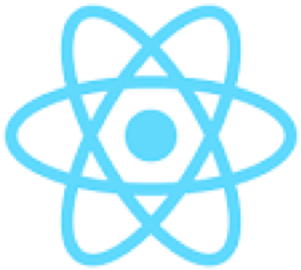


ReactJS.

The Component model

Topics

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

Component DATA

- **Two sources of data for a component:**
  1. **Props - Passed in to a component; Immutable; the** props **object.**
  2. *State* **- Managed internally by the component; Mutable; can be any data type (primitive, array, object)**
     - ***** The basis for dynamic and interactive Uis *****

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

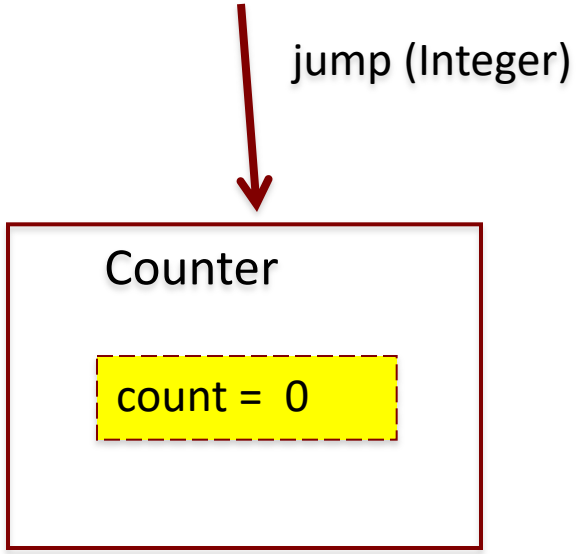
- **State-related features:**
  - **Initialization.**
  - **Mutation – setter method.**
    - **Performs an overwrite operation, not a merge.**
    - ***** Automatically causes component to re-render. *****

Component Data - Example

- **The Counter component.**
- **Ref.** 06_state.js (**from lecture archive)**

- **The useState() method:**
  - **A React hook.**
  - **Setter name arbitrary.**
  - **State variable (count) immutable.**

- **JS features:**
  - **Static function property, e.g.** defaultProps, proptypes



React's event system.

- **Cross-browser support.**
- **Event handlers receive** SyntheticEvent **– a cross-browser wrapper for the browser's native event.**
- **React event naming convention slightly different from 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 (incrementCounter) executed*

→ *state is changed (setCount())*

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

Modifying the DOM

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

- **Traditional performance best practice:**
  1. **Minimize access to the DOM.**
  2. **Avoid expensive DOM operations.**
  3. **Update elements offline, then reinsert into 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 developer from these concerns.**

The Virtual DOM

- **How React works:**
  1. **It create a lightweight, efficient form of the DOM – the Virtual DOM.**
  2. **Your app changes the V. DOM via components' JSX.**
  3. **React engine:**
     1. **Perform *diff* operation between current and previous V. DOM state.**
     2. **Compute 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*

