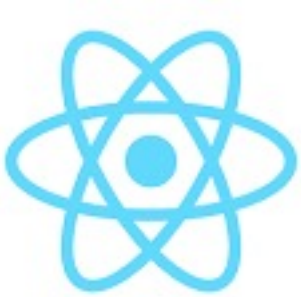


ReactJS.

The Component model

Topics

- **Component State.**
  - **Basis for dynamic, interactive UI.**
- **Data Flow patterns.**
- **Hooks.**
- **The Virtual DOM**

Component DATA

- **A component has two sources of data:**

  1. Props **- Passed in to a component; Immutable; the** props **object.**
  2. *State* **- Internal to the component; Causes the component to** re-render **when changed / mutated.**

  – **Both can be any data type - primitive, object, array.**

- **Props-related features:**
  – **Default values.**
  – **Type-checking.**

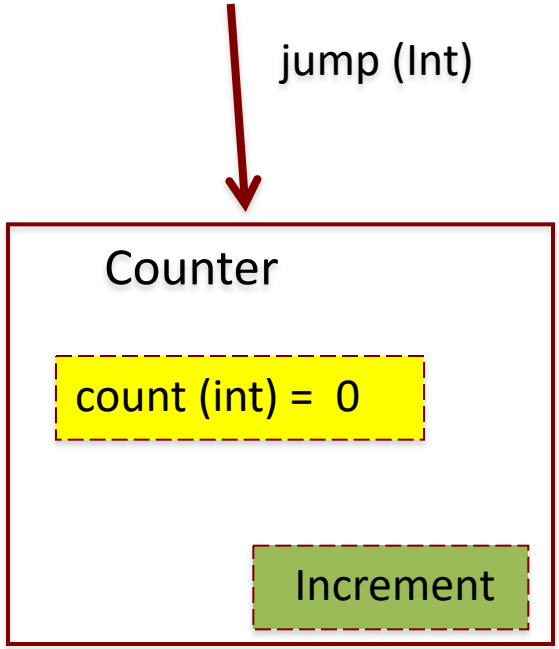
- **State-related features:**
  – **Initialization.**
  – **Mutation – using a** setter **method.**
    – Automatically causes component to re-render. ***
    – **Performs an overwrite operation, not a merge.**

Stateful Component Example

- **The Counter component.**
- **Ref.** basicReactLab **samples – sample 06.**
- **The** useState() **function:**
  - **Declares a state variable.**
  - **Returns a Setter / Mutator method.**
  - **Termed a React** hook.

- **Aside: Static function property,**

  e.g. defaultProps, proptypes



React's event system.

- **Cross-browser support.**
- **Event handlers receive a** SyntheticEvent **– a cross-browser wrapper for the browser's native event.**
- **React event naming convention slightly different to native:**

React	Native
onClick	onclick
onChange	onchange
onSubmit	onsubmit

- See https://reactjs.org/docs/events.html for full details,

Automatic Re-rendering.

- **EX.: The Counter component.**

User clicks button

→ *onClick event handler executes (incrementCounter)*

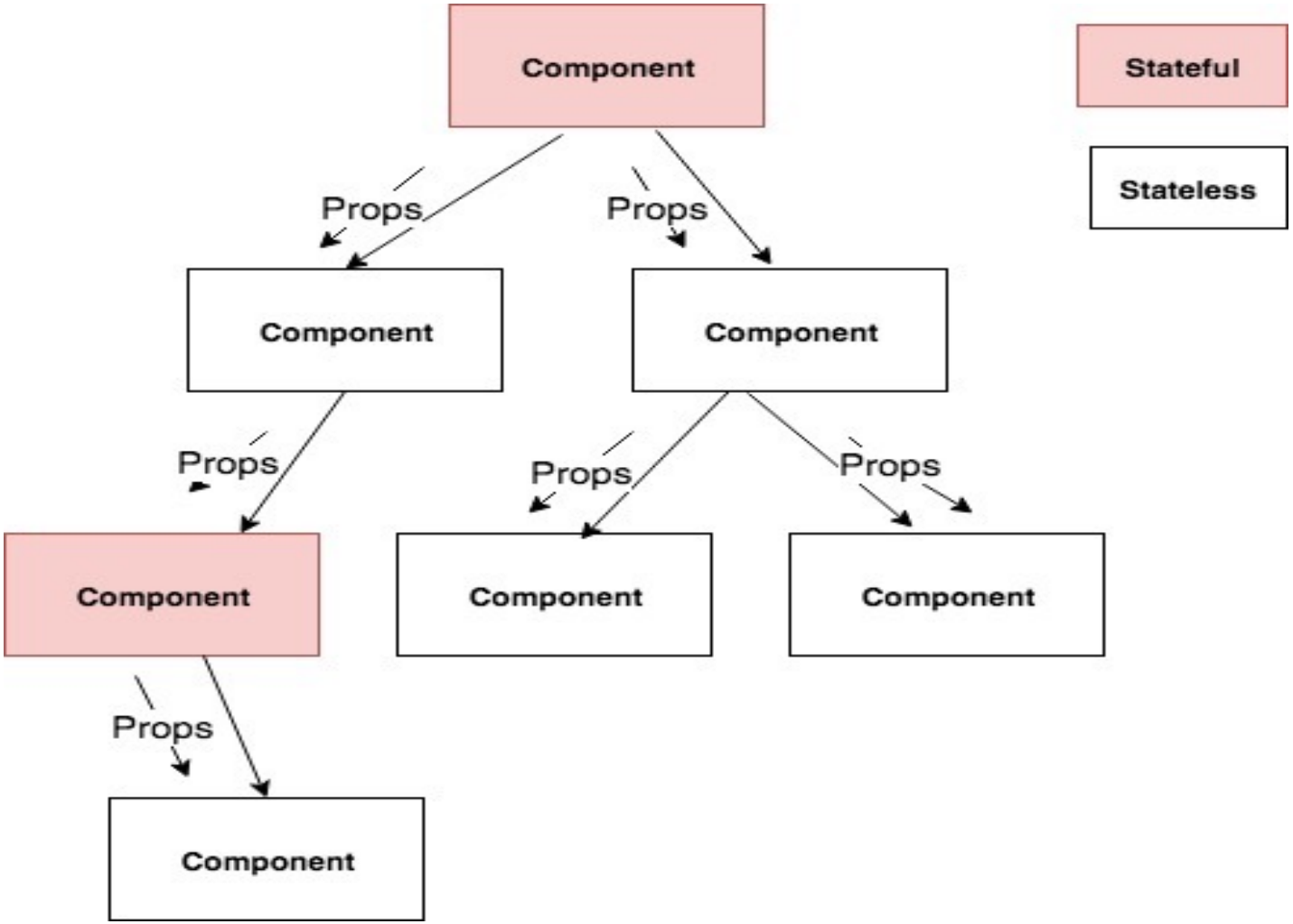
→ *handler changes component's state (setCount())*

