September 20, 2021 • 15 min read

Managing state in React is one of the main issues you'll be facing while developing React websites. `useState` is of course the most common way to create and manage state in (functional) React components.

There's also a lot of libraries offering opinionated ways to manage your entire (or part of) state, like Redux, Mobx, Recoil or XState.

But before jumping to a library to help you manage your state issues, you should be aware of another native way to manage your state in React: `useReducer`. It can be very powerful when used in the right way and for the right purpose. In fact, it's so powerful that the famous Redux library can be thought of as just a big, optimized `useReducer` (as we'll see).

In this article, we'll start by explaining what `useReducer` is and how to use it, giving you a good mental model and examples. We'll then go over a `useState` vs `useReducer` comparison to learn when to use which.

And for the TypeScript users out there, we'll also see how to use TypeScript and `useReducer` together.

Let's dive in!

What is React `useReducer` hook and how to use it

As mentioned in the introduction, **useState** and **useReducer** are the two native ways of managing state in React. You are probably already quite familiar with the former, so it's helpful to start there to understand **useReducer**.

useState and useReducer: a quick comparison

They are very similar at first glance. Let's see them side by side:

```
const [state, setState] = useState(initialValue);

const [state, dispatch] = useReducer(reducer, initialValue);
```

As you can see, in both cases the hook returns an array with two elements. The first is the **state**, and the second is a function that lets you modify the state: **setState** for **useState**, and **dispatch** for **useReducer**. We'll learn about how **dispatch** works later on.

An initial state is provided both for **useState** and **useReducer**. The main difference in the hook arguments is the **reducer** provided to **useReducer**.

For now, I'll just say that this **reducer** is a function that will handle the logic of how the state should be updated. We'll also learn about it in detail later in the article.

Now let's see how to change the state using either **setState** or **dispatch**. For this, we'll use the tried and tested example of a counter - we want to increment it by one when a button is clicked:

```
// with `useState`
<button onClick={() => setCount(prevCount => prevCount + 1)}>
  +
</button>

// with `useReducer`
<button onClick={() => dispatch({type: 'increment', payload: 1})}>
  +
</button>
```

version might look a bit strange.

Why are we passing an object with `type` and `payload` properties? Where is the (magic?) value `'increment'` coming from? Fret not, the mysteries will be explained!

For now, you can notice that both versions are still pretty similar. In either case, you update the state by calling the update function (`setState` or `dispatch`) with information on how exactly you want to update the state.

Let's now explore on a high level how the `useReducer` version exactly works.

`useReducer`: a backend mental model

In this section I want to give you a good mental model of how the `useReducer` hook works. This is important because when we're knee-deep in the implementation details, things can get a bit overwhelming. Especially if you've never worked with similar structures before.

One way to think about `useReducer` is to think of it as a backend. It might sound a bit strange, but bear with me: I'm very happy with this analogy and I think it explains reducers well.

A backend is usually structured with some way to persist data (a database) and an API that lets you modify the database.

That API has HTTP endpoints you can call. GET requests let you access the data, and POST requests let you modify it. When you do a POST request you can also give some parameters; for example if you want to create a new user you'll typically include the username, email and password of that new user in the HTTP POST request.

So, how is `useReducer` similar to a backend? Well:

- `state` is the database. It stores your data.

- **`dispatch`** is equivalent to the API endpoints being called to modify the database.
  - You can choose which endpoint to call by specifying the **`type`** of the call.
  - You can provide additional data with the **`payload`** property, which corresponds to the **`body`** of a POST request.
  - Both **`type`** and **`payload`** are properties of an object which is given to the **`reducer`**. That object is called the **`action`**.
- **`reducer`** is the logic of the API. It's called when the backend receives an API call (a **`dispatch`** call), and handles how to update the database based on the endpoint and request content (the **`action`**).

Here's a complete example of **`useReducer`** usage. Take a moment to take it in, and to compare it to the backend mental model described above.

