2024-07-22 React Journey

JavaScript has “`async`” and “`await`” methods, used for promises.

Assume the following example,

```
async function fetchData(link)
{
        try
        {
                let reponse = await fetch(link)
                let data    = await response.json();

                console.log(data)
        }

        catch (error)
        {
                console.error("Error fetching data:", error)
        }

}


fetchData('https://jsonplaceholder.typicode.com/todos/1')
```

Explanation:

1. **Async Function**: The `async` keyword before the function declaration makes it an asynchronous function, which means it will return a promise.
2. **Await Keyword**: The `await` keyword is used to pause the execution of the async function until the promise is resolved. In this example, `await fetch(...)` waits for the fetch request to complete, and `await response.json()` waits for the response to be parsed as JSON.
3. **Error Handling**: The `try...catch` block is used to handle any errors that might occur during the asynchronous operations.

This example fetches data from a placeholder API and logs it to the console. If there's an error during the fetch, it will be caught and logged.

Feel free to try this out in your code! If you have any specific scenarios or further questions, let me know.

Assume some object:

```
  const someObject = {
      title: "The Lord of the Rings",
      publicationDate: "1954-07-29",
      author: "J. R. R. Tolkien",
      genres: [
        "fantasy",
        "high-fantasy",
        "adventure",
        "fiction",
        "novels",
        "literature",
      ],
      hasMovieAdaptation: true,
      pages: 1216,
      translations: {
        spanish: "El señor de los anillos",
        chinese: "魔戒",
        french: "Le Seigneur des anneaux",
      },
      reviews: {
        goodreads: {
          rating: 4.52,
          ratingsCount: 630994,
          reviewsCount: 13417,
        },
        librarything: {
          rating: 4.53,
          ratingsCount: 47166,
          reviewsCount: 452,
        },
      },
    },
```

it is then possible to de-structure it as such:

```
const {title, author} = someObject
```

This allows for quick access to the data stored directly in the object.

Rest Operator

The rest operator allows you to represent an indefinite number of arguments as an array. It's often used in function parameters.

Assume the following function,

$$Sum(1,2,3,4) = 10$$

In JavaScript, this function can be written as follows:

```
function Sum( ... numbers){
        return nubmers.reduce((acc, curr) => acc + curr, 0)
}
```

Spread Operator

The spread operator allows you to expand an iterable (like an array) into individual elements. It's useful for copying arrays, combining arrays, or passing array elements as arguments to a function.

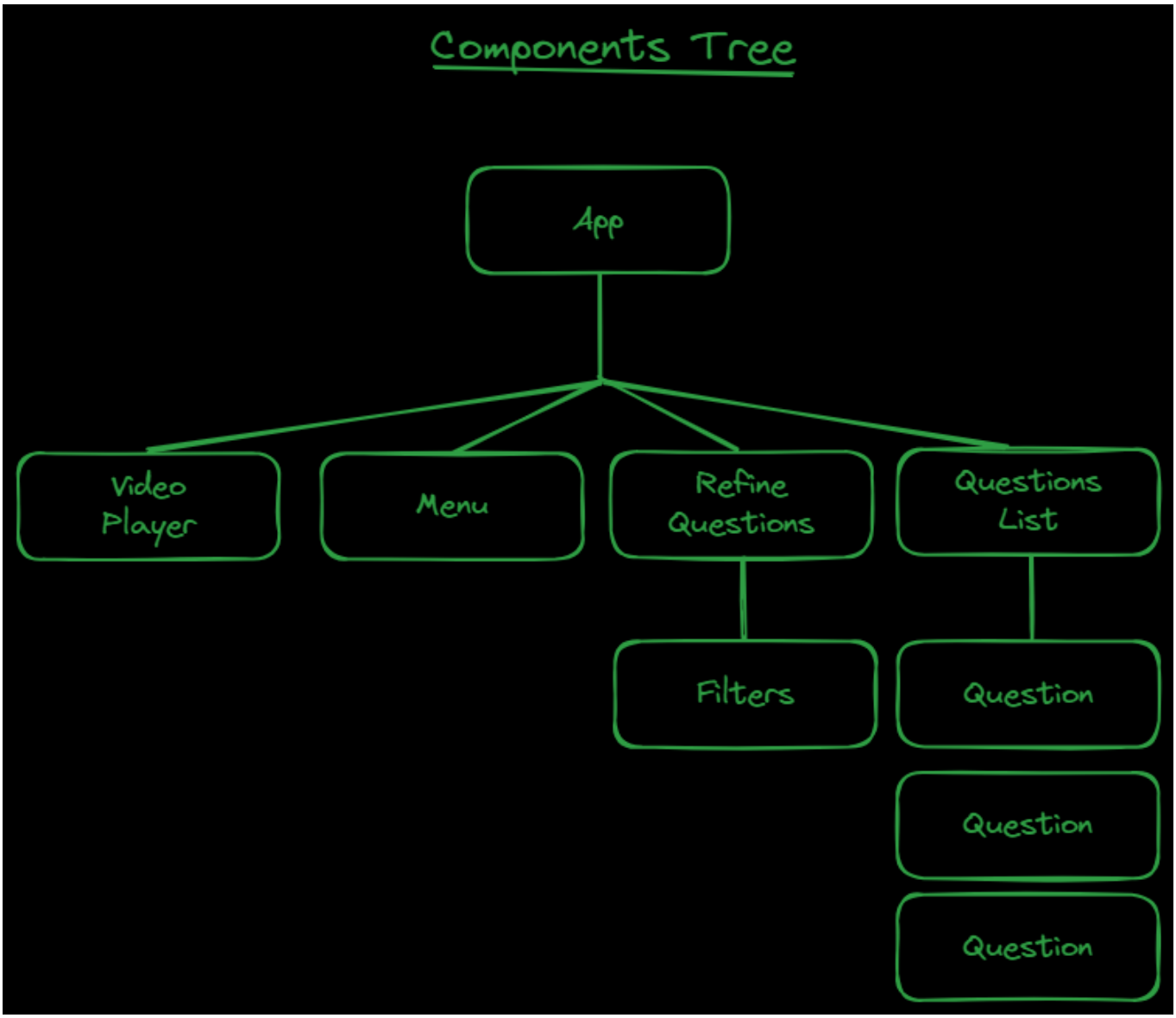
Assume the following procedure,

$$C = A \cup B$$

In JavaScript, it is written as,

```
const A = [1,2,3]
const B = [4,5,6]

const C = [ ... A,  ... B] // [1,2,3,4,5,6]
```



What is JSX?

