**Creating and nesting components**

React apps are made out of components. A component is a piece of the UI (user interface) that has its own logic and appearance. A component can be as small as a button, or as large as an entire page.

React components are JavaScript functions that return markup:

function MyButton() {

return (

<button>I'm a button</button>

);}

* Notice that <MyButton /> starts with a capital letter. That’s how you know it’s a React component. **React component names must always start with a capital letter, while HTML tags must be lowercase.**
* The **export default** keywords specify the main component in the file.
* **JSX is stricter than HTML**. You have to close tags like <br />. Your component also can’t return multiple JSX tags. You have to wrap them into a shared parent, like a <div>...</div> or an empty <>...</> wrapper.

**Adding styles**

In React, you specify a CSS class with className. It works the same way as the HTML class attribute:

<img className="avatar" />

/\* In your CSS \*/

.avatar {

border-radius: 50%;

}

**Displaying data**

JSX lets you put markup into JavaScript. Curly braces let you “escape back” into JavaScript so that you can embed some variable from your code and display it to the user. For example, this will display user.name:

return (

<h1> {user.name} </h1>

);

You can also “escape into JavaScript” from JSX attributes, but you have to use curly braces instead of quotes. For example, className="avatar" passes the "avatar" string as the CSS class, but src={user.imageUrl} reads the JavaScript user.imageUrl variable value, and then passes that value as the src attribute:

return (

<img className="avatar" src={user.imageUrl} />

);

**Full code**

const user = {

name: 'Hedy Lamarr',

imageUrl: 'https://i.imgur.com/yXOvdOSs.jpg',

imageSize: 90,

};

export default function Profile() {

return (

<>

<h1>{user.name}</h1>

<img className="avatar" src={user.imageUrl}

alt={'Photo of ' + user.name}

style={{ width: user.imageSize,height: user.imageSize}}/>

</> );}

In the above example, **style={{}}** is not a special syntax, but a regular {} object inside the style={ } JSX curly braces. You can use the style attribute when your styles depend on JavaScript variables.

**Conditional rendering**

* In React, there is no special syntax for writing conditions. Instead, you’ll use the same techniques as you use when writing regular JavaScript code. For example, you can use an if statement to conditionally include JSX:

let content;

if (isLoggedIn) {content = <AdminPanel />;}

else {content = <LoginForm />;}

return (

<div>{content} </div>

);

* If you prefer more compact code, you can use the conditional ? operator. Unlike if, it works inside JSX:

<div>{isLoggedIn ? (<AdminPanel />) : (<LoginForm />)}</div>

* When you don’t need the else branch, you can also use a shorter logical && syntax:

<div>{isLoggedIn && <AdminPanel />}</div>

**Rendering lists**

You will rely on JavaScript features like for loop and the array map() function to render lists of components.

For example, let’s say you have an array of products:

const products = [

{ title: 'Cabbage', id: 1 },

{ title: 'Garlic', id: 2 },

{ title: 'Apple', id: 3 },

];

Inside your component, use the **map() function** to transform an array of products into an array of <li> items:

const listItems = products.map(product =>

<li key={product.id}>

{product.title}

</li>);

return (

<ul>{listItems}</ul>

);

Notice how <li> has a key attribute. For each item in a list, you should pass a string or a number that uniquely identifies that item among its siblings. Usually, a key should be coming from your data, such as a database ID. React uses your keys to know what happened if you later insert, delete, or reorder the items.

**Responding to events**

You can respond to events by declaring event handler functions inside your components:

function MyButton() {

function handleClick() {alert('You clicked me!'); }

return (

<button onClick={handleClick}> Click me</button>

);}

Notice how onClick={handleClick} has no parentheses at the end! Do not call the event handler function: you only need to pass it down. React will call your event handler when the user clicks the button.

**Updating the screen**

Often, you’ll want your component to “remember” some information and display it. For example, maybe you want to count the number of times a button is clicked. To do this, add state to your component.

First, import useState from React:

import { useState } from 'react';

Now you can declare a state variable inside your component:

function MyButton() {

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

// ...

You’ll get two things from useState: the current state (count), and the function that lets you update it (setCount). You can give them any names, but the convention is to write [something, setSomething].

