REACT

Some Javascript Concept may uses in react

Array.from()

Array.from() is a built-in JavaScript method that creates a new array from:

- An array-like object (like { length: 5 })
- Or an iterable (like a string, Set, Map, etc.)
- Optionally, you can also transform items while creating the array using a mapping function.
- Array.from(arrayLike, mapFunction);

📌 1. From Array-like Object

```
js                                                    Copy  Edit

Array.from({ length: 5 });
// Output: [undefined, undefined, undefined, undefined, undefined]
```

Each item is `undefined` unless you map it.

📌 2. With Mapping Function (to get numbers 1–5)

```
js                                                    Copy  Edit

Array.from({ length: 5 }, (_, i) => i + 1);
// Output: [1, 2, 3, 4, 5]
```

- `(_, i)` means:
  - `_` : the actual value (we ignore it)
  - `i` : the index (0–4)
  - `i + 1` : we turn it into 1–5

Why react came in picture

🤔 First, Understand Two Styles of Programming

Style	What it means	How it feels
Imperative	You tell the computer how to do things step-by-step.	Like giving cooking instructions.
Declarative	You tell the computer what you want, and it figures out the how.	Like saying "Make me a pizza"—you care about the result, not the steps.

🔍 So Why Use React When We Have JS?

Feature	JavaScript (DOM)	React
DOM updates	Manual	Automatic
Code style	Imperative	Declarative
State management	You do it yourself	`useState`, `useEffect` etc.
Reusability	Harder	Easy (Components)
Scaling apps	Difficult	Much easier
Performance	Okay	Fast with Virtual DOM
Maintainability	Can become messy	Cleaner structure

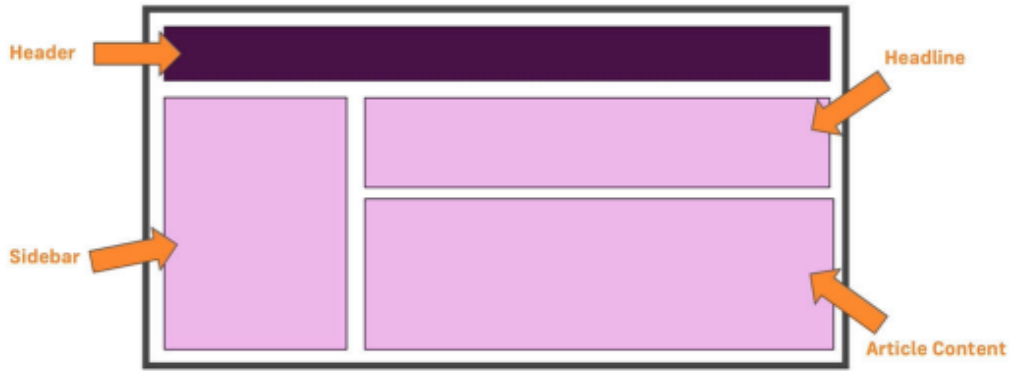
📜 JavaScript vs React (Summary Table)

Feature	JavaScript (Imperative)	React (Declarative)
Programming Style	Imperative (step-by-step instructions)	Declarative (describe what UI should look like)
DOM Manipulation	Manual (e.g., `document.getElementById`)	Automatic via Virtual DOM
UI Updates	You handle state + DOM updates yourself	React handles re-rendering automatically
Component Reuse	Harder to structure and reuse	Easy with components
State Management	Manual (global vars, event listeners)	Built-in with `useState`, `useReducer`, etc.
Code Maintenance	Becomes complex as app grows	Cleaner and more manageable
Learning Curve	Lower initially	Slightly higher, but pays off in large apps
Performance	Depends on how you write code	Efficient updates with Virtual DOM
Best For	Small/simple scripts or legacy projects	Modern, scalable, dynamic web apps

REACT

What is React?

- From the official React page : A JavaScript library for building user interfaces
- Its not a framework; React does only one thing – create awesome UI!
- React is used to build single page applications.
- React.js is a JavaScript library. It was developed by engineers at Facebook.
- React is a declarative, efficient, and flexible JavaScript library for building user interfaces.
- It lets you compose complex UIs from small and isolated pieces of code called “components”.

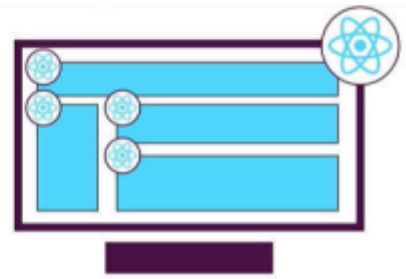


- It uses VirtualDOM instead of RealDOM considering that RealDOM manipulations are expensive.
- Supports server-side rendering.
- Follows Unidirectional data flow or data binding.
- Uses reusable/composable UI components to develop the view.

Components

- A **Component** is one of the core building blocks of React.
- Its just a **custom HTML element**!

- Every application you will develop in React will be made up of pieces called components.
  - Components make the task of building UIs much easier. You can see a UI broken down into multiple individual pieces called components and work on them independently and merge them all in a parent component which will be your final UI.

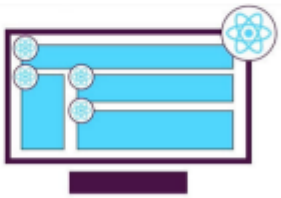


Creating a functional component

- Components are the essential building blocks of any app created with React
  - A single app most often consists of many components.
- A component is in essence, a piece of the UI - splitting the user interface into reusable and independent parts, each of which can be processed separately.
  - Components are independent and reusable bits of code.
  - It's an encapsulated piece of logic.
  - They serve the same purpose as JavaScript functions, but work in isolation and return HTML via a render function.

```
function ExpenseItem(){
   return <h2>Expense Item</h2>
}
export default ExpenseItem;
```

```
const ExpenseItem = () => {
   return <h2>Expense Item</h2>
}
export default ExpenseItem;
```



Components & JSX

- When creating components, you have the choice between two different ways:
- Functional components (also referred to as "presentational", "dumb" or "stateless" components)

```
const cmp = () => {
   return <div>some JSX</div>
}
```

using ES6 arrow functions as shown here is recommended but optional

- class-based components (also referred to as "containers", "smart" or "stateful" components)

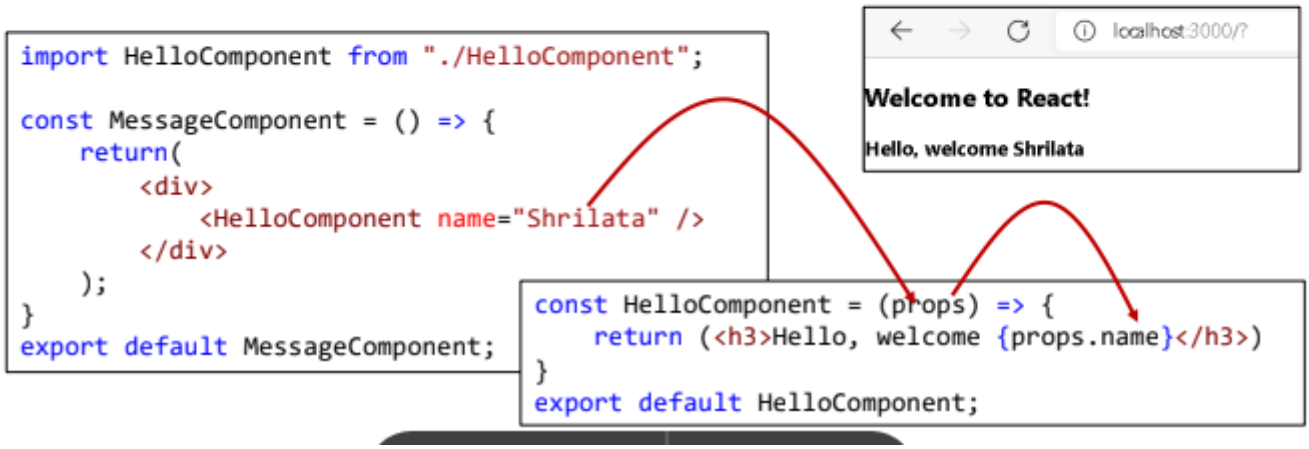
```
class Cmp extends Component {
   render () {
        return <div>some JSX</div>
   }
}
```

PROPS

Props (short for "properties") are read-only inputs that you pass from a parent component to a child component in React.

Passing data via 'Props'

- "Props" stands for properties.
  - It is a special keyword in React used for passing data from one component to another.
  - Props are arguments passed into React components.
  - props are read-only. So, the data coming from a parent component can't be changed by the child component.
  - Props are passed to components via HTML attributes.
  - Props can be used to pass any kind of data such as: String, Array, Integer, Boolean, Objects or, Functions



They allow components to be reusable and dynamic by giving them custom data.

