

ReactJS.

The Component model

Topics

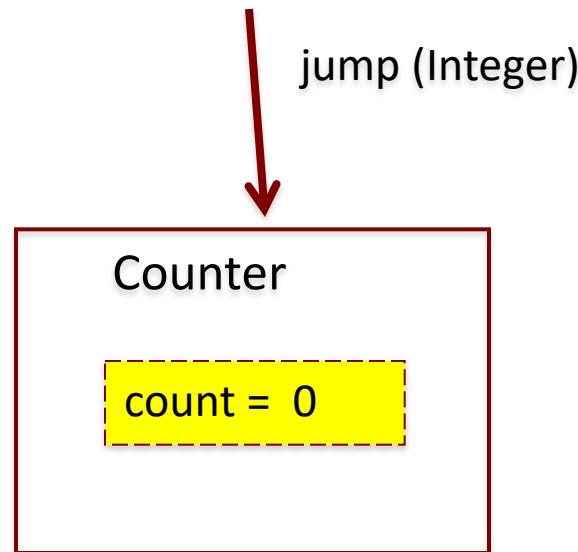
- **Component State.**
 - Basis for dynamic, interactive UI.
- **The Virtual DOM.**
- **Data Flow patterns.**
- **Hooks.**

Component DATA

- **Two sources of data for a component:**
 1. **Props - Passed in to a component; Immutable; an object (props).**
 2. **State - Managed internally by the component; Mutable; any 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.** samples/06_state.js
- **useState:**
 - **A React hook.**
 - **Setter name arbitrary.**
 - **State variable (count) immutable.**
- **JS features:**
 - **Static function property,**
e.g. defaultProps.



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; mirrors the state of the UI; always in sync.
- 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?
 1. Create a lightweight, efficient form of the DOM – the Virtual DOM.
 2. React app changes V DOM
 3. React engine:
 1. Perform *diff* operation between current and previous V DOM state.
 2. Compute the minimal set of changes to apply to (real) DOM.
 3. Batch execute all updates to real DOM.
- Benefits:
 - a) Clean – Clean, descriptive programming model.
 - b) Fast - Optimized DOM updates and reflows.

Automatic Re-rendering (detail)

- EX.: The Counter component.

User clicks button

→ *onClick event handler executed*

→ *state is changed*

→ *component function re-executed*

→ *The Virtual DOM has changed*

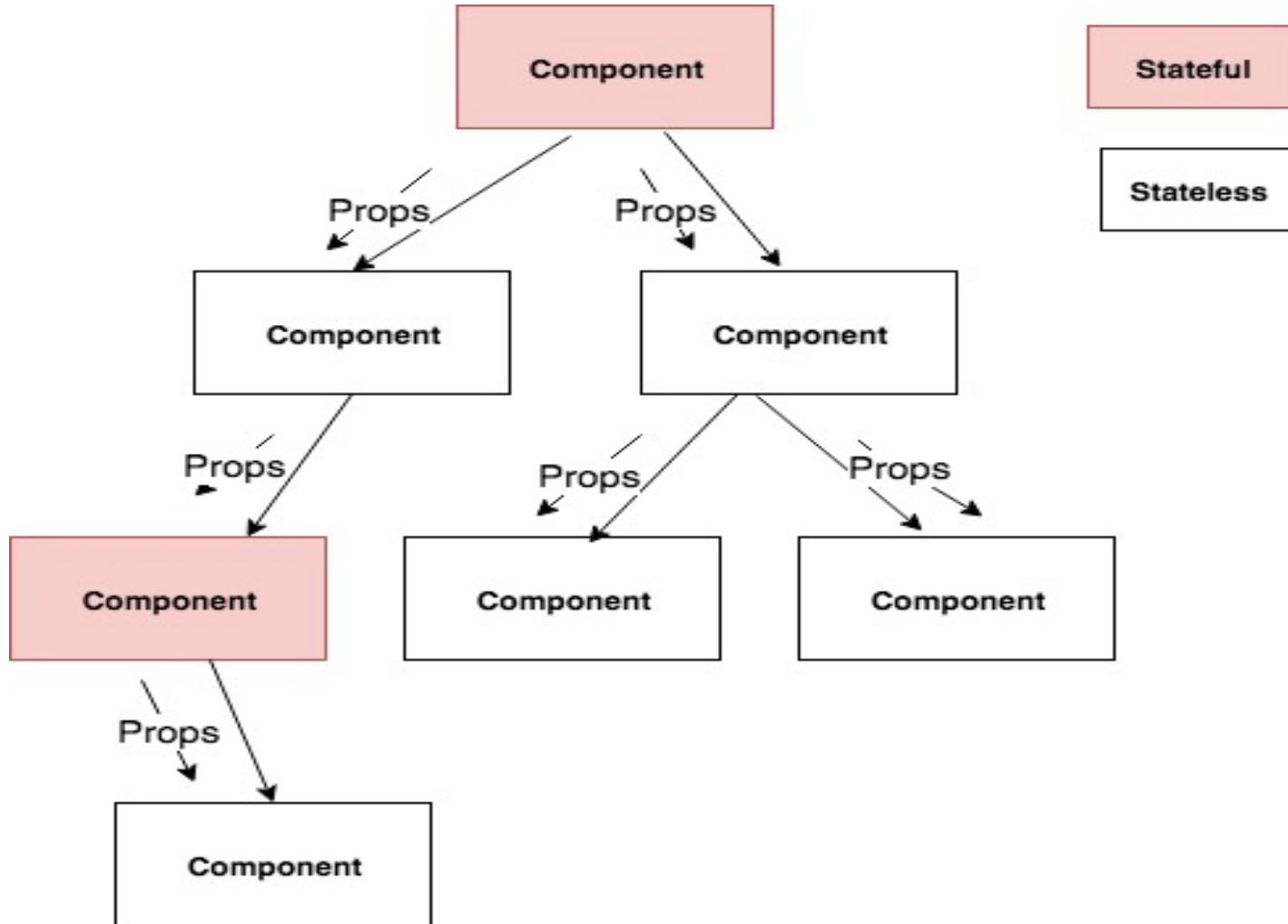
→ *React diffs the changes (between the current and previous Virtual DOM)*