The first time the button is displayed, count will be 0 because you passed 0 to useState(). When you want to change state, call setCount() and pass the new value to it. Clicking this button will increment the counter:

function MyButton() {

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

function handleClick() {

setCount(count + 1);

}

return (

<button onClick={handleClick}>

Clicked {count} times

</button>

);

}

React will call your component function again. This time, count will be 1. Then it will be 2. And so on.

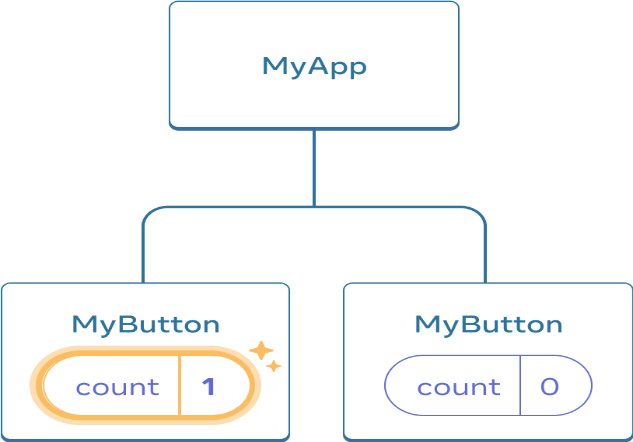
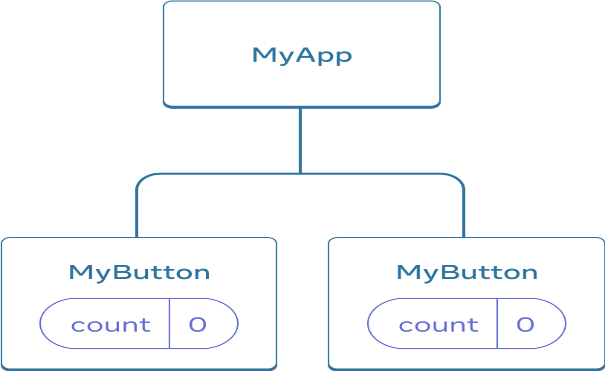
**Using Hooks**

Functions starting with use are called Hooks. useState is a built-in Hook provided by React. You can find other built-in Hooks in the API reference. You can also write your own Hooks by combining the existing ones.

Hooks are more restrictive than other functions. You can only call Hooks at the top of your components (or other Hooks). If you want to use useState in a condition or a loop, extract a new component and put it there.

**Sharing data between components**

In the previous example, each MyButton had its own independent count, and when each button was clicked, only the count for the button clicked changed:

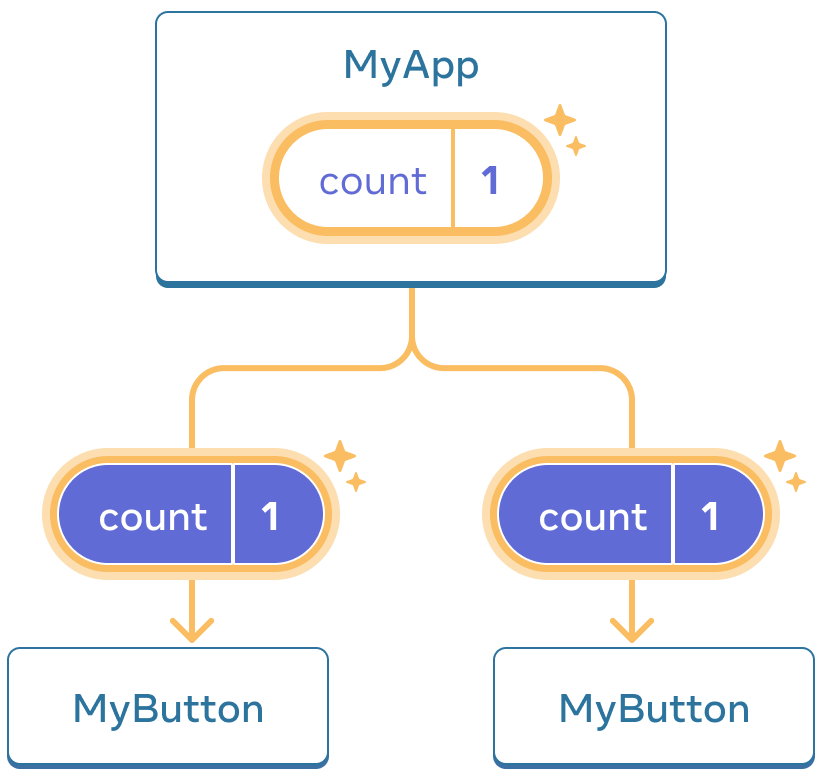
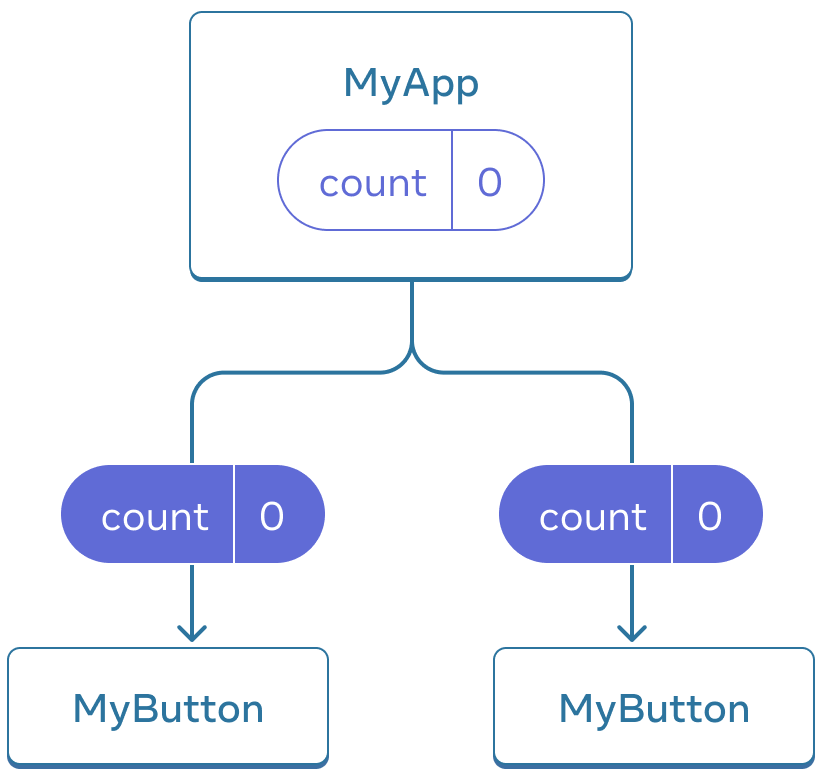


Initially, each MyButton’s count state is 0 The first MyButton updates its count to 1

However, often you’ll need components to share data and always update together.

To make both MyButton components display the same count and update together, you need to move the state from the individual buttons “upwards” to the closest component containing all of them.

In this example, it is MyApp:



Initially, MyApp’s count state is 0 and is passed down to both children

On click, MyApp updates its count state to 1 and passes it down to both children

Now when you click either button, the count in MyApp will change, which will change both of the counts in MyButton. Here’s how you can express this in code.

First, move the state up from MyButton into MyApp:

export default function MyApp() {

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

function handleClick() {

setCount(count + 1);

}

return (

<div>

<h1>Counters that update separately</h1>

<MyButton />

<MyButton />

</div>

);

}

function MyButton() {

// ... we're moving code from here ...

}

Then, pass the state down from MyApp to each MyButton, together with the shared click handler. You can pass information to MyButton using the JSX curly braces, just like you previously did with built-in tags like <img>:

export default function MyApp() {

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

function handleClick() {

setCount(count + 1);

}

return (

<div>

<h1>Counters that update together</h1>

<MyButton count={count} onClick={handleClick} />

<MyButton count={count} onClick={handleClick} />

</div>

);}

The information you pass down like this is called props. Now the MyApp component contains the count state and the handleClick event handler, and passes both of them down as props to each of the buttons.

Finally, change MyButton to read the props you have passed from its parent component:

function MyButton({ count, onClick }) {

return (

<button onClick={onClick}>

Clicked {count} times

</button>

);

}

When you click the button, the onClick handler fires. Each button’s onClick prop was set to the handleClick function inside MyApp, so the code inside of it runs. That code calls setCount(count + 1), incrementing the count state variable. The new count value is passed as a prop to each button, so they all show the new value. This is called “lifting state up”. By moving state up, you’ve shared it between components.

