ReactJS.

The Component model

Topics

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

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.

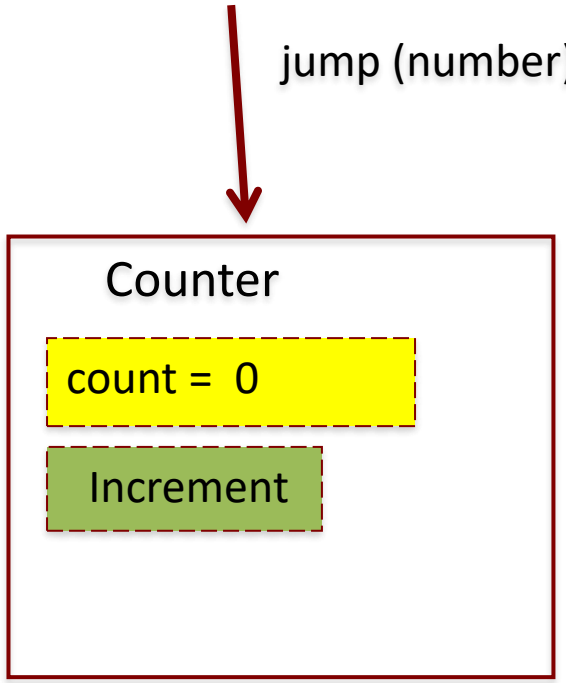
  – Both can be any data type - primitive, object, array.

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

- State-related features:
  - Initialisation.
  - 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.




```tsx
import React, { useState } from "react";

interface CounterProps {
    jump: number,
};

const Counter: React.FC<CounterProps> = (props) => {
    const [count, setCount] = useState(0);
    console.log(`${count} ${props.jump}`)

    const incrementCount = () => {
        setCount(count + props.jump)
    };

    return (
        <>
            <h2>Count: {count}</h2>
            <h3>Increment size: {props.jump}</h3>
            <button type="button" onClick={incrementCount}>
                Increment
            </button>
        </>
```

React’s event system.

- **Cross-browser support.**
- **Event handlers receive a React Event Object – 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://react.dev/learn/responding-to-events and https://react.dev/reference/react-dom/components/common#react-event-object

for full details

Automatic Re-rendering.

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

User clicks button

→ *onClick event handler executes (incrementCounter)*

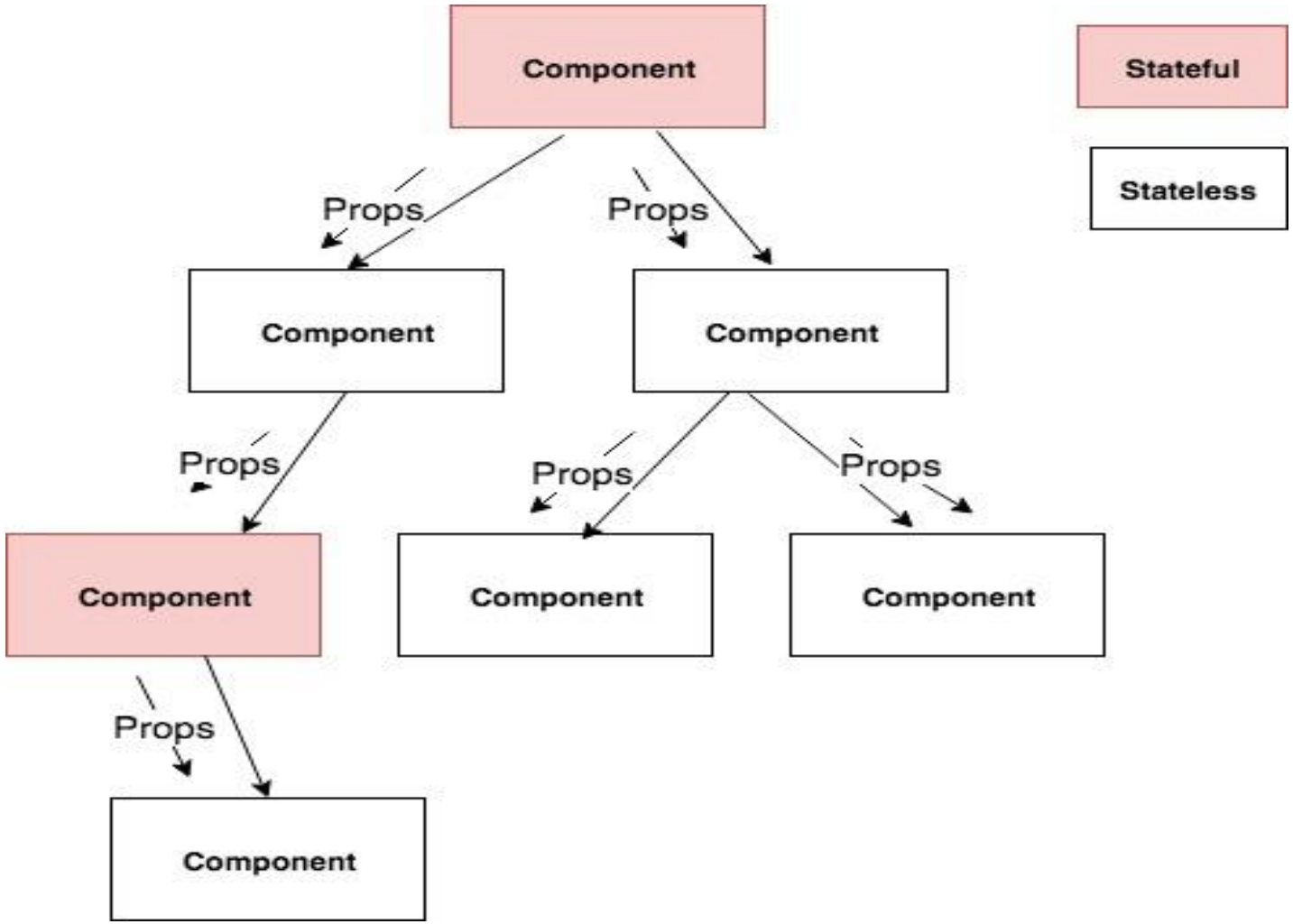
→ *handler changes component's state (setCount())*