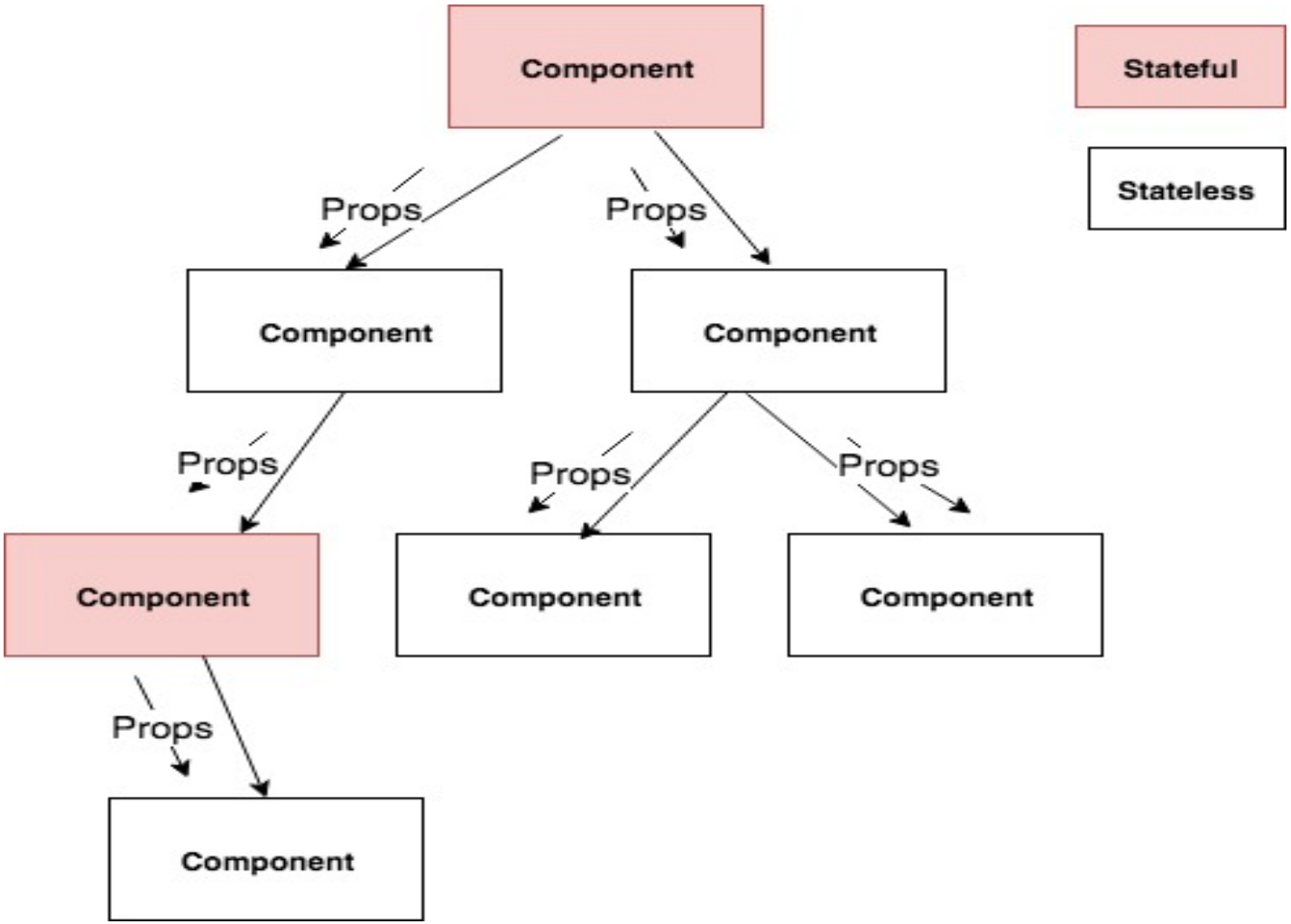
→ *React diffs the changes between the current and previous Virtual DOM*

→ *React batch updates the Real DOM*

Topics

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

Unidirectional data flow



Unidirectional data flow

- **In a React app data flows uni-directionally ONLY.**
  - **Most other SPA frameworks use two-way data binding.**

- **In a multi-component app a common pattern is:**
  - **A small subset (maybe only 1) of components will be stateful; the rest are stateless.**

- **Typical Stateful component execution flow:**
  1. **Calls *state setter method* to update its state variable(s).**
  2. **Re-renders automatically.**
  3. **Passes updated props to subordinate components.**
  4. *React guarantees subordinate components are also re-rendered with new (perhaps unchanged!!) props.*

Topics

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

React Hooks

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

- **React Hooks are:**
  1. **functions (some HOF);**
  2. **that allow us easily manipulate the state and manage the lifecycle of a component:**
  3. **(without needing to convert them into class components.)**

- **Examples: useState, useEffect, useContext, useRef, etc**
  - **Names muat start with 'use' for linting purposes.**

