

ReactJS.

Thinking in React

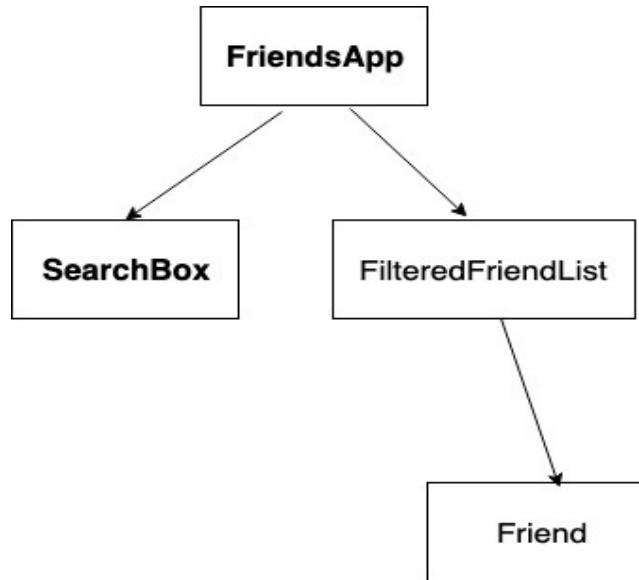
Developing a React web app

- **Step 1: Break the UI into a component hierarchy.**
- **Step 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 (Data down, Actions up), 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.

