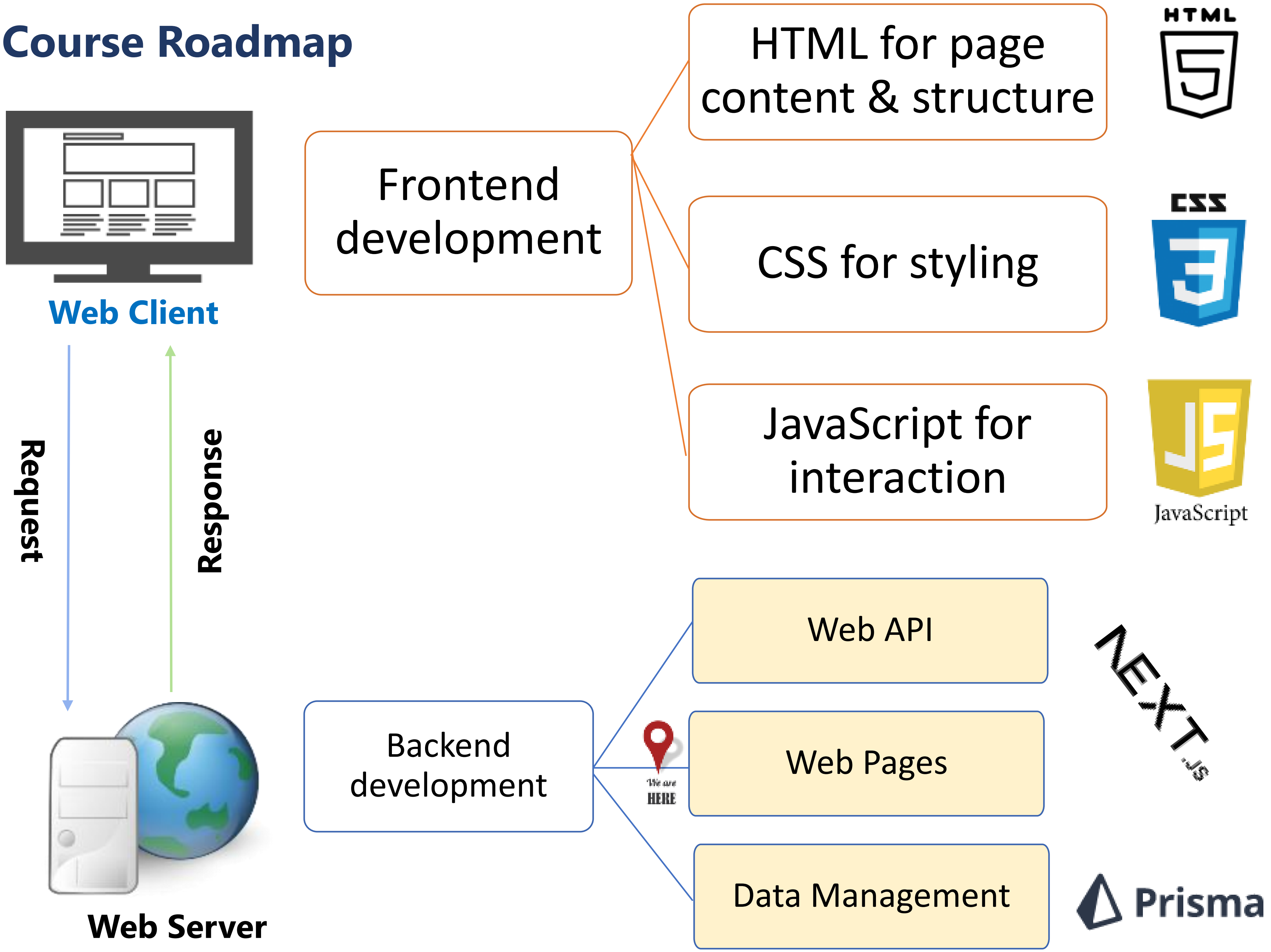
React Fundamentals

Outline

Course Roadmap



React Introduction

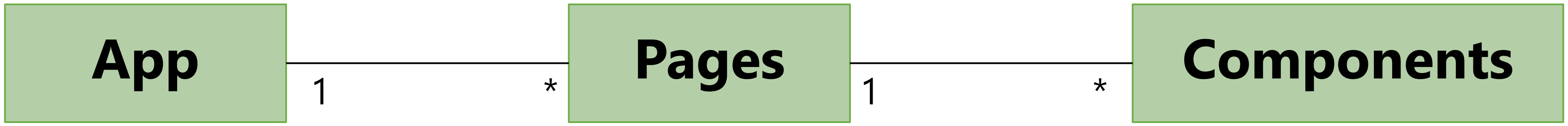
Used by Facebook, Instagram, Netflix, Dropbox, Outlook, Yahoo, Khan Academy, ...