- Declarative syntax to describe what components look like and how they work

- Components must return a block of JSX
- Extension of JavaScript that allows us to embed JavaScript, CSS, and React components into html
- Each JSX element is converted to a `React.createElement` function call

Props

Props are used to pass data from *parent components* to *child components*

> (down the components tree)

Essential tool to configure and customize components

> (like function parameters)

With props, parent components control how child components look and work

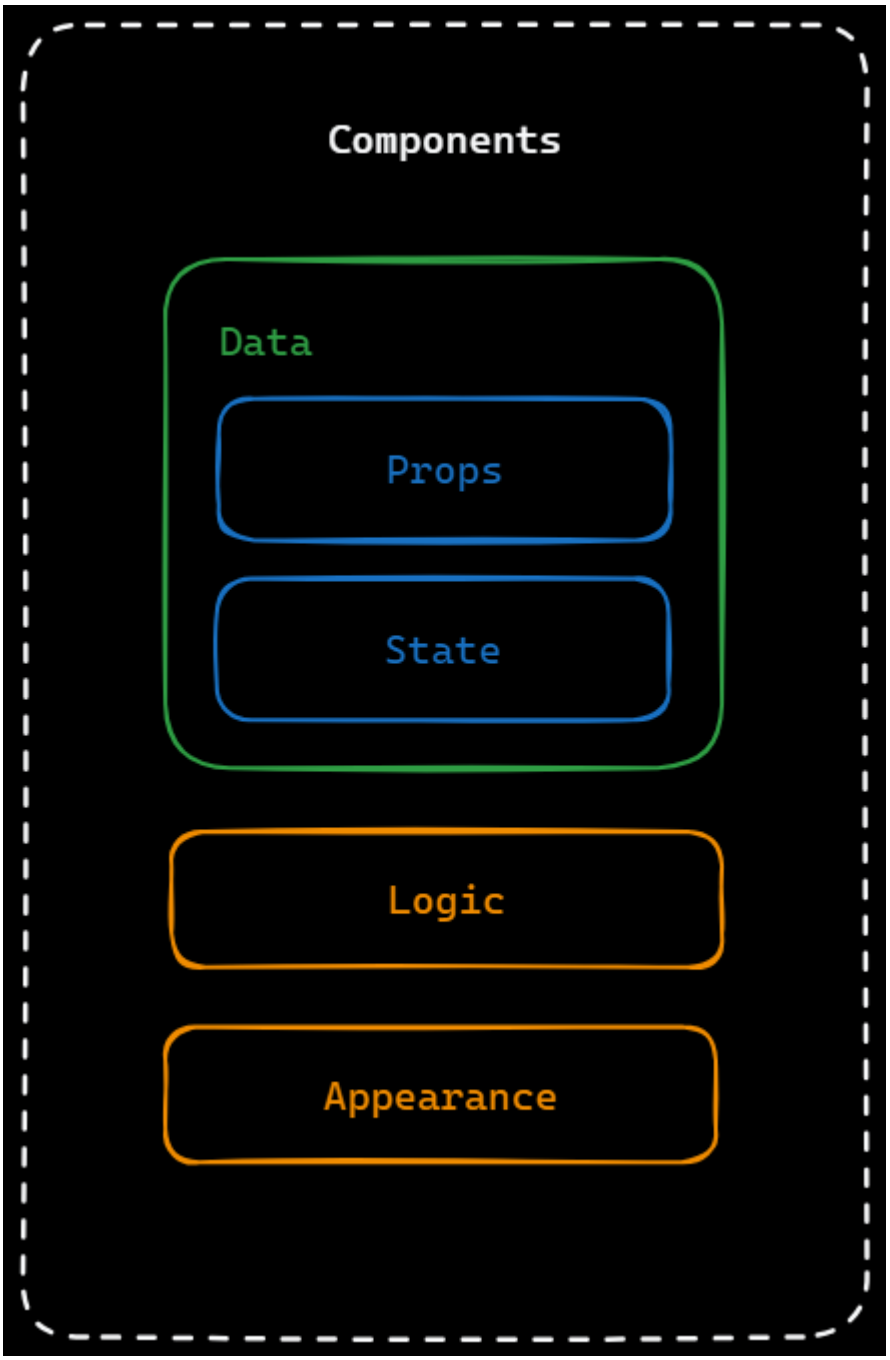
Anything can be passed as props: single values, arrays, objects, functions, even other components

```
function CourseRating()
{
        const [rating, setRating] = useState(0);

        return (
                <Rating
                        text="Course Rating"
                        currentRating={Rating}
                        numOptions={3}
                        options={["Terrible", "Okay", "Amazing"]}
                        allRatings={{num:2390, avg:4.8}}
                        setRating={setRating}
                        component={Star}
                />
        )
}
```

Props is data coming from the ***outside***, and can only be updated by the ***parent component***.

![Props are read-only, they are immutable! This is one of React's strict rules]
![If you need to mutate props, you actually need state]




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

const messages = [
  "Learn React ⚛",
  "Apply for jobs 😍",
```

```
  "Invest your new income 🤑",
]


export default function App()
{
  const [step, setStep] = useState(1)

  function handlePrevious()
  {
    if(step > 1)
      setStep(step - 1);
  }

  function handleNext()
  {
    if(step < 3)
      setStep(step + 1);
  }


  return <>
    <div className="steps">

      <div className="numbers">
        <div className={`${   step >= 1 ? "active" : ""   }`}>1</div>
        <div className={`${   step >= 2 ? "active" : ""   }`}>2</div>
        <div className={`${   step >= 3 ? "active" : ""   }`}>3</div>
      </div>

      <p className="message">
        Step {step} : {messages[step-1]}
      </p>

      <div className="buttons">


        <button
            style={{backgroundColor:"#7950f2", color:"#fff"}}
            onClick={() => handlePrevious()}
          >Previous</button>


        <button
              style={{backgroundColor:"#7950f2", color:"#fff"}}
              onClick={() => handleNext()}
```

```
            >Next</button>
        </div>

    </div>


  </>
}
```

Props

Props (short for properties) are used to pass data from a parent component to a child component. They are read-only and cannot be modified by the child component. Props are immutable, meaning their values cannot be changed once set.

> `ParentComponent.js`

```
import React from 'react';
import ChildComponent from './ChildComponent';


function ParentComponent() {
    return (
        <ChildComponent name="John Doe" age={30} />
    );
}
```