→ *component function re- executed (**re-rendering**)* 

Topics

- **Component State.** ✔
- **Data Flow patterns.**
- **Hooks.**
- **The Virtual DOM**

Unidirectional data flow



Unidirectional data flow

- **In React, data flow is** unidirectional **ONLY.**
  - **Other SPA frameworks use** two-way data binding**.**

- **Typical React app pattern: A** small subset **of the components are** stateful**; the majority are** stateless**.**
- **Stateful component execution flow:**
  1. **User interaction causes a component state change.**
  2. **Component re-renders (re-executes).**
  3. **Component recomputes the props for its subordinate components.**
  4. **Subordinate components re-render, and recomputes props for its subordinates.**
  5. **etc.**

Topics

- **Component State.** ✔
- **Data Flow patterns.** ✔ *(more later)*
- **Hooks**
- **The Virtual DOM**

React Hooks

- **Introduced in version 16.8.0 (February 2019)**

- **React Hooks are:**
  1. **Functions (some are HOFs).**
  2. **That performs component** *state manipulation* **and** *lifecycle management***.**

- **Examples: useState, useEffect, useContext, useRef, etc**
  - **'use' prefix is necessary for linting tools.**

- **Usage Rule:**
  - **Can only call hooks at the 'top level' in a component.**
    - **i.e. Don't call hooks inside loops or condition statements.**

useEffect Hook

- **Used when a component needs to perform** side effects**.**

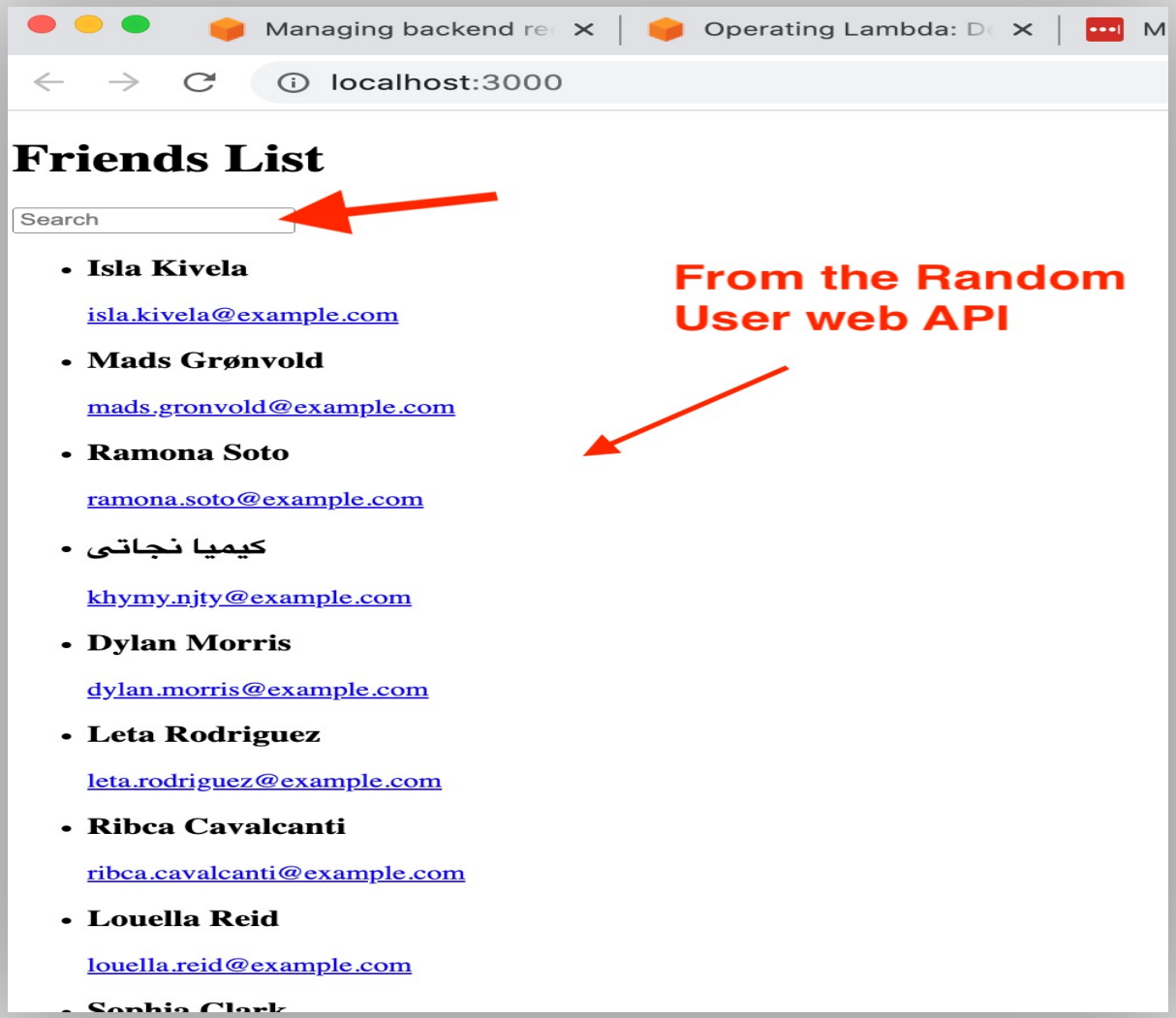
- **Side Effect example:**
  - **fetching data from a web API.**
  - **Subscribe to a browser events, e.g. window resize.**

