React Client-Components

Outline

1. State
2. Components Communication
3. React Tools and Component Libraries

State

Component State

- A component can store its own local data (**state**)
  - Private and fully controlled by the component
  - Can be passed as **props** to children
- Use `useState` hook to create a ***state variable*** and an **associated *function*** to update the state

```
const [count, setCount] = useState(0);
```

`useState` returns a state variable ***`count`*** initialized with 0 and a function ***`setCount`*** to be used to update it

  - Calling ***`setCount`*** causes React to **re-render the app components** and **update the DOM** to reflect the state changes

Never change the state directly by assigning a value to the state variable => otherwise React will NOT re-render the UI

State

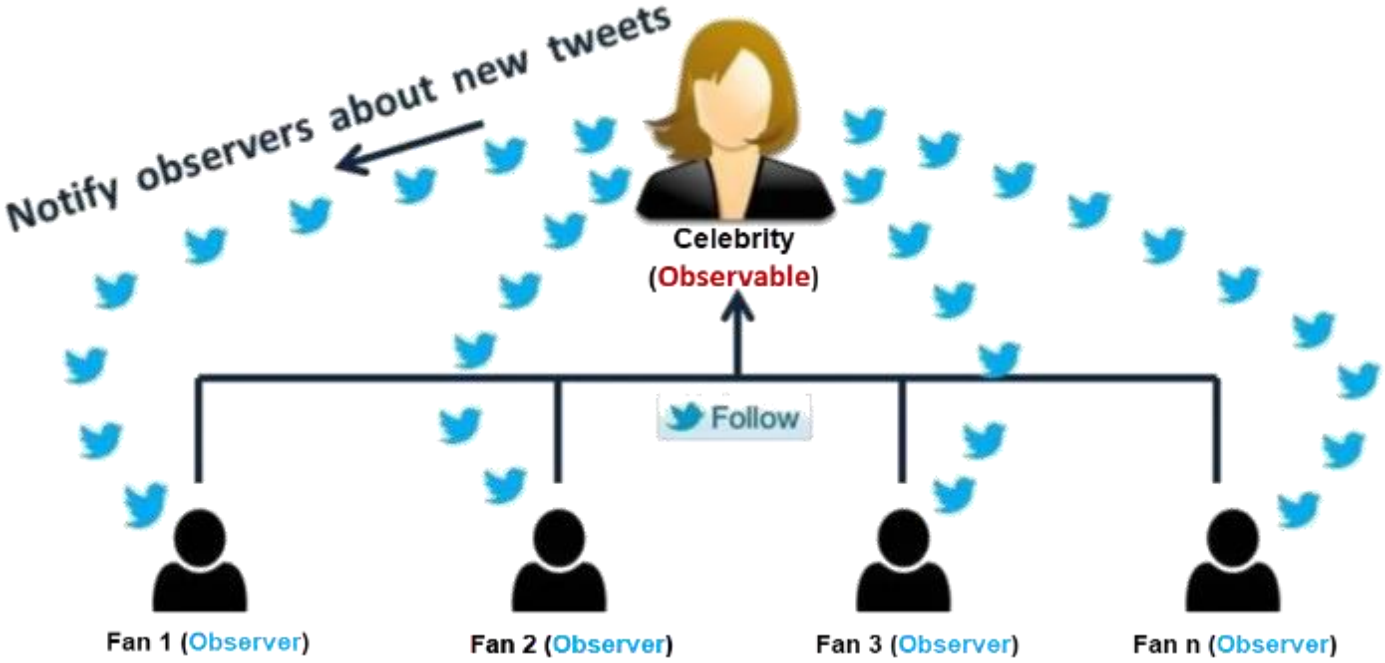
- State = any value that can change overtime
- State variable must be declared using **useState** hook to act as **Change Notifiers**
  - They **are observed** by the React runtime
  - Any change of a state variable will trigger the **re-rendering** of any functions that **reads** the state variable
  - Both props and state changes trigger a render update

=> UI is **auto-updated** to reflect the updated app state

Observer Pattern at the heart of Jetpack Compose

Observer Pattern Real-Life Example: A celebrity who has many fans on Tweeter

- Fans want to get all the latest updates (posts and photos)
- Here fans are **Observers** and celebrity is an **Observable** (analogous **state variable in React**)
- **A State variable** is an **observable data holder**: React runtime **observes its changes** and updates the Ul accordingly



Imperative UI vs. Declarative UI

- Imperative UI – manipulate DOM to change its internal state / UI

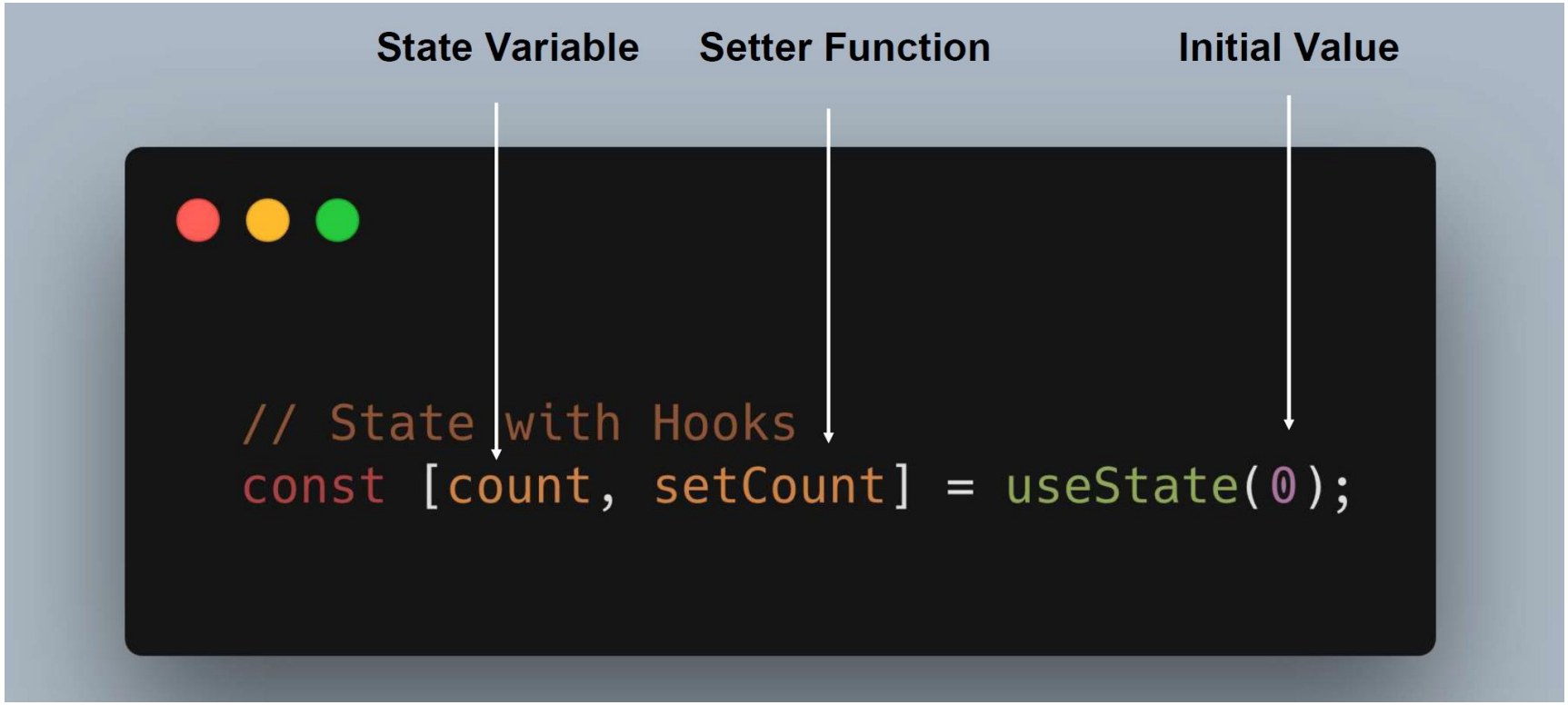
```
document.querySelector('#bulbImage').src = 'images/bulb-on.png';
document.querySelector('#switchBtn').value = "Turn off";
```