💡 Key Points about Props

Feature	Explanation
Read-only	Props cannot be changed inside the child component.
Passed down	Always flow from parent → child.
Reusable	Make components flexible with different data.
Destructuring	Common to use: `function Component({ title }) {}`

🔒 Can I Modify Props?

- ❌ No — Props are immutable inside the component.
- If you need to change data, you should use state (useState) instead.

```jsx
function Menu() {
  return (
    <main className="menu">
      <h2>Our Menu</h2>
      <Pizza
        name="Pizza Funghi"
        ingredient="Tomato, mozarella, mushrooms, and onion"
        photoName="pizzas/funghi.jpg"
        price={100}
      />
      <Pizza
        name="Pizza Prosciutto"
        ingredient="Tomato, mozarella, ham, aragula, and burrata cheese"
        photoName="pizzas/prosciutto.jpg"
        price={180}
      />
    </main>
  );
}
```

```jsx
function Pizza(props) {
  console.log(props);
  return (
    <div className="pizza">
      <img src={props.photoName} alt={props.name} />
      <div>
        <h3>{props.name}</h3>
        <p>{props.ingredient}</p>
        <span>{props.price + 3}</span>
      </div>
    </div>
  );
}
```

Destructuring Props

Props destructuring in React is the practice of extracting specific properties from the props object directly, so you can access them without repeatedly writing props. in your component.

- Props destructuring makes React components more concise, readable, and maintainable by letting you directly extract needed values from the props object.

✅ Why We Need It:

We use props destructuring to:

1. **Write cleaner and shorter code**
   → Avoid repeating `props.name`, `props.age`, etc.
2. **Improve readability**
   → You see exactly which props the component uses — right in the function signature.
3. **Avoid clutter inside JSX**
   → Especially useful when there are multiple props.
4. **Make code more maintainable**
   → Easier to update, refactor, and debug.

🔁 Comparison Table

Aspect	Without Destructuring	With Destructuring
Access syntax	`props.name`, `props.age`	`name`, `age`
Readability	👎 Cluttered with `props.` prefix	👍 Clear and focused
Code length	Longer	Shorter
Best in JSX?	No — looks noisy	Yes — looks clean
Common use?	Less common	✅ Preferred in modern React

🛠️ Without Destructuring

jsx Copy Edit

```jsx
function Welcome(props) {
  return (
    <h1>
      Hello {props.name}, you are {props.age} years old!
    </h1>
  );
}
```

- You access values with `props.name`, `props.age`, etc.
- Works fine — but becomes **verbose** with many props.

✨ With Destructuring

✅ Inside the function body:

```jsx
function Welcome(props) {
  const { name, age } = props;

  return (
    <h1>
      Hello {name}, you are {age} years old!
    </h1>
  );
}
```

- You pull out `name` and `age` from `props` into their own variables.
- This makes the JSX cleaner.

✅ Even Cleaner: Destructure in the parameter list

```jsx
function Welcome({ name, age }) {
  return (
    <h1>
      Hello {name}, you are {age} years old!
    </h1>
  );
}
```

- This is the **most common** and cleanest way.
- You skip `props` entirely and pull out only what you need.

LIST RENDERING

📚 Rendering a List in React

To render a list of items in React, you typically:

1. Use the `.map()` function to loop over an array.
2. Return **JSX** for each item.
3. Provide a **unique** `key` **prop** to each list item.

Why Use a key Prop?

The key helps React identify which items changed, were added, or removed, which improves performance and avoids rendering bugs.

- 🔑 Key should be unique and stable.
- Avoid using index as key if the list can change.

🎯 Key Concepts Explained

1. `.map()`

- A JavaScript array method.
- Takes a callback and returns a new array.
- Used to convert an array of data → array of JSX.

```js
array.map(item => <JSX />)
```

2. `key` Prop

React needs a unique key to track each element:

```jsx
<li key={book.id}>...</li>
```

```jsx
function Menu() {
  return (
    <main className="menu">
      <h2>Our Menu</h2>
      <div>
        <ul className="pizzas">
          {pizzaData.map((pizza) => (
            <Pizza pizzaObject={pizza} key={pizza.name} />
          ))}
        </ul>
      </div>
    </main>
  );
}

function Pizza(props) {
  console.log(props);
  return (
    <li className="pizza">
      <img src={props.pizzaObject.photoName} alt={props.pizzaObject.name} />
      <div>
        <h3>{props.pizzaObject.name}</h3>
        <p>{props.pizzaObject.ingredients}</p>
        <span>{props.pizzaObject.price + 3}</span>
      </div>
    </li>
  );
}
```

🔁 `.map()` is **Pure**

What does "pure" mean?

> A **pure function** is one that **does not change anything outside itself** and **always returns a new value** based only on its input.

✅ `.map()`:

- Takes an array.
- Processes each element.
- **Returns a new array** with the transformed results.
- **Does not change** the original array.
- **No side effects.**

🧠 Example:

```
js                                                    Copy    Edit

const numbers = [1, 2, 3];
const doubled = numbers.map(n => n * 2);
console.log(doubled); // [2, 4, 6]
console.log(numbers); // [1, 2, 3] - original array unchanged
```

✔ This is **pure and declarative** — you declare what the new data should look like.

🔄 `.forEach()` is **Imperative**

What does "imperative" mean?

> An **imperative function** tells the program **how to do something**, often involving **side effects** (modifying something outside the function).

❌ `.forEach()`:

- Loops over an array.
- **Does not return anything.**
- Meant for running side effects (e.g., logging, changing variables).
- Often used when you're **doing something**, not **producing a new result**.

🧠 Example:

```
js                                                    Copy    Edit

const numbers = [1, 2, 3];
const doubled = [];

numbers.forEach(n => {
  doubled.push(n * 2); // Side effect: modifying `doubled` array
});

console.log(doubled); // [2, 4, 6]
```

✔ This is **imperative** — you manually push items, m ge state, and give instructions.

Why Use map() Instead of forEach() in React for Rendering Lists

Fundamental Difference

- `map()` returns a new array with transformed elements
- `forEach()` returns undefined and only executes a function for each element
- Always use map() when you need to render a list in JSX
- Use forEach only for side effects when you don't need the return value

🧠 Why Use `.map()` in React (Not `forEach`):

1. **React needs JSX to be returned**
   When you're rendering a list in React, you need to **return JSX elements** (like `<li>`, `<div>`, `<Card />`) so React knows what to display on the screen.
   - `.map()` creates and **returns a new array** of these elements.
   - `.forEach()` does **not return anything**. It just runs code for each item — which is useful for side effects, but not for rendering.

---

2. `.map()` **fits naturally in JSX**
   You can use `.map()` **directly inside JSX**:

```jsx
{users.map(user => <li>{user}</li>)}
```

   But `.forEach()` doesn't return a value, so you **can't use it like this.**

3. **Cleaner, more concise code**
   `.map()` allows you to transform data into UI in a **single line.**
   With `.forEach()`, you'd need to:
   - create an empty array,
   - manually push items into it,
   - and then return it.

---

4. **React community standard**
   Using `.map()` is the **standard and recommended** way to render lists in React. It's used in almost every React codebase and documentation, so it keeps your code **idiomatic and familiar** to other developers.

✅ Summary (in one sentence):

> We use `.map()` in React to render lists because it returns an array of JSX elements, which React uses to display content — while `.forEach()` does not return anything useful for rendering.

Conditional Rendering?

Conditional rendering means showing different UI / div / some portion based on a condition — like whether a user is logged in, or if a shop is open.

- React uses JavaScript logic (like if, ?:, &&) inside JSX to decide what to render.

1) Ternary Operator condition ? trueUI : falseUI

```
{isLoggedIn ? <p>Welcome back!</p> : <p>Please log in.</p>}
```

2) Logical AND (&&) — render only if condition is true

```
{isOpen && <p>The shop is open!</p>}
```

3) if...else (outside JSX)

- Use this if the condition is complex or the JSX is long.

```
let content;
if (isOpen) {
  content = <p>We are open</p>;
} else {
  content = <p>We are closed</p>;
}

return <div>{content}</div>;
```

4) Early return (guard clause)

```
if (!isOpen) return <p>Closed</p>;

return <p>We're open!</p>;
```

🧩 What are React Fragments?

A React Fragment lets you group multiple elements without adding an extra node to the DOM.

❓ Why Use Fragments?

Normally in React, components must return a single parent element. If you want to return multiple siblings, you usually wrap them in a `<div>` — but that adds unnecessary HTML to the DOM.

🚫 Without Fragment:

jsx Copy Edit

```jsx
return (
  <div>
    <h1>Hello</h1>
    <p>Welcome</p>
  </div>
);
```

This adds an **extra** `<div>` to the DOM.

✅ With Fragment:

jsx Copy Edit

```jsx
return (
  <>
    <h1>Hello</h1>
    <p>Welcome</p>
  </>
);
```

This adds **no extra HTML** — just the `<h1>` and `<p>`.