- **Signature:** useEffect(callback, dependency array)
  - **The** *callback* **contains the side effect code.**
  - Dependencies **determine when a hook is executed.**

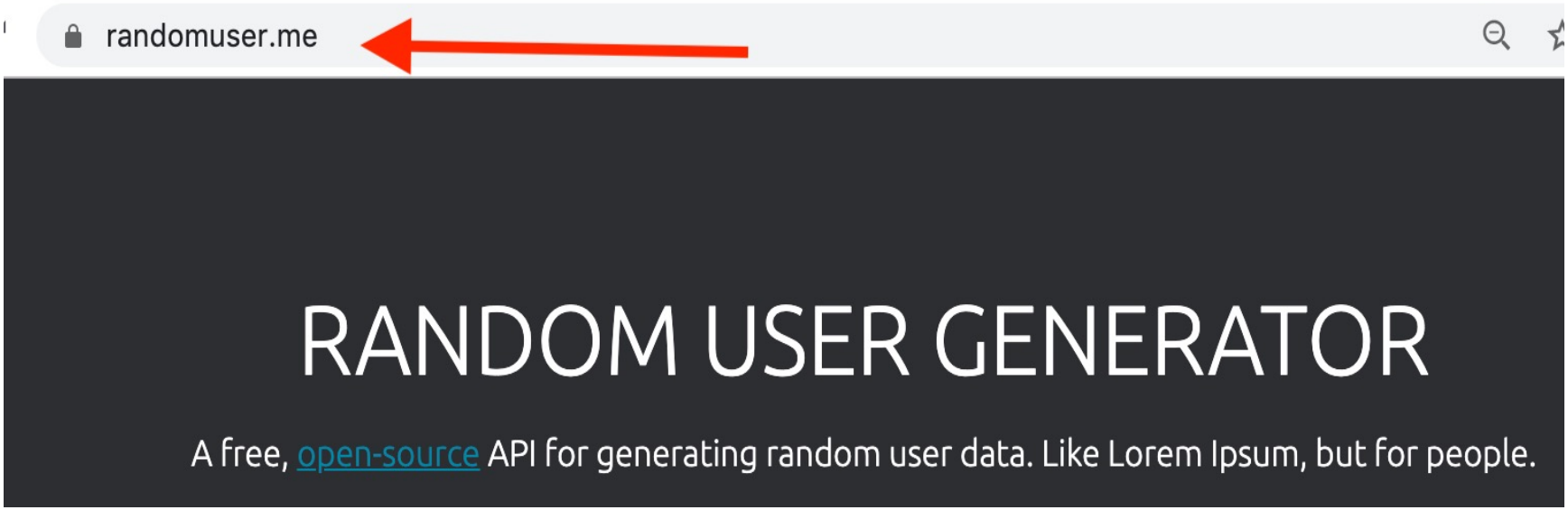
- **useEffect is executed:**
  1. **On component mounting.**
  2. **On every rendering that coincides with a dependency entry update.**

  - **An** empty dependency array **restricts execution to mount-time only.**

Sample App


Managing backend re
Operating Lambda: D
M
localhost:3000
Friends List
Search
Isla Kivela
isla.kivela@example.com
Mads Grønvold
mads.gronvold@example.com
Ramona Soto
ramona.soto@example.com
کیمیا نجاتی
khymy.njty@example.com
Dylan Morris
dylan.morris@example.com
Leta Rodriguez
leta.rodriguez@example.com
Ribca Cavalcanti
ribca.cavalcanti@example.com
Louella Reid
louella.reid@example.com
From the Random User web API