UI in React is immutable

- In react you should NOT access/update UI elements directly (as done in the imperative approach)
- Instead update the UI is by updating the state variable(s) used by the UI elements – this triggers automatic UI update
  - E.g., change the bulb image by updating the ***isBulbOn*** state variable

```
<input type="button"
       value= {isBulbOn ? "Turn off" : "Turn on"}
       onClick={() => setIsBulbOn(!isBulbOn)} />
```

useState Hook



Component with State + Events Handling

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

function Counter({ startValue }) {
    const [count, setCount] = useState(startValue);
    const increment = () => { setCount(prev => prev + 1); };

    const decrement = () => { setCount(prev => prev = - 1); };

    return <div>
            Count: {count}
            <button type="button" onClick={increment}>+</button>
            <button type="button" onClick={decrement}>-</button>
        </div>
}
export default Counter;
```

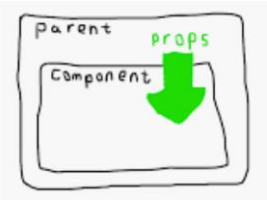
Count: 4 + -

Handling events is done the way events are handled on DOM elements

- Use the **Counter** component

```
<Counter startValue={3}/>
```

Uni-directional Data Flow: Props vs. State



- **Props** = data passed to the child component from the parent component

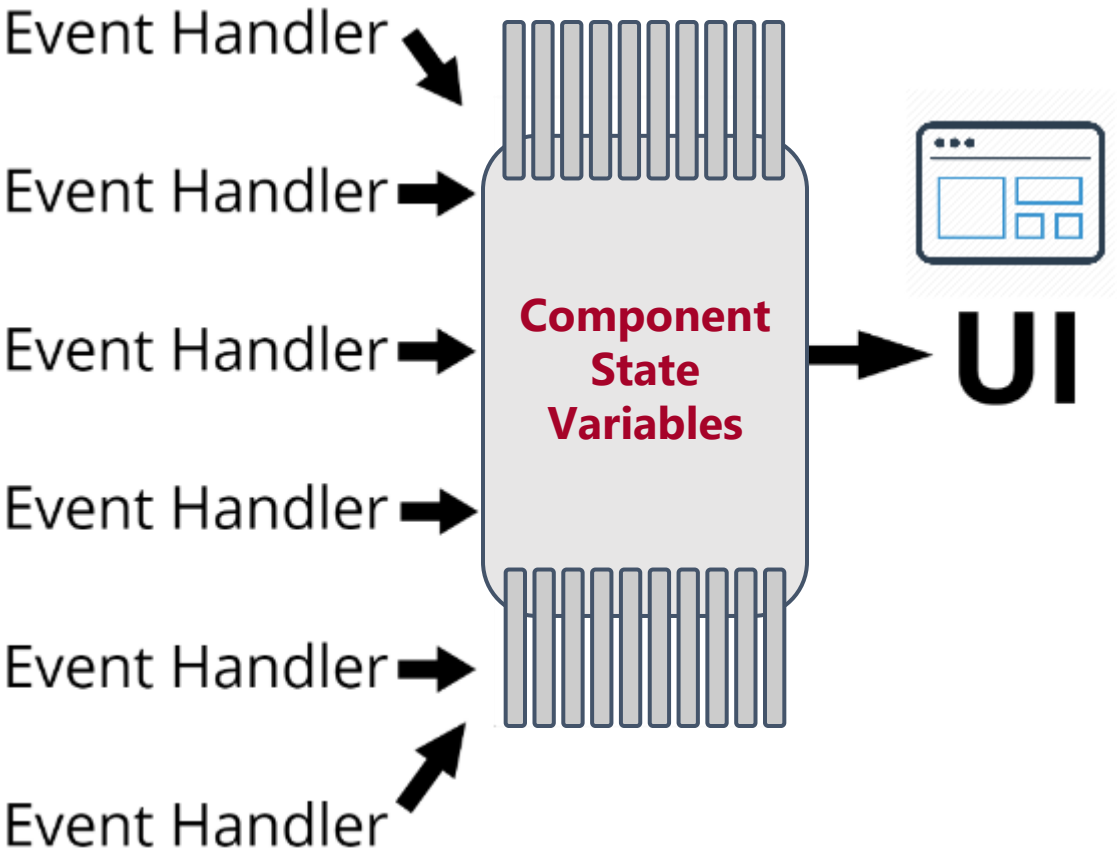
- **Props** parameters are **read only**

- **State** = internal data managed by the component (cannot be accessed and modified outside of the component)

- **State** variables are **Private** and **Modifiable** inside the component only (through **set** functions returned by useState)

👍 **React automatically re-render the UI whenever state or props are updated**

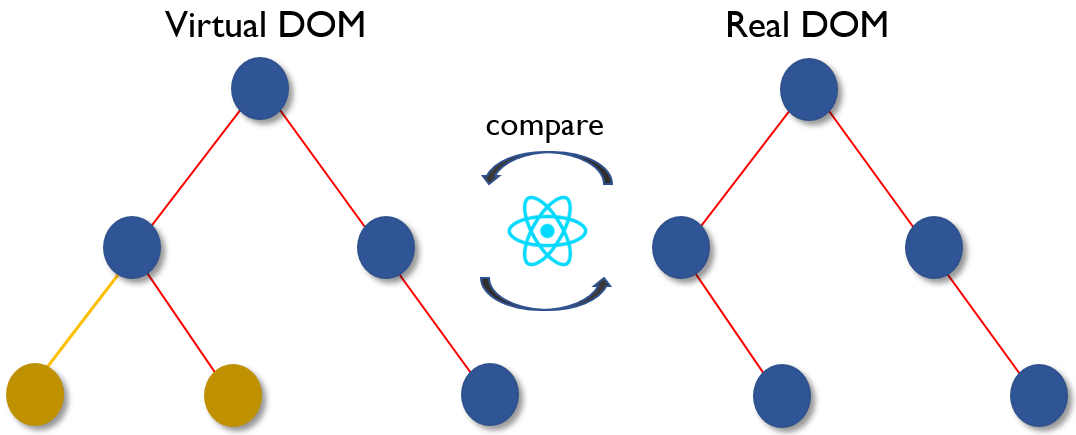
Event Handlers update the State and Reacts updates the UI



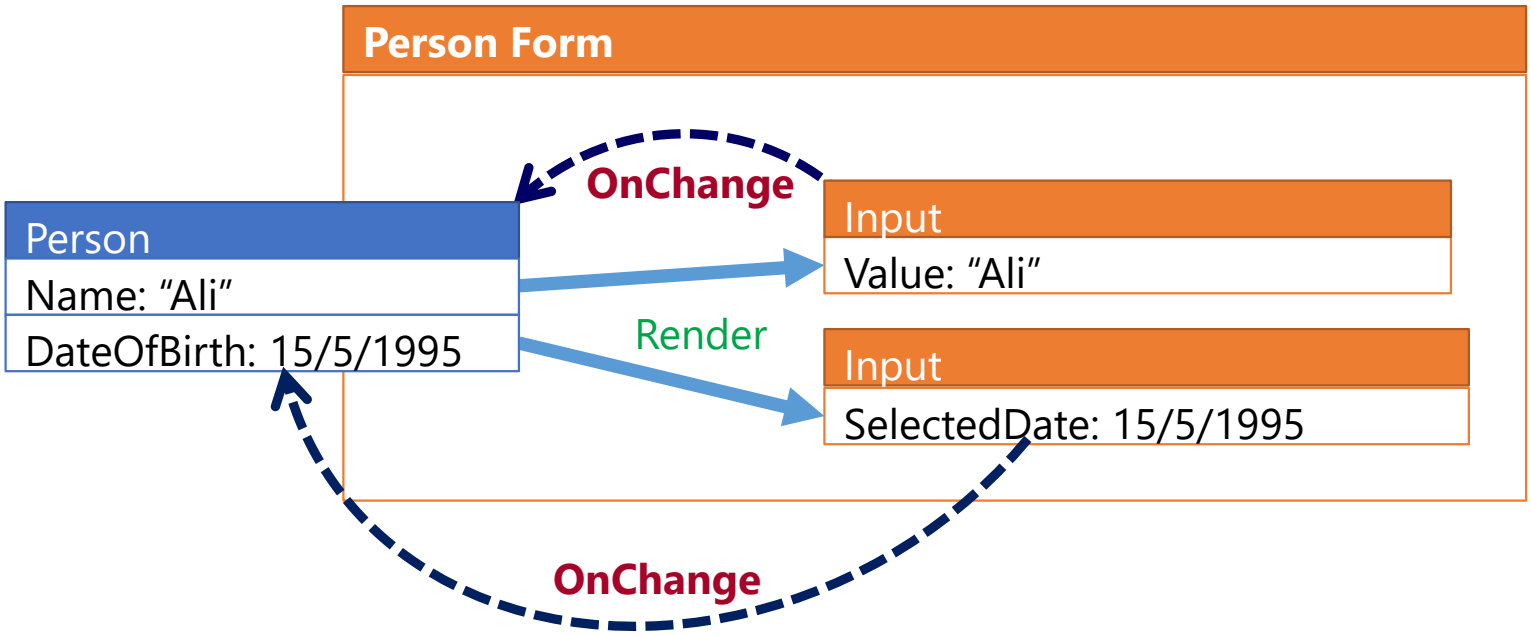
Every place a state variable is displayed is guaranteed to be auto-updated

Virtual DOM

- Virtual DOM = Pure JavaScript lightweight DOM, totally separate from the browser's slow JavaScript/C++ DOM API
- Every time the component **updates its state** or **receives new data via props**
  - A new virtual DOM tree is generated
  - New tree is **diffed** against old…
  - …producing a minimum set of changes to be performed on real DOM to bring it up to date



Unidirectional Data Flow in Forms



Common Events: `onClick - onSubmit – onChange`

Forms with React

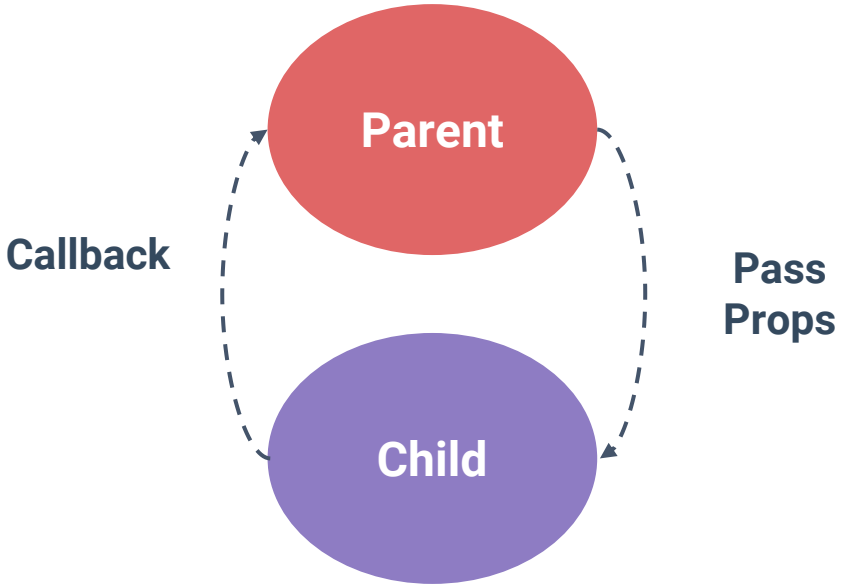
Form UI

