React State Management

OCTOBER 14, 2019 BY ROBIN WIERUCH - EDIT THIS POST

Follow on Twitter 17k Follow on Facebook

State in React is one of the most important topics when learning React. State breathes live into your React application. It's what makes your application grow beyond static content being displayed on a website, because a user can interact with it. Every interaction of a user with your application may change the underlying state which lead to changes in the UI represented by the state.

In this extensive walkthrough, I want to guide you through all the possibilities of state management in React. We will start with simple state management that is co-located to React components, exploring all its features with React Hooks, and end with more complex global state in React managed by React itself or third-party libraries.

Once you went through this state management in React walkthrough, you should have a good grasp of what's state in React. Maybe it goes beyond this and you get a good idea about how state should be managed in an ideal scenario in your next React application.

TABLE OF CONTENTS

WHAT IS STATE IN REACT?

The UI of a frontend application is a representation of its state. State is just a snapshot in time. If a user changes state by interacting with your application, the UI may look completely different afterward, because it's represented by this new state rather than the old state.

```
State => UI
```

State can be various things:

- 1) A boolean which tells the UI that a dialog/modal/popover component is opened or closed.
- 2) An user object which reflects the currently signed in user of the application.
- 3) Data from a remote API (e.g. an object/list of users), that is fetched in React and displayed in your UI.

State is just another fancy word for a JavaScript data structure representing the state with JavaScript primitives and objects. For instance, a simple state could be a JavaScript boolean whereas a more complex UI state could be a JavaScript object:

```
// 1)
const isOpen = true;

// 2)
```

```
const user = {
  id: '1',
  firstName: 'Robin',
  lastName: 'Wieruch',
  email: 'hello@robinwieruch.com',
};

// 3)
const users = {
  2: {
    firstName: 'Dennis',
    lastName: 'Wieruch',
    email: 'hello@denniswieruch.com',
  },
  3: {
    firstName: 'Thomas',
    lastName: 'Wieruch',
    email: 'hello@thomaswieruch.com',
  },
};
```

Every of these states could be managed by a single React component which is mainly doing three things:

A) storing the state
B) enabling the user to modify the state
C) updating the UI once the state has been changed

This can be done *within* a React component with React Hooks. I am saying *within* here, because it's co-located state to the React component by using Hooks. Later you will learn about other state that is managed *globally and outside* of React components. Let's explore the React Hooks for state first.

REACT STATE: USESTATE

React's useState hook is for many React beginners their first encounter with state in React:

```
import React from 'react';

const App = () => {
  const [counter, setCounter] = React.useState(42);

  const handleClick = () => {
    setCounter(counter + 5);
  };

  return (
    <>
      <p>{counter}</p>

      <button type="button" onClick={handleClick}>
        Increase by 5
      </button>
    </>
  );
};
```

The useState hook takes an initial state as argument, just for the first time the React component renders, and returns an array with two values: the current state and the state update function. Whereas the current state is used to display it somewhere within your React component, the state update function is used to change the current state (e.g. HTML button `onClick`).

Taking it one step further, it cannot be just used to increase an integer, but also to capture more dynamic state of an input HTML element when typing into it. Because the input HTML element takes the current state as value, it becomes a controlled component/element. Not the internal HTML manages the state anymore, but React's state management:

```
import React from 'react';

const App = () => {
  const [text, setText] = React.useState('Hello React');

  const handleChange = event => {
    setText(event.target.value);
  };

  return (
    <>
      <p>{text}</p>

      <input type="text" value={text} onChange={handleChange} />
    </>
  );
};
```

After all, React's useState is your gateway into state management with React. Everything that follows from here is more powerful yet more complex.

Exercises:

- Read more about React's useState Hook
- Read more about Controlled Components in React

REACT STATE: USEREDUCER

React's useReducer derives from the concept of a JavaScript Reducer. The idea: A reducer function takes the current state and an action with payload and computes it to a new state:

```
(state, action) => newState
```

A reducer function may look like the following for managing the state of a list of todo items and their `complete` status:

```
const todoReducer = (state, action) => {
  switch (action.type) {
    case 'DO_TODO':
      return state.map(todo => {
        if (todo.id === action.id) {
          return { ...todo, complete: true };
        } else {
          return todo;
        }
      });
    case 'UNDO_TODO':
      return state.map(todo => {
        if (todo.id === action.id) {
          return { ...todo, complete: false };
        } else {
          return todo;
        }
      });
    default:
      return state;
  }
};
```

Depending on the incoming action's type, one of the switch cases is taken to either complete or incomplete a todo item. The action's payload, here the `id` property, tells the reducer which todo item in the list, which is the `state` itself, should be toggled. All the other todo items are not changed.

Now consider the following initial state for a React component:

```
const initialTodos = [
  {
    id: 'a',
    task: 'Learn React',
    complete: false,
  },
  {
    id: 'b',
    task: 'Learn Firebase',
    complete: false,
  },
];
```

A React component using this reducer function with React's useReducer hook may look like the following:

```
const App = () => {
  const [todos, dispatch] = React.useReducer(
    todoReducer,
    initialTodos
  );

  const handleChange = todo => {
    dispatch({
      type: todo.complete ? 'UNDO_TODO' : 'DO_TODO',
      id: todo.id,
    });
  };

  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}>
          <label>
            <input
              type="checkbox"
              checked={todo.complete}
              onChange={() => handleChange(todo)}
            />
            {todo.task}
          </label>
        </li>
      ))}
    </ul>
  );
};
```

In this scenario, there is only an initial list of todo items where an individual item can be toggled to completed or in-completed. The useReducer hook takes the previously defined reducer and an initial state as arguments, just for the first time the React component renders, and returns an array with two values: the current state and the state update function.