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

// initial state of the database
const initialState = { count: 0 };

// API logic: how to update the database when the
// 'increment' API endpoint is called
const reducer = (state, action) => {
  if (action.type === 'increment') {
    return { count: state.count + action.payload };
  }
};

function App() {
  // you can think of this as initializing and setting
  // up a connection to the backend
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      {/* Reading from the database */}
      Count: {state.count}
      {/* calling the API endpoint when the button is clicked */}
```

```
      </button>
    </div>
  );
}

export default App;
```

Can you see how the two are related?

Remember that the code above shouldn't be used in production. It's a minimal version of the `useReducer` hook to help you compare it with the backend mental model, but it lacks several important things that you'll learn about in this article.

Now that (hopefully) you have a good idea of how `useReducer` is working on a high level, let's explore the details further.

How does the reducer work

We'll tackle the reducer first since it's where the main logic happens.

As you may have noticed from the example above, the reducer is a function that takes two arguments. The first is the current `state`, and the second is the `action` (which in our backend analogy corresponds to the API endpoint + any body the request might have).

Keep in mind that you'll never have to provide the arguments to the reducer yourself. This is handled by the `useReducer` hook automatically: the state is known, and the `action` is just the argument of `dispatch` which is passed along to the reducer as its second argument.

The `state` has whatever format you want (usually an object, but it can be anything really). The `action` can also be whatever you want, but there's some very commonly used convention on how to structure it and I advise you to follow those conventions- we'll be learning about them later. At least until

you're familiar with them and confident that departing from those is really what you want.

So conventionally, the `action` is an object with one required property and one optional property:

- `type` is the required property (analogous to the API endpoint). It tells the reducer what piece of logic it should be using to modify the state.
- `payload` is the optional property (analogous to the body of the HTTP POST request, if any). It provides additional information to the reducer on how to modify the state.

In our previous example of a counter, `state` was an object with a single `count` property. `action` is an object whose `type` can be `'increment'`, and whose payload is the amount by which you want to increment the counter.

```
// this is an example `state`
const state = { count: 0 };

// this is an example `action`
const action = { type: 'increment', payload: 2 };
```

Reducers are usually structured with a `switch` statement on the action `type`, for example:

```
const reducer = (state, action) => {
  switch (action.type) {
    case 'increment':
      return { count: state.count + action.payload };
    case 'decrement':
      return { count: state.count - action.payload };
    case 'reset':
      return { count: 0 };
  }
};
```

In this example, the reducer accepts three kinds of action types: "increment", "decrement", and "reset". Both "increment" and "decrement" require an action

decreased. On the contrary, the "reset" type doesn't require any payload, as it resets the counter back to 0.

This is a very simple example, and real-life reducers are usually much bigger and more complex. We'll see ways to improve how we write reducers in further sections, as well as examples of what a reducer would look like in a real-life app.

How does the dispatch function work?

If you have understood how the reducer works, understanding the dispatch function is pretty simple.

Whatever argument given **`dispatch`** when you call it will be the second argument given to your **`reducer`** function (the **`action`**). By convention, that argument is an object with a **`type`** and an optional **`payload`**, as we saw in the last section.

Using our last reducer example, if we wanted to make a button that decreases the counter by 2 on click it would look like this:

```
<button onClick={() => dispatch({ type: 'decrement', payload: 2 })}>-</
```

And if we wanted to have a button that resets the counter to 0, still using our last example, you can omit the **`payload`**:

```
<button onClick={() => dispatch({ type: 'reset' })}>reset</button>
```

One important thing to note on **`dispatch`** is that React guarantees that its identity won't change between renders. That means that you don't need to put it in dependency arrays (and if you do, it won't ever trigger the dependency array). This is the same behaviour as the **`setState`** function from **`useState`**.

Info

If you're a bit fuzzy about that last paragraph, I've got you covered with this article on dependency arrays!

`useReducer` initial state

We haven't mentioned it a lot so far but **`useReducer`** also takes a second argument, which is the initial value you want to give to the **`state`**.

It's not a required parameter per se, but if you don't provide it the state will be **`undefined`** at first and that's rarely what you want.

You usually define the full structure of your reducer state in the initial state. It's commonly an object, and you shouldn't be adding new properties to that object inside your reducer.

In our counter example the initial state was simply:

```
// initial state of the database
const initialState = { count: 0 };

. . .

