LEARN REACT >

Managing State

INTERMEDIATE

As your application grows, it helps to be more intentional about how your state is organized and how the data flows between your components. Redundant or duplicate state is a common source of bugs. In this chapter, you'll learn how to structure your state well, how to keep your state update logic maintainable, and how to share state between distant components.

You will learn

- How to think about UI changes as state changes
- How to structure state well
- How to "lift state up" to share it between components
- How to control whether the state gets preserved or reset
- How to consolidate complex state logic in a function
- How to pass information without "prop drilling"
- How to scale state management as your app grows

Reacting to input with state

With React, you won't modify the UI from code directly. For example, you won't write commands like "disable the button", "enable the button", "show the success message", etc. Instead, you will describe the UI you want to see for the different visual states of your component ("initial state", "typing state", "success state"), and

Here is a quiz form built using React. Note how it uses the `status` state variable to determine whether to enable or disable the submit button, and whether to show the success message instead.

App.js Download Reset Fork

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


export default function Form() {
  const [answer, setAnswer] = useState('');
  const [error, setError] = useState(null);
  const [status, setStatus] = useState('typing');


  if (status === 'success') {
    return <h1>That's right!</h1>
  }


  async function handleSubmit(e) {
    e.preventDefault();
    setStatus('submitting');
    try {
      await submitForm(answer);
```

Show more

City quiz

In which city is there a billboard that turns air into drinkable water?

Submit

Ready to learn this topic?

Read More

Choosing the state structure

Structuring state well can make a difference between a component that is pleasant to modify and debug, and one that is a constant source of bugs. The most important principle is that state shouldn't contain redundant or duplicated information. If there's some unnecessary state, it's easy to forget to update it, and introduce bugs!

For example, this form has a **redundant** `fullName` state variable:

App.js Download Reset Fork

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

export default function Form() {
  const [firstName, setFirstName] = useState('');
  const [lastName, setLastName] = useState('');
  const [fullName, setFullName] = useState('');

  function handleFirstNameChange(e) {
    setFirstName(e.target.value);
    setFullName(e.target.value + ' ' + lastName);
  }

  function handleLastNameChange(e) {
    setLastName(e.target.value);
    setFullName(firstName + ' ' + e.target.value);
  }
```

Show more

App.js Download Reset Fork

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

export default function Form() {
  const [firstName, setFirstName] = useState('');
  const [lastName, setLastName] = useState('');

  const fullName = firstName + ' ' + lastName;

  function handleFirstNameChange(e) {
    setFirstName(e.target.value);
  }

  function handleLastNameChange(e) {
    setLastName(e.target.value);
  }
```

Show more

This might seem like a small change, but many bugs in React apps are fixed this way.

Ready to learn this topic?

Read **Choosing the State Structure** to learn how to design the state shape to avoid bugs.

Read More

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.

In this example, only one panel should be active at a time. To achieve this, instead of keeping the active state inside each individual panel, the parent component holds the state and specifies the props for its children.

App.js

Download Reset Fork

```
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 largest c
      </Panel>
      <Panel
        title="Etymology"
```

Show more

Ready to learn this topic?

Read **Sharing State Between Components** to learn how to lift state up and keep components in sync.

Preserving and resetting state

When you re-render a component, React needs to decide which parts of the tree to keep (and update), and which parts to discard or re-create from scratch. In most cases, React's automatic behavior works well enough. By default, React preserves the parts of the tree that "match up" with the previously rendered component tree.

However, sometimes this is not what you want. For example, in this app, typing a message and then switching the recipient does not reset the input. This can make the user accidentally send a message to the wrong person:

App.js ContactList.js Chat.js Reset Fork

```
import { useState } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';

export default function Messenger() {
  const [to, setTo] = useState(contacts[0]);
  return (
    <div>
      <ContactList
        contacts={contacts}
        selectedContact={to}
        onSelect={contact => setTo(contact)}
      />
      <Chat contact={to} />
    </div>
  )
```

Show more

recipient is different, it should be considered a *different* `Chat` component that needs to be re-created from scratch with the new data (and UI like inputs). Now switching between the recipients always resets the input field—even though you render the same component.

App.js ContactList.js Chat.js Reset Fork

```
import { useState } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';

export default function Messenger() {
  const [to, setTo] = useState(contacts[0]);
  return (
    <div>
      <ContactList
        contacts={contacts}
        selectedContact={to}
        onSelect={contact => setTo(contact)}
      />
      <Chat key={to.email} contact={to} />
    </div>
  )
```

Show more

Ready to learn this topic?

Read **Preserving and Resetting State** to learn the lifetime of state and how to control it.

Read More

Extracting state logic into a reducer

Components with many state updates spread across many event handlers can get overwhelming. For these cases, you can consolidate all the state update logic outside your component in a single function, called “reducer.” Your event handlers become concise because they only specify the user “actions.” At the bottom of the file, the reducer function specifies how the state should update in response to each action!

App.js Reset Fork

```
import { useReducer } from 'react';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';

export default function TaskBoard() {
  const [tasks, dispatch] = useReducer(
    tasksReducer,
    initialTasks
  );

  function handleAddTask(text) {
    dispatch({
      type: 'added',
      id: nextId++,
      text: text,
    });
```

Show more

Ready to learn this topic?

Read **Extracting State Logic into a Reducer** to learn how to consolidate logic in the reducer function.

Passing data deeply with context

Usually, you will pass information from a parent component to a child component via props. But passing props can become inconvenient if you need to pass some prop through many components, or if many components need the same information. Context lets the parent component make some information available to any component in the tree below it—no matter how deep it is—without passing it explicitly through props.

Here, the `Heading` component determines its heading level by "asking" the closest `Section` for its level. Each `Section` tracks its own level by asking the parent `Section` and adding one to it. Every `Section` provides information to all components below it without passing props—it does that through context.

App.js Section.js Heading.js LevelContext.js Reset Fork

```
import Heading from './Heading.js';
import Section from './Section.js';

export default function Page() {
  return (
    <Section>
      <Heading>Title</Heading>
      <Section>
        <Heading>Heading</Heading>
        <Heading>Heading</Heading>
        <Heading>Heading</Heading>
        <Section>
          <Heading>Sub-heading</Heading>
          <Heading>Sub-heading</Heading>
          <Heading>Sub-heading</Heading>
```

Ready to learn this topic?

Read **Passing Data Deeply with Context** to learn about using context as an alternative to passing props.

Read More

Scaling up with reducer and context

Reducers let you consolidate a component's state update logic. Context lets you pass information deep down to other components. You can combine reducers and context together to manage state of a complex screen.

With this approach, a parent component with complex state manages it with a reducer. Other components anywhere deep in the tree can read its state via context. They can also dispatch actions to update that state.

App.js TasksContext.js AddTask.js TaskList.js Reset Fork

```
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';
import { TasksProvider } from './TasksContext.js';

export default function TaskBoard() {
  return (
    <TasksProvider>
      <h1>Day off in Kyoto</h1>
      <AddTask />
```

```
}
```

Ready to learn this topic?

Read **Scaling Up with Reducer and Context** to learn how state management scales in a growing app.

Read More

What's next?

Head over to Reacting to Input with State to start reading this chapter page by page!

Or, if you're already familiar with these topics, why not read about Escape Hatches?

PREVIOUS
Updating Arrays in State

NEXT
Reacting to Input with State

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

Acknowledgements

Meet the Team

More

React Native

Privacy

Terms