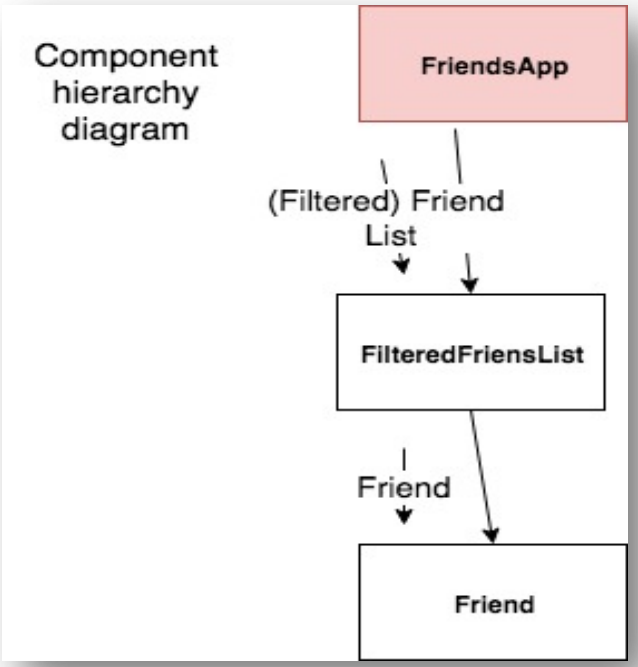
- **Usage rules:**
  1. **Only Call Hooks at the Top Level.**
     - **Don't call Hooks inside loops, conditions, or nested function.**
  2. **Only Call Hooks from React Functions.**
     - **Don't call Hooks from regular JavaScript functions.**

useEffect Hook

- useEffect **lets a component perform** side effects**.**
- **Side Effect example:**
  - **fetching some data from a web API.**
  - **Subscribe to browser events, e.g. window resize.**
- **Signature:** useEffect(callback, dependency array)
- **Execution times:**
  1. **On mounting.**
  2. **On every rendering,**
     - **Unless a dependency array is specified,**
       - **then only when a dependency variable changes value.**
  - **Use an** empty dependency array **to restrict execution to mount-time only.**

Sample App 1

(see lecture archive)



FriendsApp **component:**

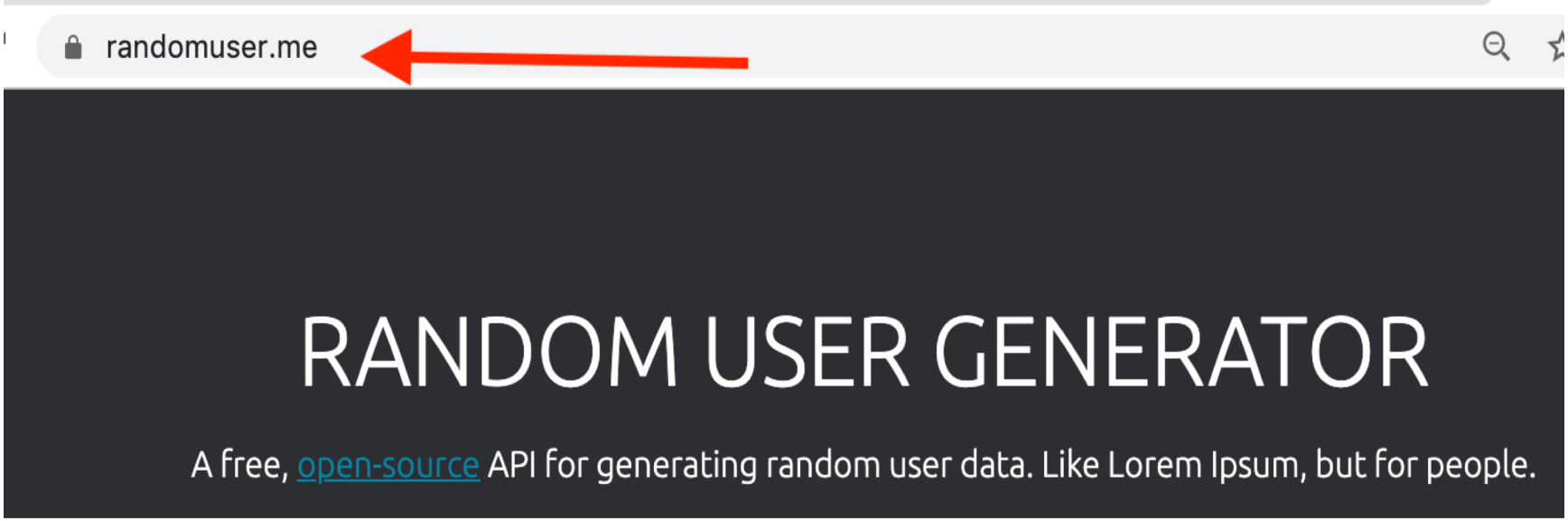
1. Manages app's state **(i.e. text box, full list of friends).**