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

Topics

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

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**

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**.**

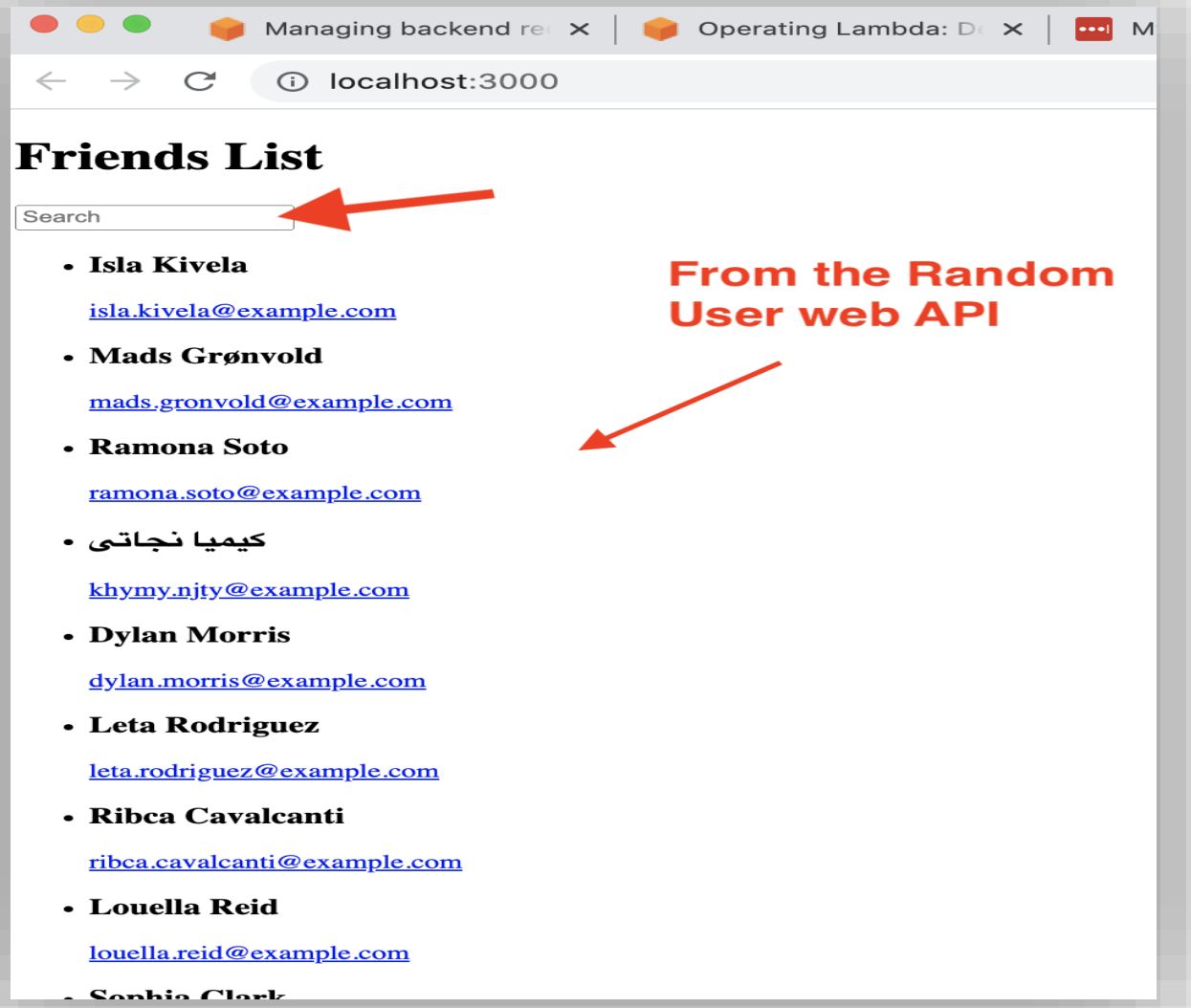
- **Side Effect example:**
  - **fetching data from a web API.**
  - **Subscribe to a browser events, e.g. window resize.**