→ *React batch updates the Real DOM*

Topics

- Component State. ✓
- The Virtual DOM. ✓
- Data Flow patterns.
- Lifecycle methods.

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.
- Stateful components:
 - Calls *state setter method* to update its state.
 - Re-renders itself automatically.
 - Passes updated props to subordinate components.
 - React guarantees subordinate components are also re-rendered with updated (perhaps unchanged!!) props.

Topics

- Component State. ✓
- The Virtual DOM. ✓
- Data Flow patterns. ✓
- Hooks

React Hooks

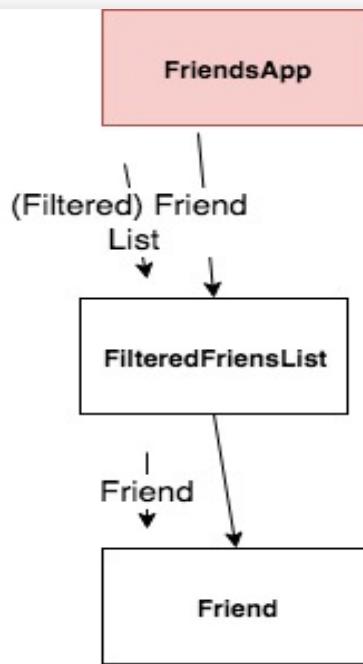
- **Introduced in version 16.8.0 (February 2019)**
- **React Hooks are:**
 1. functions (HOF).
 2. that allow us to easily manipulate the state of our functional component.
 3. without needing to convert them into class components.
- **Examples: useState, useEffect, useContext, useRef, etc**
 - Names must 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 functions
 2. Only Call Hooks from React Functions
 - Don’t call Hooks from regular JavaScript functions.

useEffect Hook

- The **useEffect** lets us perform side effects from a component.
- **Side Effect example:**
 - fetching some data from an API
 - listening for upcoming browser events.
- **Signature:** `useEffect(callback, dependency array)`
- **Execution times:**
 - On mounting.
 - On every rendering,
 - Unless a dependency array is specified, then
 - only when a dependency variable changes value.
 - Use an empty dependency array to restrict to mount time only.

Sample App

Component hierarchy diagram



FriendsApp component:

1. Manages app's state (i.e. text box, full list of friends).
2. useEffect hook to get API data
3. Computes matching friends
4. Controls list re-rendering.

Friends List

Search

- Joe Bloggs

jbloggs@here.con

- Paula Smith

psmith@here.con

- Catherine Dwyer

cdwyer@here.con

- Paul Briggs

pbriggs@here.con

useEffect Hook

- **useEffect runs AFTER component has mounted.**
 - First rendering occurs BEFORE data is available.
 - Must allow for this in implementation.