RandomUser open API

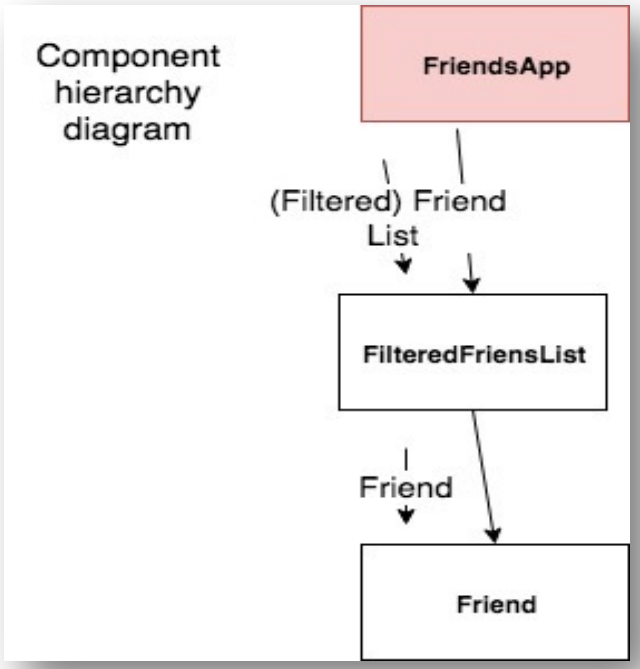


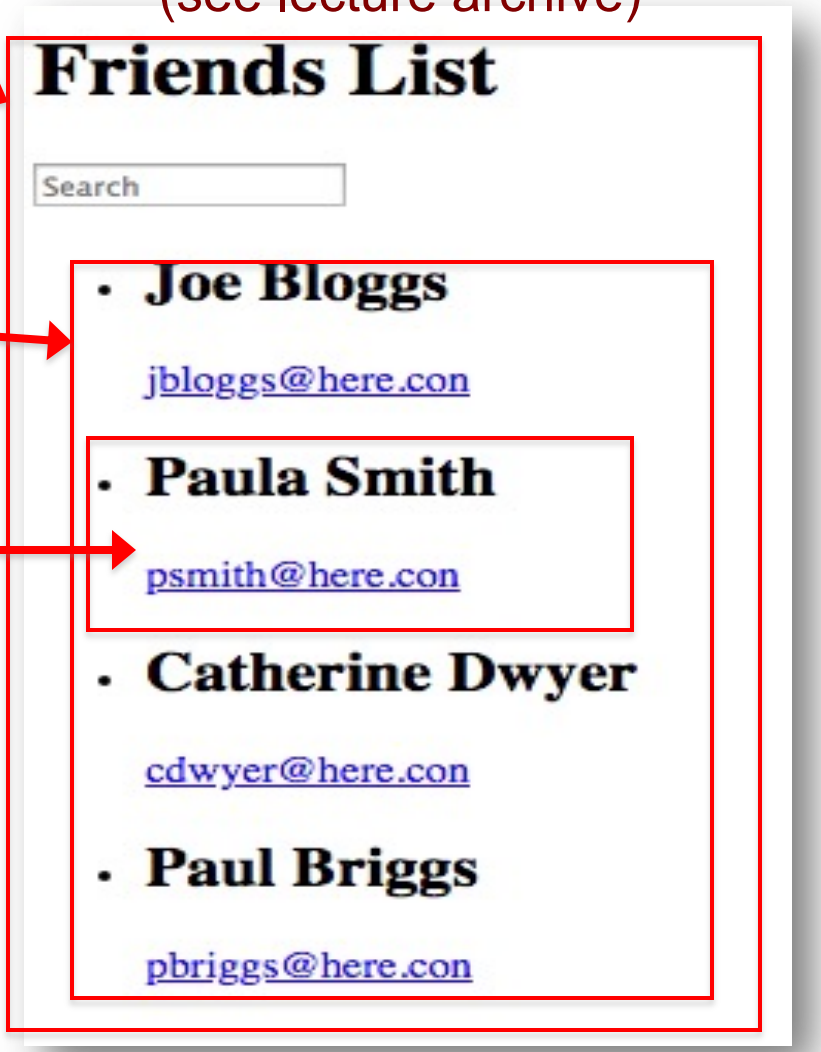
- Returns an auto-generates list of user profiles (friends).
- e.g. Get 10 user profiles:

  GET https://randomuser.me/api/?results=10

Sample App 1

(see lecture archive)