```
<form onSubmit={handleSubmit}>
    <input
        name="email"
        type="email" required
        value={state.user}
        onChange={handleChange} />
    <input
        name="password"
        type="password" required
        value={state.password}
        onChange={handleChange} />
    <input type="submit" />
</form>
```

Form State and Event Handlers

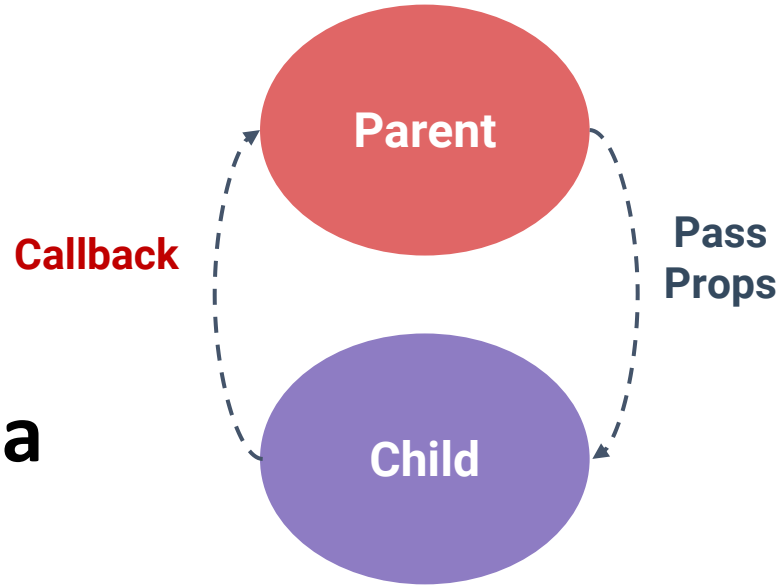
```
const [state, setState] = useState({ email: "", password: "" });

const handleChange = e => {
    const name = e.target.name;
    const value = e.target.value;
    //Merge the object before change with the updated property
    setState({ ...state, [name]: value });
};

const handleSubmit = e => {
    e.preventDefault();
    alert(JSON.stringify(state));
};
```

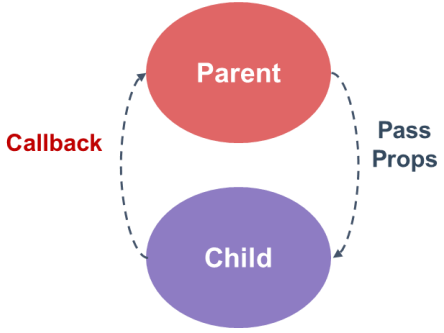
Components Communication



Composing Components

- Components are meant to be used together, most commonly in parent-child relationships
- Parent passes data down to the child via **props**
- The child notify its parent of **a state change via callbacks** (a parent must pass the child a callback as a parameter)





Parent-Child Communication

Parent

```
function Main => <Counter startValue={3}
        onChange={count => console.log(`Count from the child component: ${count}`)}/>
```

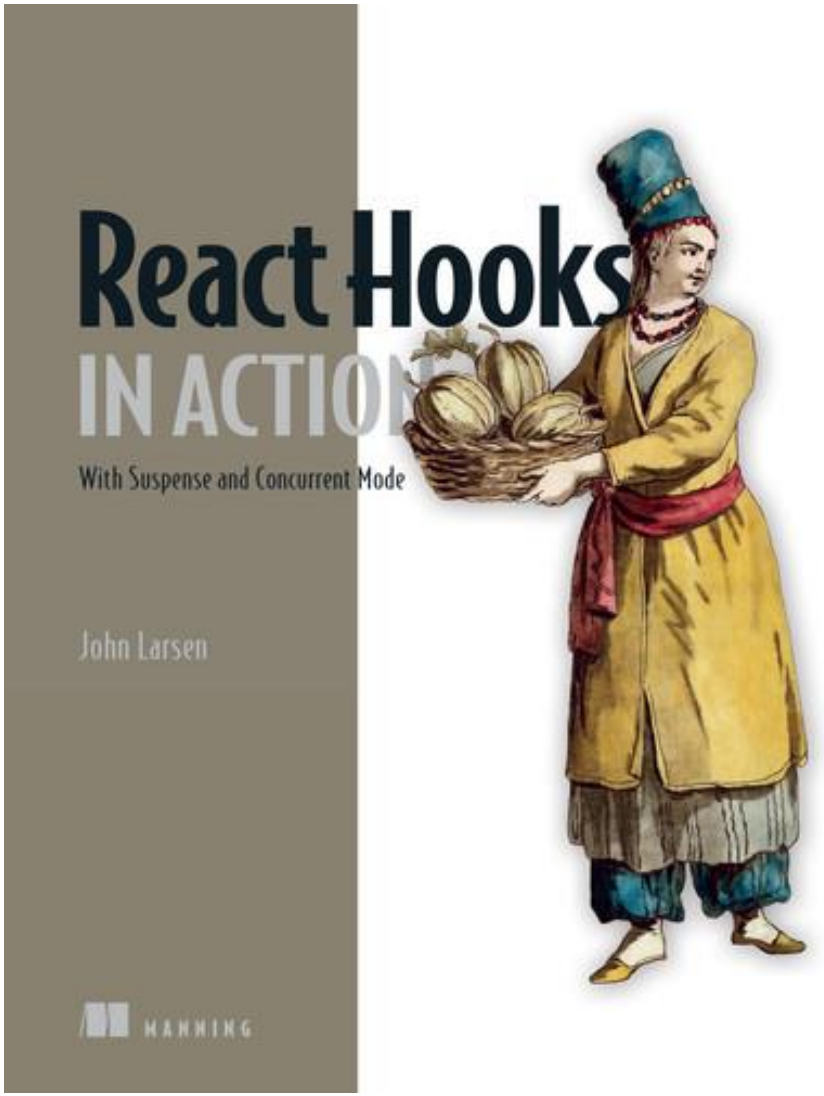
Child