Where to Use Fragments?

1. Inside .map() when rendering a list of elements that don't need a wrapper

```
const items = ["Pizza", "Pasta"];

return (
  <ul>
    {items.map((item, index) => (
      <React.Fragment key={index}>
        <li>{item}</li>
        <hr />
      </React.Fragment>
    ))}
  </ul>
);
```

2. Inside a component when returning multiple sibling elements.

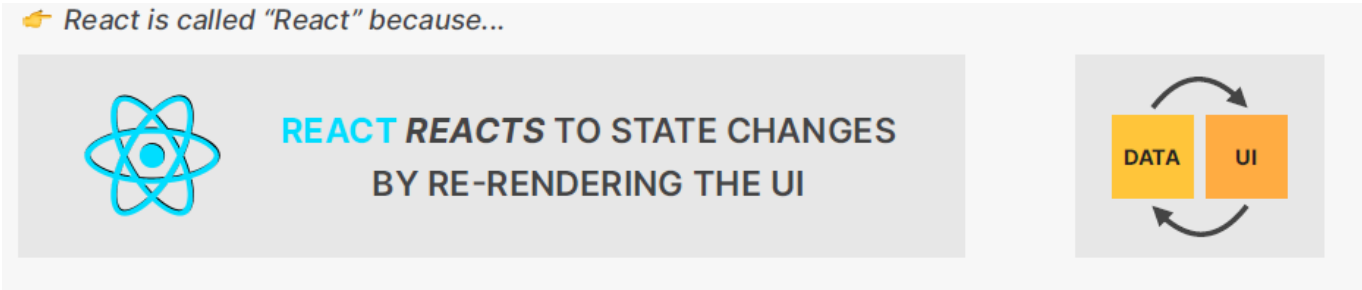
```
function Welcome() {
  return (
```

```
    <>
      <h1>Hi!</h1>
      <p>Welcome to our store.</p>
    </>
  );
}
```

3. In tables, where adding extra
   would break structure — instead use fragments to return and properly.

```
function Row() {
  return (
    <div>
      <tr>
        <td>Pizza</td>
        <td>$10</td>
      </tr>
    </div>
  );
}
```

REACT STATE



React State

- The state is a built-in React object that is used to contain data or information about the component.
- A component's state can change over time; whenever it changes, the component re-renders.
  - The change in state can happen as a response to user action or system-generated events and these changes determine the behavior of the component and how it will render.
- A component with state is known as stateful component.

- State allows us to create components that are dynamic and interactive.
  - State is private, it must not be manipulated from the outside.
  - Also, it is important to know when to use 'state', it is generally used with data that is bound to change.

Each component has and manages its own state, no matter how many times we render the same component

React Hooks

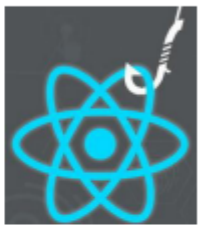
- Hooks allow us to "hook" into React features such as state and lifecycle methods
  - React Hooks are special functions provided by React to handle a specific functionality inside a React functional component.
  - Eg React provides *useState()* function to manage state in a functional component.
  - When a React functional component uses React Hooks, React Hooks attach itself into the component and provides additional functionality.

- You must import Hooks from react
  - Eg : import React, { useState } from "react"; Here - useState is a Hook to keep track of the application state.

- There are some rules for hooks:
  - Hooks can only be called inside React function components.
  - Hooks can only be called at the top level of a component.
  - Hooks cannot be conditional
  - Hooks will not work in React class components.
  - If you have stateful logic that needs to be reused in several components, you can build your own custom Hooks

Working with “state” in functional component

- The React useState Hook allows us to track state in a function component.
- To use the useState Hook, we first need to import it into our component.
  - import { useState } from "react";
  - We initialize our state by calling useState in our function component.



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

const UseStateComponent = () => {
    useState(); //hooks go here
}
```

  - useState accepts an initial state and returns two values:
  1. The current state.
  2. A function that updates the state.
  - Eg:

```
function FavoriteColor() {
    const [color, setColor] = useState("");
}
```

    - The first value, color, is our current state.
    - The second value, setColor, is the fuction that is used to update our state.
    - Lastly, we set the initial state to an empty string: useState("")

Working with “state” in functional component

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

const UseStateComponent = () => {
    const [counter, setCounter] = useState(0);  //hooks go here

    const btnHandler = () => {
        setCounter(counter+1);
        console.log(counter, "  button clicked")
     }
    return(
           <div>
              Counter : {counter}   
              <button onClick={btnHandler}>increment counter</button>
           </div>
        );
}
export default UseStateComponent;
```

LOOP HOLE

1. ☑ const [step, setStep] = useState(1);

- We use const here because:
  - useState() returns an array with two values: the current state (step) and the updater function (setStep).
  - You are not reassigning the [step, setStep] pair

2. ❌ let [step, setStep] = useState(1);
   - React state is immutable. Directly updating step like this:

```
step = step + 1;
```

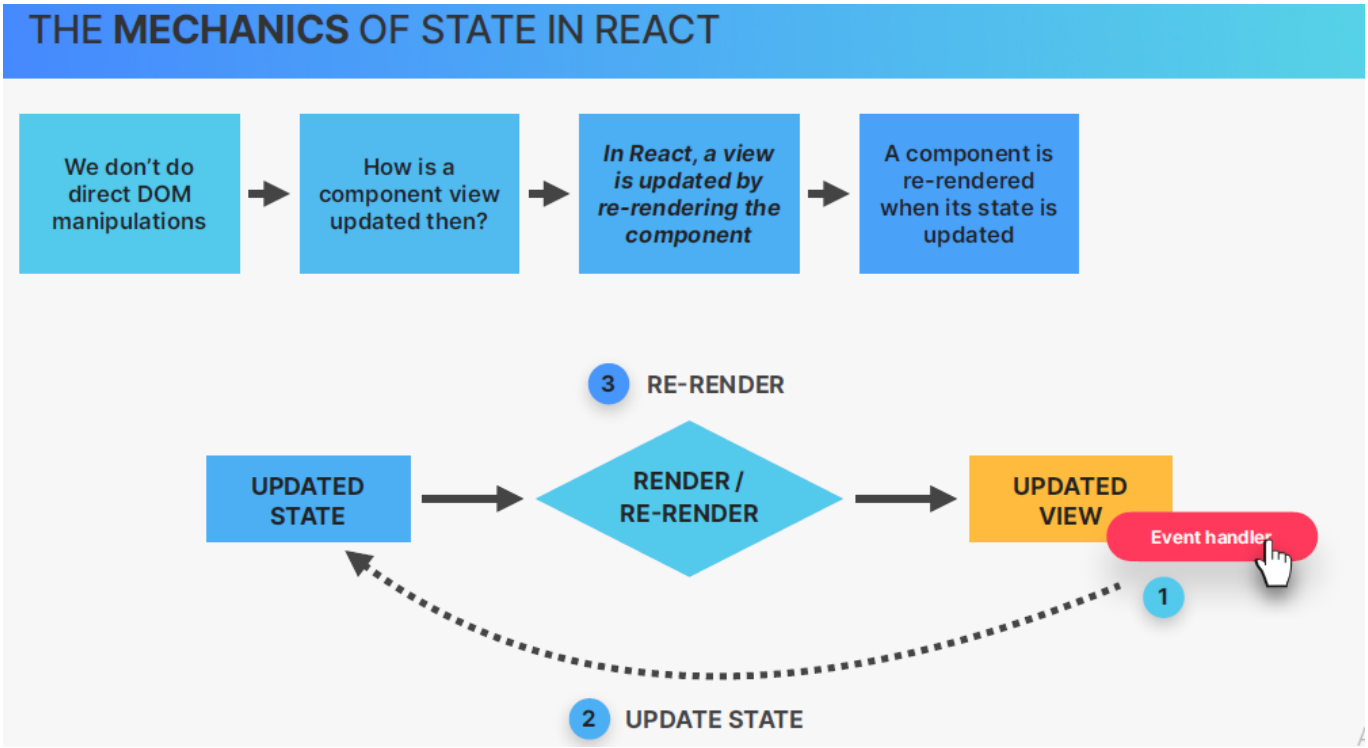
   - Doesn't work — it just reassigns a local variable.
   - Does not trigger a re-render, so the UI won't update.
   - Breaks React's rules, which expect you to use the setStep() function to schedule state changes.