FriendsApp **component:**

1. Manages state **(i.e. text box, full list of friends).**

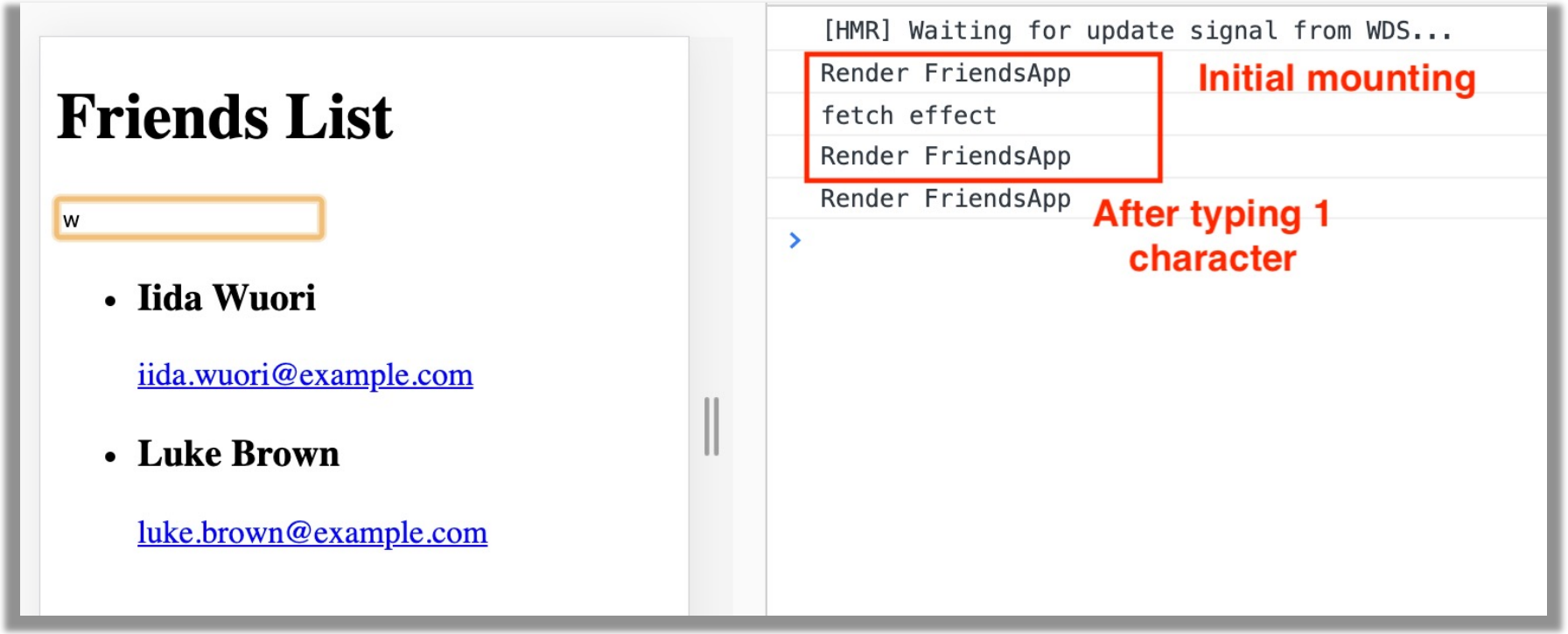
2. Fetches data from a web API – side effect.

3. Computes matching friends.

4. Controls list rendering.

Sample App - *useEffect* Hook

- **useEffect runs AT THE END of a component's mount process.**
  i.e. First rendering occurs BEFORE the API data is available.
  - **We must accommodate this in the implementation.**



Sample App - *useEffect* Hook

- **You must allow for asynchronous nature of API calls:**
  1. **UI must remain interactive while waiting for API data.**
  2. **Components should render without errors before API data is available.**

- **Correct approach:**

  const [friends, setFriends] = useState(***[ ]***) ; // Store API data

- **Incorrect approach:**

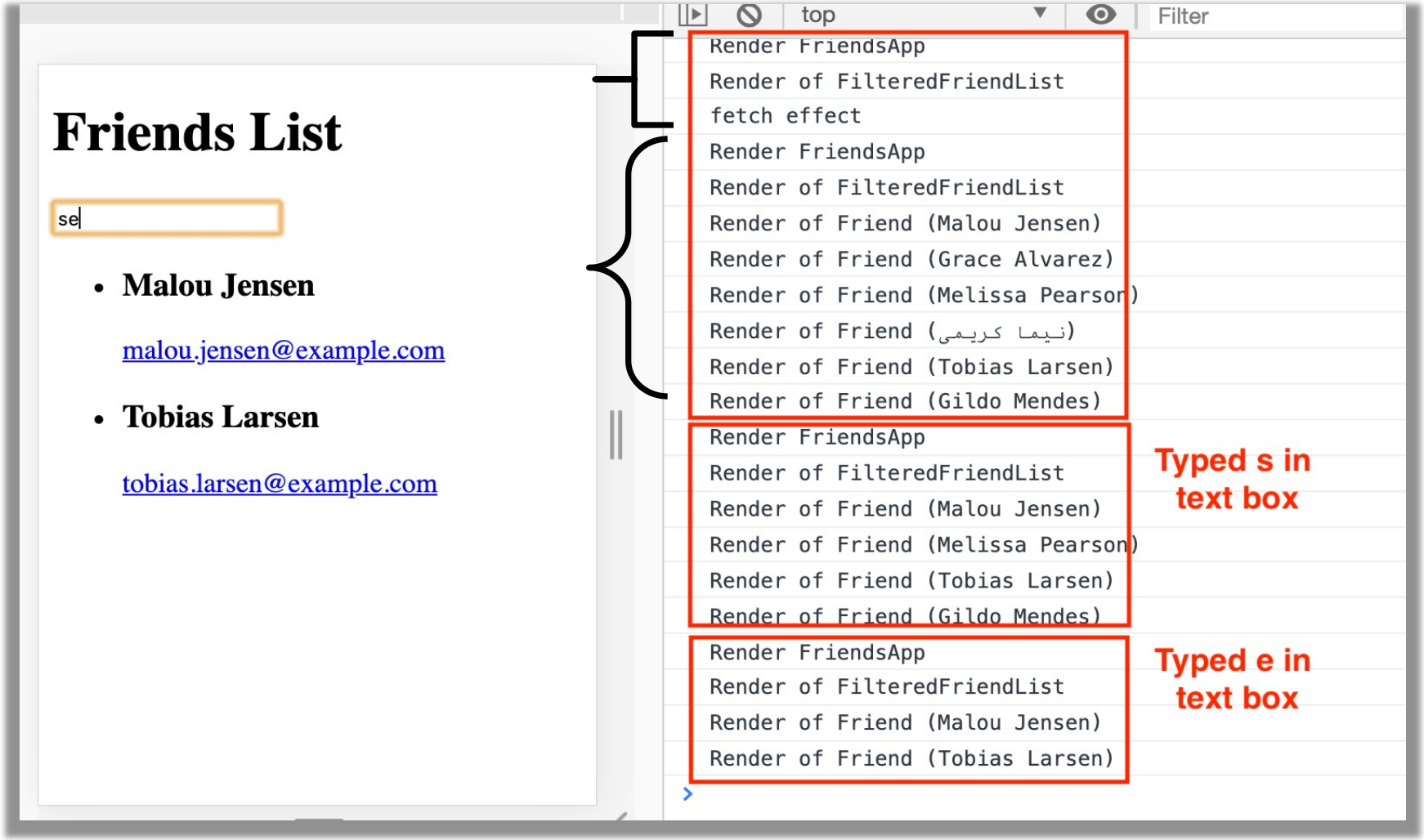
  const [friends, setFriends] = useState(***null***);