- **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.**

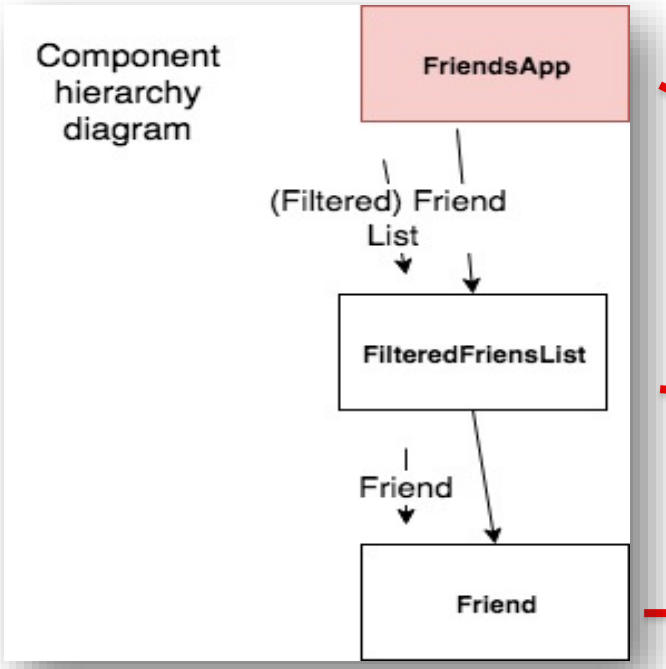
  - **An** empty dependency array **restricts execution to mount-time only.**

