# Understanding the React useReducer Hook

Stephen Hartfield

> If you are at all familiar with reducers, you should have a good understanding of what React's `useReducer` hook does. Plain and simple, it allows functional components in React access to reducer functions from your state management. If you are not familiar with reducers, read this Reducers Introduction first.

The basic set up of the useReducer hook is the following:

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

## Example Store

Let's look at an example using a reducer. It always starts with state management - your store of data somewhere away from your main components. Here's a store of data, where we have four live people (this is our `initialState`):

```
const people = [
  {name: 'Jay', alive: true},
  {name: 'Kailee', alive: true},
  {name: 'John', alive: true},
  {name: 'Mia', alive: true}
]
```

And there you have it, four lively people. But not for long, 'cause that's no fun. Let's create a reducer to destroy them:

```
const reducer = (people, action) => {
  if(action.type == 'chomp') {
    return people.map(person => {
      if(person.name == action.payload) {
        person.alive = false;
      }
      return person;
    })
```

```
  }
  if(action.type == 'revive') {
    return people.map(person => {
      if(person.name == action.payload) {
        person.alive = true;
      }
      return person;
    })
  }
}
```

You can choose your method of murder: `'incinerate'`, `'hang'`, etc. The important thing is that we can detect the method of murder and record that we killed our target. We also have the ability to revive them if they are dead - that's probably a nice idea since we are using them for our experiment.

You'll notice in our reducer the conventional `action.payload`. You can change this name along with `action.type` to whatever fits your app. The important thing to notice here is what we pass to our reducer function: people (our initial state) and action (you could think of this as a `setState` method). The `action.type` identifies what type of action we want to do and `action.payload` identifies for us which person we want to perform the action on.

As a side note, we could use our reducer function to remove a person from our array completely when we kill them ☠ ☠ ☠. In our example, we are switching them to either dead or alive, using a boolean. Either way is easy to manage with reducers.

## *useReducer Hook*

Now that we have our reducer and store setup, let's implement them with our useReducer Hook. Like the other React hooks, we can import `useReducer` from `react`:

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

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

The `reducer` here in `useReducer(reducer, people)` is the constant we defined earlier. These names need to match in order to use the reducer we defined. Same with our initial state as `people`, which we also defined before. Name the two constants whatever suits your taste, as long as they match up. `state` represents the people in our store being passed in and `dispatch` is sort of an alias for action in our reducer.

## JSX

You can pass useReducer to your components through the Context API or whatever means you choose. You could have the useReducer in your component file and simply import your reducer/initialState. For context, you can bring in your constants hooked up to useReducer using the useContext Hook:

```
const {state, dispatch} = useContext(StoreContext);
```

More on useContext here.

Once you've got your reducer imported to your JSX, you can implement it and setup our dispatch functions to be rendered to the UI.

```
return (
  <div>
    {state.map((person, idx) => (
      <div key={idx} style={{display: 'flex', width: '50%', justifyContent: 'space
        <div>{person.name}</div>
        {person.alive ?
        <div> ✨✨ ALIVE! ✨✨</div> :
        <div> ☠️ ☠️ DEAD ☠️ ☠️</div>}
      </div>
    ))}
  </div>
)
```

The `state` here, which we `map` over, is the state constant we defined in our useReducer and imported over. It represents the current state of our people store, so state is a proper name. Each person in our "people" store has a name and an alive boolean to determine whether they are alive.

Next, let's implement our dispatch functions. Remember how we setup the action, the info that we need in order to update our store. We used a `type` to identify the action, and a `payload` to identify who was our next victim. It's important that we dispatch actions with matching identifiers.

```
function devour(name) {
  dispatch({ type: 'chomp', payload: name });
```

```
}

function spitOut(name) {
  dispatch({ type: 'revive', payload: name });
}

return (
  <div>
    {state.map((person, idx) => (
      <div key={idx} style={{ display: 'flex', width: '50%', justifyContent: 'spac
        <div>{person.name}</div>
        {person.alive ?
          <div> ✨✨ ALIVE! ✨✨ <button onClick={() => devour(person.name)}> 🔪
          <div> ☠️ ☠️ DEAD ☠️ ☠️ <button onClick={() => spitOut(person.name)}> 🤮 S
      </div>
    ))}
  </div>
)
```

The `dispatch` method receives an object that represents the action we desire to be done. We ultimately pass our action to our reducer through the useReducer. Our reducer returns updated state.

## *The Third Parameter*

There is also a third parameter `useReducer` can receive. It's the optional init function, which will allow you to lazily create the initial state. This is helpful if you want the initial state to be different depending on a situation; instead of using our people constant above, we could create the initial state anywhere, perhaps dynamically, and it will override our initial state.

```
const deadPeople = () => ([
  { name: 'Jay', alive: false },
  { name: 'Kailee', alive: false },
  { name: 'John', alive: false },
  { name: 'Mia', alive: false }
])

// ...
// wherever our useReducer is located
const [state, dispatch] = useReducer(reducer, people, deadPeople);
```

Notice, this new initial state is a function, not just an array. The third parameter takes a function to override initial state.

## *Further Reading*

Please see the official docs to learn more. It's always good to read straight from the creators, especially when people's lives hang on the balance.

*hot react picks*

React Hooks

styled-components

emotion

Suspense

React + Bulma

React + Axios

React Motion

*latest react posts*

A Sneak Peek at React Router v6

Building a CRUD App with React Hooks & the Context API

A Simple Guide to Error Boundaries in React

A Simple Guide to Error Boundaries in React

5 Tips to Improve the Performance of Your React Apps

all react posts

follow us @alligatorio

Published: November 27, 2019