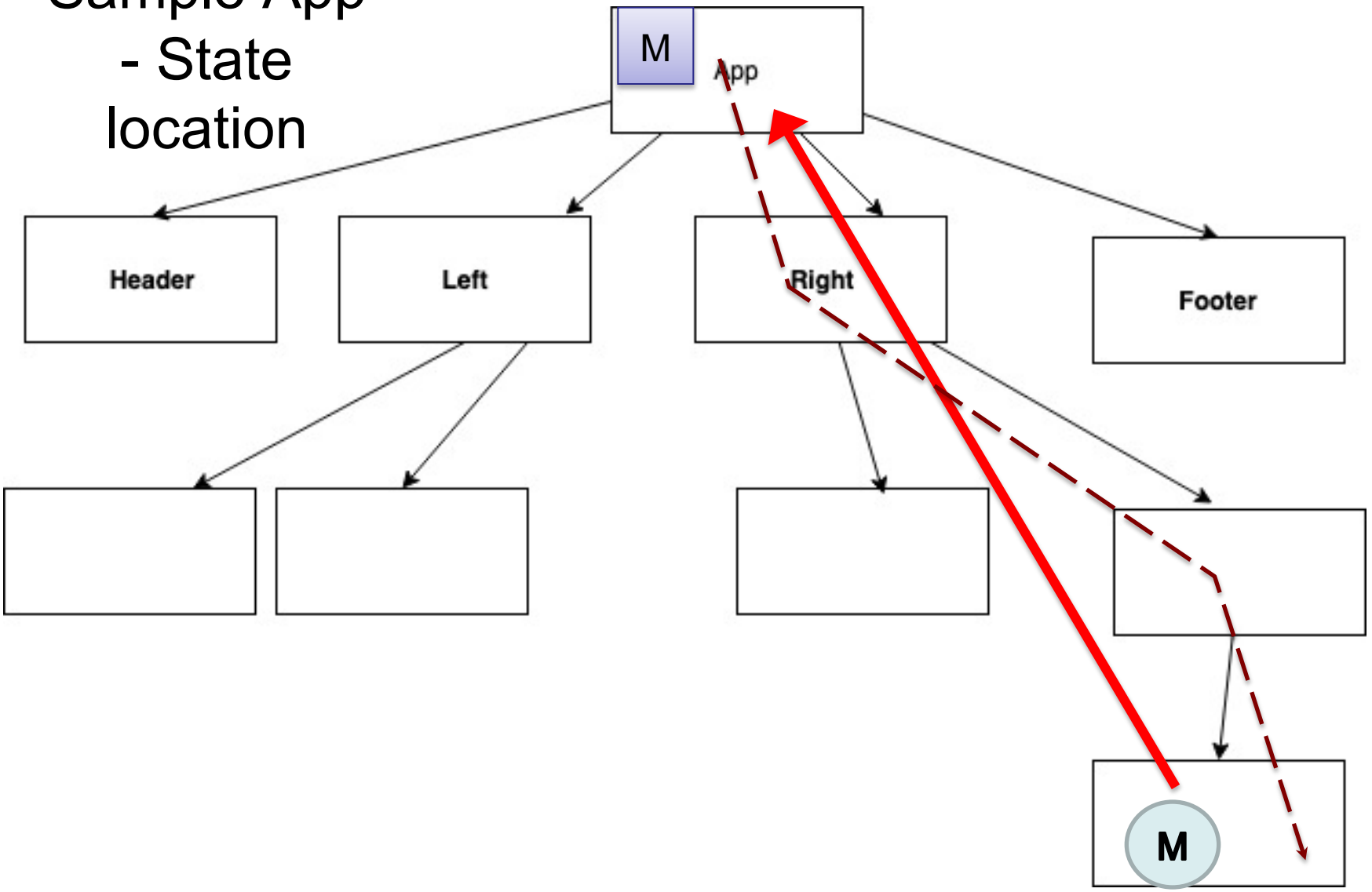
Step 5: Add inverse data flow

- Problem: **A component's state changes when the user interacts with a deeper nested component.**
  - **The nested component must communicate the** event **to the (superordinate) stateful component.**

- Solution: **Inverse data flow pattern:**
  - **Statefull component passes (as a prop) a local function reference to the nested component.**
  - **Nested component calls function when event fires.**

- **Update Storybook stories to reflect additional props (function refs)**

Sample App - State location



Developing a React web app

- **Step 1: Break the UI into a** component hierarchy**.**
- **Stap 2: Build a static version in React.**
- **Step 3: Identify the** minimal **representation of UI state.**
- **Step 4: Identify where your state should live.**
- **Step 5: Add inverse data flow, if required.**