2. Gets friends list (useEffect)

3. Computes matching friends.

4. Controls list re-rendering.

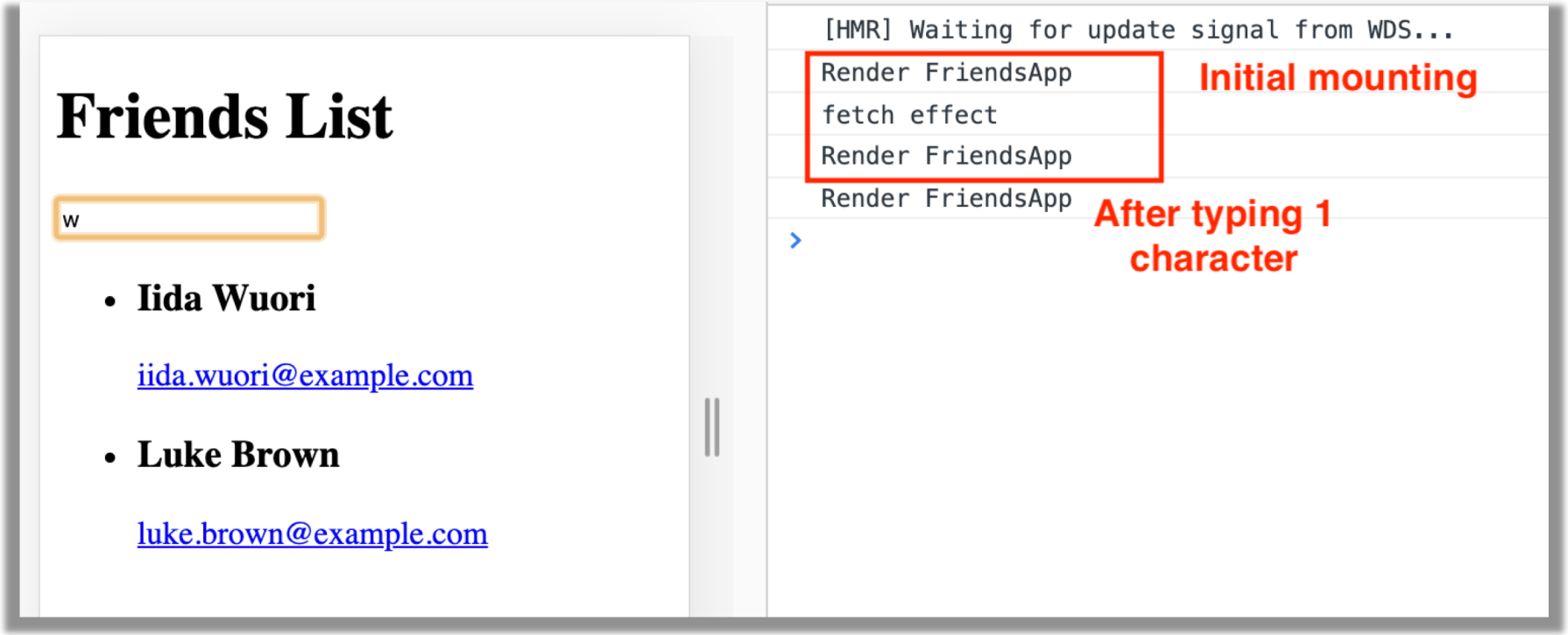
Recall – RandomUser open API



- We use it to provide a list od friends for the sample app

Sample App 1 - *useEffect* Hook

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



Sample App 1 (version 2)

- **App feature changes (**ref App2.js**):**
  1. **A 'Reset' button to trigger loading a new set of friends from API.**
  2. **Browser tab title shows # of matching friends (side effect).**

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

- **2 side effects:**
  1. **'Fetch API data' effect - dependent on** reset button **state change.**
  2. **'Set browser title' effect - dependent on matching list size change.**