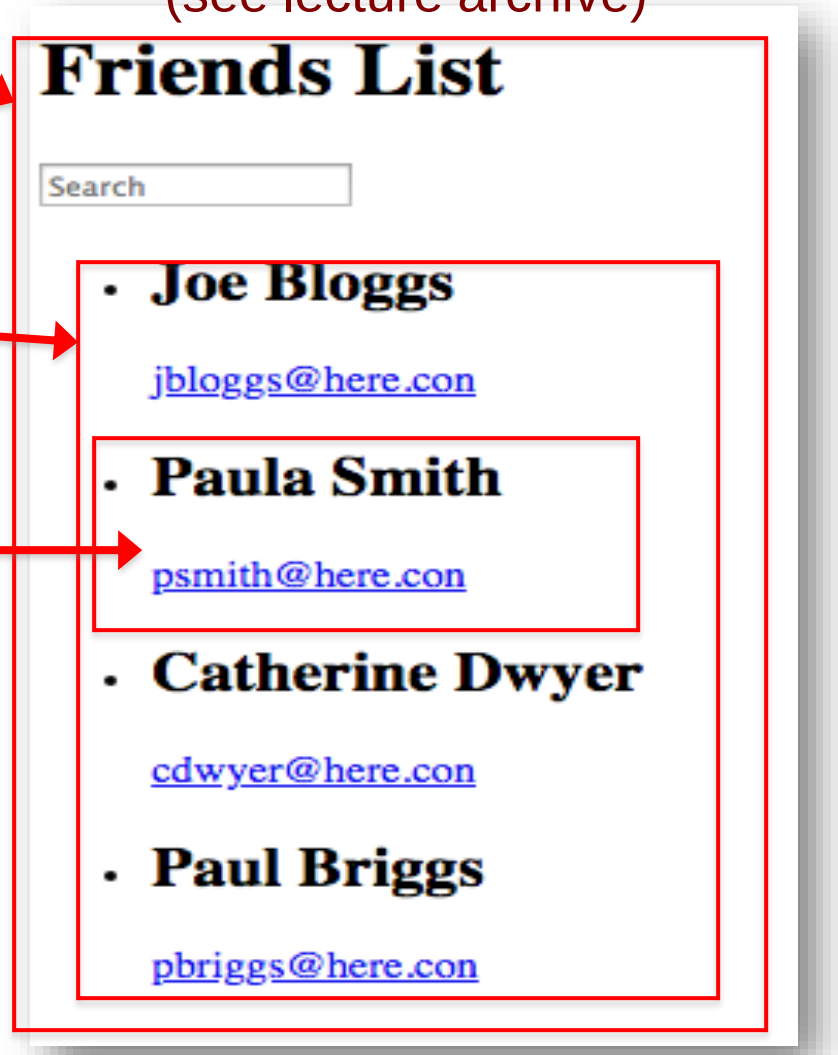
Sample App


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
كيميا نجاتى
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

Sample App 1

(see lecture archive)





FriendsApp **component:**

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

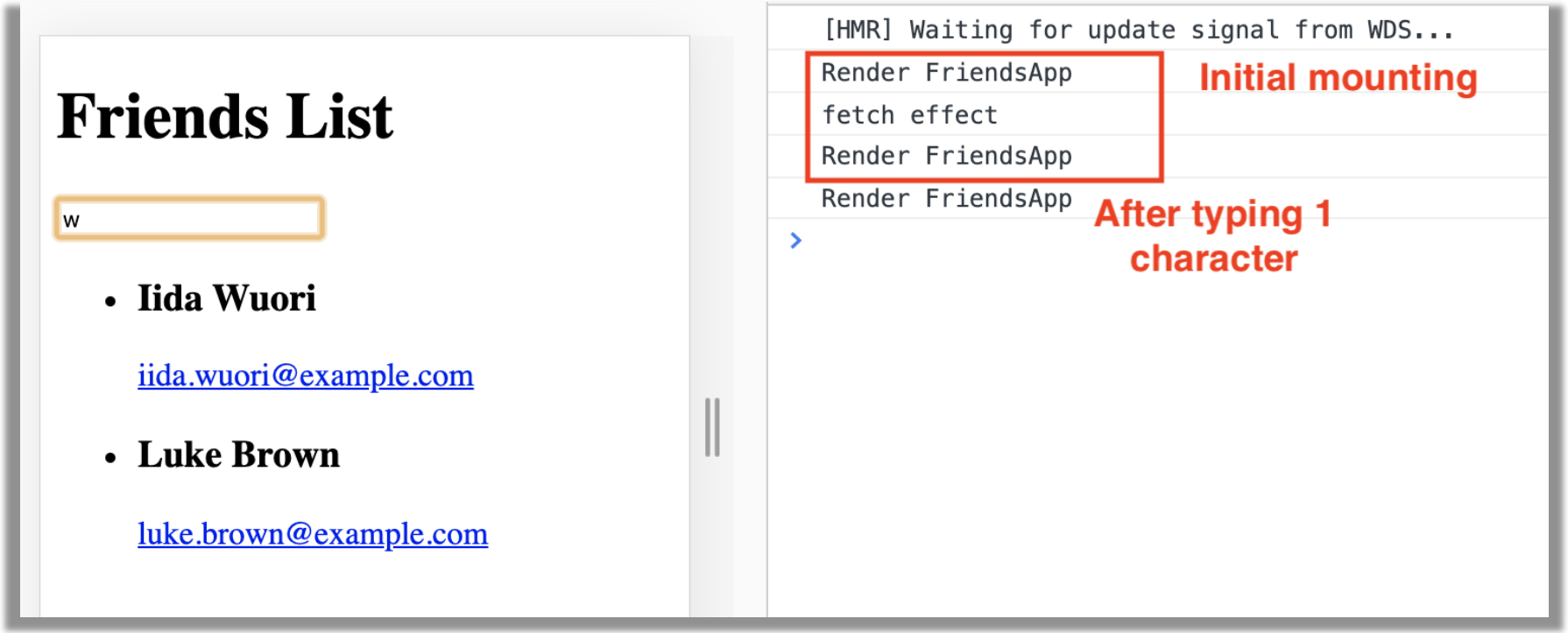
2. Fetches data from a web API – side effect.