https://intellisoft.io/15-popular-sites-built-with-react-js/

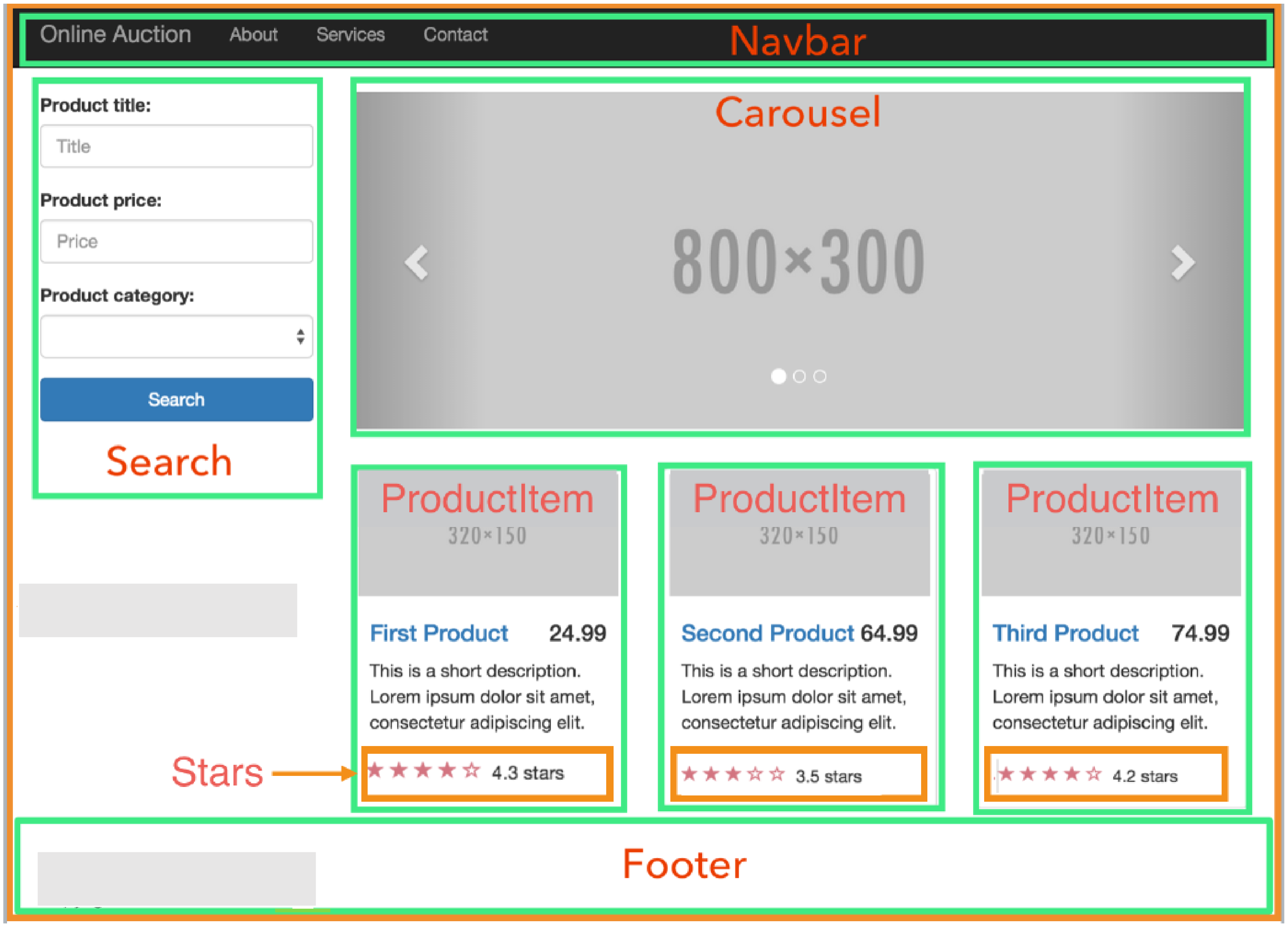
< Back

Web App

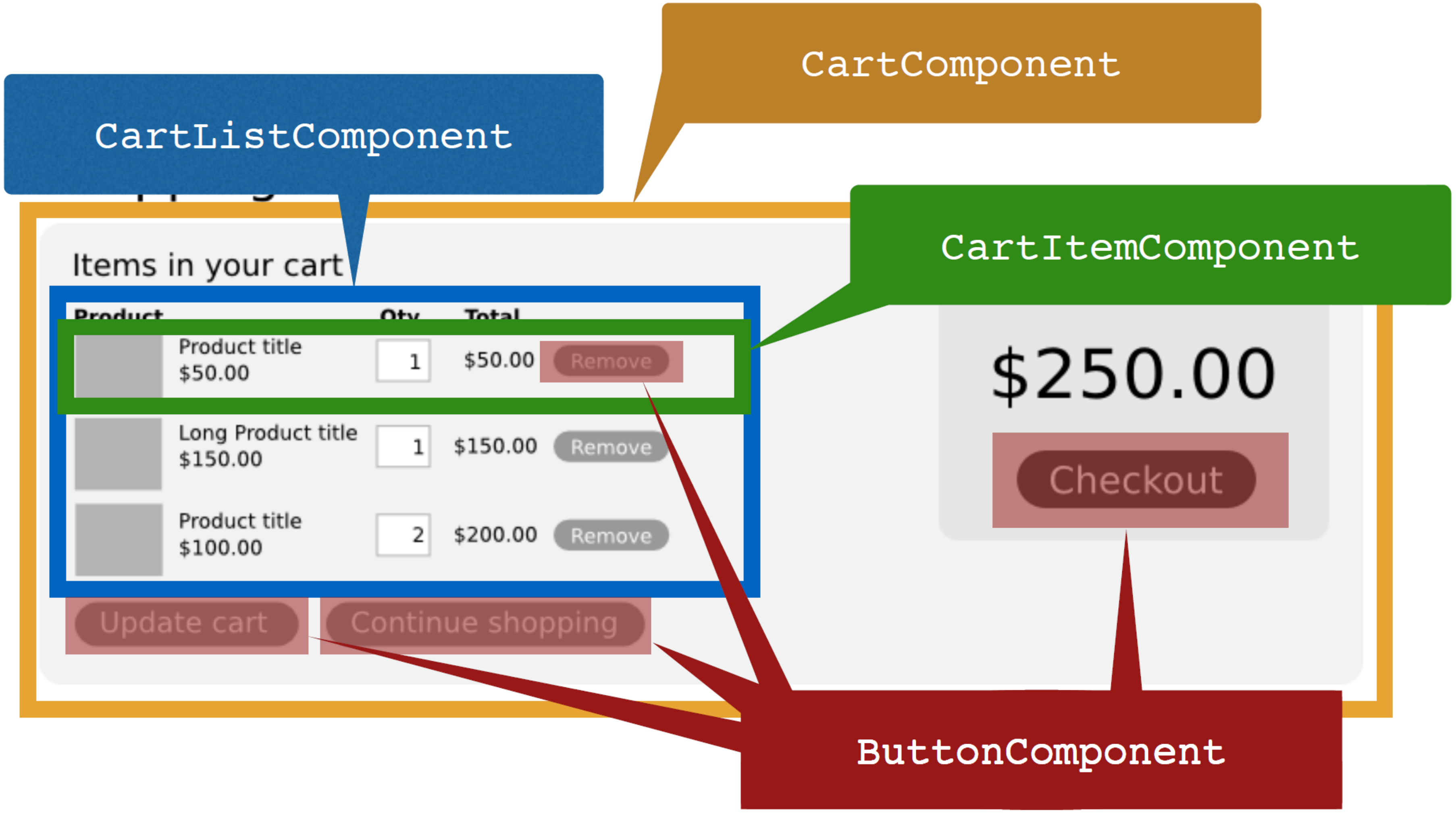
Web app has **pages**, and a page is composed on **components**



A page = a composition of components



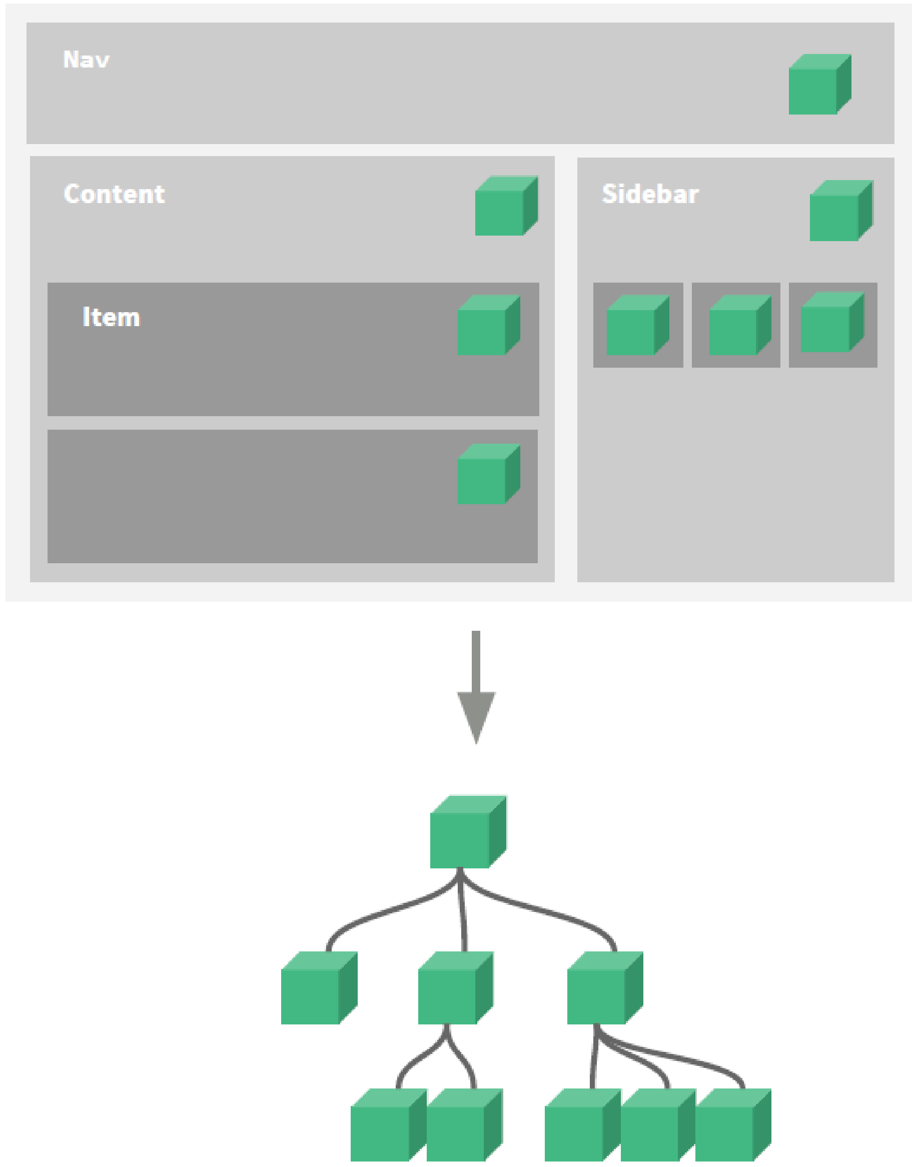
A component could be a tree of components



What is React?

- React is an open-source JavaScript library for building **components-based user interfaces (UI)**
  - UI is **composed** of small reusable **components**
  - A component encapsulates **UI elements** and the **behavior** associated with them
- React components can be rendered either on the server-side or on the client-side
- Open-sourced by Facebook mid-2013 - https://reactjs.org/
- Competing with Angular http://angular.io and Vue.js https://vuejs.org/

React Components



< Back

React Component

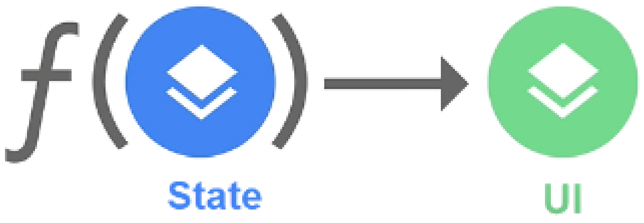
- React App = composition of **components**
- A ***component***:
    - Return **HTML *elements*** to provide the UI
    - Encapsulate **state** (internal component data) and **functions** to ***handle events*** raised from the UI elements
- Component = UI + display logic
- Components allows creating new '**HTML tags**'

How to define a piece of UI?