```
function Counter(props) {
    const [count, setCount] = useState(props.startValue);

    const increment = () => {
        const updatedCount = count + 1;
        setCount(updatedCount);
        props.onChange(updatedCount);
    };

    return <div>
        Count: {count}
        <button type="button" onClick={increment}>+</button>
    </div>
}
```

Outline

1. useState
2. useEffect
3. useRef
4. useReducer
5. useContext

Slides are based on



https://learning.oreilly.com/library/view/react-hooks-in/9781617297632/

What is Hook?

- A Hook is a special function that lets you **hook** into React features such as state and lifecycle methods
- There are 3 rules for hooks:
  - Hooks can only be called at the top level of a component
  - Hooks cannot be conditional

Common Hooks

useState: creates a state variable

- Used for basic state management inside a component

```
const [state, setState] = useState(initialState)
```

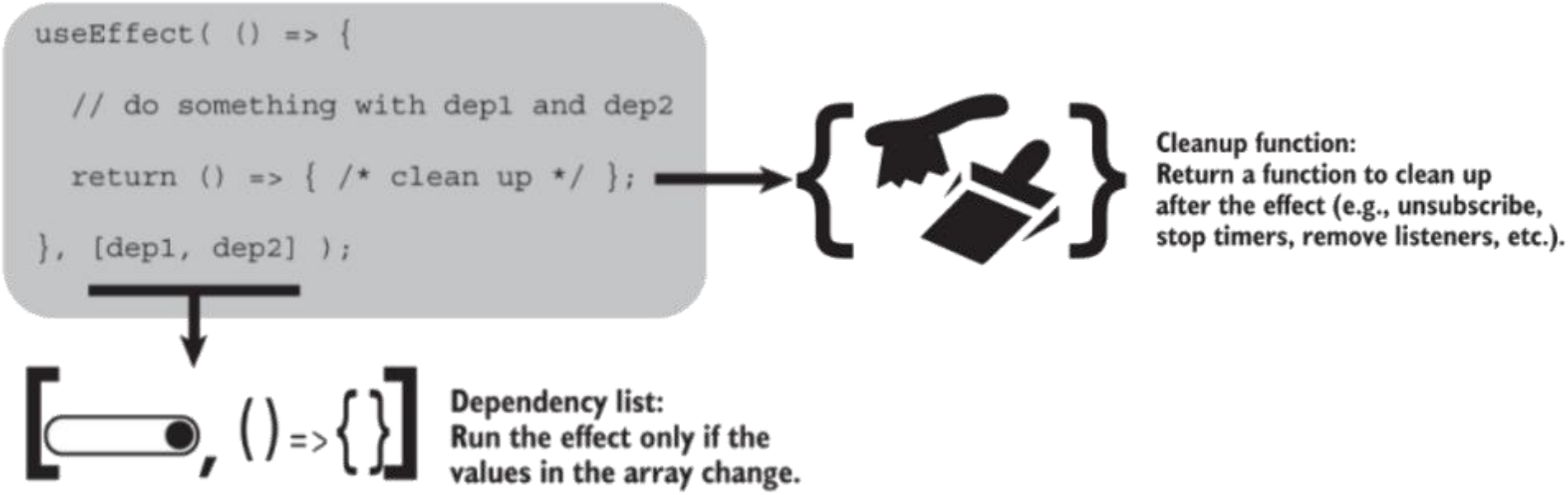
The name of your state

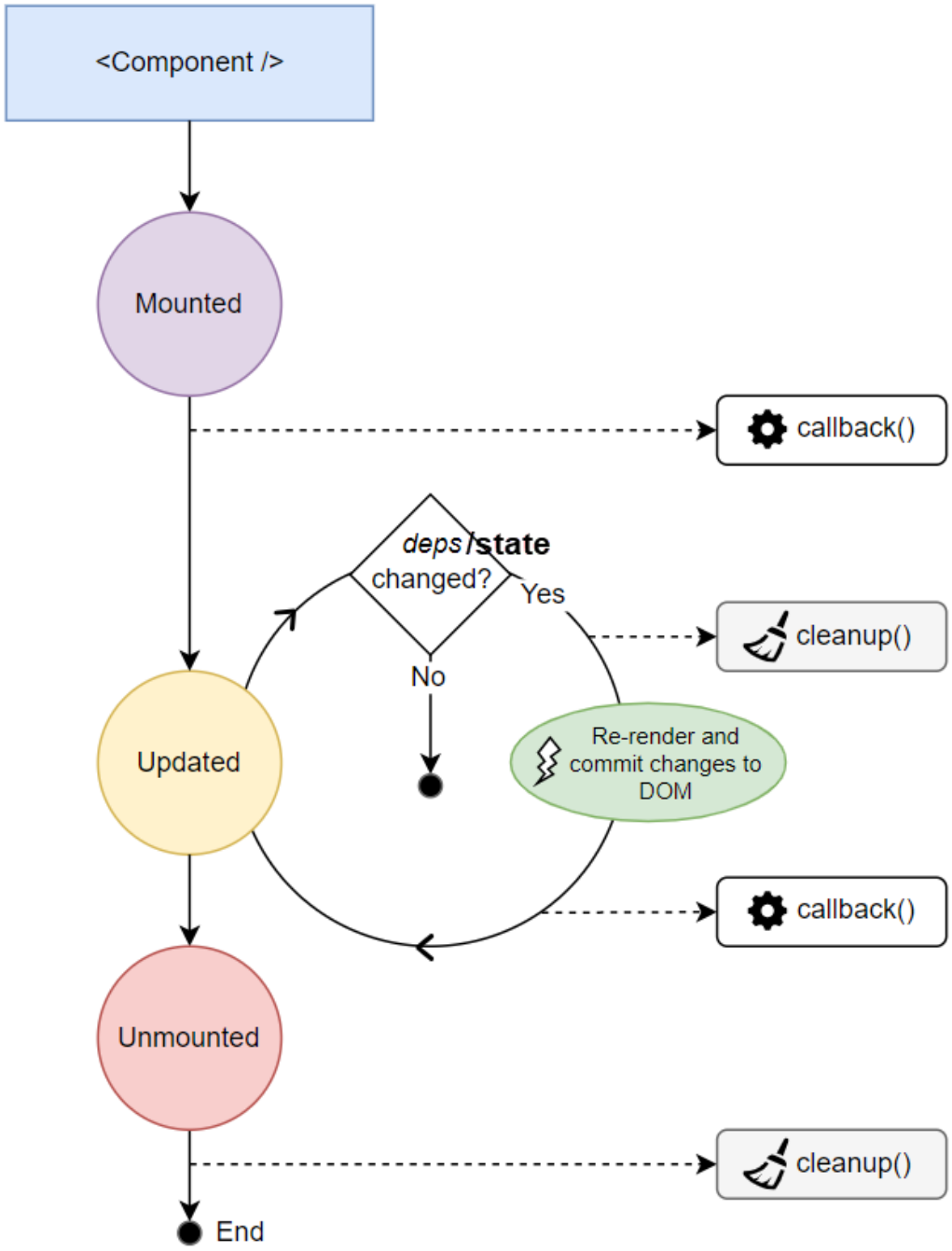
The function you'll eventually use to change the value of this state

The initial value of your state

useEffect

- For doing stuff when a component is mounts/unmounts/updates
- Ideal for fetching data when the component is mounted





A) After initial rendering, `useEffect()` invokes the callback having the side-effect. cleanup function is not invoked

B) On later renderings, before invoking the next side-effect callback, `useEffect()` invokes the cleanup function from the previous side-effect execution (to clean up everything after the previous side-effect), then runs the current side-effect

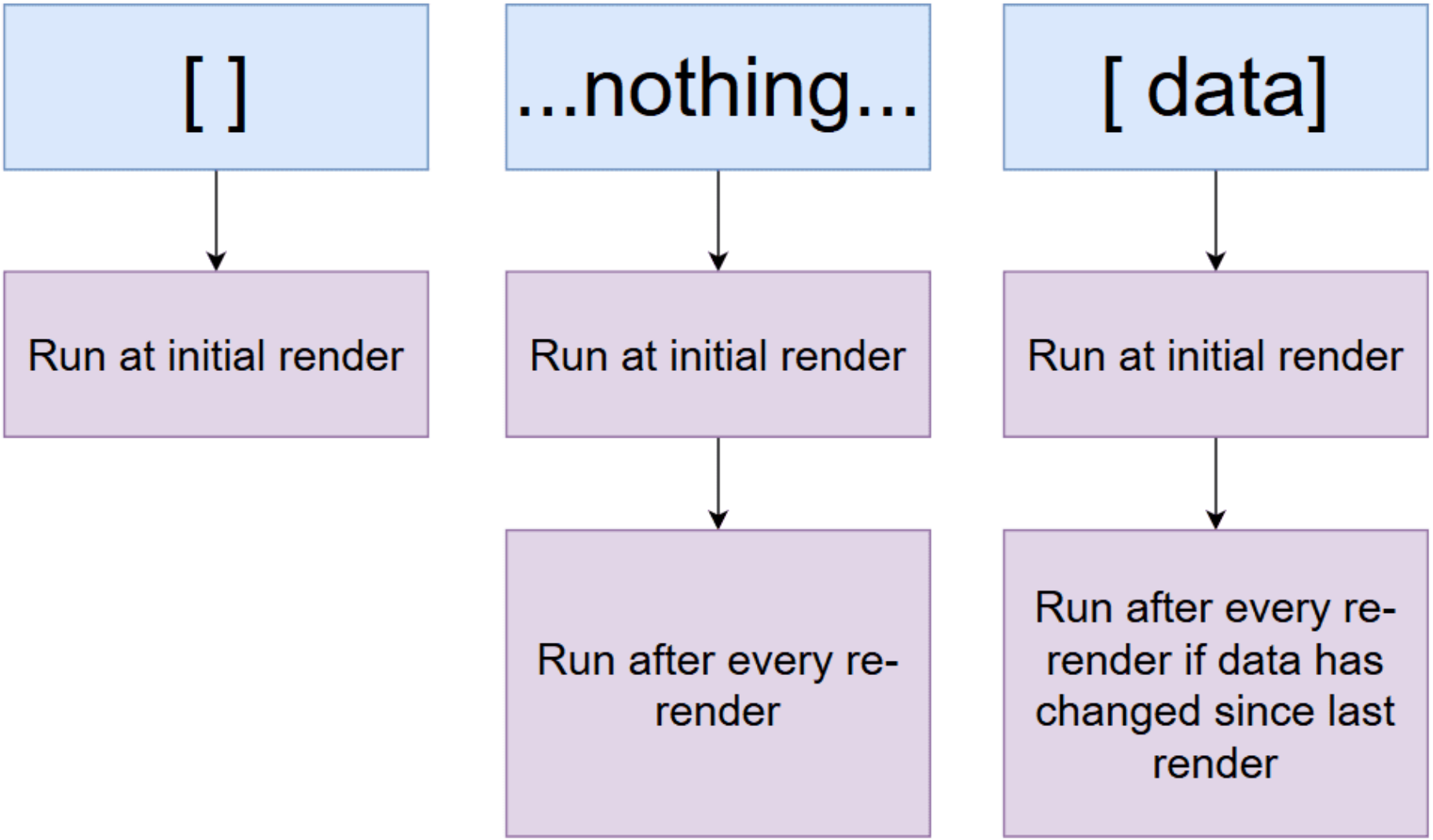
C) Finally, after unmounting the component, `useEffect()` invokes the cleanup function from the latest side-effect

Common side effects

Common side effects include:

- Setting the page title imperatively
- Working with timers like `setInterval` or `setTimeout`
- Logging messages to the console or other service
- Fetching data or subscribing and unsubscribing to services
- Setting or getting values in local storage

useEffect - 2nd argument



Use cases for the useEffect hook

Call pattern	Code pattern	Execution pattern

No second argument	`useEffect(() => {`
	`  // perform effect`
	`});`
		Run after every render.

Empty array as second argument	`useEffect(() => {`
	`  // perform effect`
	`}, []);`
		Run once, when the component mounts.

Dependency array as second argument	`useEffect(() => {`
	`  // perform effect`
	`  // that uses dep1 and dep2`
	`}, [dep1, dep2]);`
		Run whenever a value in the dependency array changes.

Return a function	`useEffect(() => {`
	`  // perform effect`
	`  return () => {/* clean-up */};`
	`}, [dep1, dep2]);`
		React will run the cleanup function when the component unmounts and before rerunning the effect.

useEffect – Executes code during Component Life Cycle

- **Initialize state data** when the component loads