// usage inside of the component
const [state, dispatch] = useReducer(reducer, initialState);
```

We'll see more examples of this further down the road.

`useReducer` tips and tricks

There are several ways in which we can improve our use of **`useReducer`**. Some of those are things you should really be doing, others are more matters of personal taste.

I've roughly classified them from important to optional, starting with the most important ones.

The reducer should throw an error for unknown action types

In our counter example we had a switch statement with three cases:

editor, you might have noticed ESLint being mad at you.

> Tip
>
> You have **ESLint** right? If you don't you really should set it up!

ESLint (rightly) wants switch statements to have a default case. So, what should be the reducer default case when it's handling an unknown action type?

Some people like to simply return the state:

```
const reducer = (state, action) => {
  switch (action.type) {
    case 'increment':
      return { count: state.count + action.payload };
    case 'decrement':
      return { count: state.count - action.payload };
    case 'reset':
      return { count: 0 };
    default:
      return state;
  }
};
```

But I really don't like that. Either the action type is something you expect and should have a case for, or it's not, and returning the `state` is not what you want. This is basically creating a **silent error** when an incorrect action type is provided, and silent errors can be very hard to debug.

Instead, your default reducer case should be throwing an error:

```
const reducer = (state, action) => {
  switch (action.type) {
    case 'increment':
      return { count: state.count + action.payload };
    case 'decrement':
      return { count: state.count - action.payload };
    case 'reset':
      return { count: 0 };
```

```
  }
};
```

That way, you won't miss a typo or forget about a case.

You should spread the state in every action

So far we have only seen a very simple `useReducer` example, in which the state is an object with only one property. Usually though, `useReducer` use cases call for state objects with at least a few properties.

A common `useReducer` usage is to handle forms. Here's an example with two input fields, but you could imagine the same with many more fields.

(Beware! The code below has a bug. Can you spot it?)

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

const initialValue = {
  username: '',
  email: '',
};

const reducer = (state, action) => {
  switch (action.type) {
    case 'username':
      return { username: action.payload };
    case 'email':
      return { email: action.payload };
    default:
```

```
};

const Form = () => {
  const [state, dispatch] = useReducer(reducer, initialValue);
  return (
    <div>
      <input
        type="text"
        value={state.username}
        onChange={(event) =>
          dispatch({ type: 'username', payload: event.target.value })
        }
      />
      <input
        type="email"
        value={state.email}
        onChange={(event) =>
          dispatch({ type: 'email', payload: event.target.value })
        }
      />
    </div>
  );
};

export default Form;
```

The bug is in the reducer: updating **`username`** will completely override the previous state and delete **`email`** (and updating **`email`** will do the same to **`username`**).

The way to solve this issue is to remember to keep all the previous state every time you update a property. This can be achieved easily with the **spread syntax**:

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

const initialValue = {
  username: '',
  email: '',
};

const reducer = (state, action) => {
```

```
      return { ...state, username: action.payload };
    case 'email':
      return { ...state, email: action.payload };
    default:
      throw new Error(`Unknown action type: ${action.type}`);
  }
};

const Form = () => {
  const [state, dispatch] = useReducer(reducer, initialValue);
  return (
    <div>
      <input
        value={state.username}
        onChange={(event) =>
          dispatch({ type: 'username', payload: event.target.value })
        }
      />
      <input
        value={state.email}
        onChange={(event) =>
          dispatch({ type: 'email', payload: event.target.value })
        }
      />
    </div>
  );
};