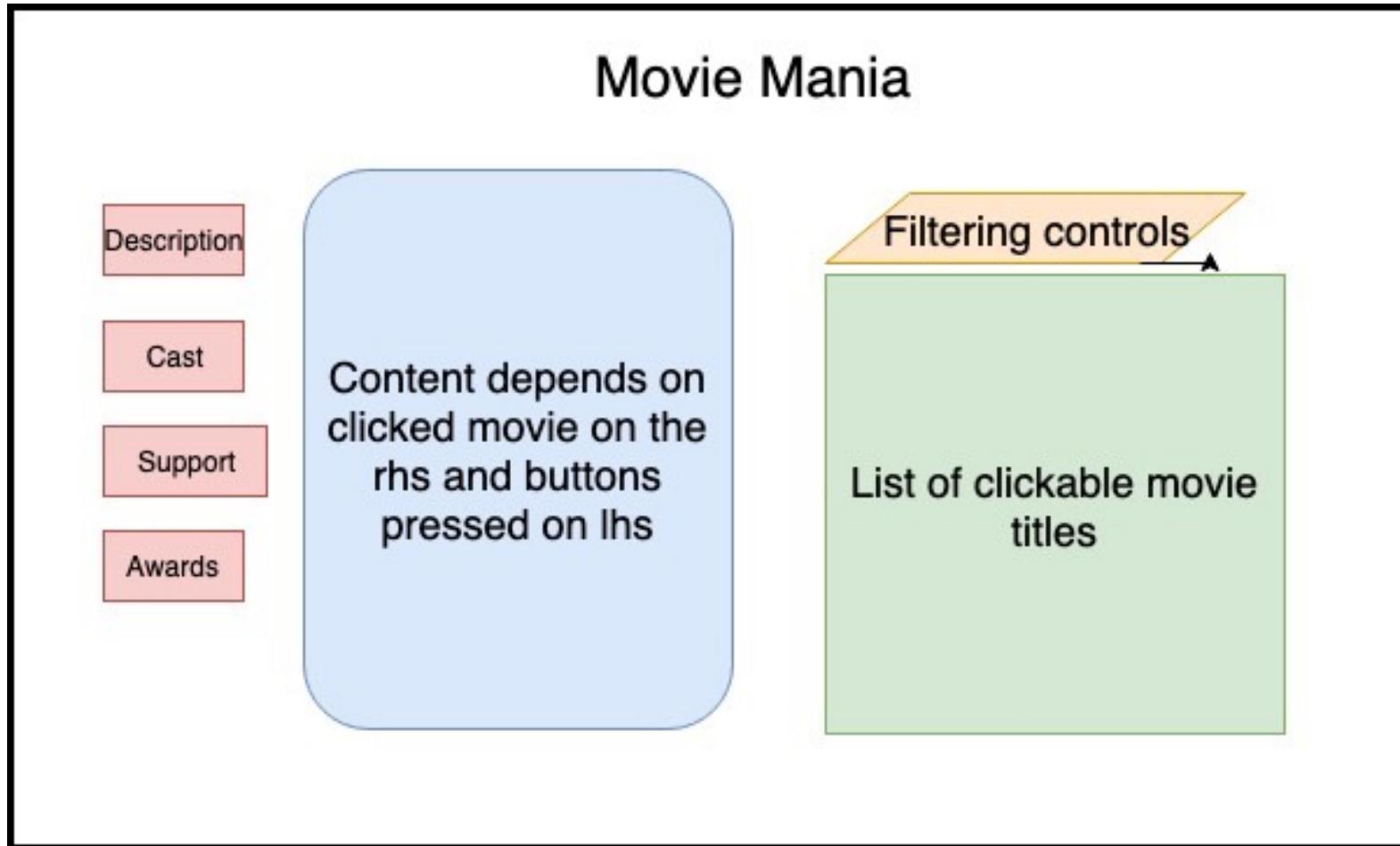


The screenshot shows a user interface titled 'Friends List'. At the top is a search bar labeled 'Search'. Below it is a list of four friends, each represented by a red-bordered box:

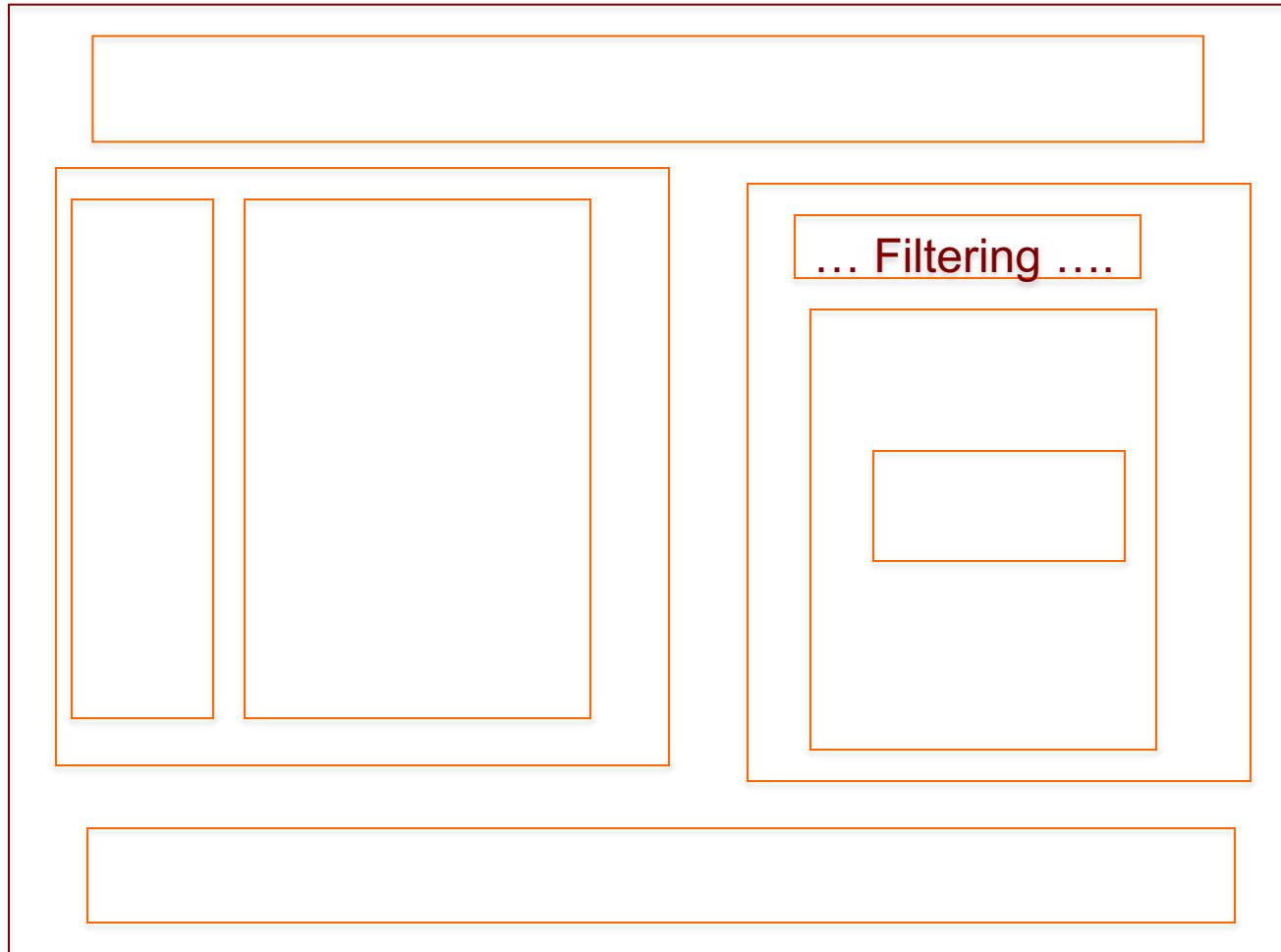
- **Joe Bloggs**
jbloggs@here.con
- **Paula Smith**
psmith@here.con
- **Catherine Dwyer**
cdwyer@here.con
- **Paul Briggs**
pbriggs@here.con