```
useEffect(() => {
    async function fetchData() {
        const url = "https://api.github.com/users";
        const response = await fetch(url);
        setUsers( await response.json() ); } // set users in state
        fetchData();
}, [ ]); // pass empty array to run this effect once when the component is first mounted to the DOM.
```

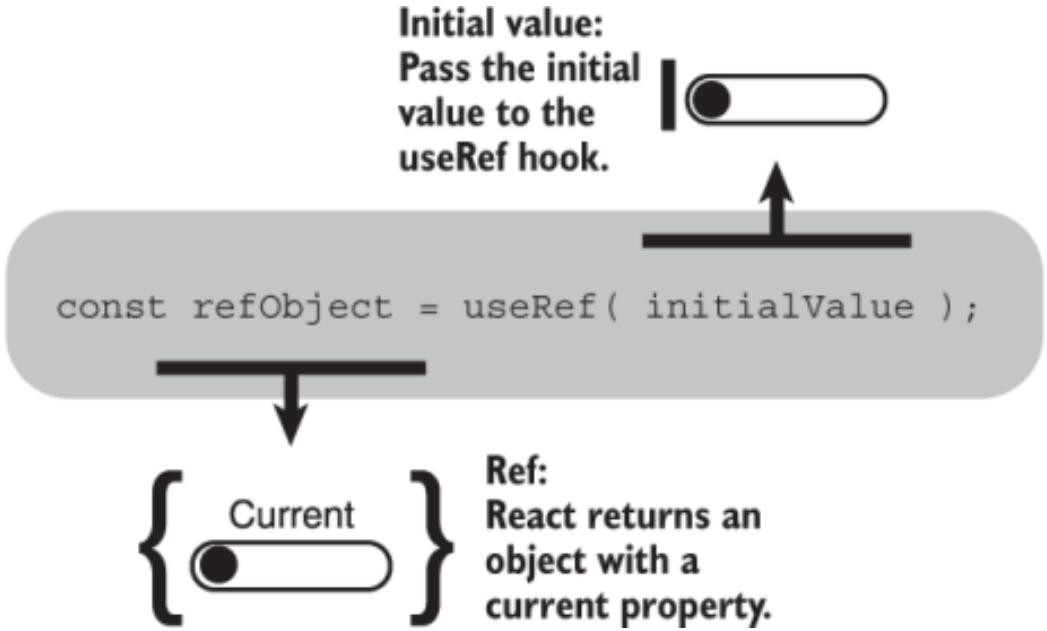
- **Executing a function every time a state variable changes**

```
useEffect(() => {
    async function fetchData() {
        const url = `https://hn.algolia.com/api/v1/search?query=${query}`;
        const response = await fetch(url);
        const data = await response.json();
        setNews(data.hits);
    }
    fetchData();
}, [query]);
```

If 2nd parameter is not set, then the useEffect function will run on every re-render

useRef

- useRef() hook to create **persisted mutable values** as well as directly **access DOM elements** (e.g., focusing an input)
  - The value of the reference is persisted (stays the same) between component re-renderings;
  - Updating a reference doesn't trigger a component re-rendering.



useRef for Mutable values

- useRef(initialValue) accepts one argument as the initial value and returns a reference. A reference is an object having a special property current