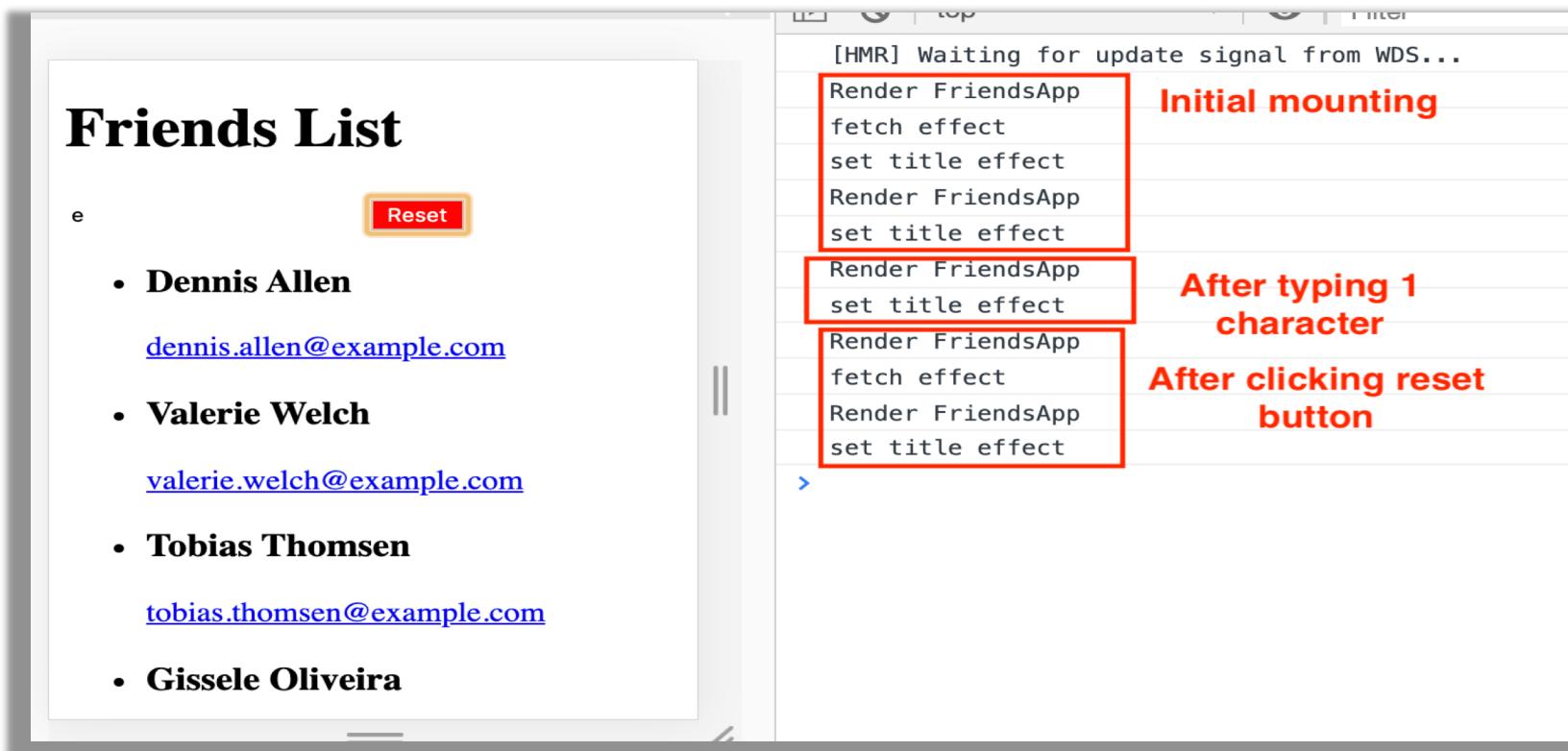
The screenshot shows a 'Friends List' application interface on the left and a developer tools timeline on the right. The application displays two friends: Iida Wuori and Luke Brown, each with an email link. The developer tools timeline shows the following sequence:

- [HMR] Waiting for update signal from WDS...
- Render FriendsApp
- fetch effect
- Render FriendsApp
- Render FriendsApp

A red box highlights the first three entries: 'Render FriendsApp', 'fetch effect', and 'Render FriendsApp'. To the right of this box, the text 'Initial mounting' is written in red. Below the timeline, a blue arrow points to the fourth 'Render FriendsApp' entry, with the text 'After typing 1 character' written in red to its right.

useEffect Hook

- App changes (ref App2.js):
 1. Reset button to triggers load more data/friends from API.
 2. Browser tab title shows # of matching friends (side effect)
- 'Fetch' effect dependent on reset button state change
- 'Set title' effect dependent on matching list size change



Unidirectional data flow & Re-rendering

The screenshot shows a development environment with two main panes. On the left is a component tree for a 'Friends List' application, and on the right is a list of rendered components.