```
export default ParentComponent;
```

`ChildComponent.js`

```
import React from 'react';


function ChildComponent(props) {
    return (
        <div>
            <p>Name: {props.name}</p>
            <p>Age: {props.age}</p>
        </div>
    );
}


export default ChildComponent;
```

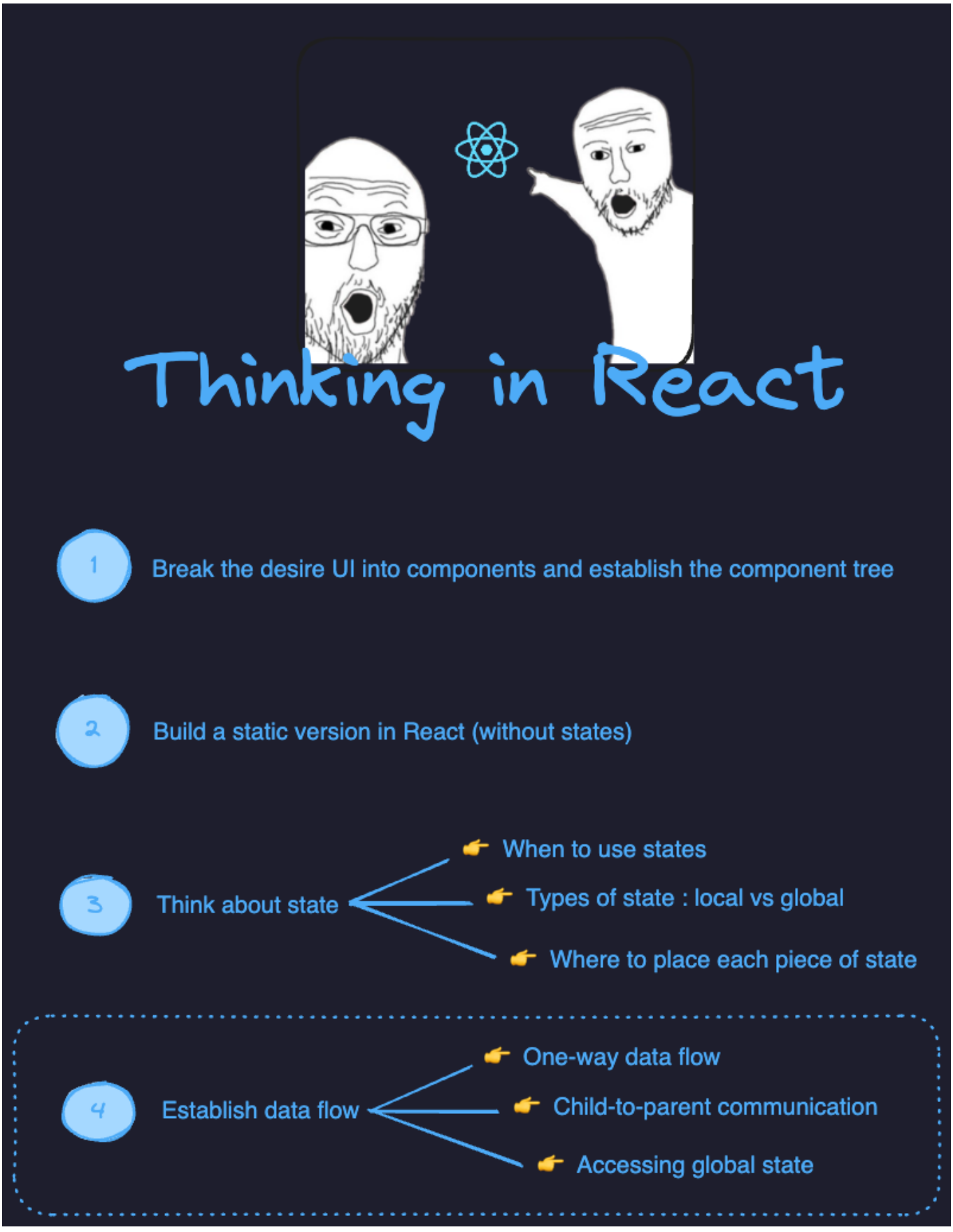
State

State is used to manage data that can change over time within a component. Unlike props, state is mutable and can be updated using the `setState` method (or `useState` hook in functional components). State is local to the component and cannot be accessed or modified by other components.

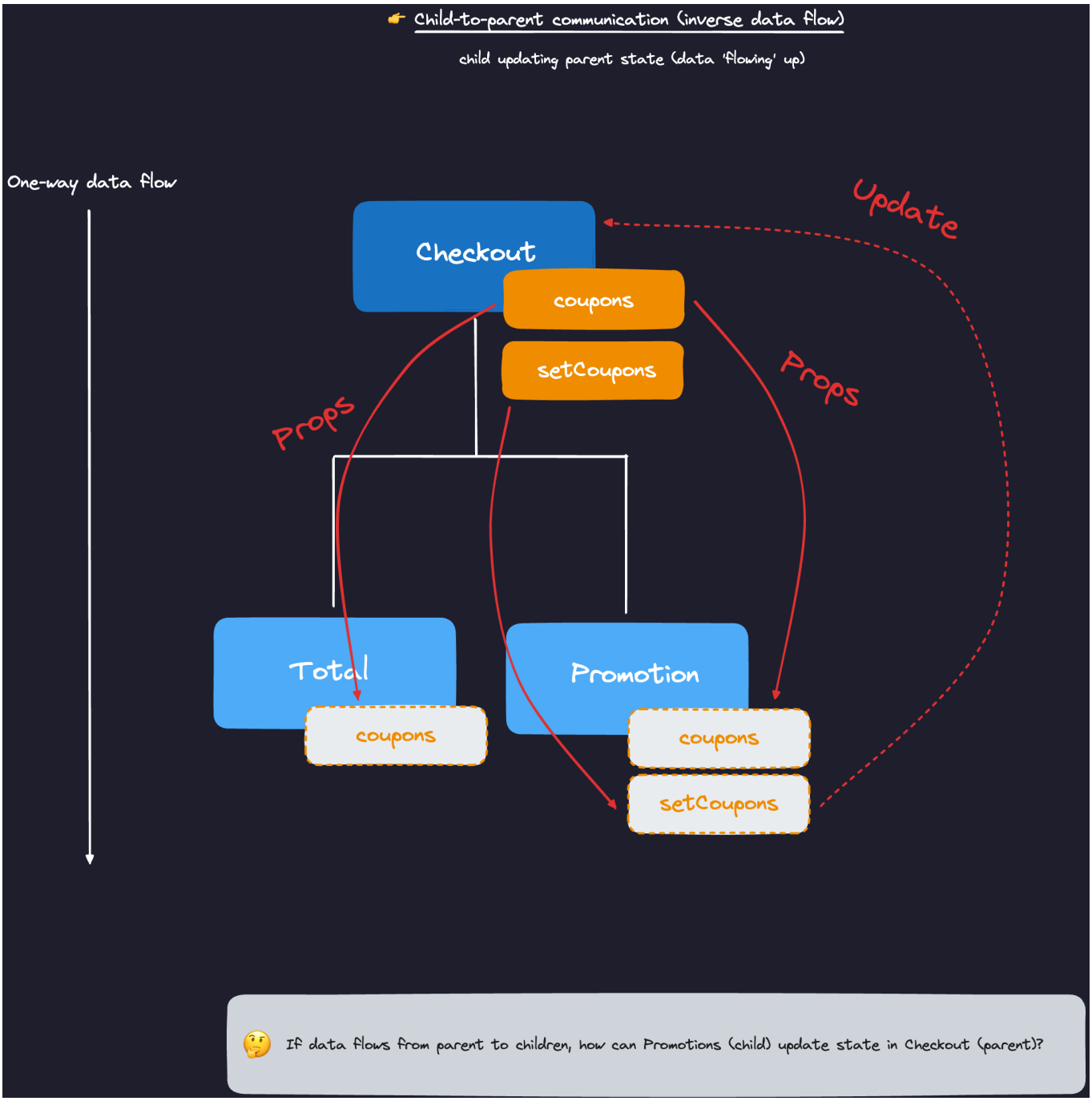
```
import React, { useState } from 'react';