UI is **composed** of small reusable **components**

UI Component = a **function**:

- Takes some inputs and emits a piece of UI
- Function that converts the state (i.e., app data) into UI



- **UI = f(state) : UI is a visual representation of state**
  (e.g., display a tweet and associated comments)
- 👍 **State changes trigger automatic update of the UI**

Component Example

- Create a **Welcome** component
  - Returns **JSX** : an HTML-like syntax to define the component UI
  - Can accept a parameter called ***props***
    - to configure the component with different content / attributes - just like how HTML works (makes the component reusable)
    - **props** are read-only
  - Component name must start with a capital letter

```
import React from "react";
function Welcome(props) {
    return (<h1>Welcome to {props.appName}</h1>);
}
export default Welcome;
```

You can embed JavaScript expressions in JSX

- Use the **Welcome** component

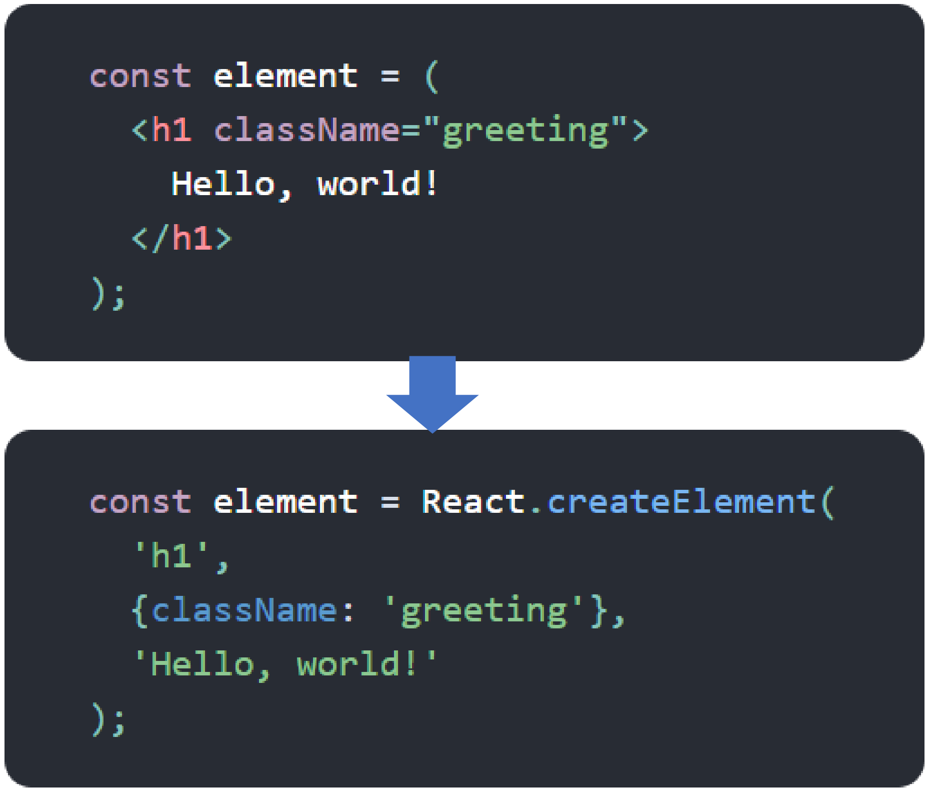
```
<Welcome appName='React Demo App' />
```

What is JSX?

- React uses JSX (JavaScript XML) HTML-like markup to describe the component’s UI
- JSX allows us to write HTML like syntax which gets transformed to JavaScript objects

JSX

JavaScript



```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

Props destructuring

- In a **react** component you can destructure props into variables

```
function UserInfo(props) {
    return (
        <div>
            First Name: {props.firstName}
            Last Name: {props.lastName}
        </div>
    );
}
```

Becomes

```
function UserInfo({ firstName, lastName }) {
    return (
        <div>
            First Name: {firstName}
            Last Name: {lastName}
        </div>
    );
}
```

Special "children" Prop

- The children property holds the content you might have provided between the component's opening and closing tags
  - A special children property auto-added by react

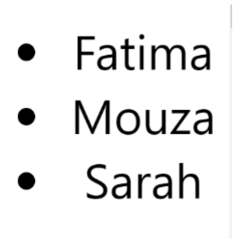
```
<Welcome name="Ali Faleh">
  <h2>Welcome to QU</h2>
  <img src="http://www.qu.edu.qa/.../logotype.png" />
</Welcome>
```

```
function Welcome({name, children}) {
    return (
        <>
            <h1>Welcome {name}</h1>
            {children}
        </>
    );
}
```

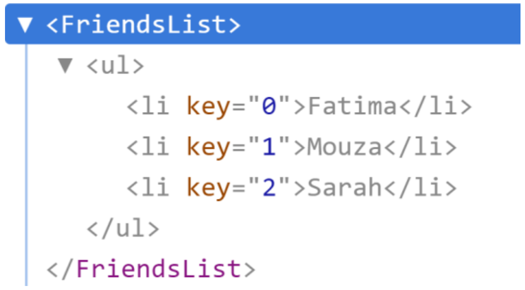
Rendering a List of items (with .map())

Lists are handled using **.map** array function

```
function FriendsList({friends}) {
  return <ul>
            {friends.map( (friend, i) =>
                <li key={i}>{friend}</li>
            )}
         </ul>
}
```



- Fatima
- Mouza
- Sarah