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

function Counter() {
  // ❌ Using let (misleading — still works, but not needed)
  let [count, setCount] = useState(0);

  function increment() {
    // ❌ Direct assignment — doesn't update UI
    count = count + 1;
    console.log("Count is:", count);
  }

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increase</button>
    </div>
  );
}
```

❌ **What's wrong here?**

- `count = count + 1` only changes the local variable.
- **React doesn't know** the state changed.
- UI won't update!
- You're misusing `let` — it implies you plan to reassign `count`, which you shouldn't manually do in React.

pair



🖼 Differences between the regular form vs functional form of setState in React

React state updates are **asynchronous and batched**. That means when you call:

```js
setStep(step + 1);
setStep(step + 1);
```

Both updates are:

- **Scheduled** at the same time,
- Using the same current value of `step` (say it's `1`),
- So both evaluate as `setStep(2)` — even though you're calling it twice.

> Result: Only **1 increment**, not 2. ❌

✅ The Solution: Use Functional Updates

The functional form ensures **each update uses the latest state**, even when updates are queued.

```js
setStep((prevStep) => prevStep + 1);
setStep((prevStep) => prevStep + 1);
```

Now, React:

- Runs first update: `prevStep = 1 → 2`
- Runs second update: `prevStep = 2 → 3`

> Result: Final value = `3` ✅

When updating state based on the **previous value**, it's important to use the **functional update form** to avoid stale values.

Method	Looks at old value?	Updates individually?	Final Result (step from 1)
`setStep(step + 1)`	❌ (stale value)	❌ (same value reused)	2 (wrong)
`setStep(s => s + 1)`	☑ (fresh state each time)	☑ (depends on previous state)	3 (correct)

- **Conclusions**

📝 Summary

Aspect	Regular Form (`step + 1`)	Functional Form (`s => s + 1`)
Uses current variable value	✅	❌ (uses fresh state each time)
Safe for back-to-back calls	❌	✅
Works well with batching	❌	✅
Triggers multiple updates	Sometimes ❌	✅ Always
Recommended?	❌ For multiple updates	✅ Yes

🧠 Rule of Thumb:

> Use **functional updates** (`setState(prev => ...)`) when:
>
> - **"You're updating state multiple times in a row"**
> - **"You're relying on previous state"**
> - **"You're in async code, timers, or effects"**

📌 Props vs State

Props and State are core concepts in React. They are the only triggers that cause React to re-render components and potentially update the DOM in the browser

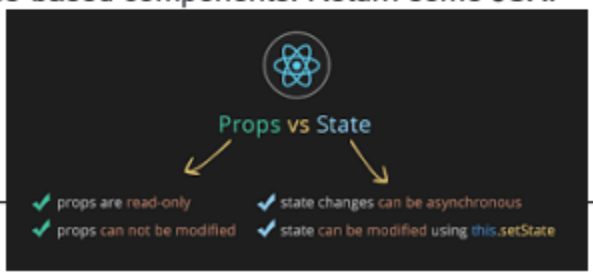
- Props : allow you to pass data from a parent (wrapping) component to a child component.
  - Eg : AllPosts Component : “title” is the custom property (prop) set up on the custom Post component.
  - Post Component: receives the props argument. React will pass one argument to your component function; an object, which contains all properties you set up on <Post ... /> .
  - {props.title} then dynamically outputs the title property of the props object - which is available since we set the title property inside AllPosts component

```
//AllPosts
const posts = () => {
    return (
        <div>
            <Post title="My first Post" />
            <Post title="My second Post" />
        </div>
    );
}
```

```
//Post
const post = (props) => {
    return (
        <div>
            <h1>{props.title}</h1>
        </div>
    );
}
```

- State : While props allow you to pass data down the component tree (and hence trigger an UI update), state is used to change the component’s, well, state from within.
  - Changes to state also trigger an UI update.
  - Example: NewPost Component: this component contains state . Only class-based components can define and use state . You can of course pass the state down to functional components, but these then can't directly edit it.

```
class NewPost extends Component { // state can only be accessed in class-based components!
    state = {
        counter: 1
    };

    render () { // Needs to be implemented in class-based components! Return some JSX!
        return (
            <div>{this.state.counter}</div>
        );
    }
}
```

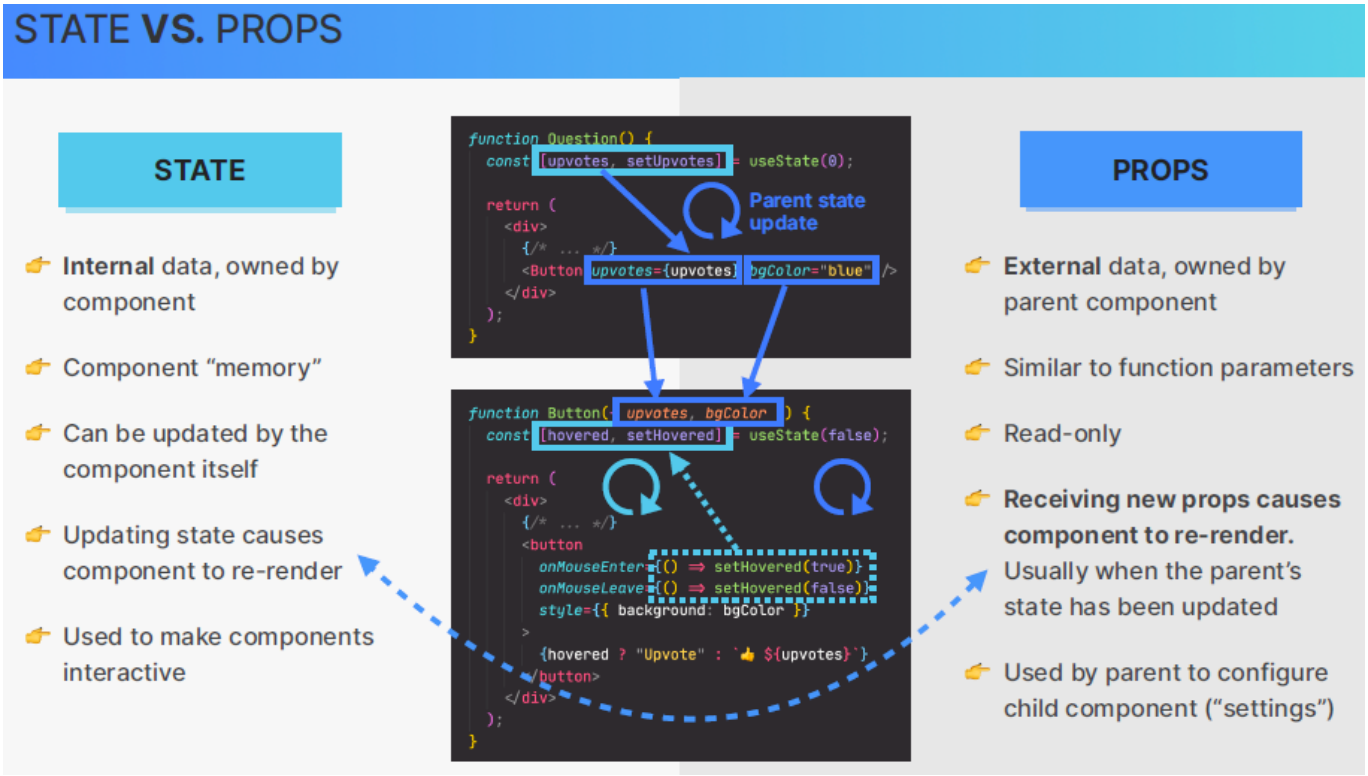


props and state

- Props are immutable.
- They should not be updated by the component to which they are passed.
- They are owned by the component which passes them to some other component.

- State is something internal and private to the component.
- State can and will change depending on the interactions with the outer world.
- State should store as simple data as possible, such as whether an input checkbox is checked or not or a CSS class that hides or displays the component

🧩 Props vs 🧠 State



Feature	Props	State
Mutability	Immutable – cannot be changed by the receiving component	Mutable – can be updated using `useState` or similar
Ownership	Owned by the **parent** component	Owned and managed by the **same** component
Purpose	Pass data and configuration to child components	Store dynamic data that may change during lifecycle

Feature	Props	State
Usage	Passed via JSX: `<Component title="Hello" />`	Managed internally: `const [count, setCount] = useState(0)`
Example Use	Text, props like `title`, `id`, etc.	Form inputs, toggle states, visibility, active tabs, etc.
Modification	Cannot be modified by the child component	Can be updated by event handlers, API responses, etc.

Controlled Elements in React

The `<input>, <select>, or <textarea>` maintain their state in the dom by themselves.

- The value is managed by the DOM, not React.
- React has no idea what the user typed unless you manually query the DOM (like document.querySelector or use a ref).
- This breaks the "React way" of managing everything with state. In react we want all state to centralized at one place means inside the react application not in the DOM. To do that we use controlled element

A controlled element is a form element where React fully manages the value via state like `<input>, <select>, or <textarea>`

Step 1: Create state

Use `useState` to hold the value of the form input.

```js
const [inputValue, setInputValue] = useState("");
```

Step 2: Set the `value` of the element

Connect the state to the form element using the `value` attribute.