```
import { useRef } from 'react';

function LogButtonClicks() {
  const countRef = useRef(0);

  const handle = () => {
    countRef.current++;
    console.log(`Clicked ${countRef.current} times`);
  };

  console.log('I rendered!');

  return <button onClick={handle}>Click me</button>;
}
```

- `reference.current` accesses the reference value, and `reference.current = newValue` updates the reference value
- The value of the reference is persisted (stays the same) between component re-renderings
- Updating a reference doesn't trigger a component re-rendering

useRef for accessing DOM elements

- useRef() hook can be used to access DOM elements

```
import { useRef, useEffect } from 'react';

function InputFocus() {
  const inputRef = useRef();

  useEffect(() => {
    inputRef.current.focus();
  }, []);

  return (
    <input
      ref={inputRef}
      type="text"
    />
  );
}
```

- Define the reference to access the element

```
const inputRef = useRef();
```

- Assign the reference to **ref** attribute of the element:

```
<input  ref={inputRef} />
```

- After mounting, `inputRef.current` points to the DOM element

=> In this example, we access the input to focus on it when the component mounts. After mounting we call `inputRef.current.focus()`

useRef vs. useState

- useState, useReducer, and useContext hooks triggering re-renders when a state variable changes
- useRef remembers the state value but change of value does not trigger rerender
  - The values of refs persist (specifically the current property) throughout render cycles

useReducer

- useReducer allows extracting the state management out of the component to separate the state management and the rendering logic
  - The `useReducer(reducer, initialState)` hook accept 2 arguments: the reducer function and the initial state. The hook then returns an array of 2 items: the current state and the dispatch function
- The **dispatch** function dispatches an action object that describes how to update the state. Typically, the action object would have:
  - **type** property: a string describing what kind of state update the reducer must do
