Sharing State Between Components

Sometimes, you want the state of two components to always change together. To do it, remove state from both of them, move it to their closest common parent, and then pass it down to them via props. This is known as *lifting state up,* and it's one of the most common things you will do writing React code.

You will learn

- How to share state between components by lifting it up
- What are controlled and uncontrolled components

Lifting state up by example

In this example, a parent `Accordion` component renders two separate `Panel` s:

- `Accordion`
    - `Panel`
    - `Panel`

Each `Panel` component has a boolean `isActive` state that determines whether its content is visible.

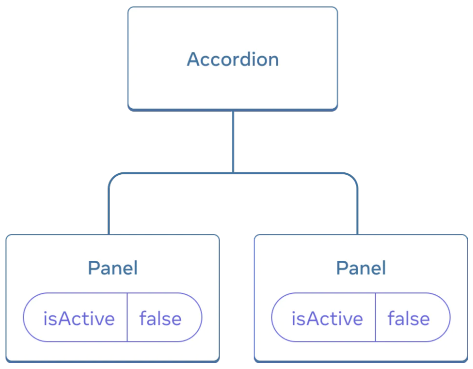
Press the Show button for both panels:

App.js Download Reset

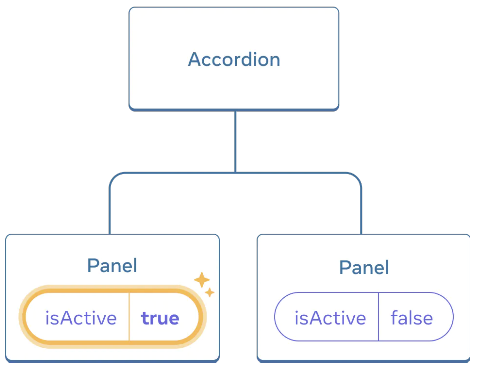
```
import { useState } from 'react';

function Panel({ title, children }) {
  const [isActive, setIsActive] = useState(false);
  return (
    <section className="panel">
      <h3>{title}</h3>
      {isActive ? (
        <p>{children}</p>
      ) : (
        <button onClick={() => setIsActive(true)}>
          Show
        </button>
      )}
    </section>
  );
}

export default function Accordion() {
  return (
    <>
      <h2>Almaty, Kazakhstan</h2>
      <Panel title="About">
        With a population of about 2 million, Almaty is Kazakhstan's la
      </Panel>
      <Panel title="Etymology">
        The name comes from <span lang="kk-KZ">алма</span>, the Kazakh
      </Panel>
    </>
  );
}

```

Show less

Notice how pressing one panel's button does not affect the other panel—they are independent.



Initially, each `Panel`'s `isActive` state is `false`, so they both appear collapsed



Clicking either `Panel`'s button will only update that `Panel`'s `isActive` state alone

But now let's say you want to change it so that only one panel is expanded at any given time. With that design, expanding the second panel should collapse the first one. How would you do that?

the first one. How would you do that?

To coordinate these two panels, you need to “lift their state up” to a parent component in three steps:

1. **Remove** state from the child components.
2. **Pass** hardcoded data from the common parent.
3. **Add** state to the common parent and pass it down together with the event handlers.

This will allow the `Accordion` component to coordinate both `Panel` s and only expand one at a time.

Step 1: Remove state from the child components

You will give control of the `Panel` ’s `isActive` to its parent component. This means that the parent component will pass `isActive` to `Panel` as a prop instead. Start by **removing this line** from the `Panel` component:

```
const [isActive, setIsActive] = useState(false);
```

And instead, add `isActive` to the `Panel` ’s list of props:

```
function Panel({ title, children, isActive }) {
```

Now the `Panel` ’s parent component can *control* `isActive` by passing it down as a prop. Conversely, the `Panel` component now has *no control* over the value of `isActive` —it’s now up to the parent component!