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

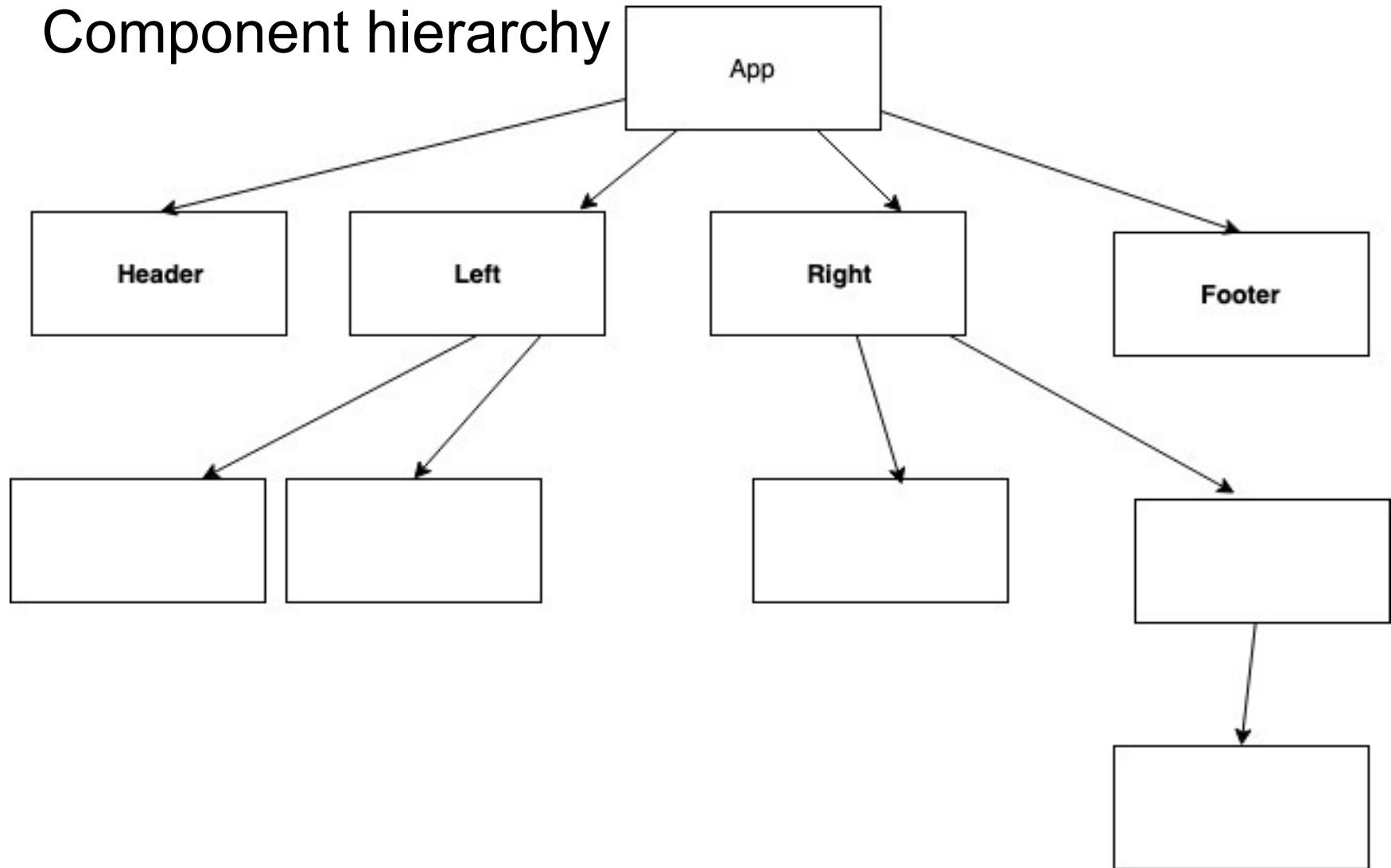
Sample App – Mock UI



Sample App – Component breakdown



Sample App - Component hierarchy



Screenshot of a web browser showing a TMDB Client application. The browser has multiple tabs open, including 'My LastPass Vault', 'AppBar API - Material-L...', 'My Home', 'diarmuidconnor/wad2', 'ICT Skills 2', and 'React App'. The 'React App' tab is active.

The TMDB Client interface features a navigation bar with 'HOME', 'UPCOMING', and 'FAVORITES' buttons. A sidebar on the left contains a search field ('Filter the movies.') and a dropdown menu ('Genre: All'). Below the sidebar are movie cards for 'Miraculous World: New York, United HeroeZ' and 'The Midnight Sky'.

The main content area is titled 'Discover Movies' and displays a grid of movie posters. The movies shown are: 'Wonder Woman 1984' (1), 'Soul' (1), 'Cosmoball' (1), 'Vanguard' (1), 'The Doorman' (4). Each movie card includes a release date, rating, and a 'MORE INFO ...' button.

A detailed diagram at the bottom right, titled 'App (Default)', shows the component hierarchy:

```
graph TD; App[App (Default)] --- SiteHeader[SiteHeader]; App --- HomePage[HomePage]; App --- MovieListHeader[MovieListHeader]; App --- MovieList[MovieList]; App --- FilteringCard[FilteringCard]; App --- MovieCard[MovieCard];
```

The diagram also includes a timestamp 'You, seconds ago' and a small image of a movie poster.

Step 1: Break the UI into a component hierarchy.

- **Additional criteria 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.]

Step 2: Build a static version.

- **Use Storybook to build a component library.**
 - Helps determine component prop requirements. ****
 - Start with ‘leaf’ components, and work up the hierarchy, e.g. MovieCard → MovieList -> FilterControls
 - Consider multiple stories for a component, e.g. prop boundary values, default value.
- **Using a sample data set, render the UI but ignore all interactivity.**
- **Components should only have a return statement.**
 - No hooks, event handlers, algorithms, 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. **Computed data, e.g. subset of matching friends.**
 2. **Copies of props:** Props are the ‘source of truth’.
 - Unless props’ previous value(s) effects rendering.
 3. **React components;** State should always be JSON-serializable.
 4. **Domain Data** - data retrieved from a Web API/service.
 - **Use a 3rd party state management module, e.g. react-redux, or a React Context feature.**