```
▼ <FriendsList>
  ▼ <ul>
      <li key="0">Fatima</li>
      <li key="1">Mouza</li>
      <li key="2">Sarah</li>
    </ul>
  </FriendsList>
```

Key helps identify which items have changed, added or removed

- Use the **FriendsList** component

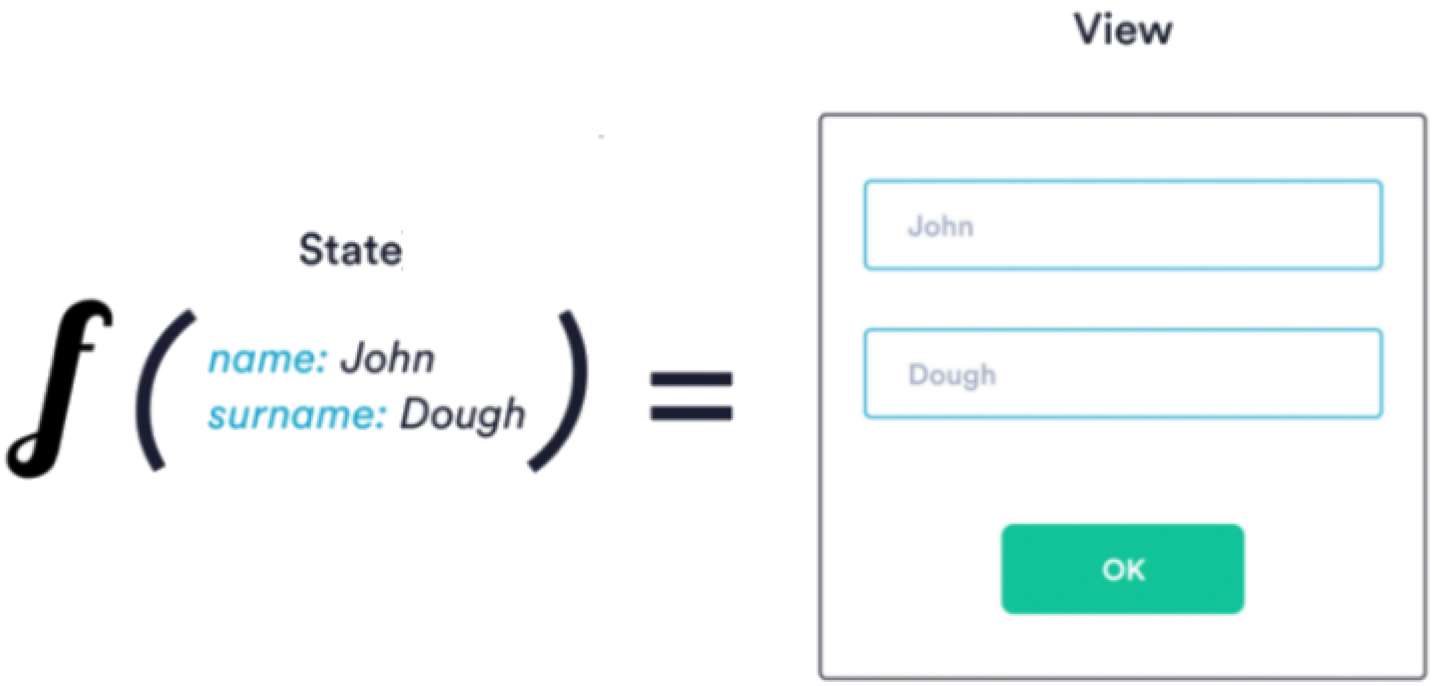
```
<FriendsList friends={['Fatima', 'Mouza', 'Sarah']}/>
```

List of item keys

Keys are very important in lists for the following reasons:

- A key is a unique identifier used to identify which list items have changed, are added, or are deleted from the list
- It also helps to determine which components need to be re-rendered instead of re-rendering all the components every time.
    - Therefore, it increases performance, as only the updated components are re-rendered

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

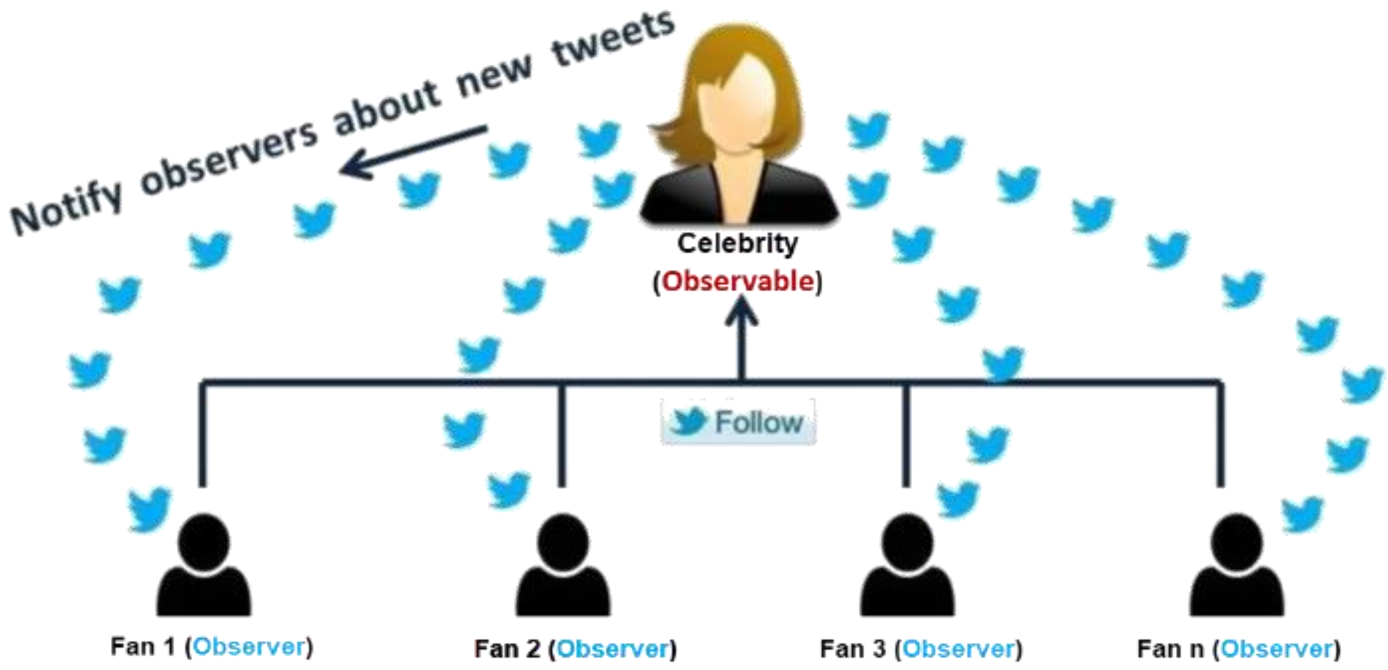
- State = any value that can change overtime
- State variable must be declared using **useState** hook to act as **Change Notifiers**
  - 👍 They **are observed** by the React runtime
  - Any change of a state variable will trigger the **re-rendering** of any functions that **reads** the state variable
  - 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 UI 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";
```

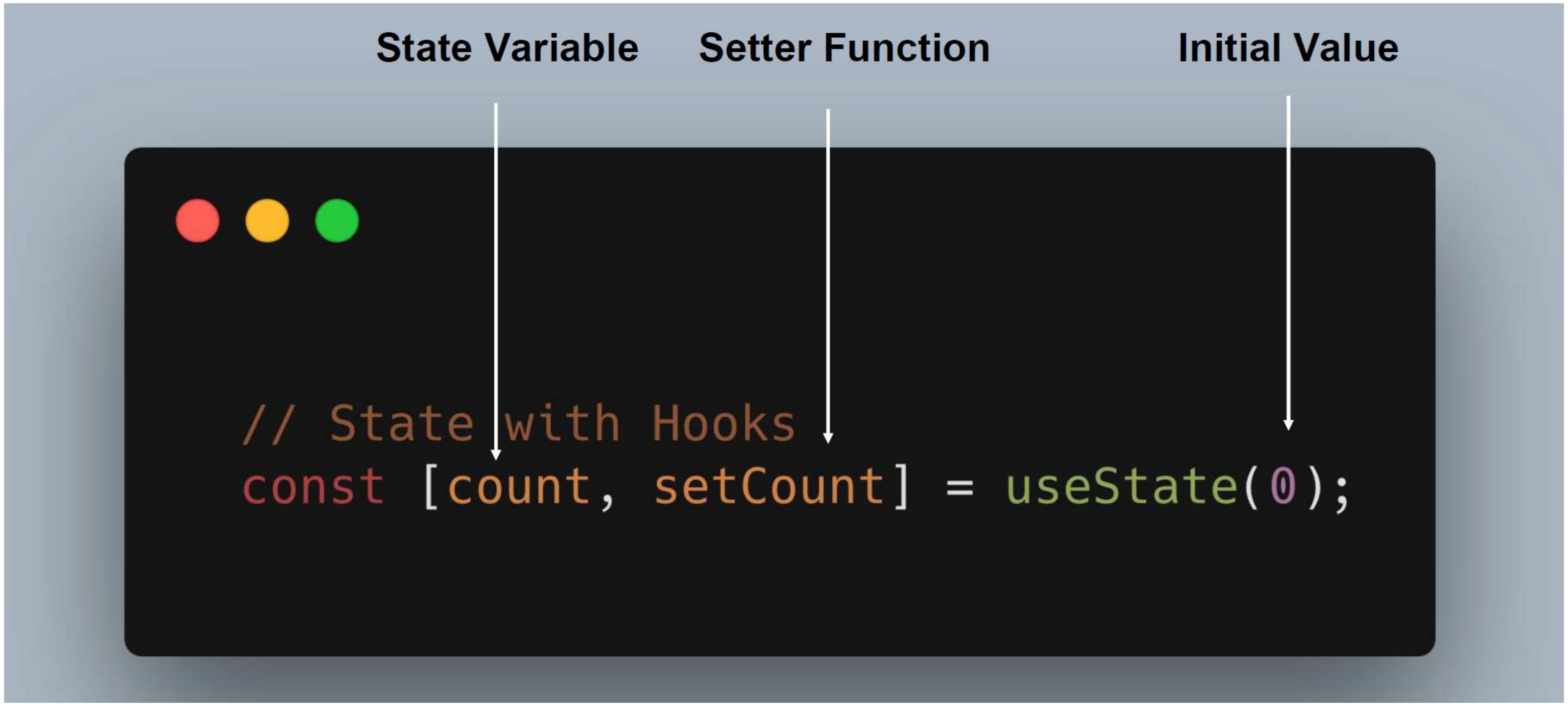
UI in React is immutable

- 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: creates a state variable

- Used for basic state management inside a component



```
// State with Hooks
const [count, setCount] = useState(0);
```