```jsx
<input type="text" value={inputValue} />
```

Step 3: Handle changes with `onChange`

Update the state when the input changes.

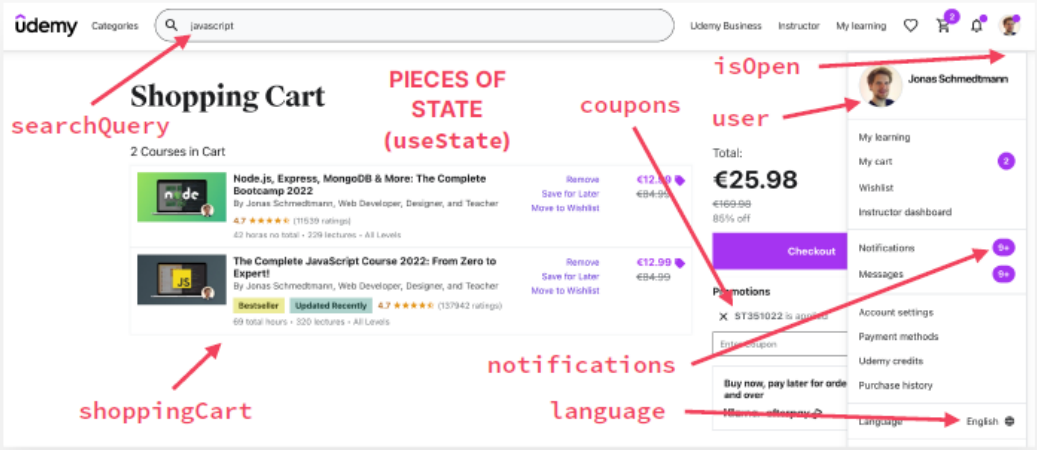
```jsx
<input
  type="text"
  value={inputValue}
  onChange={(e) => setInputValue(e.target.value)}
/>
```

Step	Action	Purpose
1	`useState()`	Create state for input value
2	`value={state}`	Connect input value to state
3	`onChange={...}`	Update state on user input

State Management / Types / When & Where

WHAT IS STATE **MANAGEMENT?**

👉 **State management:** Deciding **when** to create pieces of state, what **types** of state are necessary, **where** to place each piece of state, and how data **flows** through the app → 🏠 Giving each piece of state a **home**



TYPES OF STATE: LOCAL VS. GLOBAL STATE

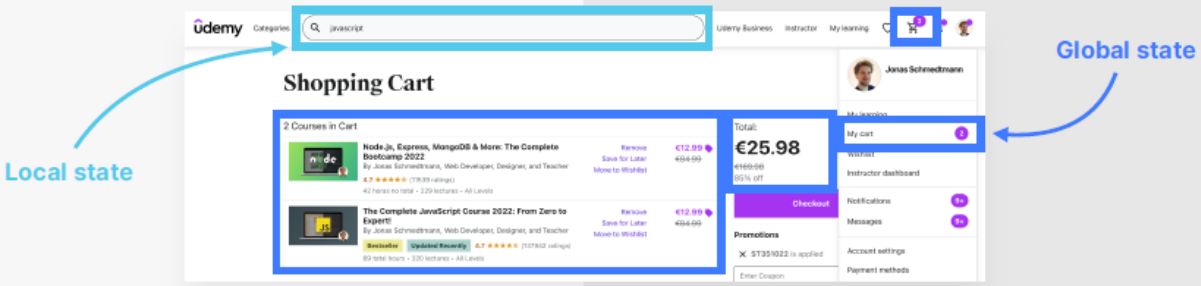
LOCAL STATE

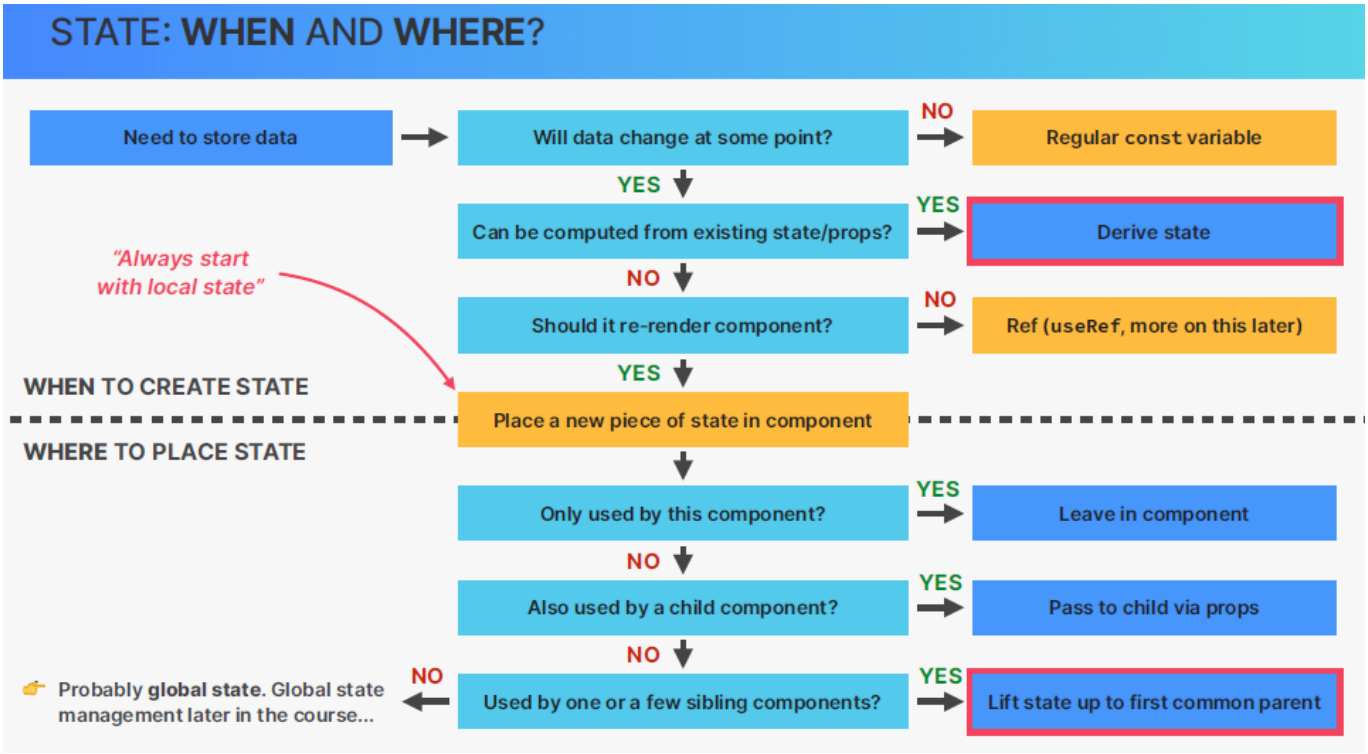
- 👉 State needed **only by one or few components**
- 👉 State that is defined in a component and **only that component and child components** have access to it (by passing via props)
- ☝️ ***We should always start with local state***

GLOBAL STATE

- 👉 State that **many components** might need
- 👉 **Shared** state that is accessible to **every component** in the entire application

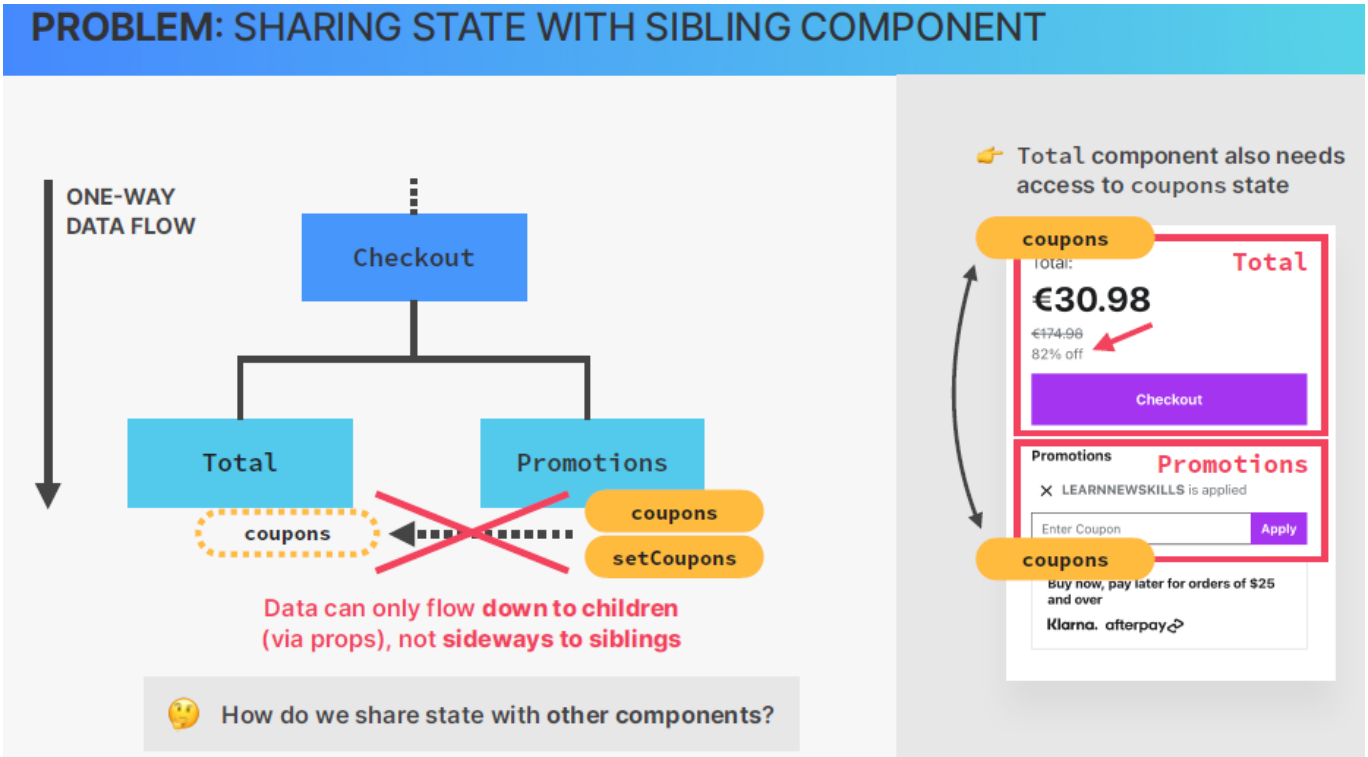
Context API Redux





Lifting State Up in React

Lifting State Up means moving state to the closest common parent component so that multiple child components can share and communicate through it.



Why Do We Lift State?

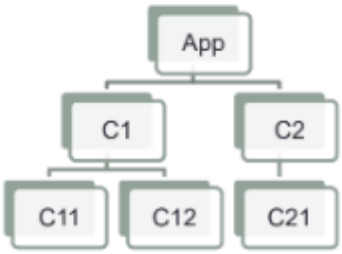
Imagine two components need access to the same data. Instead of duplicating state in both, we **"lift"** it **up** to a shared parent.

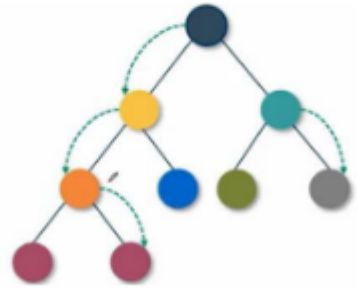
This allows:

- ✅ Synchronization
- ✅ Reusability
- ✅ Clean data flow

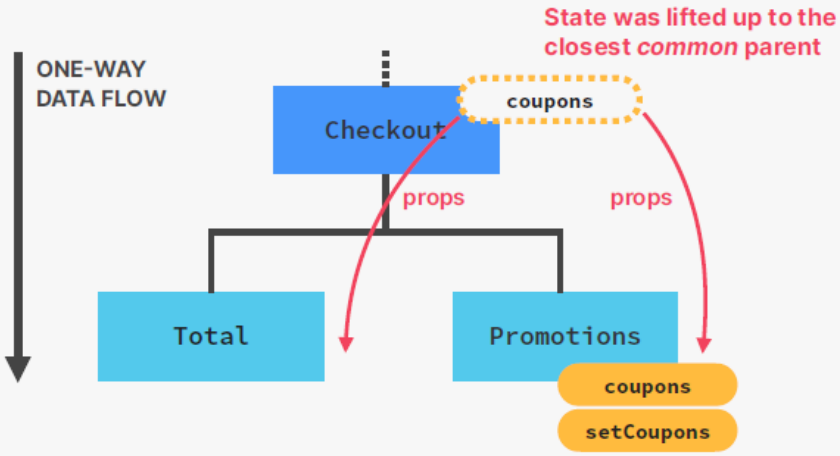
Lifting state up in React.js

- In a typical application, you pass a lot of state down to child components as props.
  - These props are often passed down multiple component levels.
  - That's how state is shared vertically in your application.
- Often there will be a need to share state between different components.
  - The common approach to share state between two components is to move the state to common parent of the two components.
  - This approach is called as lifting state up in React.js
  - React components can manage their own state
  - Often components need to communicate state to others
  - Siblings do not pass state to each other directly
  - State should pass through a parent, then trickle down



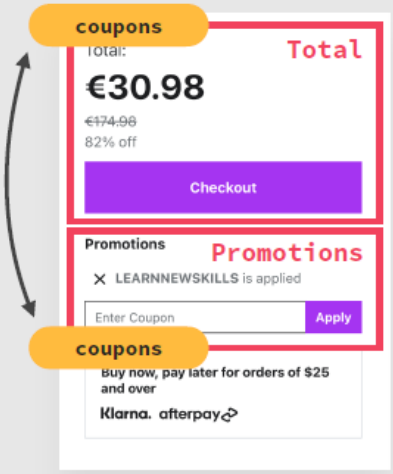


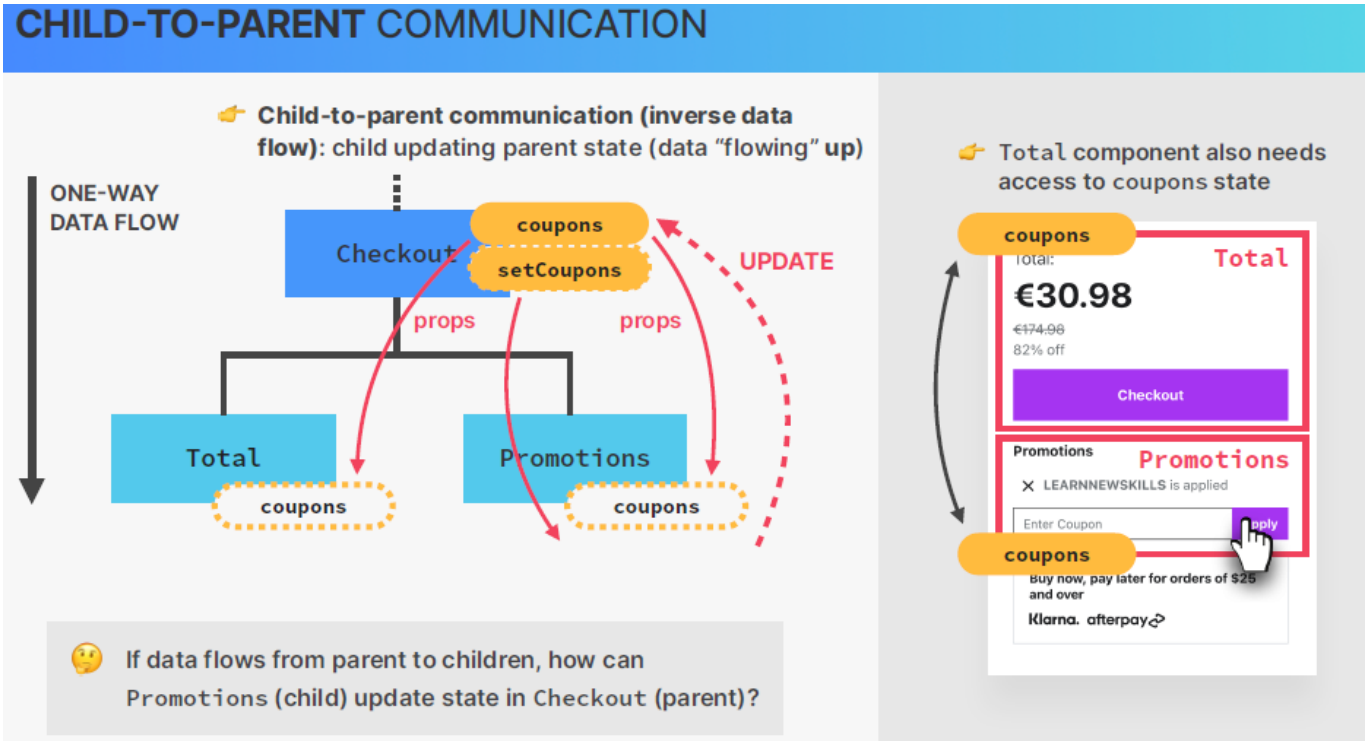
SOLUTION: LIFTING STATE UP



By lifting state up, we have successfully shared one piece of state with multiple components in different positions in the component tree

Total component also needs access to coupons state





Example: Without Lifting State

Let's say we have a form that adds items and a list that shows them:

```jsx
// ❌ This won't work — both components have separate state

function Form() {
  const [items, setItems] = useState([]);
  // Adds items here
}

function PackingList() {
  const [items, setItems] = useState([]);
  // Displays items here
}
```

This means the form and list aren't connected — they both have **separate copies** of `items`.

✅ With Lifting State Up

Instead, move `items` to the **common parent**, and pass state + callbacks down:

jsx Copy Edit

```jsx
function App() {
  const [items, setItems] = useState([]);

  function handleAddItem(newItem) {
    setItems((items) => [...items, newItem]);
  }

  return (
    <>
      <Form onAddItem={handleAddItem} />
      <PackingList items={items} />
    </>
  );
}
```

Now both `Form` and `PackingList` are **in sync** ✅

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

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

  return (
    <div>
      <CounterButton count={count} setCount={setCount} />
      <DisplayCount count={count} />
    </div>
  );
}

function CounterButton({ count, setCount }) {
  return (
    <div>
      <h2>Counter Button</h2>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

function DisplayCount({ count }) {
  return (
    <div>
      <h2>Display Count</h2>
      <p>Current Count: {count}</p>
    </div>
  );
}

export default App;
```

preventDefault()

The preventDefault() method cancels the event if it is cancelable, meaning that the default action that belongs to the event will not occur.