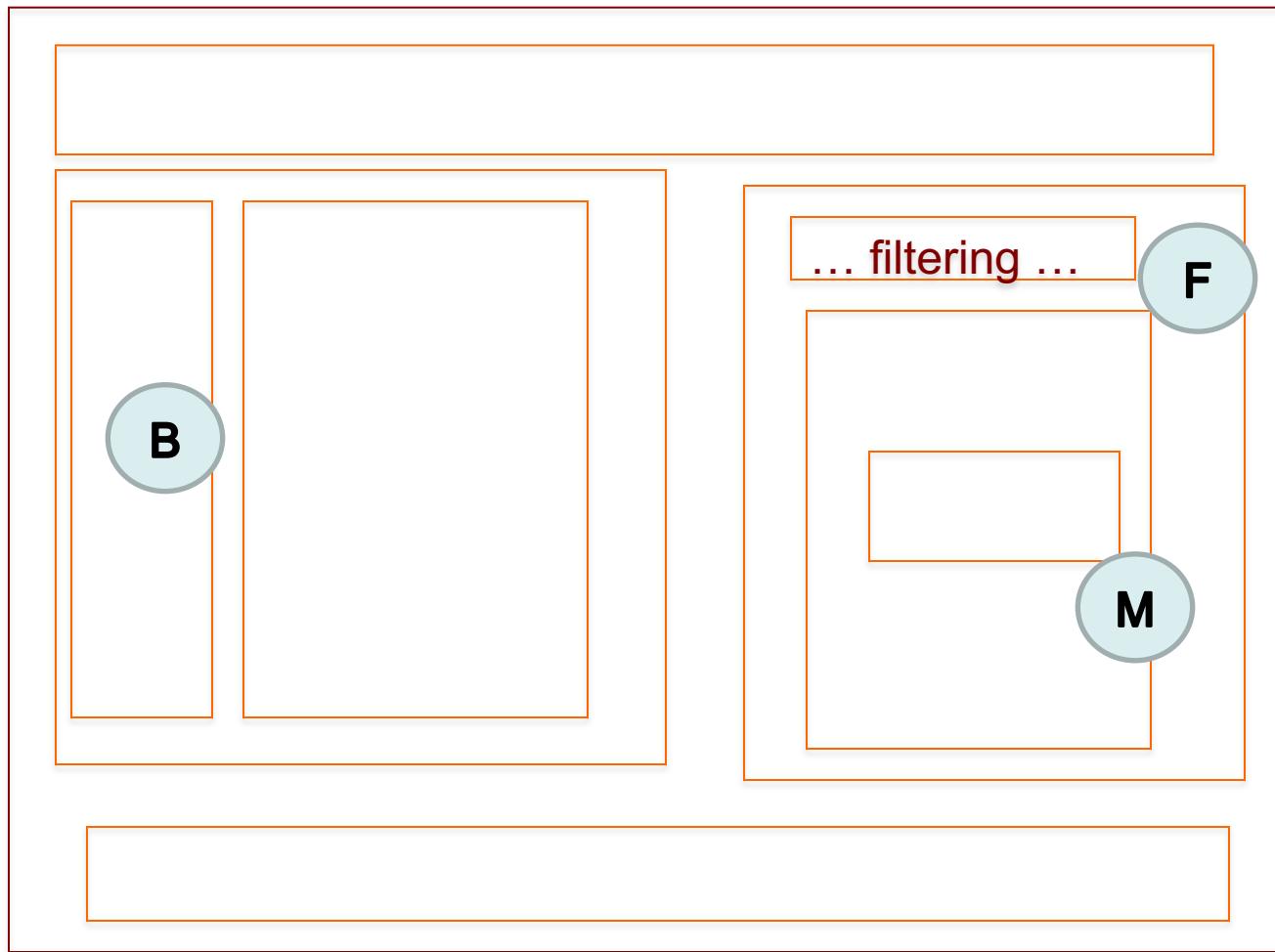
Step 3: Identify UI state.

- **What should go in state?**
 - User inputs – 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/interactive? 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.
 - We should have used a Context instead.
 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