export default Form;
```

This example can actually be optimized further. You might have noticed that we are repeating ourselves a bit in the reducer: both the **`username`** and **`email`** cases have essentially the same logic. This isn't too bad for two fields but we could have many more.

There's a way to refactor the code to have only one action for all inputs, using the [ES2015 feature of computed keys]:

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

const initialValue = {
```

```
};

const reducer = (state, action) => {
  switch (action.type) {
    case 'textInput':
      return {
        ...state,
        [action.payload.key]: action.payload.value,
      };
    default:
      throw new Error(`Unknown action type: ${action.type}`);
  }
};

const Form = () => {
  const [state, dispatch] = useReducer(reducer, initialValue);
  return (
    <div>
      <input
        value={state.username}
        onChange={(event) =>
          dispatch({
            type: 'textInput',
            payload: { key: 'username', value: event.target.value },
          })
        }
      />
      <input
        value={state.email}
        onChange={(event) =>
          dispatch({
            type: 'textInput',
            payload: { key: 'email', value: event.target.value },
          })
        }
      />
    </div>
  );
};

export default Form;
```

update) and `value` (the value to update the `key` by).

Pretty neat if you ask me!

> Tip
>
> You might notice that we have one more place where we repeat ourselves in this code: the `onChange` event handler. The only thing that's changing is the `payload.key`.
>
> And indeed, you could further extract that into a reusable action to which you only have to provide the `key`.
>
> I tend to only have reusable actions when the reducer starts to get really big, or if very similar actions are repeated a lot.
>
> This is a very common pattern though, and we'll show an example of it later on in the article.

Stick to the conventional action structure

What I mean by "conventional action structure" is the structure we've been using so far in this article: `action` should be an object literal with a required `type` and an optional `payload`.

This is the Redux way of structuring actions and is also the most commonly used. It's tried and tested, and a very good default for all of your `useReducer`s.

The main con of that structure is that it can sometimes be a bit verbose. But

the Redux way.

Sugar syntax: deconstruct `type` and `payload` from action

This is a quality of life thing. Instead of repeating **`action.payload`** (and potentially **`action.type`**) everywhere in your reducer, you could directly deconstruct the reducer's second argument, like so:

```
const reducer = (state, { type, payload }) => {
  switch (type) {
    case 'increment':
      return { count: state.count + payload };
    case 'decrement':
      return { count: state.count - payload };
    case 'reset':
      return { count: 0 };
    default:
      throw new Error(`Unknown action type: ${type}`);
  }
};
```

You could even go a step further and also deconstruct the state. This is only handy if your reducer state is small enough, but it can be nice in those cases.

```
const reducer = ({ count }, { type, payload }) => {
  switch (type) {
    case 'increment':
      return { count: count + payload };
    case 'decrement':
      return { count: count - payload };
    case 'reset':
      return { count: 0 };
    default:
      throw new Error(`Unknown action type: ${type}`);
  }
};
```

That's it for the tips and tricks!

useReducer third parameter: lazy initialization

It's good to know that `useReducer` has an optional third argument. This argument is a function used to initialize the state lazily if you need to.

This is not used very often but it can be quite useful when you actually need it. The react documentation has a **good example of how to use that lazy initializaiton**.

`useState` vs `useReducer`: when to use which

Now that you know how `useReducer` works and how to use it in your components, we need to address an important question. Since `useState` and `useReducer` are two ways of managing state, which should you choose when?

These kinds of questions are always a tricky topic because the answer will usually change depending on who you ask, and it's also highly context-dependent. However, there are still guidelines that can orient you in your choice.

First off, know that `useState` should remain your default choice for managing React state. Only switch to `useReducer` if you start having trouble with `useState` (and if that trouble can be solved by switching to `useReducer`). At least until you're experienced enough with `useReducer` to know in advance which one to use.