For example, this can be useful when:

- Clicking on a "Submit" button, prevent it from submitting a form
- Clicking on a link, prevent the link from following the URL
- Note: Not all events are cancelable. Use the cancelable property to find out if an event is cancelable.
- Note: The preventDefault() method does not prevent further propagation of an event through the DOM. Use the stopPropagation() method to handle this.

🧪 Example 1: Prevent form from refreshing the page

jsx Copy Edit

```jsx
function FormExample() {
  function handleSubmit(e) {
    e.preventDefault(); // stops page reload
    alert("Form submitted without reloading!");
  }

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" placeholder="Type something..." />
      <button type="submit">Submit</button>
    </form>
  );
}
```

🔁 **Without** `e.preventDefault()`, the browser would reload the page after form submission — losing all data/state.

🧪 Example 2: Prevent anchor tag navigation

jsx Copy Edit

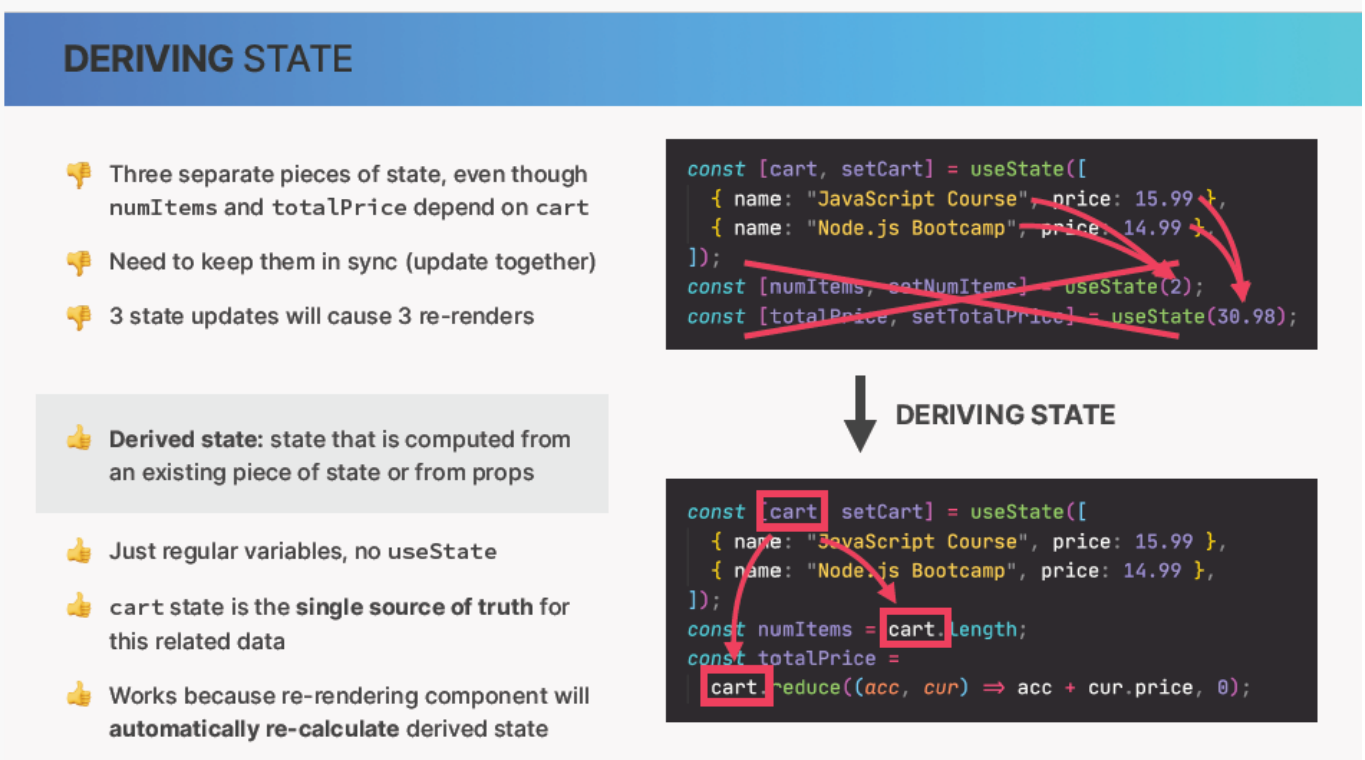
```jsx
function LinkExample() {
  function handleClick(e) {
    e.preventDefault(); // prevents navigation
    alert("Link click blocked!");
  }

  return <a href="https://google.com" onClick={handleClick}>Click me</a>;
}
```

🔁 **Without** `e.preventDefault()`, clicking the link would open Google.

Deriving State

Derived state means a value that is computed based on other state or props, not stored independently.

Instead of storing a duplicate piece of data in useState, you derive it from existing state or props inside the render.



1. Without Derived State (Bad Practice) Stores both products and filteredProducts in state — redundant.

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

const ProductList = () => {
  const [searchQuery, setSearchQuery] = useState('');
  const [products] = useState([
    { id: 1, name: 'Laptop' },
    { id: 2, name: 'Smartphone' }
  ]);
  const [filteredProducts, setFilteredProducts] = useState(products);

  const handleSearch = (e) => {
    const query = e.target.value;
    setSearchQuery(query);
    setFilteredProducts(
      products.filter(p =>
        p.name.toLowerCase().includes(query.toLowerCase())
      )
    );
  };

  return (
    <>
      <input value={searchQuery} onChange={handleSearch} />
      <ul>
```

```
        {filteredProducts.map(p => <li key={p.id}>{p.name}</li>)}
      </ul>
    </>
  );
};

export default ProductList;
```

2. ☑ With Derived State (Good Practice) Filters products directly during render — cleaner and simpler.

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

const ProductList = () => {
  const [searchQuery, setSearchQuery] = useState('');
  const products = [
    { id: 1, name: 'Laptop' },
    { id: 2, name: 'Smartphone' }
  ];

  const filteredProducts = products.filter(p =>
    p.name.toLowerCase().includes(searchQuery.toLowerCase())
  );

  return (
    <>
      <input value={searchQuery} onChange={(e) => setSearchQuery(e.target.value)}
/>
      <ul>
        {filteredProducts.map(p => <li key={p.id}>{p.name}</li>)}
      </ul>
    </>
  );
};

export default ProductList;
```

.sort() method

To sort elements by name in React, you can use JavaScript's .sort()

3. `.slice()` to the rescue!

```js
items
  .slice()          // → makes a *shallow copy* of the array
  .sort(...)        // → now you're sorting the copy, not the original
```

This way, the original `items` array (whether it lives in state, props, or elsewhere) stays untouched, and React can correctly recognize that you provided a "new" array to render.

Alternative using spread syntax:

```js
[...items].sort((a, b) => a.name.localeCompare(b.name))
```

```
// Or, if your objects have a `price` field:
const sortedByPrice = items
  .slice()
  .sort((a, b) => a.price - b.price);
```

main.js

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

const ProductList = () => {
  const [searchQuery, setSearchQuery] = useState('');
  const products = [
    { id: 1, name: 'Laptop' },
    { id: 2, name: 'Smartphone' }
  ];

  // Filter and then sort by name
  const filteredAndSorted = products
    .filter(p => p.name.toLowerCase().includes(searchQuery.toLowerCase()))
    .slice() // copy before sort
    .sort((a, b) => a.name.localeCompare(b.name));// Sort alphabetically

  return (
    <>
      <input
        type="text"
        value={searchQuery}
        onChange={(e) => setSearchQuery(e.target.value)}
        placeholder="Search for a product"
      />
      <ul>
        {filteredAndSorted.map(p => (
          <li key={p.id}>{p.name}</li>
        ))}
      </ul>
    </>
  );
};

export default ProductList;
```

Window: confirm() method

window.confirm() instructs the browser to display a dialog with an optional message, and to wait until the user either confirms or cancels the dialog.