Step 2: Pass hardcoded data from the common parent

To lift state up, you must locate the closest common parent component of *both* of the child components that you want to coordinate:

- `Accordion` *(closest common parent)*
  - `Panel`
  - `Panel`

In this example, it's the `Accordion` component. Since it's above both panels and can control their props, it will become the "source of truth" for which panel is currently active. Make the `Accordion` component pass a hardcoded value of `isActive` (for example, `true`) to both panels:

App.js

Download Reset

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

export default function Accordion() {
  return (
    <>
      <h2>Almaty, Kazakhstan</h2>
      <Panel title="About" isActive={true}>
        With a population of about 2 million, Almaty is Kazakhstan's la
      </Panel>
      <Panel title="Etymology" isActive={true}>
        The name comes from <span lang="kk-KZ">алма</span>, the Kazakh
      </Panel>
    </>
  );
}

function Panel({ title, children, isActive }) {
  return (
    <section className="panel">
      <h3>{title}</h3>
      {isActive ? (
        <p>{children}</p>
      ) : (
        <button onClick={() => setIsActive(true)}>
          Show
        </button>
      )}
    </section>
  );
}

```

Show less

Try editing the hardcoded `isActive` values in the `Accordion` component and see the result on the screen.

Step 3: Add state to the common parent

Lifting state up often changes the nature of what you're storing as state.

In this case, only one panel should be active at a time. This means that the `Accordion` common parent component needs to keep track of *which* panel is the active one. Instead of a `boolean` value, it could use a number as the index of the active `Panel` for the state variable:

```
const [activeIndex, setActiveIndex] = useState(0);
```

When the `activeIndex` is `0`, the first panel is active, and when it's `1`, it's the second one.

Clicking the "Show" button in either `Panel` needs to change the active index in `Accordion`. A `Panel` can't set the `activeIndex` state directly because it's defined inside the `Accordion`. The `Accordion` component needs to *explicitly allow* the `Panel` component to change its state by passing an event handler down as a prop:

```
<>
  <Panel
    isActive={activeIndex === 0}
    onShow={() => setActiveIndex(0)}
  >
    ...
  </Panel>
  <Panel
    isActive={activeIndex === 1}
    onShow={() => setActiveIndex(1)}
  >
    ...
  </Panel>
</>
```

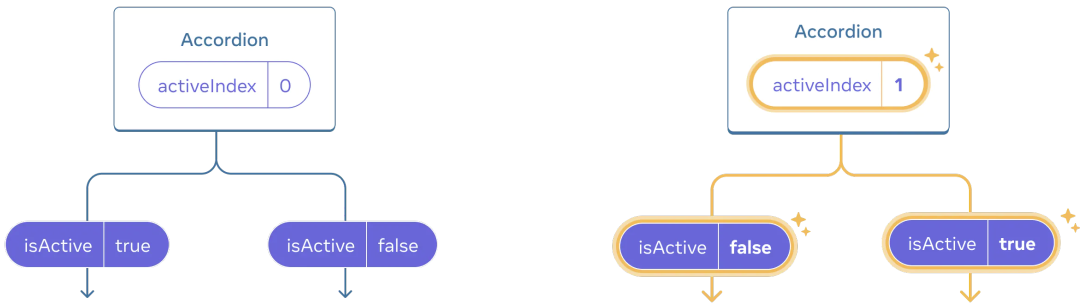
The `<button>` inside the `Panel` will now use the `onShow` prop as its click event handler:

App.js Download Reset

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

export default function Accordion() {
  const [activeIndex, setActiveIndex] = useState(0);
  return (
    <>
      <h2>Almaty, Kazakhstan</h2>
      <Panel
        title="About"
        isActive={activeIndex === 0}
        onShow={() => setActiveIndex(0)}
      >
        With a population of about 2 million, Almaty is Kazakhstan's la
      </Panel>
      <Panel
        title="Etymology"
        isActive={activeIndex === 1}
        onShow={() => setActiveIndex(1)}
      >
        The name comes from <span lang="kk-KZ">алма</span>, the Kazakh
      </Panel>
    </>
  );
}

function Panel({
  title,
  children,
  isActive,
  onShow
}) {
  return (
    <section className="panel">
      <h3>{title}</h3>
      {isActive ? (
        <p>{children}</p>
      ) : (
        <button onClick={onShow}>
          Show
        </button>
      )}
    </section>
  );
}

```

Show less

This completes lifting state up! Moving state into the common parent component allowed you to coordinate the two panels. Using the active index instead of two “is shown” flags ensured that only one panel is active at a given time. And passing down the event handler to the child allowed the child to change the parent’s state.