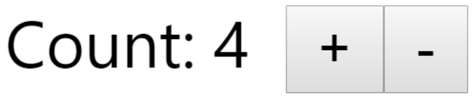
Component with State + Events Handling

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

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

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

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

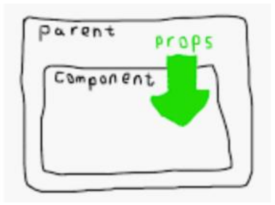


Handling events is done the way events are handled on DOM elements

- 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

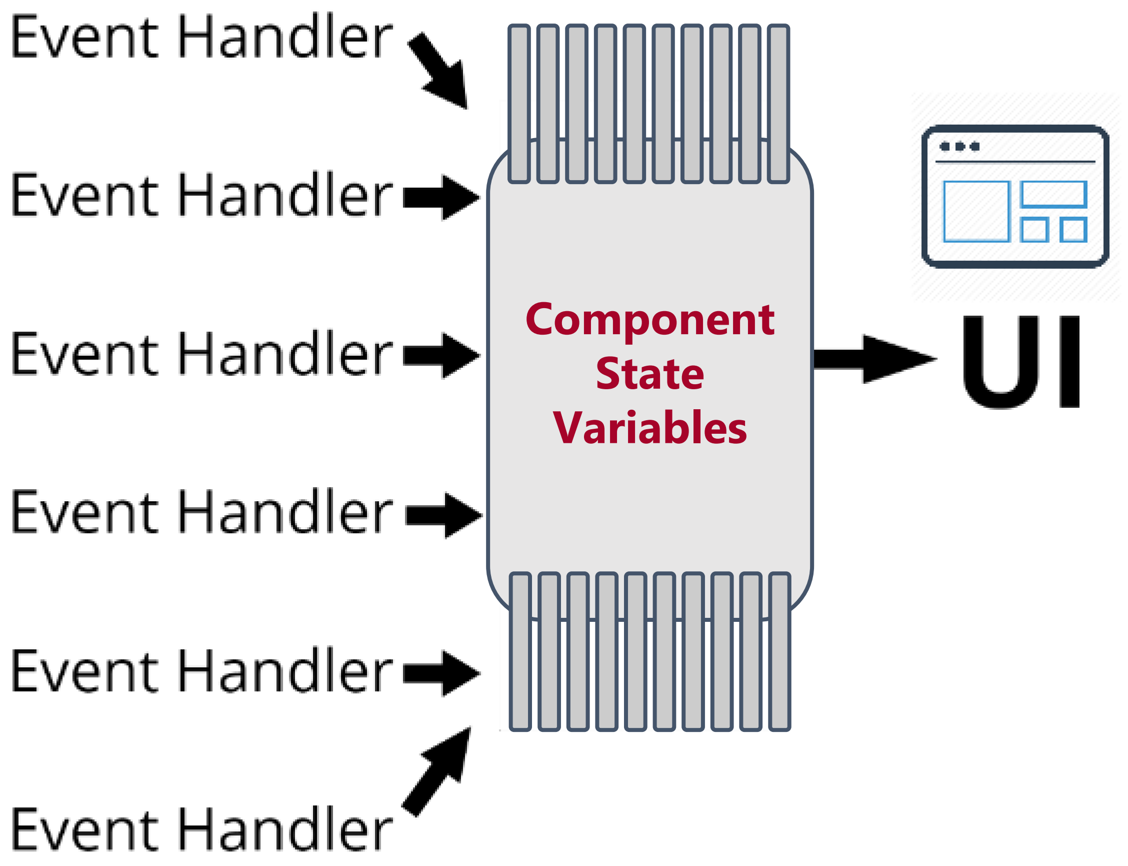
- **Props** parameters are **read only**

- **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**

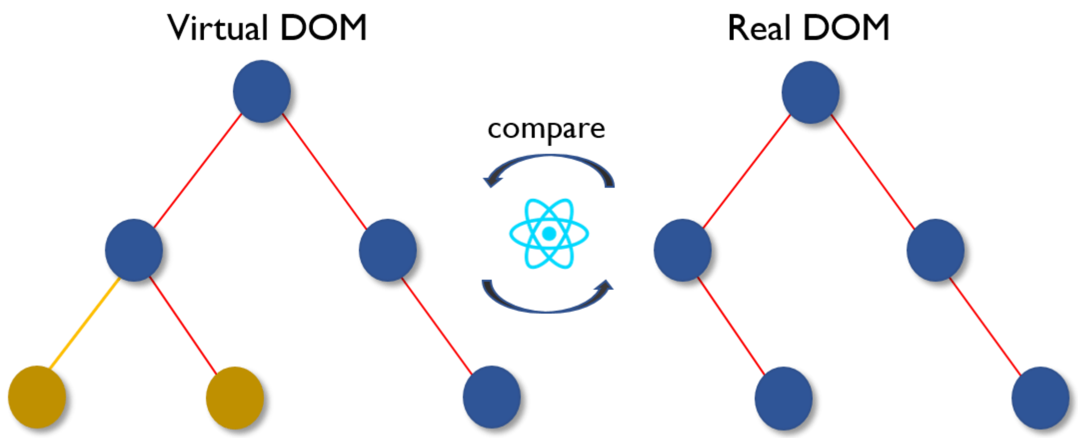
Event Handlers update the State and React updates the UI



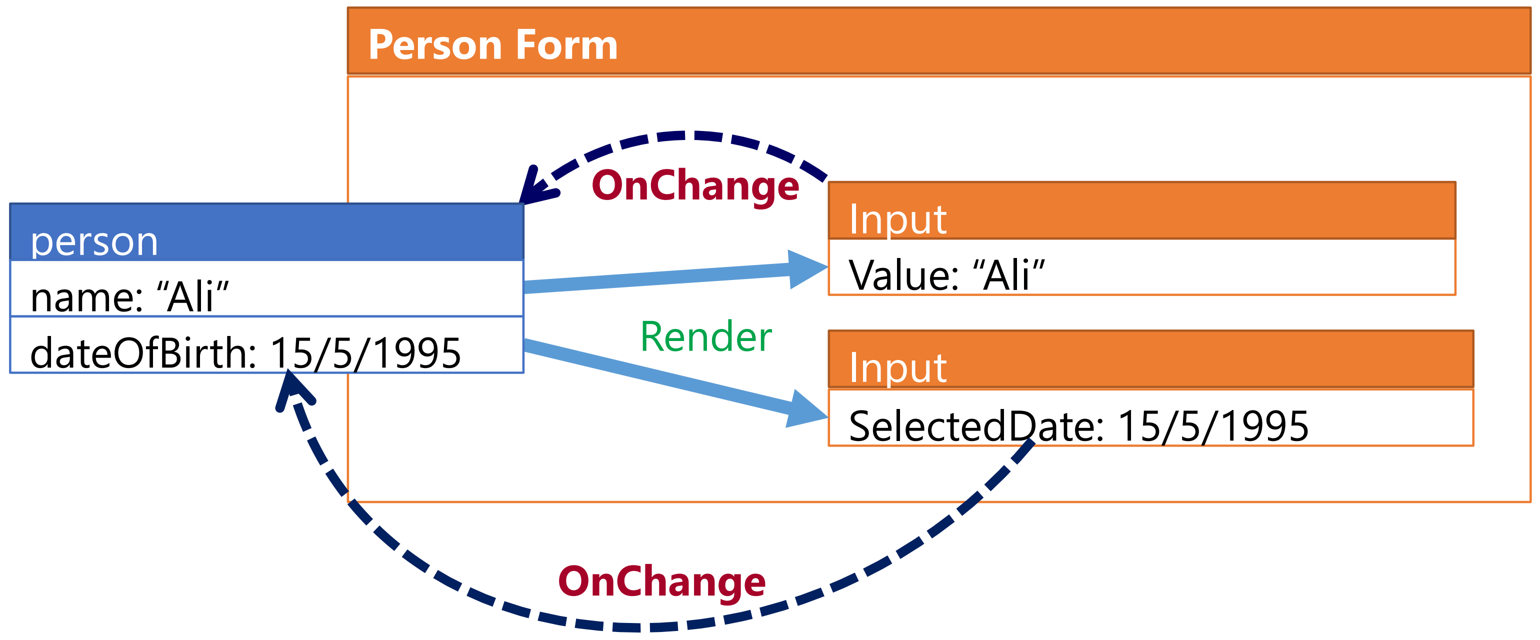
Every place a state variable is displayed is guaranteed to be auto-updated

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



Event Handlers update State Variables



Common Events: `onClick - onSubmit – onChange`

Forms with React

```
<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 UI

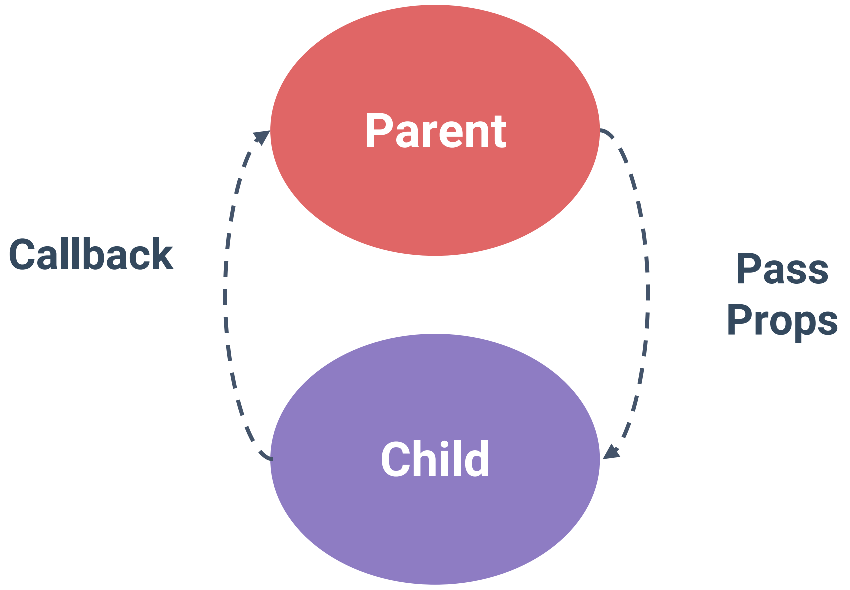
Form State and Event Handlers