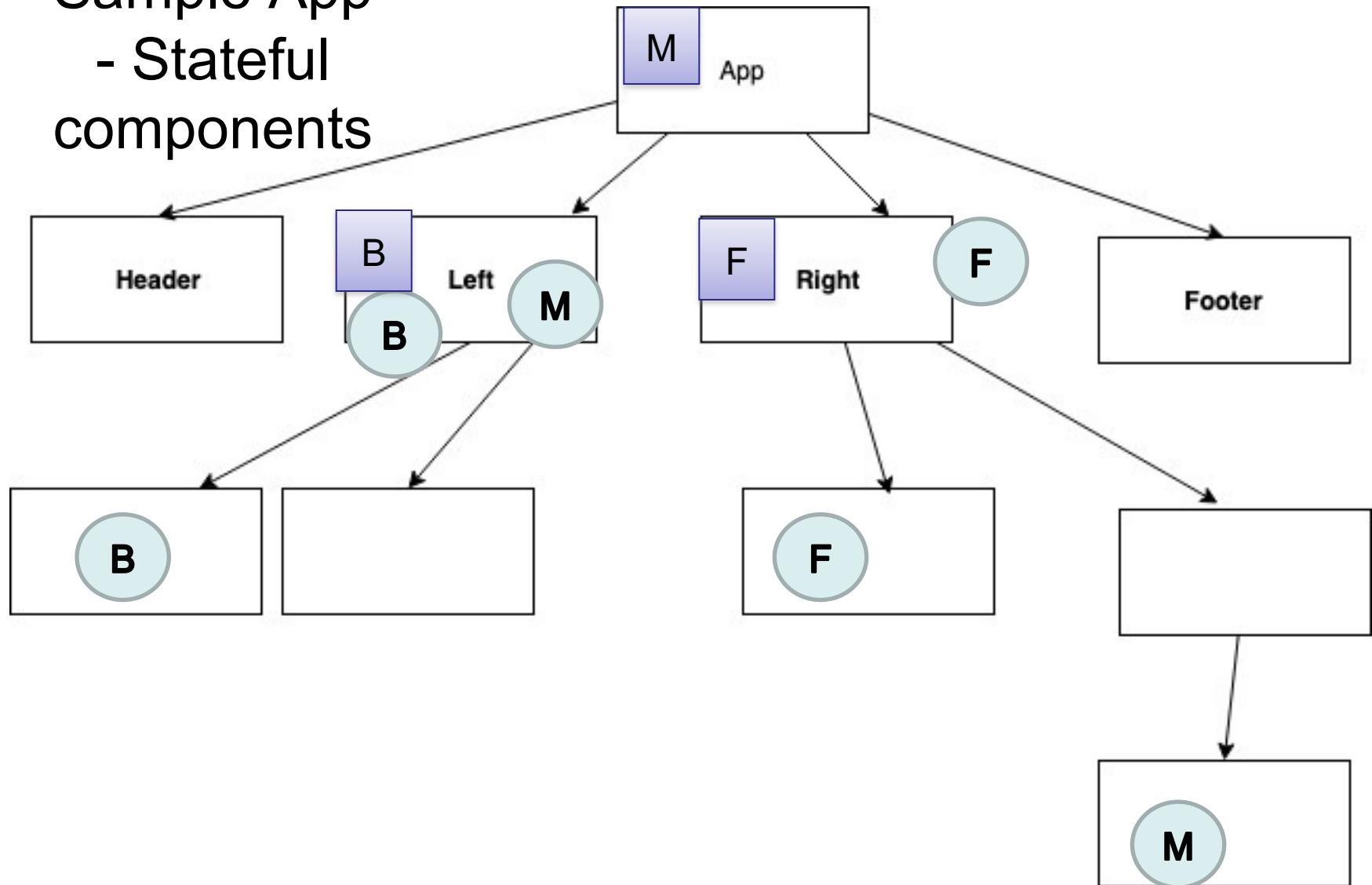
Sample App - Identify UI state



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. Add useState hook to selected ancestor component.