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

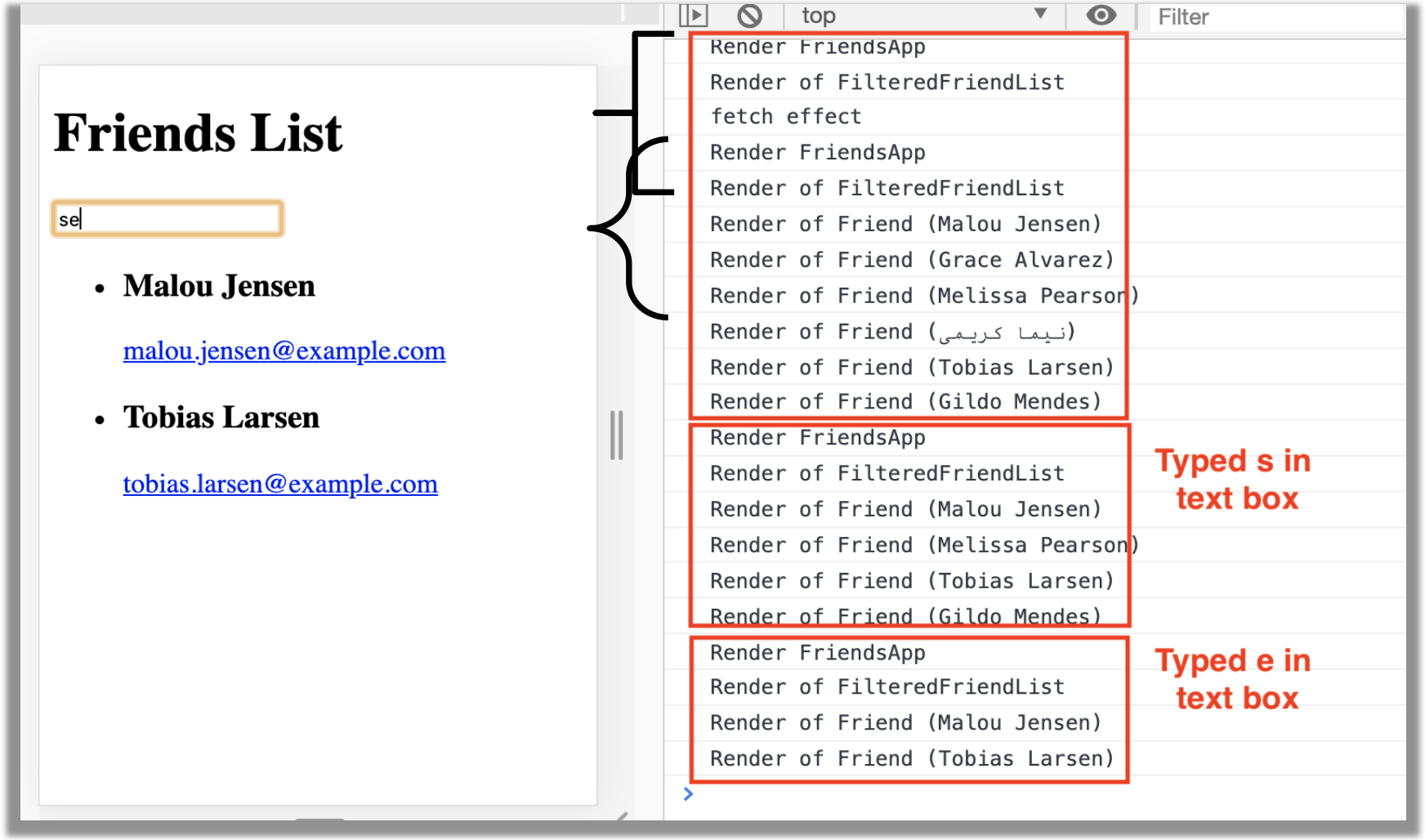
- **Incorrect approach:**

  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)



More to come ….

-