  - **payload** property: having the data to be used by the reduced to update the state

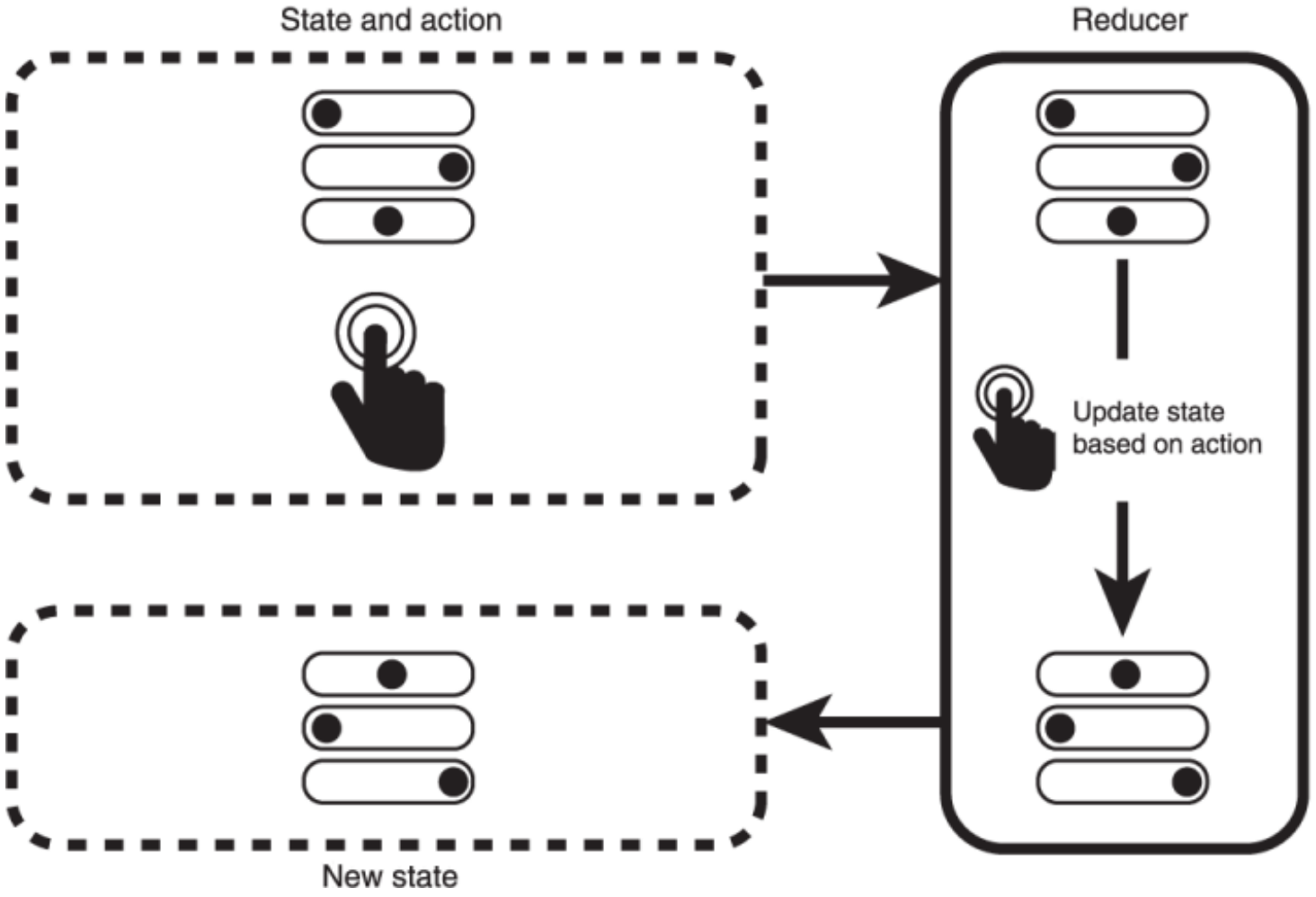
Reducer Function

- The reducer function that accepts 2 parameters: the current state and an action object
- Depending on the action object, the reducer function computes and returns the new state

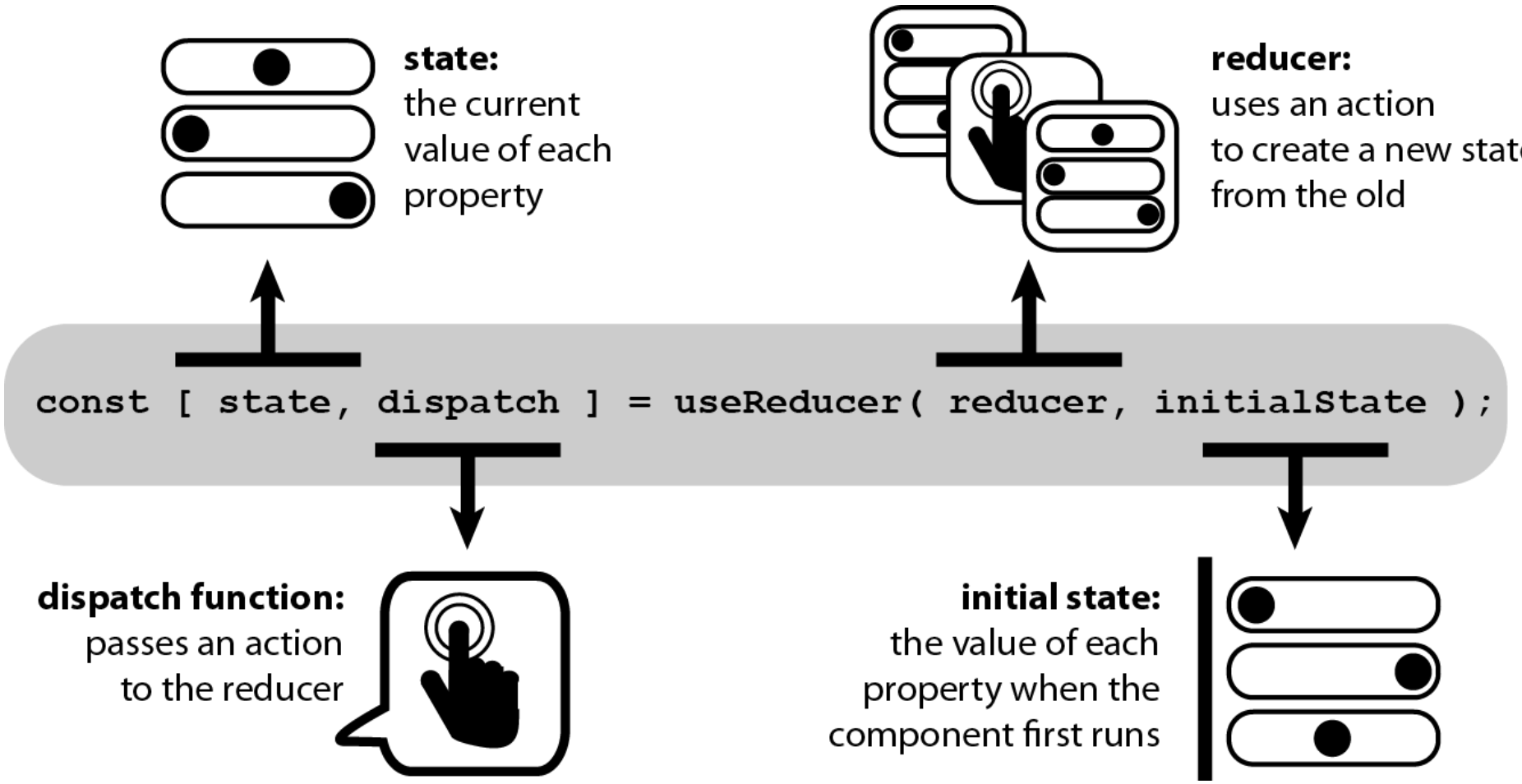
```
function countReducer(state, action) {
    switch (action.type) {
      case "increment":
        return { count: state.count + 1 };
      case "decrement":
        return { count: state.count - 1 };
      case "reset":
          return { count: 0 };
      case "init":
          return { count: action.payload };
    }
}
```

The reducer **doesn't modify** directly the current state variable, but rather creates a new state object then returns it

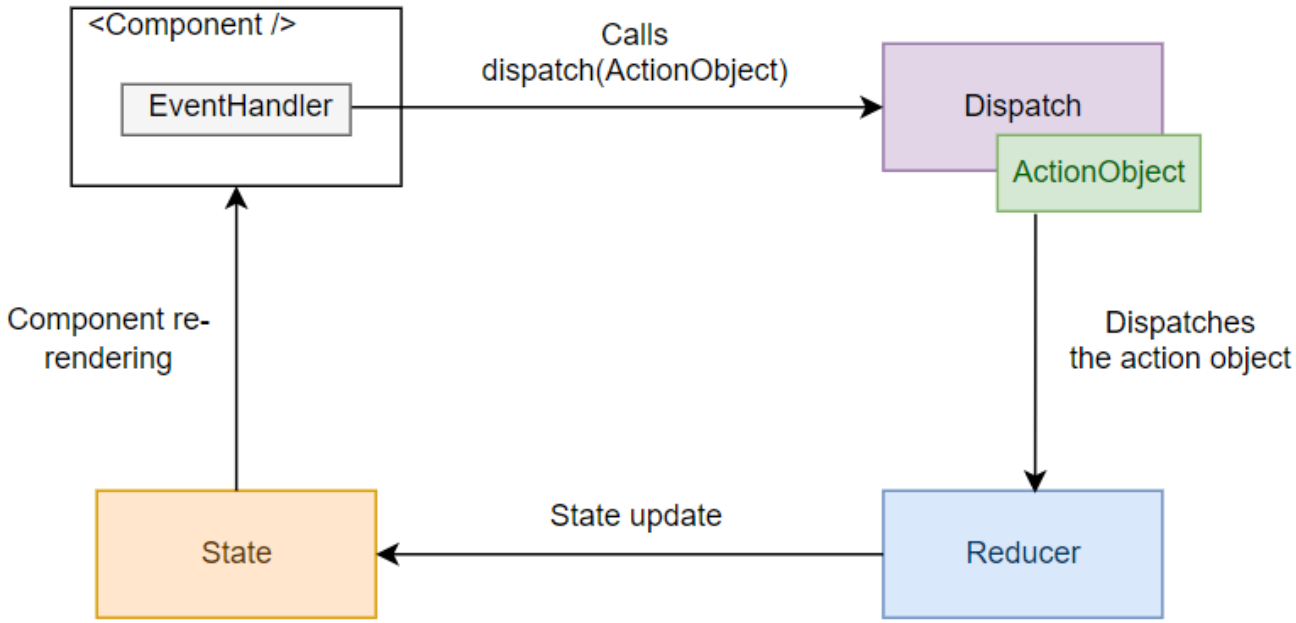
A reducer function takes a state and an action and returns a new state



useReducer: manage multiple related state variables



Whenever you want to update the state (usually from an event handler or after completing a fetch request), you simply call the **dispatch** function with the appropriate action object: dispatch(actionObject)



- As a result of an event handler, you call the dispatch function with the action object
- Then React redirects the action object and the current state value to the reducer function
- The reducer function uses the action object (having an optional payload) and performs a state update, returning the new state
- useReducer() returns the new state value: [state, ...] = useReducer(...)

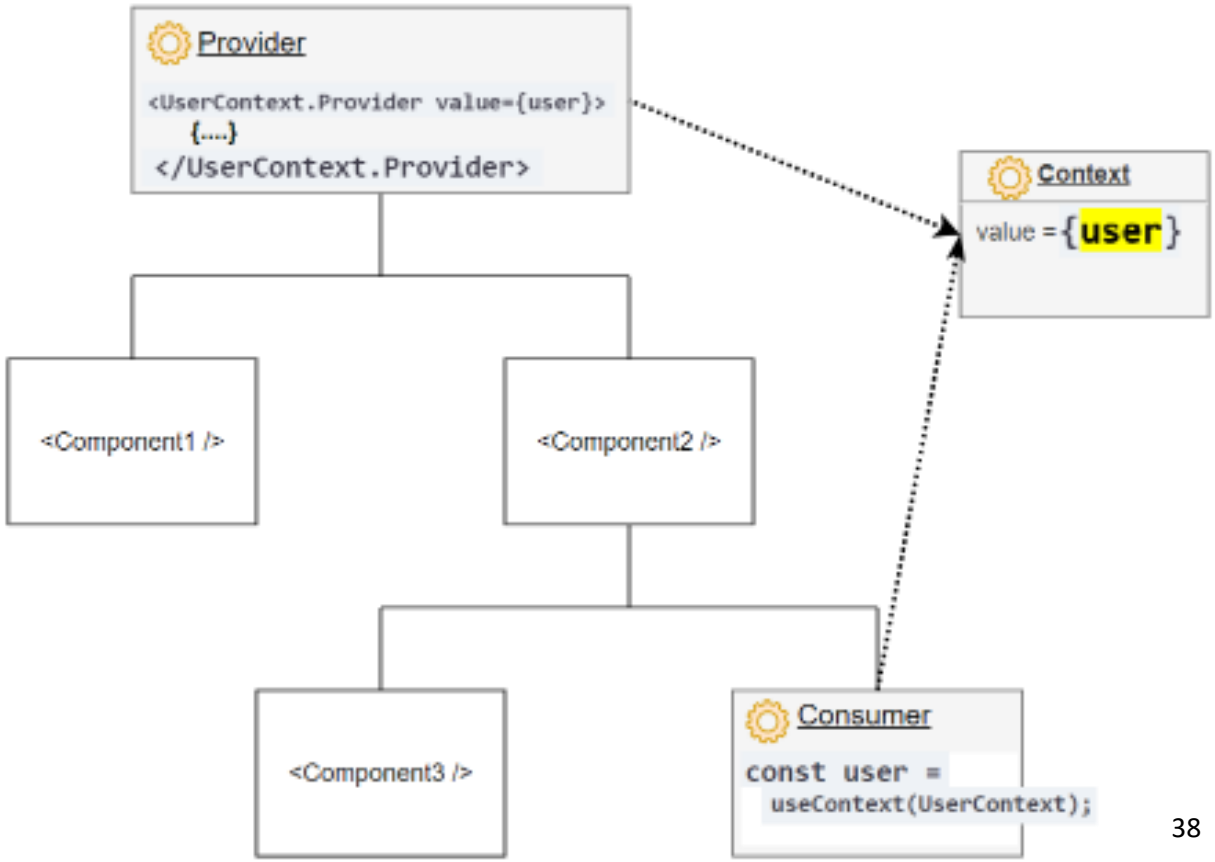
If the state has been updated, React re-renders the component

useContext

- Share state (e.g., current user, user settings) between deeply nested components more easily than prop drilling (i.e., without pass the state as props through each nested component)

- Using the context requires 3 steps: creating, providing, and consuming the context

- If the context variables change then all consumers are notified and re-rendered



useContext – provides shared variables and functions

1. **Create a context** instance (i.e., a container to hold shared variables and functions)