function Counter() {
    const [count, setCount] = useState(0);

    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={() => setCount(count + 1)}>Increment</button>
        </div>
    );
}
export default Counter;
```



Child-to-Parent communications



Deriving State

```
![State that is computed from an existing piece of state or from
props.]
```

```
const [cart, setCart] = useState([
        {name: "JavaScript Course", price: 15.99},
        {name: "Node.js Bootcamp", price: 14.99}
])
```

Bad Practice

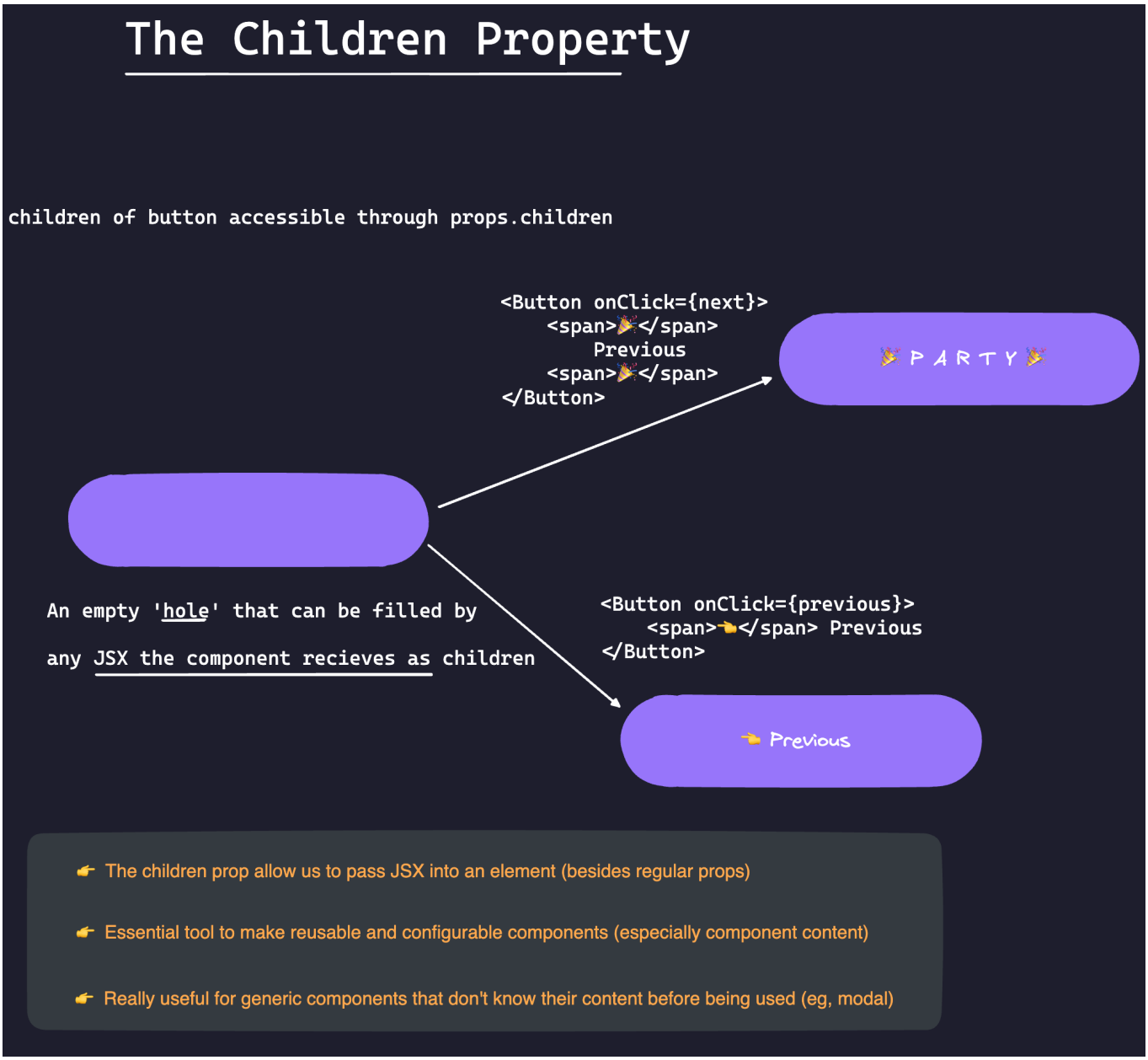
```
const [numItems, setNumItems]     = useState(2);
const [totalPrice, setTotalPrice] = useState(30.98);
```

- Three separate pieces of state, even though `numItems` and `totalPrice`
- Need to keep them in sync (update together)
- 3 state updates will cause 3 re-renders

Good Practice

```
const numItems   = cart.length;
const totalPrice = cart.reduce((acc, cur) => acc + cur.price, 0)
```

- Just regular variables, no `useState`
- Cart state is the *single source of truth for this related* data
- **Works because re-rendering component will *automatically re-calculate* derived state**



Tip Calculator

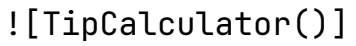
App.jsx

![App()]