  – **Resulting error**

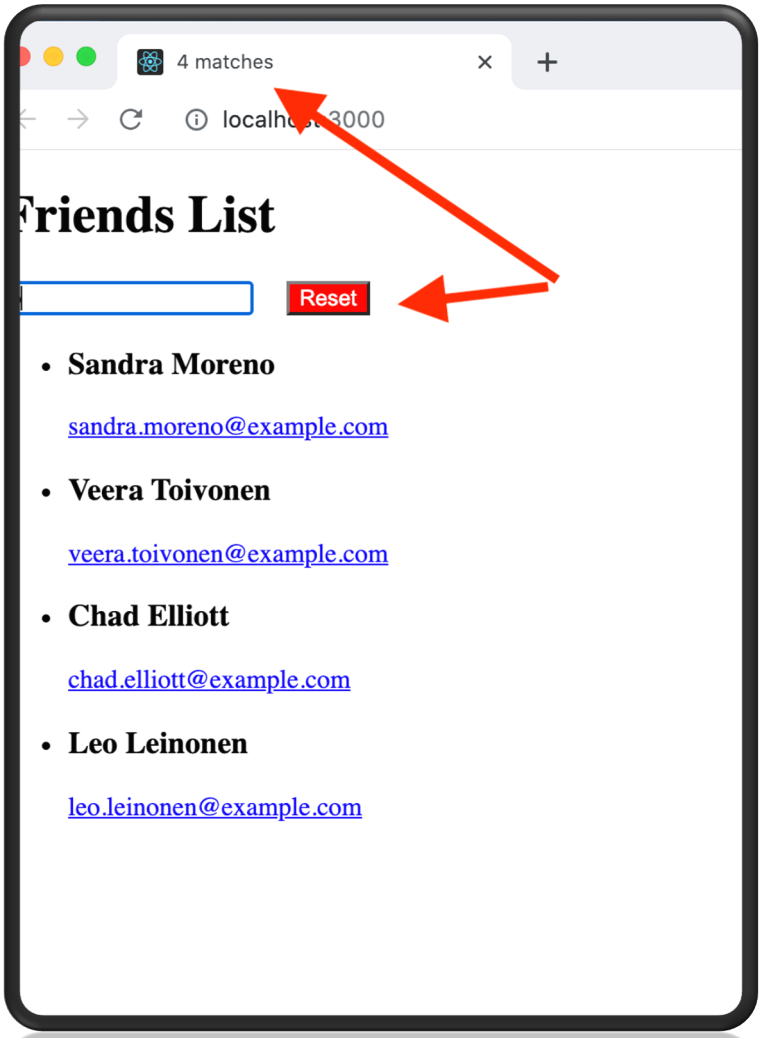
  TypeError: Cannot read property 'filter' of null

Unidirectional data flow & **Re-rendering**

(Assume we request 6 friends from web API)