I'll illustrate when to use `useReducer` over `useState` through a few examples.

One good use case for **`useReducer`** is when you have multiple pieces of state that rely on each other.

It's quite common when you're building forms. Let's say you have a text input and you want to track three things:

1. The value of the input.
2. Whether the input has already been "touched" by the user. This is useful to know whether to display an error. For example, if the field is required you want to display an error when it's empty. However, you don't want to display an error on the first render when the user has never visited the input before.
3. Whether there's an error.

With **`useState`**, you would have to use the hook three times and update three pieces of state separately each time there's a change.

With **`useReducer`**, the logic is actually quite simple:

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

const initialValue = {
  value: '',
  touched: false,
  error: null,
};

const reducer = (state, { type, payload }) => {
  switch (type) {
    case 'update':
      return {
        value: payload.value,
        touched: true,
        error: payload.error,
      };
    case 'reset':
      return initialValue;
    default:
      throw new Error(`Unknown action type: ${type}`);
  }
```

```
const Form = () => {
  const [state, dispatch] = useReducer(reducer, initialValue);
  console.log(state);
  return (
    <div>
      <input
        className={state.error ? 'error' : ''}
        value={state.value}
        onChange={(event) =>
          dispatch({
            type: 'update',
            payload: {
              value: event.target.value,
              error: state.touched ? event.target.value.length === 0 :
            },
          })
        }
      />
      <button onClick={() => dispatch({ type: 'reset' })}>reset</buttor
    </div>
  );
};

export default Form;
```

Add a bit of rudimentary CSS to style the **`error`** class, and you have the beginning of an input with good UX and simple logic, thanks to **`useReducer`**:

```
.error {
  border-color: red;
}

.error:focus {
  outline-color: red;
}
```

Manage complex state

Another good use case for **`useReducer`** is when you have a LOT of different pieces of state, and putting them all in **`useState`** would get really out of hand.

We saw earlier an example of a single reducer managing 2 inputs with the same action. We can easily scale that example up to 4 inputs.

While we're doing this, we might as well refactor the action out of each individual **`input`**:

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

const initialValue = {
  firstName: '',
  lastName: '',
  username: '',
  email: '',
};

const reducer = (state, action) => {
  switch (action.type) {
    case 'update':
      return {
        ...state,
        [action.payload.key]: action.payload.value,
      };
    default:
```

```
  }
};

const Form = () => {
  const [state, dispatch] = useReducer(reducer, initialValue);

  const inputAction = (event) => {
    dispatch({
      type: 'update',
      payload: { key: event.target.name, value: event.target.value },
    });
  };

  return (
    <div>
      <input
        value={state.firstName}
        type="text"
        name="firstName"
        onChange={inputAction}
      />
      <input
        value={state.lastName}
        type="text"
        name="lastName"
        onChange={inputAction}
      />
      <input
        value={state.username}
        type="text"
        onChange={inputAction}
        name="username"
      />
      <input
        value={state.email}
        type="email"
        name="email"
        onChange={inputAction}
      />
    </div>
  );
};
```

Seriously, how clean and clear is that code? Imagine doing this with 4 `useState` instead! All right, it wouldn't be *that* bad, but this can scale to the number of inputs you want without adding anything else than the input itself.

And you could also easily build further on that. For example, we may want to add the `touched` and `error` property of the last section to each of the four inputs in this section.

In fact, I advise you to try it yourself, it's a good exercise to cement your learnings so far!

What about doing this but with `useState` instead?

One way to get rid of a dozen of `useState` statements is to just put all of your state into one object stored in a single `useState`, and then update that.

This solution works, and sometimes it's a good way to go. But you'll often find yourself re-implementing a `useReducer` in a more awkward way. Might as well use a reducer right away.

`useReducer` with TypeScript

All right, you should be getting the hang of `useReducer` now. If you're a TypeScript user, you're probably wondering how to properly make the two play nice.