```
import { useState } from "react"

export default function App()
{
    return (
```

```
        <div>
            <TipCalculator />
        </div>
    )
}
```




```
function TipCalculator()
{
    const [bill, setBill] = useState(0)
    const [percentage1, SelectPercentage1] = useState(0)
    const [percentage2, SelectPercentage2] = useState(0)

    const tip = bill * (percentage1 + percentage2) / (2 * 100)

    return <div>
        <BillInput
                        bill={bill}
                        setBill={setBill}/>
        <SelectPercentage
                            percentage={percentage1}
                            SelectPercentage={SelectPercentage1}>
                        How did you like the service?
            </SelectPercentage>
        <SelectPercentage
                            percentage={percentage2}
                            SelectPercentage={SelectPercentage2}>
                        How did your friend like the service?
                </SelectPercentage>
        <Output bill={bill} tip={tip} />
        <Reset />
    </div>
}


function BillInput({ bill , setBill })
{
    return <div>
        <label>How much was the bill?</label>
        <input type="number" onChange={(e) =>
setBill(Number(e.target.value))}/>
    </div>
}
```

```
function SelectPercentage({ percentage , SelectPercentage , children })
{


    return <div>
        <label>{children}</label>
        <select value={percentage} onChange={e =>
SelectPercentage(Number(e.target.value))}>
            <option value={0}  >    Very dissatisfied (0%)  </option>
            <option value={5}  >    dissatisfied (5%)       </option>
            <option value={10} >    Neutral (10%)           </option>
            <option value={15} >    satisfied (15%)         </option>
            <option value={20} >    Very satisfied (20%)    </option>
        </select>
    </div>
}


function Output({ bill , tip })
{
    return <h3>You pay ${bill + tip} (${bill} + ${tip})</h3>
}


function Reset()
{
    return <button>Reset</button>
}
```

Using An API

In order to utilize APIs, you first have to ensure that `Axios` is installed:

```
npm run dev
```

Then your APIs could be written like the following, which uses the `Unsplash API`.

```
import axios from 'axios';

const api_link = "https://api.unsplash.com/search/photos"
```

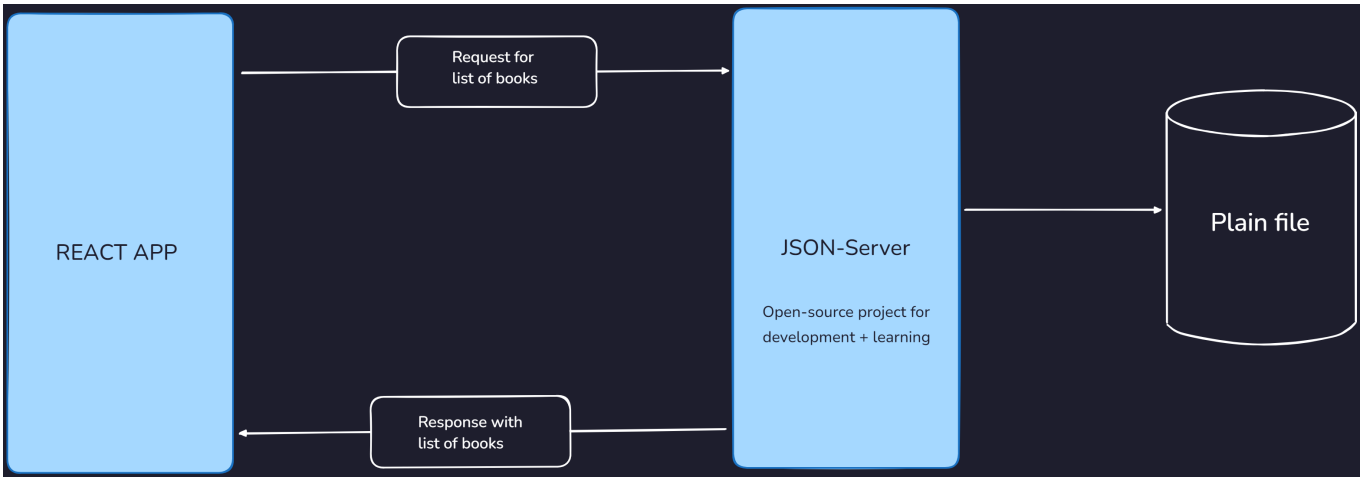
```
const access_key = "8050V7bNzfKdVixwS9W9nZVdr0VnrCv9gmeimfdvp6Y"

const searchImages = async () => {
        const response = await axios.get(api_link, {
                headers:{
                        Authorization: `Client-ID ${access_key}`
                },
                params:{
                        query: "cars"
                }
        })

        return reponse
}

export default searchImages
```

JSON Server Issues and Required version




```
npm install json-server@0
```

Creating a `context`

```
const NumberContext = React.createContext();