**Full code**

import { useState } from 'react';

export default function MyApp() {

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

function handleClick() {

setCount(count + 1);

}

return (

<div>

<h1>Counters that update together</h1>

<MyButton count={count} onClick={handleClick} />

<MyButton count={count} onClick={handleClick} />

</div>

);}

function MyButton({ count, onClick }) {

return (

<button onClick={onClick}>

Clicked {count} times

</button>

);}

**Build a static version in React**

To build a static version of your app that renders your data model, you’ll want to build [components](https://react.dev/learn/your-first-component) that reuse other components and pass data using [props.](https://react.dev/learn/passing-props-to-a-component) Props are a way of passing data from parent to child. (If you’re familiar with the concept of [state](https://react.dev/learn/state-a-components-memory), don’t use state at all to build this static version. State is reserved only for interactivity, that is, data that changes over time. Since this is a static version of the app, you don’t need it.)

There are **two types of “model” data in React: props and state**. The two are very different:

* [**Props** are like arguments you pass](https://react.dev/learn/passing-props-to-a-component) to a function. They let a parent component pass data to a child component and customize its appearance. For example, a Form can pass a color prop to a Button.
* [**State** is like a component’s memory.](https://react.dev/learn/state-a-components-memory) It lets a component keep track of some information and change it in response to interactions. For example, a Button might keep track of isHovered state.

Props and state are different, but they work together. A parent component will often keep some information in state (so that it can change it), and *pass it down* to child components as their props. It’s okay if the difference still feels fuzzy on the first read. It takes a bit of practice for it to really stick!

**NOTE**: React uses one-way data flow, passing data down the component hierarchy from parent to child component.

**{' '}**:

* This is a JavaScript expression used to insert a space character between the checkbox and the label text. It ensures proper spacing for readability.

**CLASS AND FUNCTION Components**

**COMPONENT SIDE-EFFECTS WITH REACT'S USEEFFECT HOOK.**

Class Component

componentDidUpdate() {

localStorage.setItem('myValueInLocalStorage', this.state.value);

}

Function Component

useEffect(() => {

localStorage.setItem('myValueInLocalStorage', value);

}, [value]);

React's useEffect Hook runs every time one of the values in the passed array (second argument) got changed. In our case, every time the value from the input field changes, we update the local storage with it. Also the value from the local storage is used initially to set the initial value for the input field.

Again, by nature the Function Component is way more lightweight, because it can use state and side-effects within its function body. Also the usage of the local storage moved closer in the function's body rather than having it in different class methods as before.

If the next React Component you are going to implement has to have side-effects -- like calling the local storage of the browser --, don't default to a React Class Component, but give React Function Components with React Hooks a shot.

**React Element vs Component**

React Elements, Components, and Instances are different terms in React which work closely together.

We can extract a component from another component and render it the following way. Rendering a component happens whenever we use this component as a **React element** with angle brackets (e.g. <Greeting />) in another component:

const Greeting = ({ text }) => {

return <p>{text}</p>;

};

const App = () => {

return <Greeting text="Hello React" />;

**REACT ELEMENTS IN DEPTH**

Let's take one step back and start with a simple example again:

const App = () => {

return <p>Hello React</p>;

};

Whenever a React component gets called (rendering), React calls its React.createElement() method internally which returns the following object:

console.log(App());

// {

// $$typeof: Symbol(react.element)

// "type": "p",

// "key": null,

// "ref": null,

// "props": {

// "children": "Hello React"

// },

// "\_owner": null,

// "\_store": {}

// }

Focus your attention on the **type and props** properties of this object: While the type represents the actual HTML element**, the props are all HTML attributes (plus the inner content, read: children)** which are passed to this HTML element.

When looking at the paragraph HTML element from above, you can see that no attributes are passed to it. However, React treats children as pseudo HTML attribute whereas children represents everything that's rendered between the HTML tag. This fact becomes clearer when adding an attribute to the paragraph HTML element:

const App = () => {

return <p className="danger">Hello React</p>;

};

console.log(App());

// {

// $$typeof: Symbol(react.element)

// "type": "p",

// "key": null,

// "ref": null,

// "props": {

**// "children": "Hello React",**

**// "className": "danger"**

// },

// "\_owner": null,

// "\_store": {}

// }

Essentially React translates all HTML attributes to React props in addition to adding the inner content as children property.