```
<button id="windowButton">Open new tab</button>
<pre id="log"></pre>
```

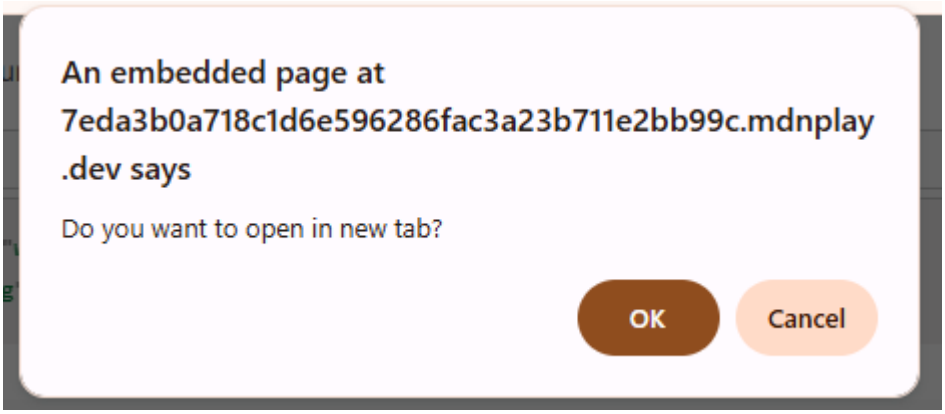
JS

```
const windowButton = document.querySelector("#windowButton");
const log = document.querySelector("#log");

windowButton.addEventListener("click", () => {
  if (window.confirm("Do you want to open in new tab?")) {
    window.open("https://developer.mozilla.org/en-US/docs/Web/API/Window/open");
  } else {
    log.innerText = "Glad you're staying!";
  }
});
```

Play

Play

Open new tab



CHILDREN PROPS

The children prop is a special prop automatically provided by React. It allows you to pass content between a component's opening and closing tags, instead of passing it as a regular prop.

In React, children is a special prop that automatically passes any JSX elements or components nested within a component's opening and closing tags. It provides a way for components to render dynamic content passed down from their parent components.

Code

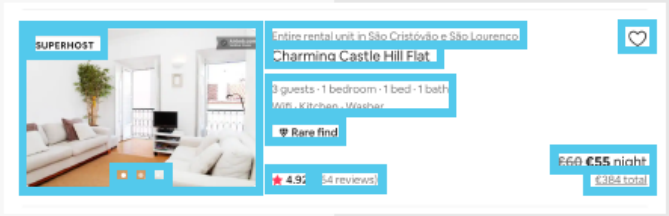
```
function Container({ children }) {
  return (

      {children}

  );
}

function App() {
  return (
    <Container>
      <h1>Hello, world!</h1>
      <p>This is some content inside the container.</p>
    </Container>
  );
}
```

In this example, the `Container` component receives the `<h1>` and `<p>` elements as its `children` prop and renders them within a `div`. This allows the `App` component to control the content displayed within the `Container` without the `Container` component needing to know the specific details of that content.

THINKING IN REACT: COMPONENTS COMPOSITION, AND REUSABILITY

COMPONENT SIZE **MATTERS**

Many small components



COMPONENT SIZE

SMALL — HUGE

Generally, we need to find the right balance between too specific and too broad

- We end up with 100s of mini-components
- Confusing codebase
- Too abstracted

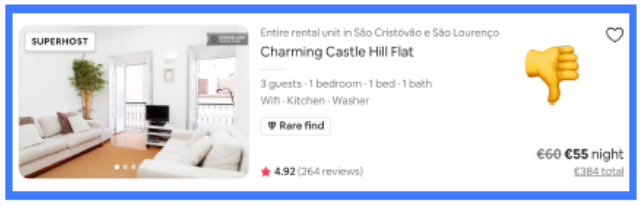
Creating something new to hide the implementation details of that thing

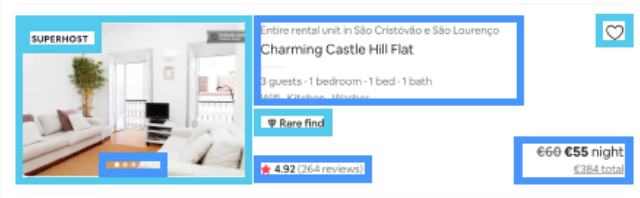
- Too many responsibilities
- Might need too many props
- Hard to reuse
- Complex code, hard to understand

HOW TO **SPLIT** A UI INTO COMPONENTS

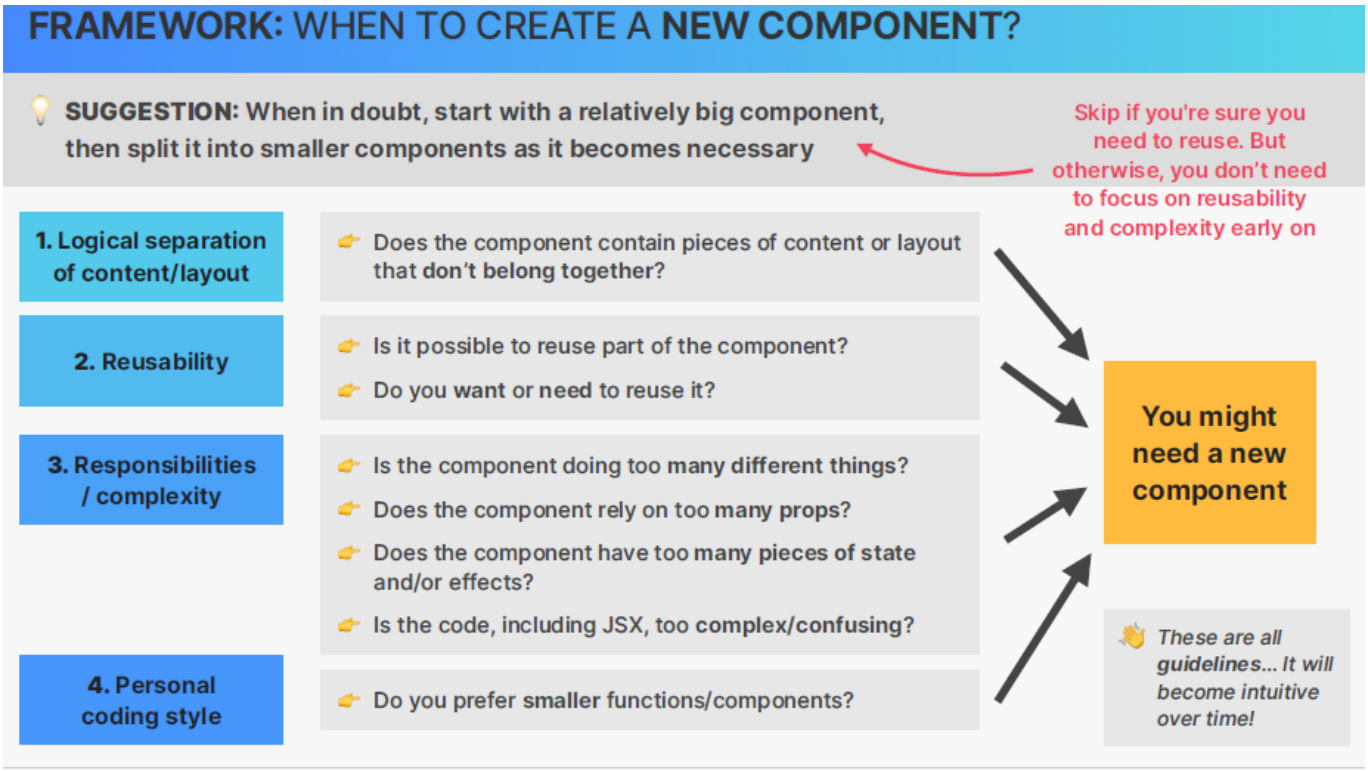
The 4 criteria for splitting a UI into components:

1. Logical separation of content/layout
2. Reusability
3. Responsibilities / complexity
4. Personal coding style





- ✅ Logical separation
- ✅ Some are reusable
- ✅ Low complexity


FRAMEWORK: WHEN TO CREATE A NEW COMPONENT?
SUGGESTION: When in doubt, start with a relatively big component, then split it into smaller components as it becomes necessary
Skip if you're sure you need to reuse. But otherwise, you don't need to focus on reusability and complexity early on
1. Logical separation of content/layout
Does the component contain pieces of content or layout that don't belong together?
2. Reusability
Is it possible to reuse part of the component?
Do you want or need to reuse it?
3. Responsibilities / complexity
Is the component doing too many different things?
Does the component rely on too many props?
Does the component have too many pieces of state and/or effects?
Is the code, including JSX, too complex/confusing?
4. Personal coding style
Do you prefer smaller functions/components?
You might need a new component
These are all guidelines... It will become intuitive over time!