function App() {
  return (
```

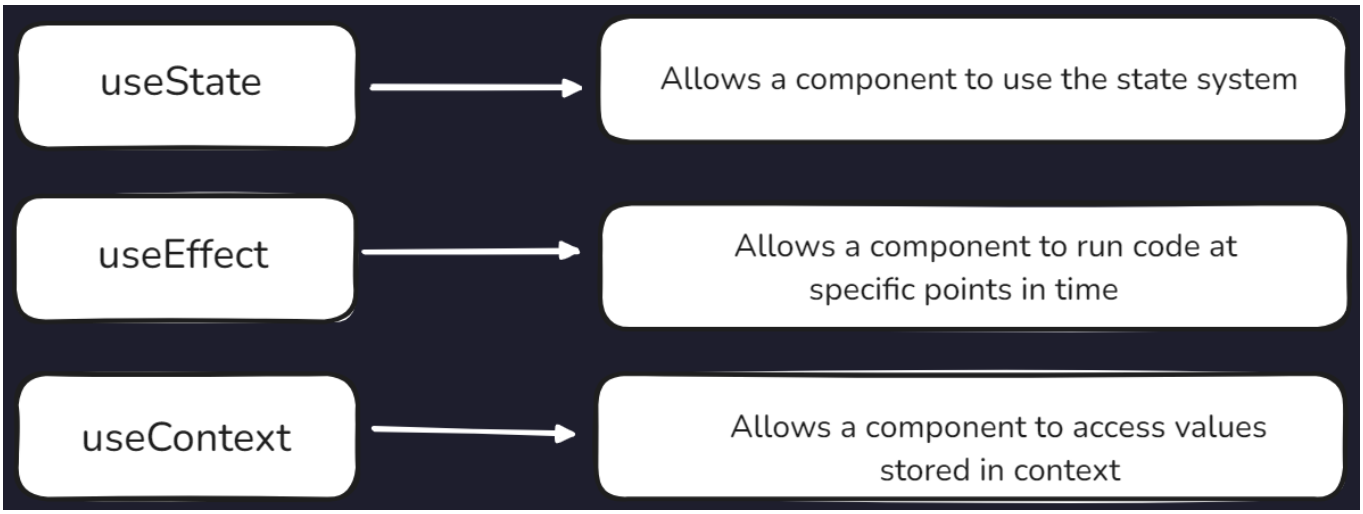
```
    <NumberContext.Provider value={42}>
      <Display />
    </NumberContext.Provider>
  );
}
```

Using `useContext`:

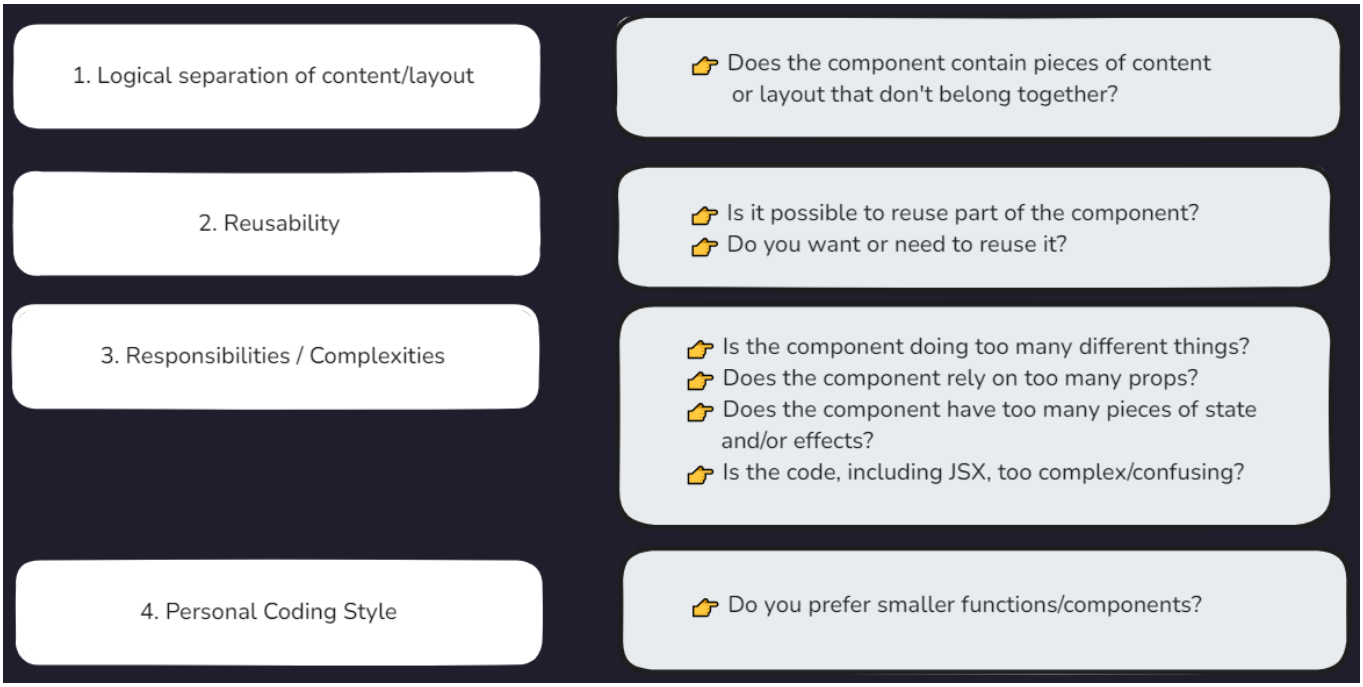
```
function Display() {
  const value = useContext(NumberContext);
  return <div>The answer is {value}.</div>;
}
```

Using `Consumer` (the traditional way):

```
function Display() {
  return (
    <NumberContext.Consumer>
      {value => <div>The answer is {value}.</div>}
    </NumberContext.Consumer>
  );
}
```



GENERAL GUIDELINES FOR COMPONENTS



💰 Be aware that creating a new component [creates a new abstraction]. Abstraction have a [cost], because [more abstractions require more mental energy] to switch back and forth between components. So try not to create new components too early.

🏷 Name a component according to [what it does] or [what it displays]. Don’t be afraid of using long component names.

🧅 Never declare a new component [inside another component!]

🍫 [Co-locate related components inside the same file.] Don’t separate components into different files too early.

↔ It's completely normal that an app has components of many different size, including very small and huge ones

Component categories

Most components will naturally fall into one of the following categories:

- Stateless / Presentation Components
  - [No state]
  - Can receive props and simply present received data or other content
  - Usually [small and reusable]
- Stateful Components
  - [Have state]
  - Can still be [reusable]
- structural Components
  - “[Pages]”, “[Layouts]”, or “[Screens]” of the app
  - Result of [composition]
  - Can be [huge and non-reusable] (but don’t have to)

React’s Cross-site scripted attack protection

React components all have a `$$typeof: Symbol(react.element)`

[symbols cannot be transmitted via JSON, so if a fake react element is sent (from attackers), it will not have this property attached.]

Calling components

---

calling components like

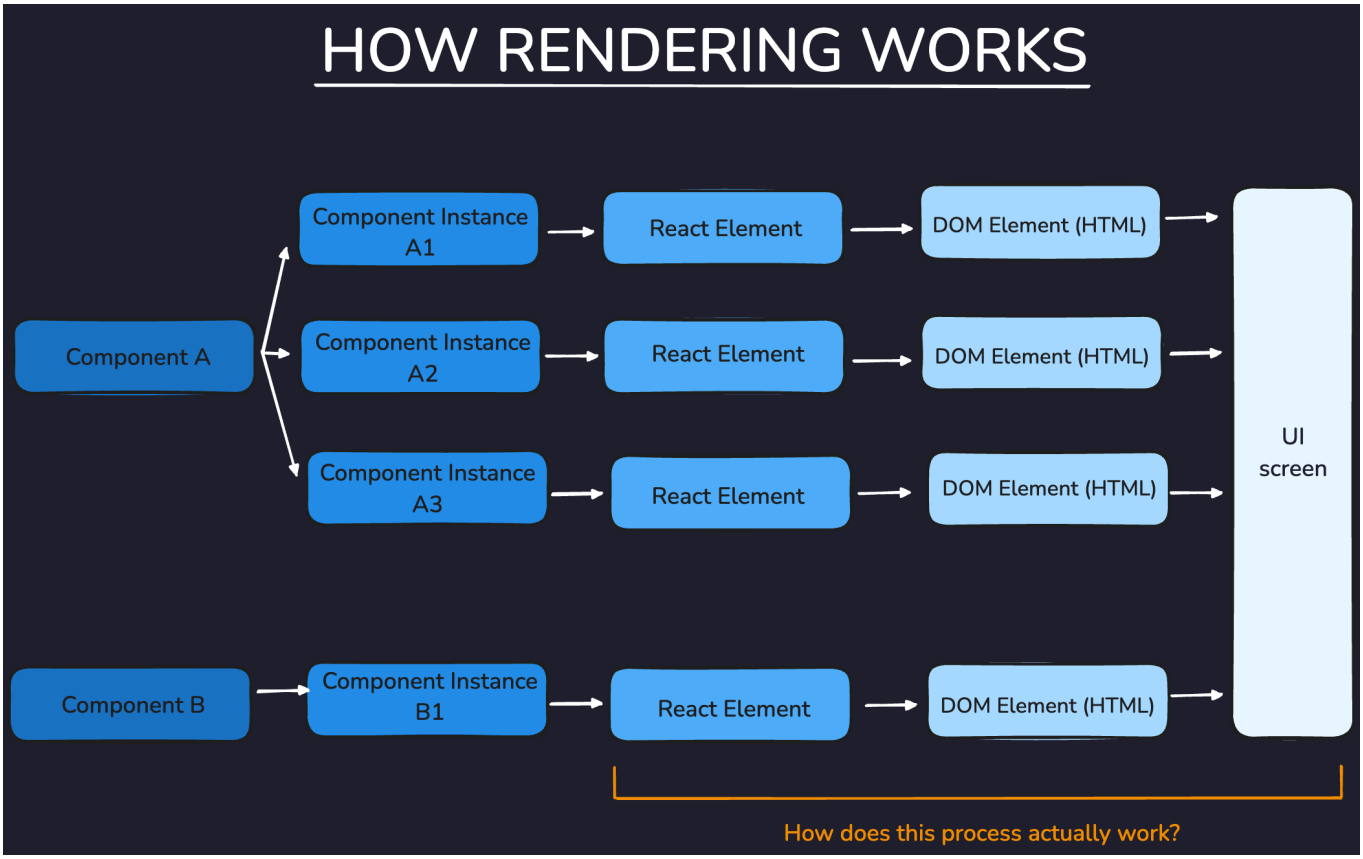
```
<this />
```

is not the same as calling it like