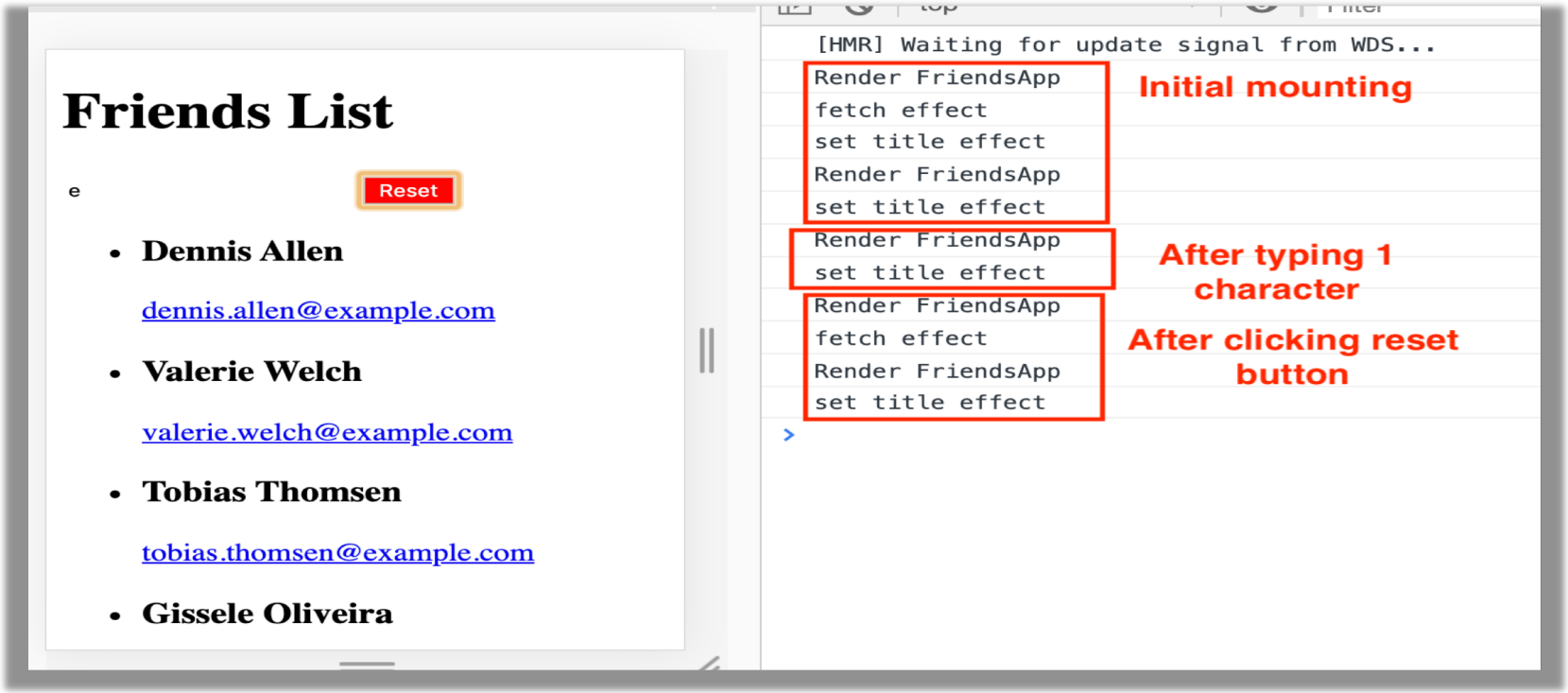
Sample App 1 (version 2) events.

- **On mounting of** FriensApp **component (Empty friends array):**
  Both effect execute → 'Fetch data' effect changes 'friends list' state → Component re-renders + 'Set browser ttile' effect executes.

- **On typing character in text box:**
  'Text' state change → FriendsApp **rerenders + Matching friends list changed → 'Set browser title' effect executes.**