Sample App - Stateful components



Sample: Filtered Friends app

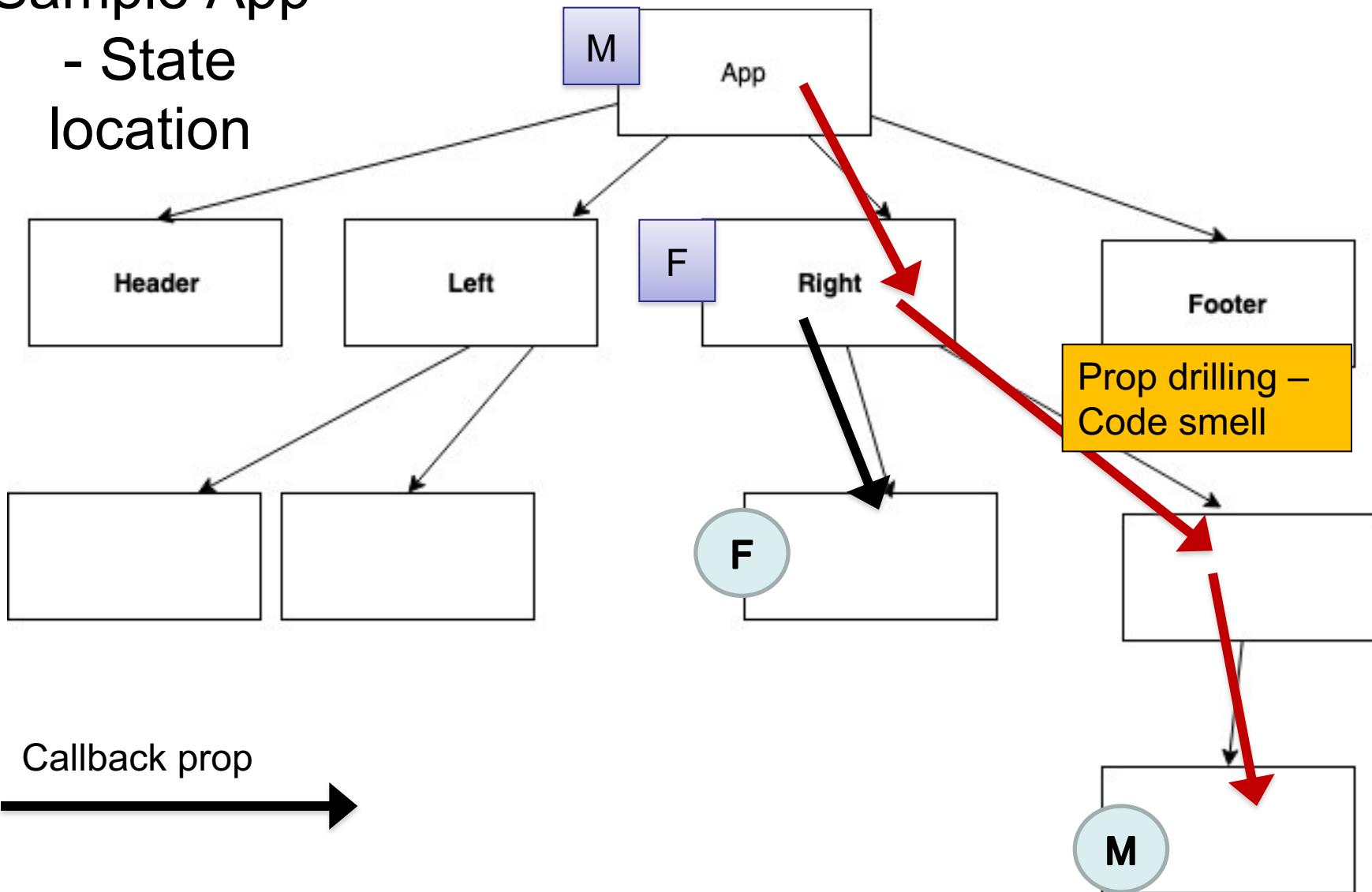
- Only 1 state variable (`searchText`).
- Design A – ‘1-way data flow’ design.
 - FriendsApp component needs to display the text & use it to compute the filtered list of friends.
 - No other component uses this state.
- Design B – ‘Data Down, Action Up’ design.
 - FriendsApp component needs it to compute the filtered list of friends.
 - SearchBox component needs to display it in the text field.
→ FriendsApp is the ‘common ancestor’ component

Step 5: Add inverse data flow

(aka Data Down, Action Up)

- Problem: **A component's state change is determined by an event that occurs in a subordinate component.**
 - The event is usually triggered by user interaction.
 - The subordinate component must communicate the event to the (superordinate) stateful component.
- Solution: **Data down, Action Up pattern:**
 - Stateful component passes a callback function (prop) to the subordinate.
 - Subordinate invokes callback when event fires.
- **Update Storybook stories to reflect callback prop,**

Sample App - State location



Developing a React web app

- **Step 1: Break the UI into a component hierarchy.**
- **Step 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 (Data down, Action Up), if required.**