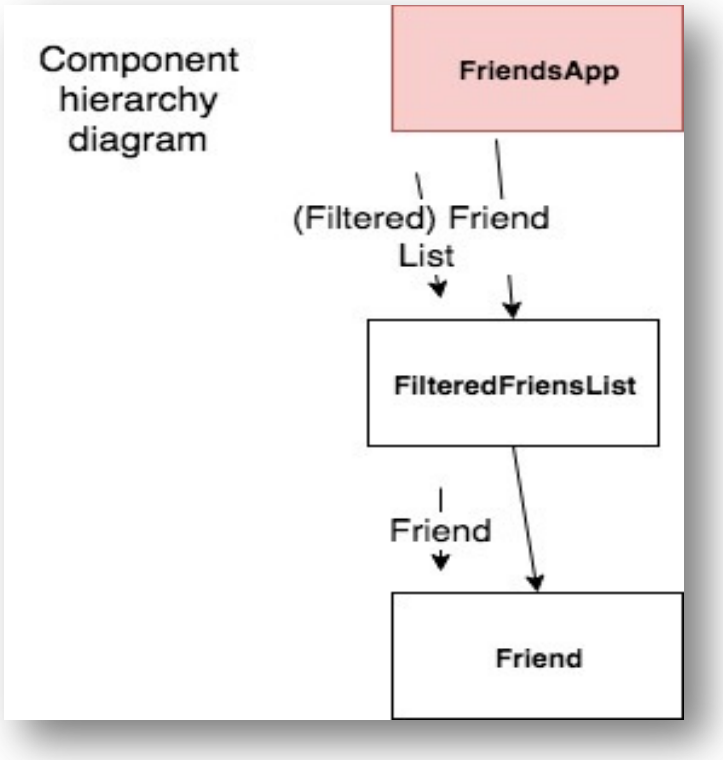


Sample App 1 – Version 2



- **App UI changes:**
  1. **A 'Reset' button – loads a new list of friends (overwriting the current list).**
  2. **Browser tab title - shows # of matching friends (side effect).**
- **Ref. lecture archive for source code**

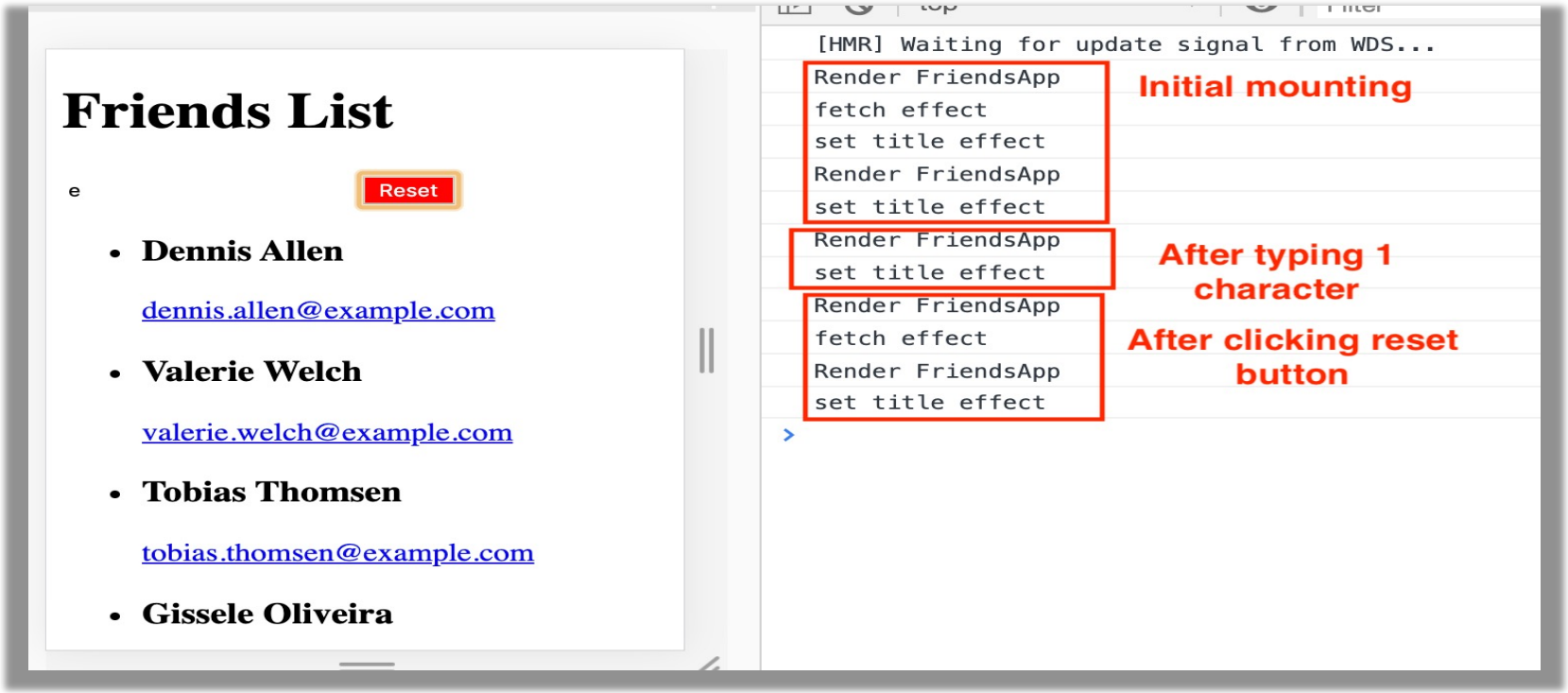
Sample App 1 (v2) - Design



- **3 state variables:**
  1. **List of friends from API.**
  2. **Text box content.**
  3. *Reset button toggle.*

- **2 side effects, both with dependency arrays:**
  1. **'Fetch API data' -** reset button **toggle dependency.**
  2. **'Set browser tab title' - matching list <u>length</u> dependency.**

Sample App 1 (v2) - Events



Sample App 1 (v2) - Events.