```
this()
```

because the top returns an [instance of a React component] whereas the bottom directly returns a [type “div” that is the raw element of that component].

React will not be able to see the component if it is passed like `this()`. So it is important to always call it using JSX like `<this />`.



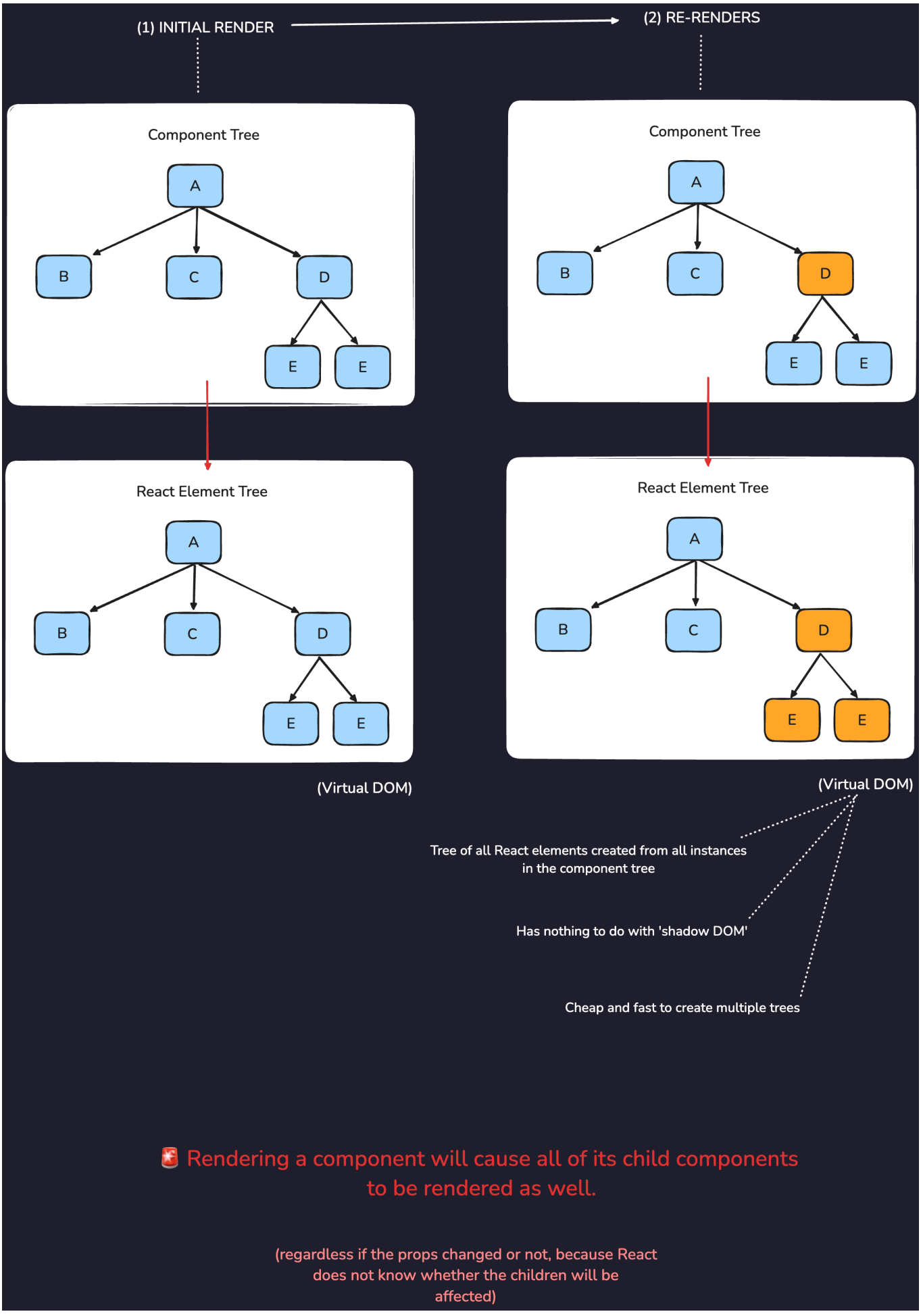
THE TWO SITUATIONS THAT TRIGGER RENDERS

1. Initial render of the application
2. State is updated in one or more component instances (re-render)

👉 The render process is triggered fro the entire application

👉 In practice, it looks like React only re-renders the component where the state update happens, but that's not how it works behind the scenes

👉 Renders are not triggered immediately, but scheduled for when the JS engine has some 'free-time'. There is also batching of multiple setState calls in event handlers


(1) INITIAL RENDER
(2) RE-RENDERS
Component Tree
A
B
C
D
E
E
Component Tree
A
B
C
D
E
E
React Element Tree
A
B
C
D
E
E
(Virtual DOM)
React Element Tree
A
B
C
D
E
E
(Virtual DOM)
Tree of all React elements created from all instances in the component tree
Has nothing to do with 'shadow DOM'
Cheap and fast to create multiple trees
🚨 Rendering a component will cause all of its child components to be rendered as well.
(regardless if the props changed or not, because React does not know whether the children will be affected)

KEY PROP

- Special props that we use to tell the diffing algorithm that an element is unique
- Allows React to distinguish between multiple instances of the same component type
- When a key stays the same across renders, the element will be kept in the DOM (even if the position in the tree changes)
- Using keys in lists:
  - When a key changes between renders, the element will be destroyed and a new one will be created (even if the position in the tree is the same as before)
- Using keys to reset state

Assume a list of items [without keys],