Initially, `Accordion`'s `activeIndex` is `0`, so the first `Panel` receives `isActive = true`

When `Accordion`'s `activeIndex` state changes to `1`, the second `Panel` receives `isActive = true` instead

DEEP DIVE

Controlled and uncontrolled components

Hide Details

It is common to call a component with some local state "uncontrolled". For example, the original `Panel` component with an `isActive` state variable is uncontrolled because its parent cannot influence whether the panel is active or not.

In contrast, you might say a component is "controlled" when the important information in it is driven by props rather than its own local state. This lets the parent component fully specify its behavior. The final `Panel` component with the `isActive` prop is controlled by the `Accordion` component.

Uncontrolled components are easier to use within their parents because they require less configuration. But they're less flexible when you want to coordinate them together. Controlled components are maximally flexible, but they require the parent components to fully configure them with props.

In practice, “controlled” and “uncontrolled” aren’t strict technical terms—each component usually has some mix of both local state and props. However, this is a useful way to talk about how components are designed and what capabilities they offer.

When writing a component, consider which information in it should be controlled (via props), and which information should be uncontrolled (via state). But you can always change your mind and refactor later.

A single source of truth for each state

In a React application, many components will have their own state. Some state may “live” close to the leaf components (components at the bottom of the tree) like inputs. Other state may “live” closer to the top of the app. For example, even client-side routing libraries are usually implemented by storing the current route in the React state, and passing it down by props!

For each unique piece of state, you will choose the component that “owns” it. This principle is also known as having a “single source of truth”. It doesn’t mean that all state lives in one place—but that for *each* piece of state, there is a *specific* component that holds that piece of information. Instead of duplicating shared state between components, you will *lift it up* to their common shared parent, and *pass it down* to the children that need it.

Your app will change as you work on it. It is common that you will move state down or back up while you’re still figuring out where each piece of the state “lives”. This is all part of the process!

To see what this feels like in practice with a few more components, read Thinking in React.

Recap

- When you want to coordinate two components, move their state to their common parent.
- Then pass the information down through props from their common parent.
- Finally, pass the event handlers down so that the children can change the parent's state.
- It's useful to consider components as "controlled" (driven by props) or "uncontrolled" (driven by state).

Try out some challenges

1. Synced inputs 2. Filtering a list

Challenge 1 of 2:

Synced inputs

These two inputs are independent. Make them stay in sync: editing one input should update the other input with the same text, and vice versa.

App.js Download Reset

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

export default function SyncedInputs() {
  return (
    <>
      <Input label="First input" />
      <Input label="Second input" />
    </>
  );
}
```

```

function Input({ label }) {
  const [text, setText] = useState('');

  function handleChange(e) {
    setText(e.target.value);
  }

  return (
    <label>
      {label}
      {' '}
      <input
        value={text}
        onChange={handleChange}
      />
    </label>
  );
}

```

Show less

Show hint Show solution Next Challenge

PREVIOUS

Choosing the State Structure

NEXT

Preserving and Resetting State

How do you like these docs?

Take our survey!

FACEBOOK

Open Source

Learn React

Quick Start

Installation

Describing the UI

Adding Interactivity

Managing State

Escape Hatches

API Reference

React APIs

React DOM APIs

Community

Code of Conduct

More

React Native

Acknowledgements

Docs Contributors

Meet the Team

Blog

Privacy

Terms