Sharing State

Situations where more than one component needs the same state -- lifting state up and the props drilling problem, with simple examples.

Intermediate 11 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/sharing-state

---

Sharing State

So far, we've always used `useState` inside a SINGLE component. This lesson covers situations where more than one component needs the same state -- and the "props drilling" problem that comes along with it.

When Two Components Need the Same State

Picture a search box (`SearchBox`) and a results list (`ResultsList`) -- both need the SAME search text. What happens if the `query` state lives inside `SearchBox` ?

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

function SearchBox() {
  const [query, setQuery] = useState("");

  return (
    <input
      type="text"
      value={query}
      onChange={(event) => setQuery(event.target.value)}
      placeholder="Search courses..."
    />
  );
}

function ResultsList() {
  // Problem: this component needs to know what the user typed into
  // SearchBox -- but the `query` state is trapped INSIDE SearchBox and
  // there is no way to reach it from here.
  return <p>Type in the search box above -- but this list has no way to know what you typed.
</p>;
}

function SeparateStateProblemExample() {
  return (
    <div>
      <SearchBox />
      <ResultsList />
    </div>
  );
}
```

`ResultsList` has no way to access `SearchBox`'s state -- each component's own state is trapped INSIDE it; sibling components can't see each other's state DIRECTLY.

Lifting State Up

The fix is to move the state to a place that's the COMMON ancestor of both components:

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

const courses = ["Java", "React", "Spring Boot"];

function SearchBox({ query, onQueryChange }) {
  return (
    <input
      type="text"
      value={query}
      onChange={(event) => onQueryChange(event.target.value)}
      placeholder="Search courses..."
    />
  );
}

function ResultsList({ query }) {
  const filtered = courses.filter((course) =>
    course.toLowerCase().includes(query.toLowerCase()),
  );

  return (
    <ul>
      {filtered.map((course) => (
        <li key={course}>{course}</li>
      ))}
    </ul>
  );
}

function LiftingStateUpExample() {
  // The `query` state no longer lives INSIDE SearchBox -- it now lives in
  // this component, the COMMON ancestor of both. This is what we call
  // "lifting state up". Both children receive this state via props.
  const [query, setQuery] = useState("");

  return (
    <div>
      <SearchBox query={query} onQueryChange={setQuery} />
      <ResultsList query={query} />
    </div>
  );
}
```

The `query` state now lives in the shared parent; both children receive it via props -- `SearchBox` gets `query` and `onQueryChange`, `ResultsList` gets just `query`. This pattern is called **lifting state up** -- it's the most fundamental way to manage shared state in React.

Seeing the Same Pattern in a Different Scenario

Lifting state up isn't limited to filtering a list -- it also applies to components that show the SAME value in TWO DIFFERENT ways:

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

function SliderInput({ rating, onRatingChange }) {
  return (
    <input
      type="range"
      min="0"
      max="5"
      value={rating}
      onChange={(event) => onRatingChange(Number(event.target.value))}
    />
  );
}

function RatingDisplay({ rating }) {
  return <p>Rating: {rating} / 5</p>;
}

function SyncedSiblingsExample() {
  // Two sibling components with different views (a slider and a text
  // display) represent the SAME value -- since the state has been
  // moved to their common ancestor, both always stay in sync.
  const [rating, setRating] = useState(0);

  return (
    <div>
      <SliderInput rating={rating} onRatingChange={setRating} />
      <RatingDisplay rating={rating} />
    </div>
  );
}
```

A slider and a text display share the SAME `rating` value -- since the state lives in the shared parent, when one changes, the other stays instantly in sync.

Props Drilling: Passing Through Intermediate Layers

As the component tree gets deeper, DELIVERING a prop to the component that needs it may require the components in between to also accept that prop:

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

const courses = ["Java", "React", "Spring Boot"];

function SearchBox({ query, onQueryChange }) {
  return (
    <input
      type="text"
      value={query}
      onChange={(event) => onQueryChange(event.target.value)}
      placeholder="Search courses..."
    />
  );
}

function ResultsList({ query }) {
  const filtered = courses.filter((course) =>
    course.toLowerCase().includes(query.toLowerCase()),
  );

  return (
    <ul>
      {filtered.map((course) => (
        <li key={course}>{course}</li>
      ))}
    </ul>
  );
}

function ResultsPanel({ query }) {
  // ResultsPanel never uses `query` ITSELF -- it only receives it to
  // PASS IT ALONG to ResultsList. This is a simple example of what we
  // call "props drilling" (passing props through intermediate layers
  // that don't need them).
  return (
    <div className="panel">
      <ResultsList query={query} />
    </div>
  );
}

function PropsDrillingExample() {
  const [query, setQuery] = useState("");

  return (
    <div>
      <SearchBox query={query} onQueryChange={setQuery} />
      <ResultsPanel query={query} />
    </div>
  );
}
```

`ResultsPanel` never uses `query` ITSELF -- it only accepts it to pass along to `ResultsList`. This is called **props drilling** -- being forced to pass a prop through intermediate layers that don't use it.

Why Props Drilling Hurts

This problem grows as the tree gets deeper:

```
function Level1({ user }) {
  return <Level2 user={user} />;
}

function Level2({ user }) {
  return <Level3 user={user} />;
}

function Level3({ user }) {
  return <Level4 user={user} />;
}

function Level4({ user }) {
  return <p>Logged in as {user}</p>;
}

function WhyPropsDrillingHurtsExample() {
  // NONE of Level1, Level2, or Level3 use `user` -- they just PASS IT
  // ALONG to the next level. Only Level4, at the very bottom, actually
  // uses it. As the tree gets deeper (or new props are added at every
  // level) this becomes tedious to write and error-prone -- in the next
  // lesson (Context API) we'll see an approach that solves this.
  return <Level1 user="Ada" />;
}
```

NONE of `Level1`, `Level2`, `Level3` use `user` -- they just pass it along. Only `Level4`, at the very bottom, actually uses it. Every new level, or every new shared value, stretches this chain further -- making the code tedious to write and fragile to change. In the next lesson (Context API), we'll see a way to solve this.

Summary and Glossary

When more than one component needs the same state, it needs to be moved to their COMMON ancestor (lifting state up) -- each child receives it via props. As the tree gets deeper, being forced to pass a prop through intermediate components that don't use it creates the "props drilling" problem -- making code tedious to write and fragile to change.

Glossary

Lifting State Up — Moving state shared by multiple components to their COMMON ancestor.

Props Drilling — Being forced to pass a prop through intermediate layer components that don't use it.