```
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



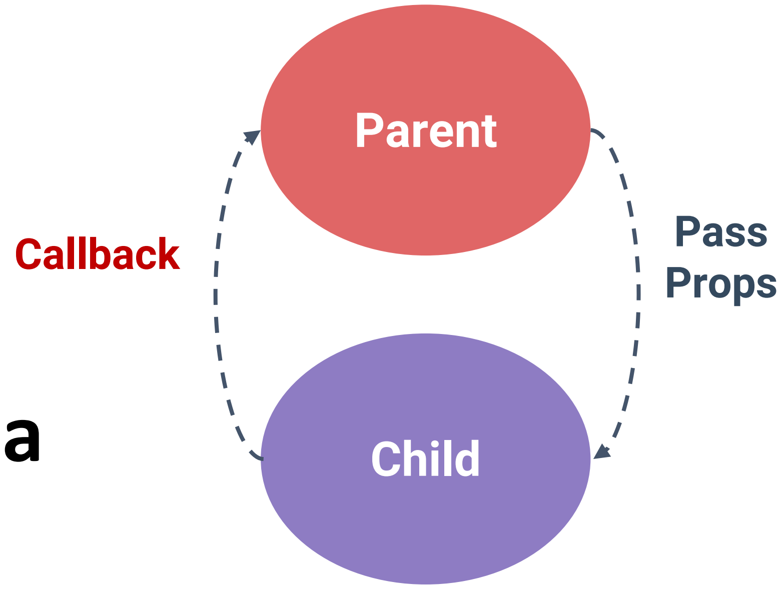
Back

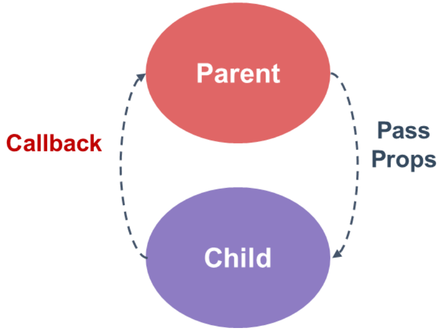
Composing Components

- 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

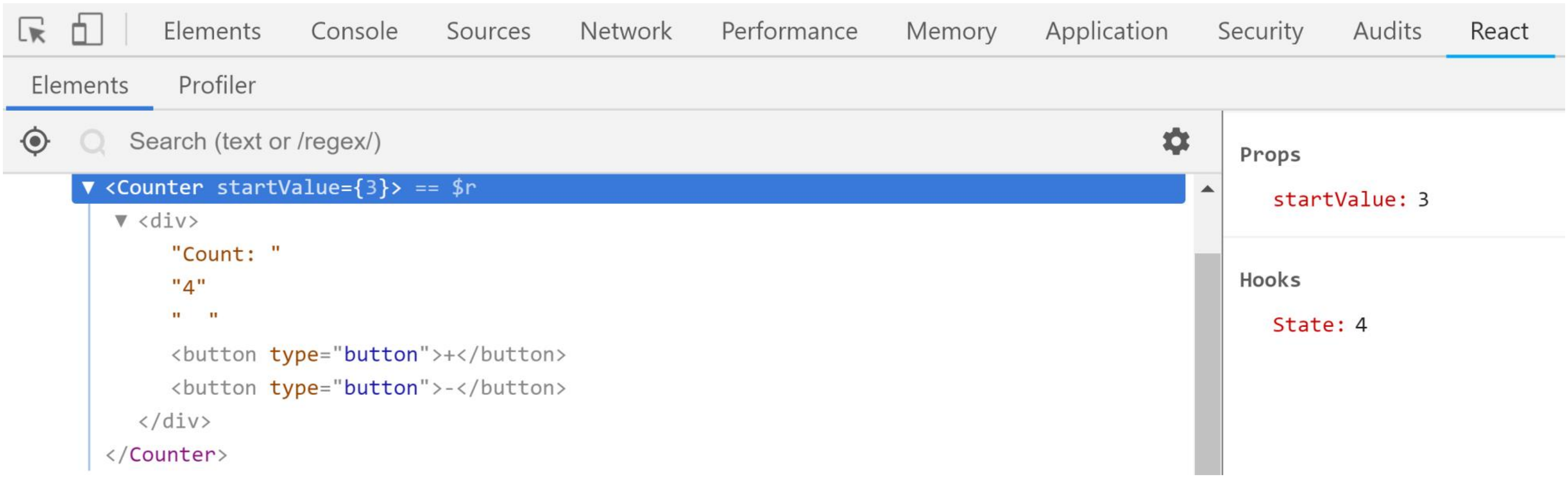
```
function Counter(props) {
    const [count, setCount] = useState(props.startValue);

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

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

React Dev Tools

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

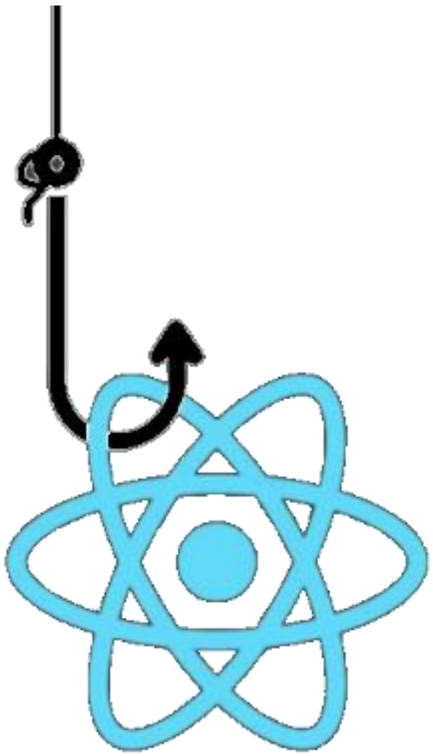


```
▼ <Counter startValue={3}> == $r
  ▼ <div>
      "Count: "
      "4"
      "  "
      <button type="button">+</button>
      <button type="button">-</button>
    </div>
</Counter>
```