```
<ul>
        <Question question={q[1]} />
        <Question question={q[2]} />
</ul>
```

Assuming a new list item is added,

```
<ul>
        <Question question={q[0]} />
        <Question question={q[1]} />
        <Question question={q[2]} />
</ul>
```

Same elements, but different positions in tree, so they are removed and recreated in the DOM (![BAD FOR PERFORMANCE])

If [keys] are used:

```
<ul>
        <Question key="q1" question={q[1]} />
        <Question key="q2" question={q[2]} />
</ul>
```

When adding a new list item,

```
<ul>
        <Question key="q0" question={q[0]} />
        <Question key="q1" question={q[1]} />
        <Question key="q2" question={q[2]} />
</ul>
```

Assume the following:

```
function Question({question}){
        const [newAnswer, setNewAnswer] = useState('')
        const numAnswers = question.answers.length ?? 0;

        const handleNewAsnwer = function(e){
                if (question.closed) return;
                setNewAnswer(e.target.value)
        }

        const createList = function(){
                return (
                        <ul>
                                {question.answers.map((q) => (
                                        <li>{q}</li>
                                ))}
                        </ul>
                )
        }

        return (
        <div>
                <h3>{question.title}</h3>
                <p>{question.body}</p>
                {question.hasAnswer} ? (
                        createList()
                ) : (
                        <input value={newAnswer} onChange={handleNewAnswer}
/>
                )
        )
}
```

Render Logic

👉 Code that lives at the top level of the component function

👉 Participates in describing how the component view looks like

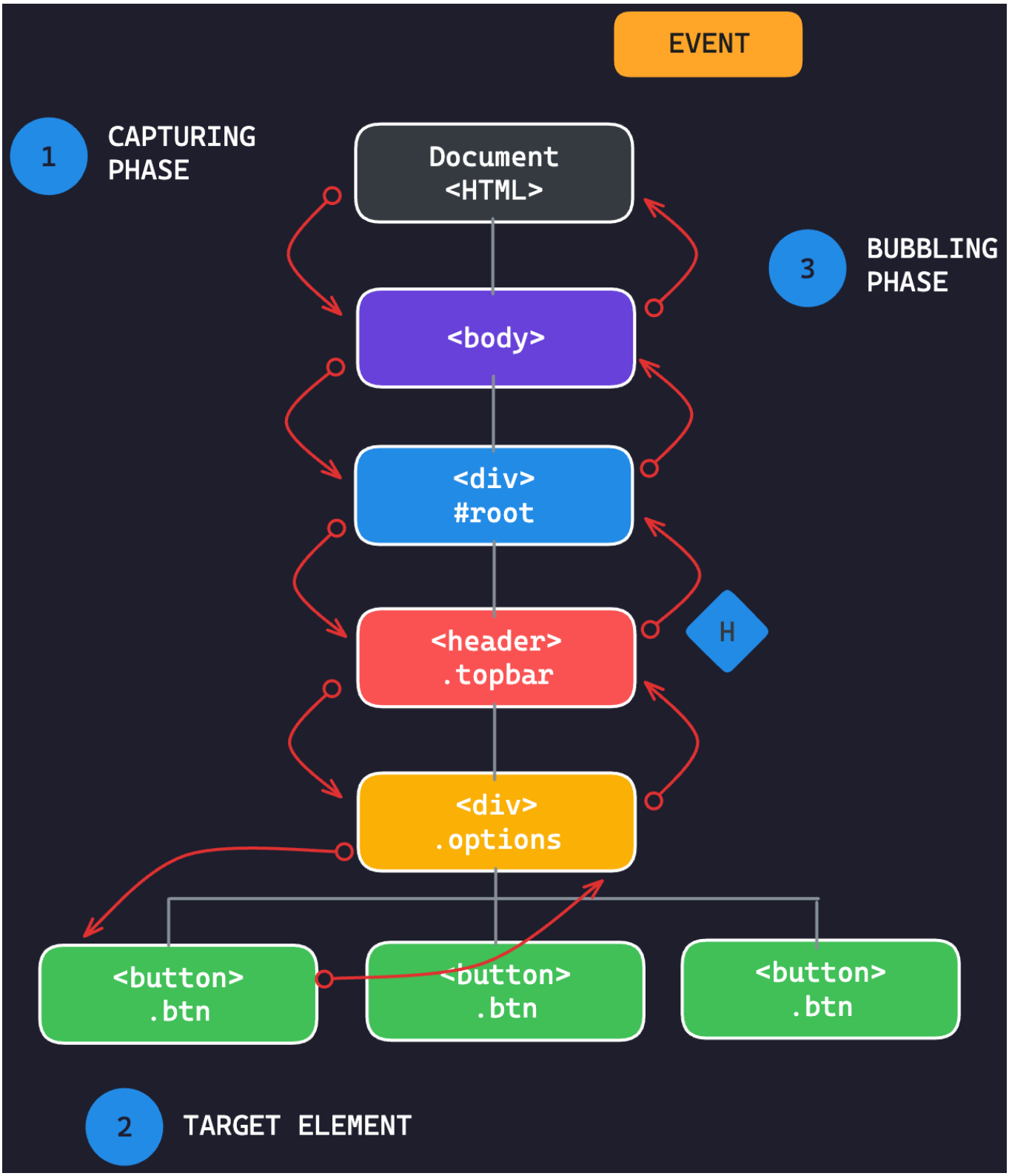
👉 Executed every time the component renders

[All `const` and `return` functions are part of the render logic]

Event Handler Function

👉 Executed as a consequence of the event that the handler is listening for (change event in this example)

👉 Code that actually does things: update state, perform an HTTP, request, read an input field, navigate to another page, etc…



- By default, event handlers listen to events on the target *and during the bubbling phase*
- We can prevent bubbling with `e.stopPropagation()`