- **On mounting of** FriensApp **component:**
  Both effects execute (Set browser tab to ’0 matches’).
  → **‘Fetch data’ effect initializes ‘friends list’ state.**
  → **Component re-renders → ‘Set browser tab’ effect executes.**
- **On typing a character in the text box:**
  ‘Text box’ state changes.
  → FriendsApp **rerenders + recomputes matching friends list**
  → **‘Set browser title’ effect executes.**
- **On clicking Reset button:**
  ‘Reset toggle’ state changes.
  → FriendsApp **re-renders.**
  → **‘Fetch data’ effect executes.**
  → **‘Friends list’ state changes.**
  → **FriendsApp re-renders + recomputes matching list**
  → **‘Set browser title’ effect executes.**

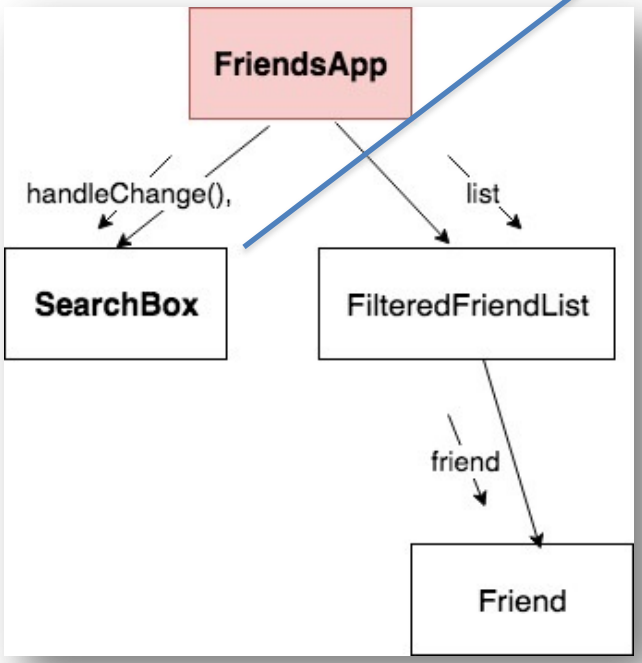
Topics

- **Component State.** ✔
- **Data Flow patterns.**
- **Hooks.**

Sample App 2

(Data down, actions up pattern or Inverse data flow pattern)

- **Often a component's** state change **is caused by an** event **in a subordinate component.**
- **Solution: Use the** data down, action up pattern.



Data down, Action up.

Pattern:

1. Stateful **component provides a callback to the subordinate.**

2. Subordinate **invokes callback when the event occurs.**

```
const FriendsApp = () => {
  const [searchText, setSearchText] = useState("");
  const [friends, setFriends] = useState([]);

  useEffect(() => {…
  }, []);

  const filterChange = text =>
    setSearchText(text.toLowerCase());

  const updatedList = friends.filter(friend => {…
  });
  return (
    <>
      <h1>Friends List</h1>
      <SearchBox handleChange={filterChange } />
      <FilteredFriendList list={updatedList} />
    </>
```

```
const SearchBox = props => {
  const onChange = event => {
    event.preventDefault();
    const newText = event.target.value.toLowerCase();
    props.handleChange(newText);
  };

  return <input type="text" placeholder="Search"
           onChange={onChange} />;
};
```

Topics

- **Component State.** ✔
- **Data Flow patterns.**
- **Hooks.**
- **The Virtual DOM**

Modifying the DOM

- **DOM – an internal data structure representing the browser's current 'display area' ; DOM always** in sync **with the display.**

- **Traditional performance best practice:**
  1. **Minimize direct accessing of the DOM.**
  2. **Avoid 'expensive' DOM operations.**
  3. **Update elements offline, then replace in the DOM.**
  4. **Avoid changing layouts in Javascript.**
  5. **. . . etc.**

- **Should the developer be responsible for low-level DOM optimization? Probably not.**
  - **React provides a *Virtual DOM* to shield developers from these concerns.**

The Virtual DOM

- **How React works:**
  1. **At app startup it create a lightweight, efficient form of the DOM, termed the *Virtual DOM*.**
  2. **The app changes the V. DOM whenever a component re-renders.**
  3. **When the re-rendering cycle is complete, the React engine:**
     1. **Performs a *diff* operation between the current and previous V. DOM instance.**
     2. **Computes the *set of changes* to apply to real DOM.**
     3. **Batch update the real DOM.**

- **Benefits:**
  a) **Cleaner, more descriptive programming model.**
  b) **Optimized DOM updates and reflows.**

Automatic Re-rendering (detail)

- **EX.: The Counter component.**

User clicks button

→ *onClick event handler executed*

→ *component state is changed*

→ *component re-executed (re-renders)*

→ *The Virtual DOM has changed*

→ *React diffs the changes between the current and previous Virtual DOM*

→ *React batch updates the Real DOM*

Re-rendering & the real DOM

- **What happens when the user types in the text box?**

User types a character in text box

→ *onChange* ***event handler executes***

→ ***Handler changes a state variable***

→ ***React re-renders*** *FriendsApp component*

→ ***React re-renders children (****FilteredFriendList****) with new prop values.***

→ ***React re-renders children of*** *FilteredFriendList****.***

(Re-rendering completed)

→ ***(Pre-commit phase) React computes the updates required to the browser's DOM***

→ ***(Commit phase) React batch updates the DOM.***

→ ***Browser repaints screen***

Summary

- **A state variable change always causes a component to re-render.**
  - **State change logic is usually part of an event handler function.**
  - **Event hadler may be in a subordinate component.**

- **Side effects:**
  - **Always execute at mount time.**
  - **The dependency array will either reference a state variable, a value computed from a state variable, or a prop.**
    - **Can be multiple entries**
  - **Callback performs the side-effect, and may also cause a state change.**

- **Data flows downward, actions flow upward.**