Component Tree (Left):

- Friends List
 - Malou Jensen
 - malou.jensen@example.com
 - Tobias Larsen
 - tobias.larsen@example.com

Rendered Components (Right):

- Render FriendsApp
- Render of FilteredFriendList
- fetch effect
- Render FriendsApp
- Render of FilteredFriendList
- Render of Friend (Malou Jensen)
- Render of Friend (Grace Alvarez)
- Render of Friend (Melissa Pearson)
- Render of Friend (نیما کریمی)
- Render of Friend (Tobias Larsen)
- Render of Friend (Gildo Mendes)
- Render FriendsApp
- Render of FilteredFriendList
- Render of Friend (Malou Jensen)
- Render of Friend (Melissa Pearson)
- Render of Friend (Tobias Larsen)
- Render of Friend (Gildo Mendes)
- Render FriendsApp
- Render of FilteredFriendList
- Render of Friend (Malou Jensen)
- Render of Friend (Tobias Larsen)

Annotations:

- A red box highlights the first group of rendered components (up to 'Render of Friend (Gildo Mendes)'). A red arrow points from the text 'Typed s in text box' to this box.
- A red box highlights the second group of rendered components (from 'Render FriendsApp' to 'Render of Friend (Gildo Mendes)'). A red arrow points from the text 'Typed e in text box' to this box.

Unidirectional data flow & Re-rendering

- What happens when user types in text box?

User types a character in text box

→ *onChange event handler executes*

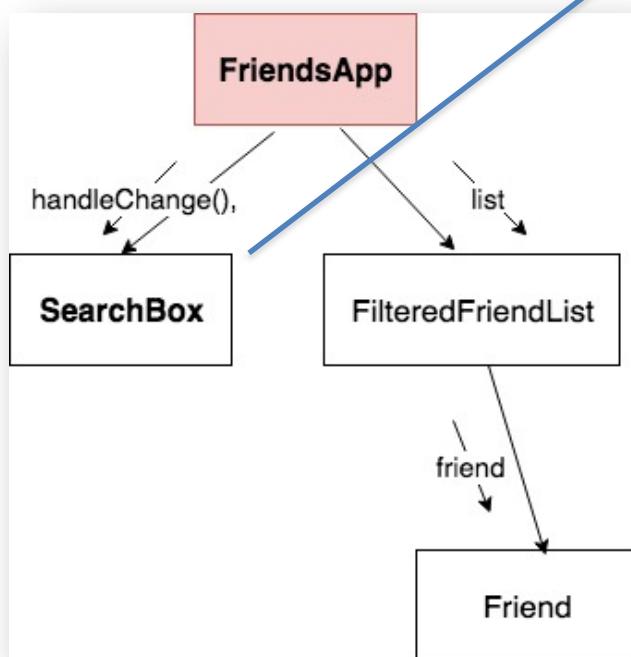
- *Handler changes state (FriendsApp component)*
 - *React re-renders FriendsApp*
 - *React re-renders children (FilteredFriendList) with new prop values.*
 - *React re-renders children of FilteredFriendList.
(Re-rendering completed)*
 - *(Pre-commit phase) React computes the new Virtual DOM*
 - *React 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) ✓

Inverse data flow

- What if a component's state is influenced by an event in a subordinate component?
- Solution: The inverse data flow pattern.



Inverse data flow

Pattern: Stateful component (FriendsApp) provides a callback for subordinate (SearchBox) to call when event (onChange) occurs

```
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}>;
};
```

```
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} />
    </>
  );
}
```

Summary

- **Component state.**
 - User's input/interaction is 'recorder' in a component's state variable.
 - State changes cause component re-execution -> re-rendering
 - Re-rendering (may) results in UI changes -> dynamic apps.
- **Hooks – allows us manipulate the 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.**