```
import React from 'react';
const UserContext = React.createContext();
export default UserContext;
```

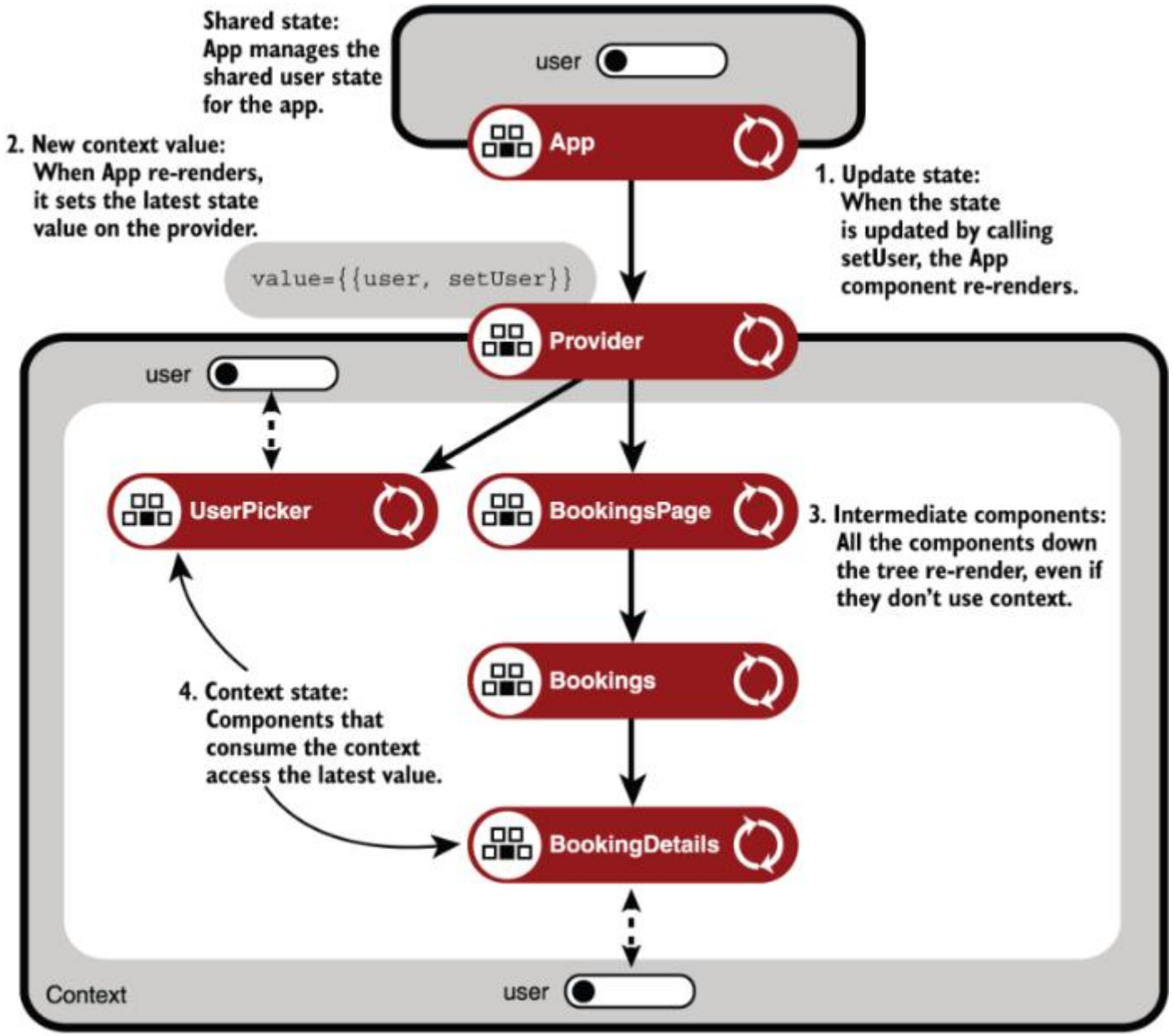
2. **Provider places shared variables / functions in the context** to make them available to child components

```
import UserContext from './components/UserContext';
function App() { return (
     <UserContext.Provider value={ user }>
            <Welcome /> …
      </UserContext.Provider>
   ); }
```

3. **Consumer access the shared variables / functions in the context**

```
import React, {useContext} from "react";  import UserContext from './UserContext';
export default function Welcome() {
    const user = useContext(UserContext);
    return <div>You are login as: {user.username}</div>;
}
```

Shared State Example



Summary

- Hooks are functions which "hook into" React state and lifecycle features from components
- **useState** : manage state
- **useEffect**: perform side effects and hook into moments in the component’s life cycle
- **useRef**: access DOM elements directly
- **useReducer:** manage multiple related state variables
- **useContext:** share data and functions with child components without prop drilling using

Resources

- **Hooks at a Glance**

https://reactjs.org/docs/hooks-overview.html

- React Hooks in Action textbook

https://learning.oreilly.com/library/view/react-hooks-in/9781617297632/

- Useful list of resources

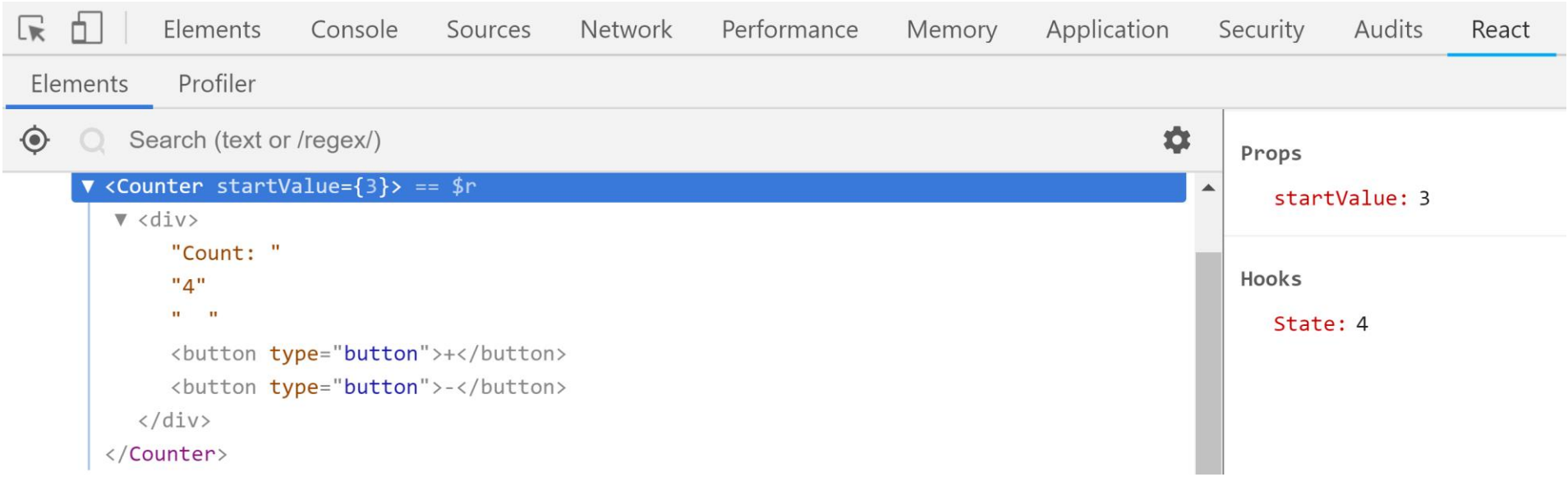
https://github.com/rehooks/awesome-react-hooks

React Tools and Component Libraries

- **React Dev Tools**
- **React Components Libraries**

React Dev Tools

- React Dev Tools
  https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en



Component Libraries

- **Shadcn**

https://ui.shadcn.com/

- **Material-UI**: React components with Material Design

https://mui.com/

Summary

- React = a declarative way to define the UI
- Decompose UI into self-contained and often reusable components
- Why React:
  - Component-based
  - Virtual DOM
  - Declarative
- React uses JSX syntax to define component's UI

Resources

- Thinking in React

https://reactjs.org/docs/thinking-in-react.html

- React Router

https://reactrouter.com/

- Useful list of resources

https://github.com/enaqx/awesome-react