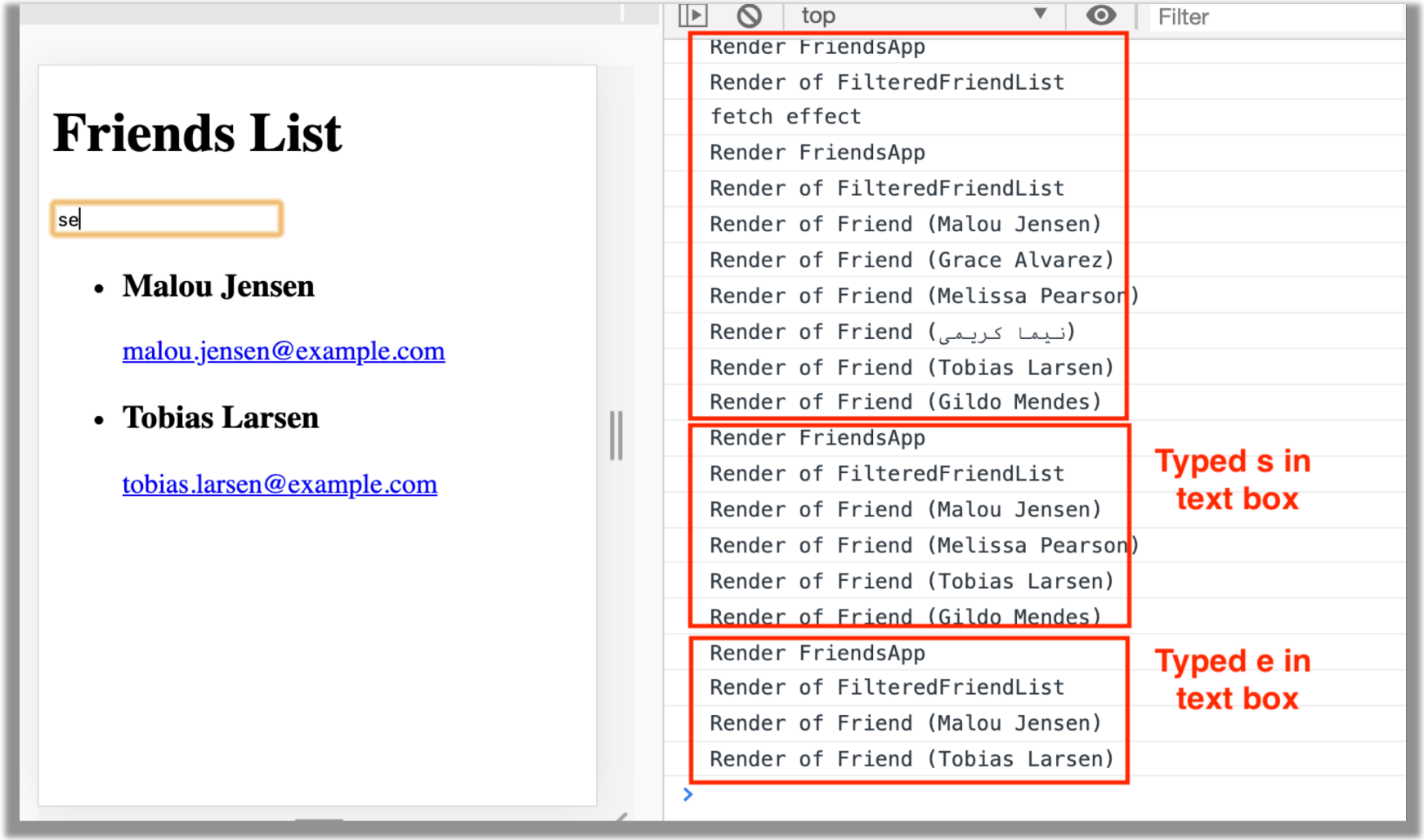
- **On clicking Reset button:**
  'Reset' state change → FriendsApp **rerenders → 'Fetch data' effect executes → 'Friends list' state changed → FriendsApp re-renders → 'Set browser title' effect executes**

Sample App 1 (version 2)



Unidirectional data flow & **Re-rendering**

(Sample app 1 – version 1)



Unidirectional data flow & **Re-rendering**

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

  User types a character in text box
  → *onChange* ***event handler executes***
  → ***Handler changes state (****FriendsApp* ***component)***
  → ***React re-renders*** *FriendsApp*
  → ***React re-renders children (****FilteredFriendList****) with new prop values.***
  → ***React re-renders children of*** *FilteredFriendList****.***
  (Re-rendering completed)
  → ***(Pre-commit phase) React computes the new Virtual DOM***
  → ***React diffs the new and previous Virtual DOMs***
  → ***(Commit phase) React batch updates the Real DOM.***
  → ***Browser repaints screen***