Thankfully it's pretty easy. Here it is:

```
import { useReducer, ChangeEvent } from 'react';

type State = {
  firstName: string;
  lastName: string;
  username: string;
  email: string;
};

type Action =
```

```
      payload: {
        key: string;
        value: string;
      };
    }
  | { type: 'reset' };

const initialValue = {
  firstName: '',
  lastName: '',
  username: '',
  email: '',
};

const reducer = (state: State, action: Action) => {
  switch (action.type) {
    case 'update':
      return { ...state, [action.payload.key]: action.payload.value };
    case 'reset':
      return initialValue;
    default:
      throw new Error(`Unknown action type: ${action.type}`);
  }
};

const Form = () => {
  const [state, dispatch] = useReducer(reducer, initialValue);

  const inputAction = (event: ChangeEvent<HTMLInputElement>) => {
    dispatch({
      type: 'update',
      payload: { key: event.target.name, value: event.target.value },
    });
  };

  return (
    <div>
      <input
        value={state.firstName}
        type="text"
        name="firstName"
        onChange={inputAction}
      />
```

```
          type="text"
          name="lastName"
          onChange={inputAction}
        />
        <input
          value={state.username}
          type="text"
          onChange={inputAction}
          name="username"
        />
        <input
          value={state.email}
          type="email"
          name="email"
          onChange={inputAction}
        />
      </div>
    );
};

export default Form;
```

If you're unfamiliar with the syntax of the `Action` type, it's [a discriminated union](#).

Redux: an overpowered `useReducer`

We're closing on the end of our `useReducer` guide (phew, it turned out way longer than I anticipated!). There's still one important thing to mention: [Redux](#).

You might have heard of Redux as this very popular state management library. Some people hate it, some people love it. But it turns out that all your brain juice that went into understanding `useReducer` is useful to understand Redux.

In fact, you can think of Redux as just a big, global, managed and optimized `useReducer` for your entire app. It's really all it is.

You have a "store", which is your state, and you define "actions" that tell a "reducer" how to modify that store. Sounds familiar!

Of course there are some important differences, but if you've understood `useReducer` well you're in very good shape to understand Redux easily.

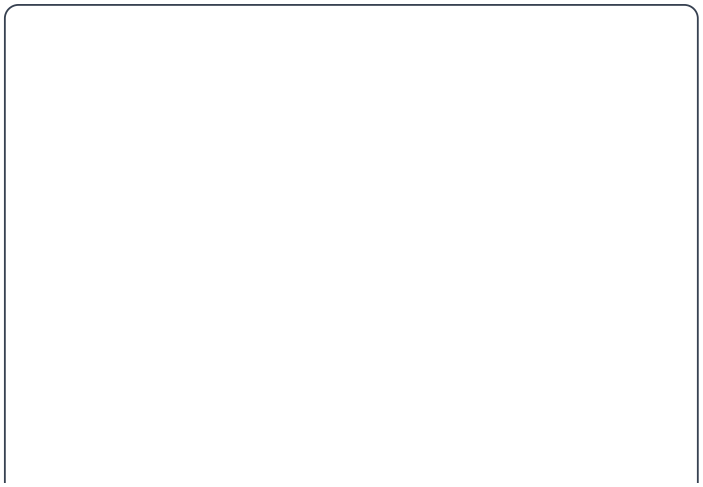
Wrap up

And that's the end of the article! I hope that it helped you learn everything you wanted about `useReducer`.

As you saw, it can be a very powerful tool in your React toolkit.

Good luck!

Package versions at the time of writing



complete guide)

How to use React Context like a pro

What are dependency arrays in React?

How to set an interval in React (with examples)

How to set a timeout in React (with examples)

How to add keyboard shortcuts to your React app

React events and TypeScript: a complete guide

Join the newsletter!

Be alerted when new articles regularly come out!

First Name *

Email *

Subscribe

Home

About