```
Props
  startValue: 3

Hooks
  State: 4
```

Common Hooks

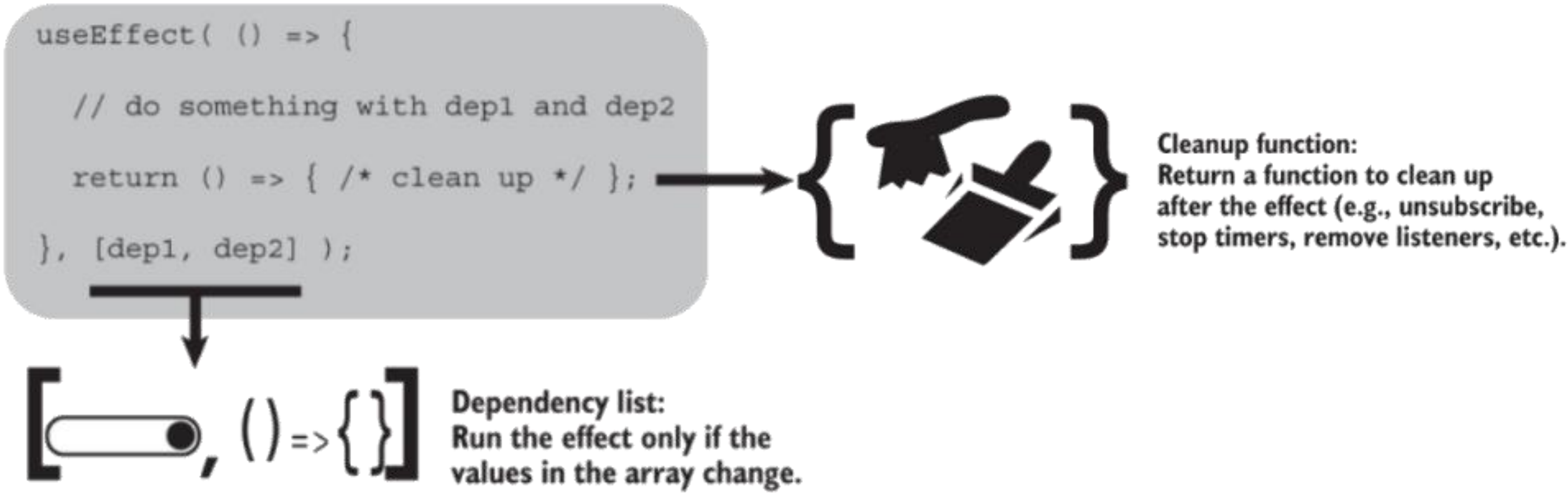


- A Hook is a special function that lets you **hook** into React features such as state and lifecycle methods

< Back

useEffect

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



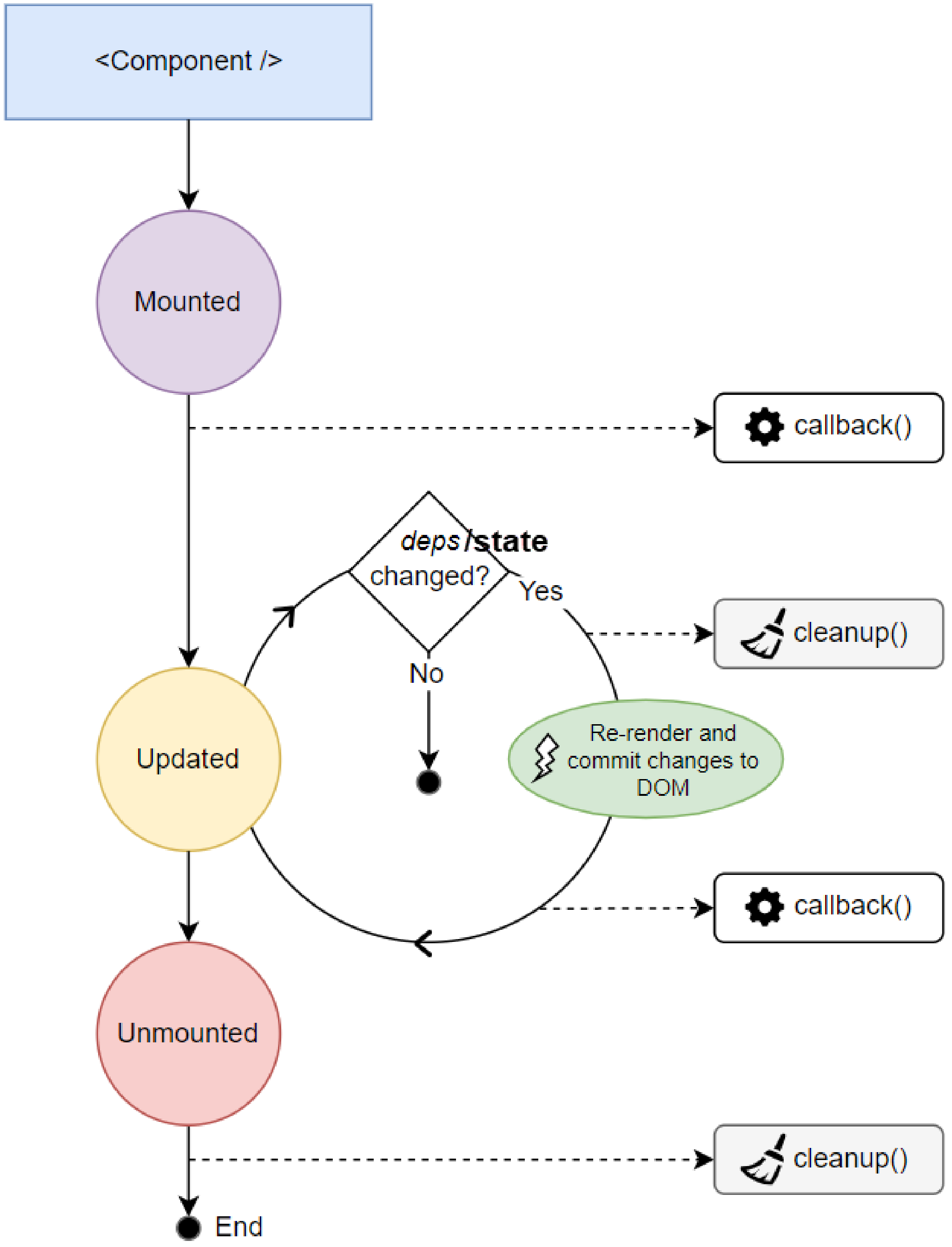
```
useEffect( () => {

  // do something with dep1 and dep2

  return () => { /* clean up */ };