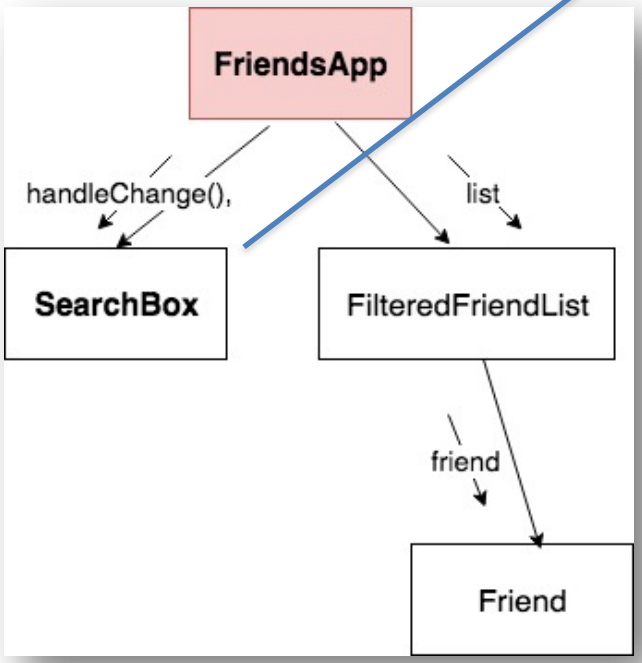
Topics

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

Sample App 2

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

- **What if a component's** state **is influenced by an** event **in a subordinate component?**
- **Solution: The** data down, action up pattern.



Data down, Action up

Pattern:

1. **Stateful component** (FriendsApp) **provides a callback to subordinate** (SearchBox)**.**
2. **Subordinate calls it when event** (onChange) **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} />;
};
```

Summary

- **Component state.**
  - **User input/interaction is 'recorder' in a component's state variable.**
  - **State changes cause component re-execution -> re-rendering.**
  - **Re-rendering (may) result in UI changes -> dynamic apps.**

- **Hooks – allows us manipulate state variables and hook into the lifecycle of a component.**
  - **useState, useEffect, etc**

- **React achieves DOM update performance improvements by managing an intermediate data structure, the Virtual DOM.**

- **Data only flows downward through the component hierarchy – this aids debugging.**
  - **Actions (Events) flow upwards**

ES6 top-up.

The spread operator (…)

- *Spread* - ***A new operator that takes a collection data structure and separates it into its individual elements.***

```
{
  // Inserting arrays
  const array1 = [3, 4];
  let array2 = [1, 2, array1, 5, 6];
  console.log(array2); // [ 1, 2, [ 3, 4 ], 5, 6 ]
  array2 = [1, 2, ...array1, 5, 6];
  console.log(array2); // [ 1, 2, 3, 4, 5, 6 ]
}
```

```
{
  // Copy an array
  const array1 = ["a", "b", "c"]
  const array2 = array1  // Not a copy
  array2.push("d")
  console.log(array1)  // [ 'a', 'b', 'c', 'd' ]
  const array3 = [...array1]  // Is a copy
  array3.push("e")
  console.log(array1)  // [ 'a', 'b', 'c', 'd' ]
}
```

Spread on Objects.

```
{
  // Insert one object's elements into another
  const partOfMe = { first: "Diarmuid", address: "1 Main street" };
  let allMe = { surname: "O Connor", partOfMe, employer: "WIT" };
  console.log(allMe);
  allMe = { surname: "O Connor", ...partOfMe, employer: "WIT" };
  console.log(allMe);
}
```

```
{
  surname: 'O Connor',
  partOfMe: { first: 'Diarmuid', address: '1 Main street' },
  employer: 'WIT'
}
{
  surname: 'O Connor',
  first: 'Diarmuid',
  address: '1 Main street',
  employer: 'WIT'
}
```

```
{
  // Trick for updating an object's properites
  const me = {
    surname: "O Connor",
    first: "Diarmuid",
    address: "1 Main street",
    employer: "WIT",
  };
  const updatedMe = { ...me, address: "2 High Street" };
  console.log(me);
  console.log(updatedMe);
}
```

```
{
  surname: 'O Connor',
  first: 'Diarmuid',
  address: '1 Main street',
  employer: 'WIT'
}
{
  surname: 'O Connor',
  first: 'Diarmuid',
  address: '2 High Street',
  employer: 'WIT'
}
```

The Rest operator (…)

- *Spread – When used in an assignment target, it gathers elements into a collection.*

```
{
    // The Rest operator
    const array = ["a", "b", "c", "d", "e"]
    const [first,second, ...rest] = array
    console.log(rest) // [ 'a', 'b', 'c', 'd' ]
}
```

```
// Rest with objects
const me = {
    surname: "O Connor",
    first: "Diarmuid",
    address: "1 Main street",
    employer: "WIT",
}
const { address, ...rest  } = me
console.log(address)
console.log(rest)
```

```
1 Main street
{ surname: 'O Connor', first: 'Diarmuid', employer: 'WIT' }
```