}, [dep1, dep2] );
```

Cleanup function:
Return a function to clean up after the effect (e.g., unsubscribe, stop timers, remove listeners, etc.).
Dependency list:
Run the effect only if the values in the array change.

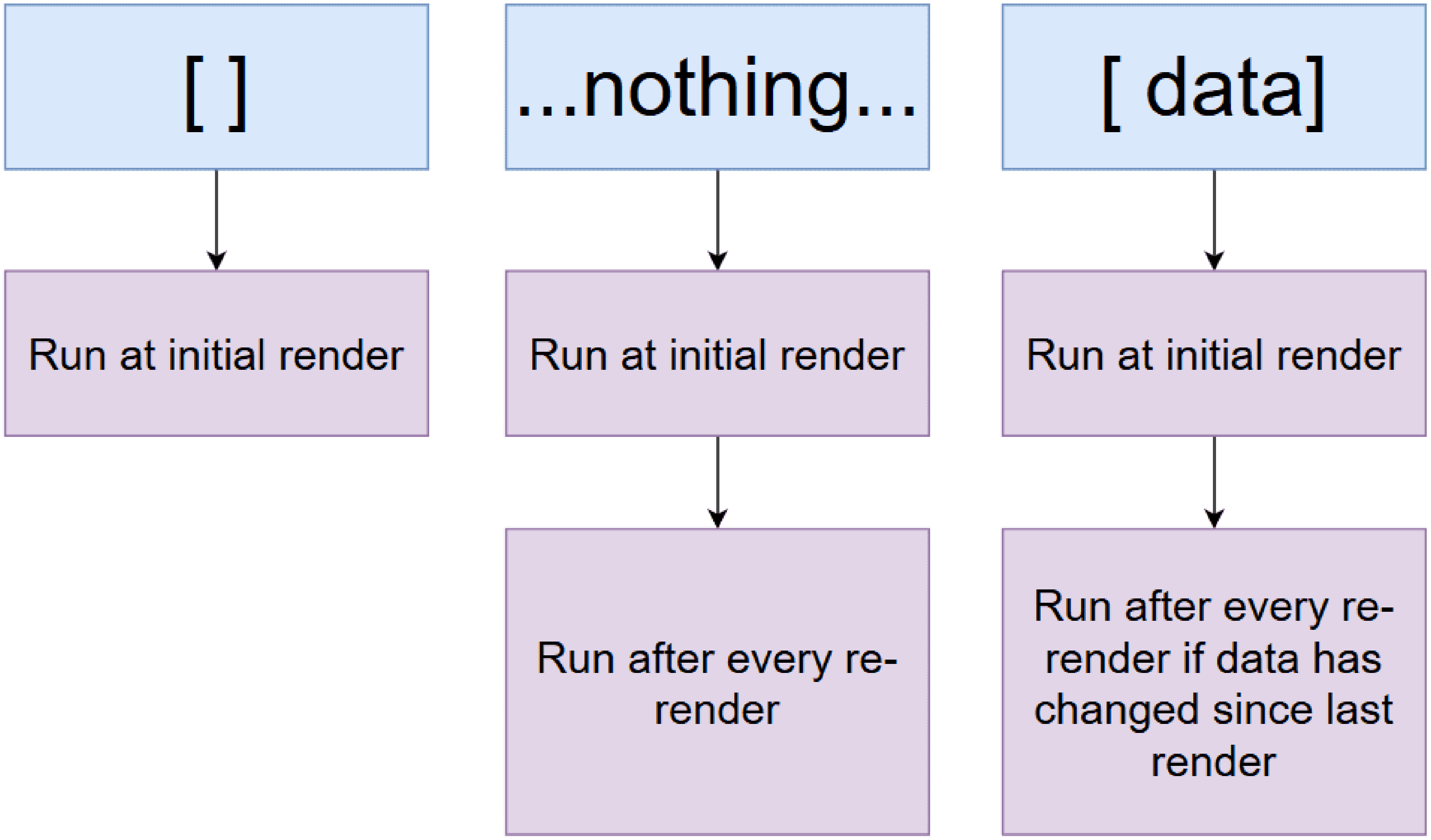


A) After initial rendering, `useEffect()` invokes the callback having the side-effect. cleanup function is not invoked

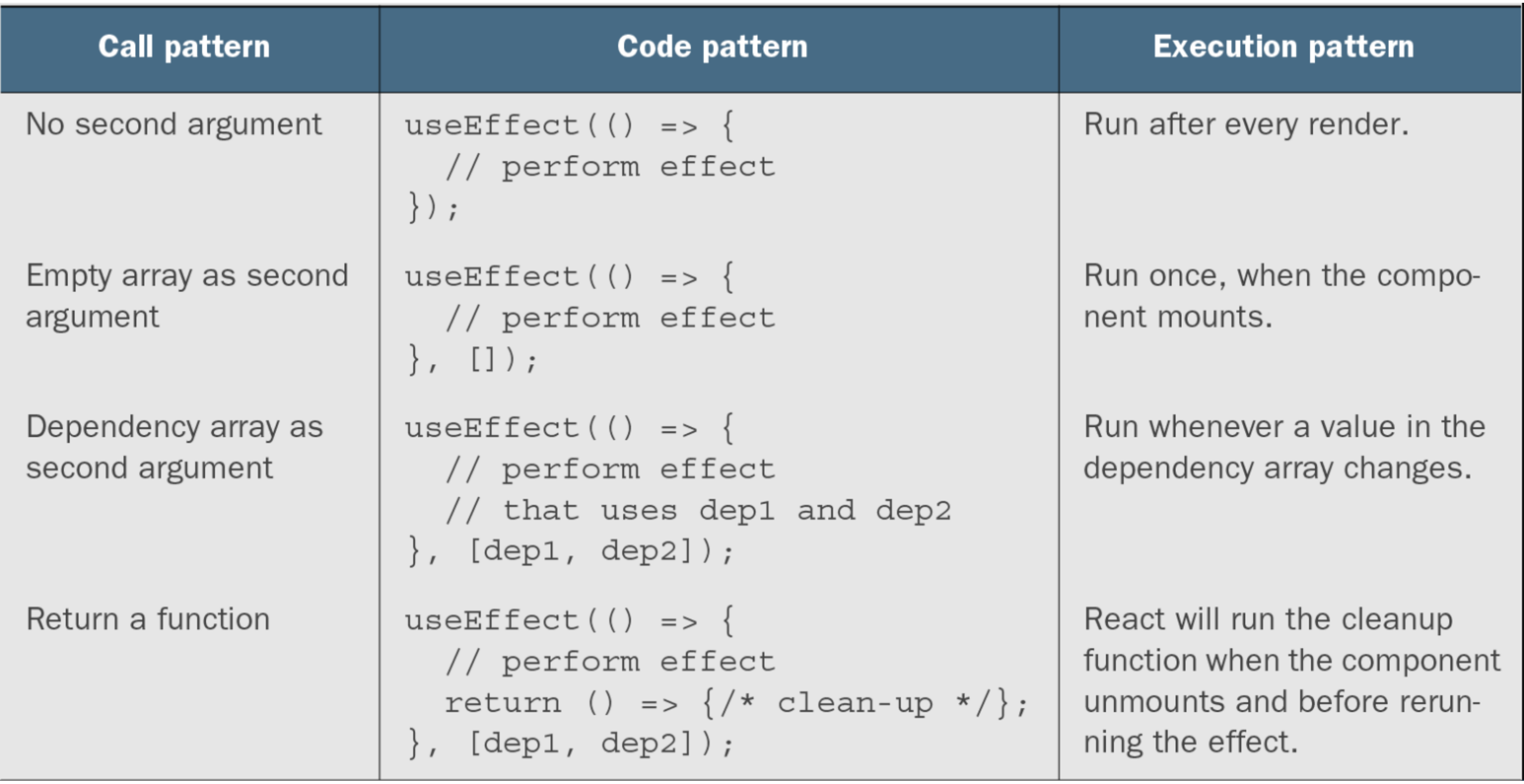
B) On later renderings, before invoking the next side-effect callback, `useEffect()` invokes the cleanup function from the previous side-effect execution (to clean up everything after the previous side-effect), then runs the current side-effect

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

useEffect - 2nd argument


[ ]
...nothing...
[data]
Run at initial render
Run at initial render
Run at initial render
Run after every re-render
Run after every re-render if data has changed since last render

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 compo-nent 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 rerun-ning 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.
```

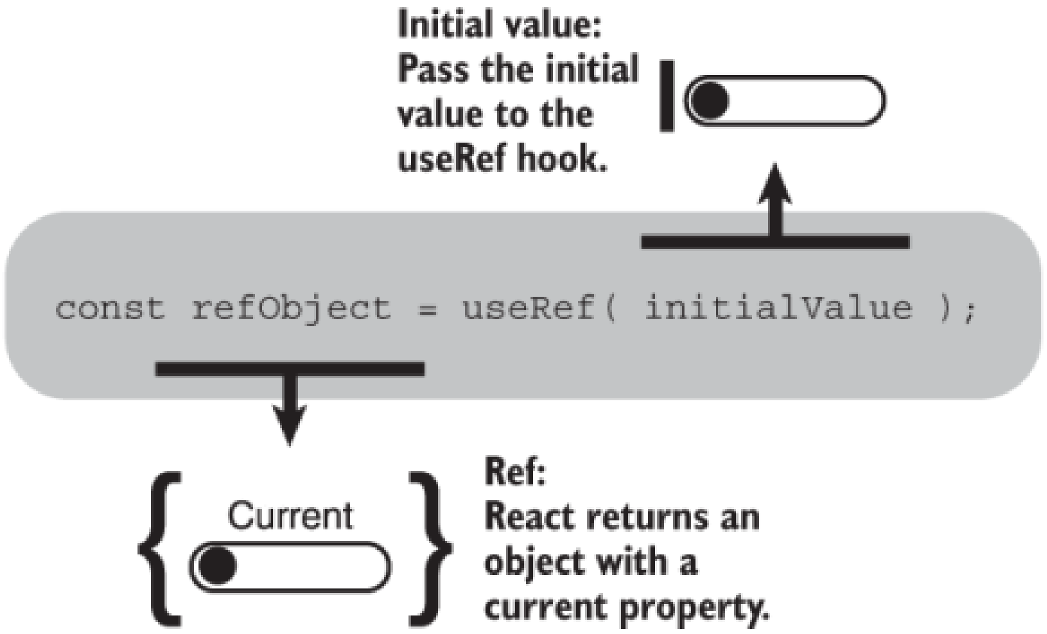
- **Executing a function every time a state variable changes**

```
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

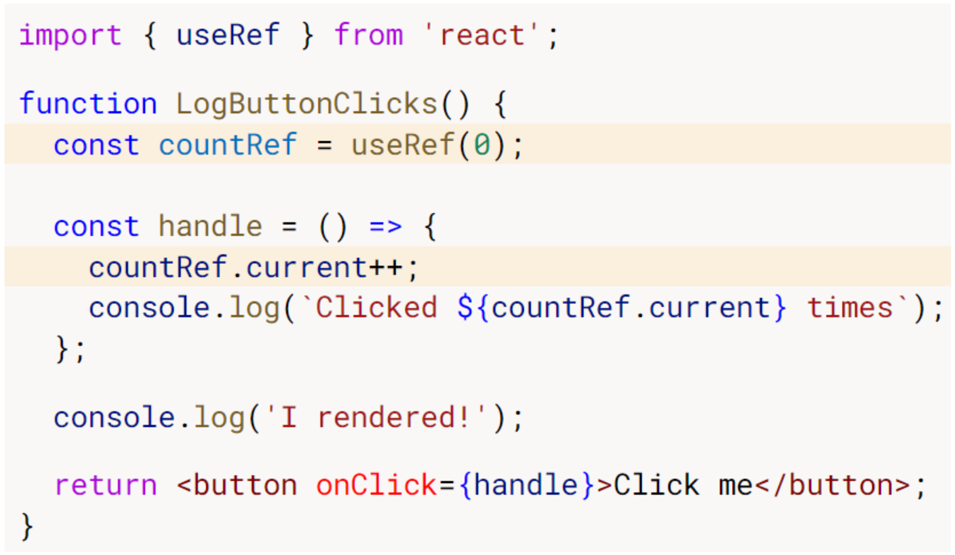
useRef

- 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.


Initial value:
Pass the initial value to the useRef hook.
const refObject = useRef(initialValue);
Current
Ref:
React returns an object with a current property.

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 hook triggers re-rendering 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

Summary

- React = a declarative way to define the UI
  - Decompose UI into self-contained and often reusable components
  - React uses JSX syntax to define component’s UI
- 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

Resources

- Thinking in React

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

- **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/enaqx/awesome-react

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