In contrast to the React's useState hook, the state update function from the useReducer hook always takes an action as argument. It's commonly called `dispatch` because it "dispatches" an action to the reducer. Each action comes with a mandatory `type` property, which is used later in the reducer function, and additional payload, which is here the todo item's `id` property.

When to use useReducer or useState? Generally speaking, React's useReducer hook can be used over React's useState hook, if (A) a state transition becomes more complex, (B) the state object isn't a simple JavaScript primitive anymore, or most importantly, if (C) multiple states are conditionally related to each other. Naturally this happens if a React application grows beyond a small-sized project.

Exercises:

- Read more about JavaScript Reducer
- Read more about React's useReducer Hook
- Read more about when to use useState or useReducer

REACT STATE: USECONTEXT

Technically React's useContext hook isn't related to state. It makes it just more convenient to pass props down the component tree. Normally React props are passed from parent to child components; however, React's Context API allows it to tunnel React components in between. Thus it's possible to pass props from a grandfather component to a grandchild component without bothering the other React components in between of the chain.

However, React's Context API mechanism is indeed used for more advanced state management in React: It tunnels the current state and the state update function -- either returned from useState or useReducer -- through many React components. This way, developers started to manage React state at top-level components with useReducer/useState and pass their returned values -- current state and state update function -- with useContext to all the React child components that are interested in this state or updating this state.

```
const App = () => {
  const [filter, dispatchFilter] = useReducer(filterReducer, 'ALL');
  const [todos, dispatchTodos] = useReducer(todoReducer, initialTodos);

  const filteredTodos = todos.filter(todo => {
    ...
  });

  return (
    <TodoContext.Provider value={dispatchTodos}>
      <Filter dispatch={dispatchFilter} />
      <TodoList todos={filteredTodos} />
      <AddTodo />
    </TodoContext.Provider>
  );
};
```

In the previous code snippet, the ability to modify todo items with the `dispatchTodos` updater function is made globally available with React's Context API. In another step, React's useContext hook can be used in any child component to retrieve the dispatch function. Follow along in the exercises of this section to explore this concept in detail.

When to combine useContext with useReducer or useState?

- 1) Generally speaking, React's useContext hook should be used with React's useState hook and/or useReducer hook, if it becomes a burden to pass state and state update function down multiple component levels.
- 2) Strategically speaking, React's useContext hook can be used to move state from being local state to global state. While state would be managed globally at a top-level component, React's useContext hook is used to pass down state and state updater function to all child components interested in it. You will read more about this later.

Exercises:

- Read more about React's useContext Hook
- Read more about useContext combined with useState and useReducer for React state

LOCAL VS GLOBAL STATE

These are the three main strategies for state management in React:

- (1) Manage state within a React component.
- (2) Manage state within a top-level React component where it gets distributed to all child components.
- (3) Manage state outside of React with a third-party state management library.

All three strategies map to the following types of state:

- (1) local state
- (2) global state, but managed in React
- (3) global state, managed by a third-party state management library

In addition, enabling all three strategies map to various features or combinations of these features within or outside of React's capabilities:

- (1) useState and useReducer
- (2) useState/useReducer with useContext
- (3) Redux, MobX and various other state management libraries

You are not limited to just one of these strategies. Whereas smaller applications start out with managing state in a React component with useState and useReducer hooks, in a growing application developers start to manage state globally too, for state that is needed by more than one React component and state that is needed to be shared among a multitude of React components.

Technically React's useReducer + useContext hooks from strategy (2) enable one to create their own state management library like Redux from strategy (3).

```
const App = () => {
  const [state, dispatch] = useCombinedReducers({
    filter: useReducer(filterReducer, 'ALL'),
    todos: useReducer(todoReducer, initialTodos),
  });

  ...
};
```

Let's explore such implementation together with `useCombinedReducers` in the exercises of this section.

Exercises:

- Read more about how to create Redux with useReducer and useContext
  - Postpone reading this tutorial to the next section, if you need more clarity about Redux first

REACT STATE: REDUX

Even though React's useReducer came into the world after Redux, its concept origins from Redux itself. Redux just takes state management on another level. One may speak of the state is truly managed globally by an external force outside of React.

```
React => Action => Reducer(s) => Store => React
```

Whereas `Action => Reducer(s) => Store` encapsulates Redux. Let's recap all parts of Redux briefly in JS. This is a Redux Reducer that acts on two Redux Actions which has no dependencies on the Redux library at all:

```
function reducer(state, action) {
  switch(action.type) {
    case 'TODO_ADD' : {
      return applyAddTodo(state, action);
    }
    case 'TODO_TOGGLE' : {
      return applyToggleTodo(state, action);
    }
    default : return state;
  }
}

function applyAddTodo(state, action) {
  return state.concat(action.todo);
}

function applyToggleTodo(state, action) {
  return state.map(todo =>
    todo.id === action.todo.id
      ? { ...todo, completed: !todo.completed }
      : todo
  );
}
```

The Redux store which knows about the Redux Reducer:

```
import { createStore } from 'redux';

const store = createStore(reducer, []);
```

Then, the Redux Store offers a small API surface to interact with it -- e.g. dispatching a Redux Action:

```
store.dispatch({
  type: 'TODO_ADD',
  todo: { id: '0', name: 'learn redux', completed: false },
});
```

Finally, in JavaScript, you can listen to changes with the Redux Store:

```
store.subscribe(() => {
  console.log(store.getState());
});
```

That's Redux in a nutshell with all its fragments: Action, Reducer, Store. If you attach the store subscription to React, the React UI can update whenever the state in Redux changes.

Another popular alternative for Redux is MobX for state in React: Both state management libraries got very popular in the early days of React. However, there are other state management libraries out there, competing with both titans, by offering a more lightweight state management solution.

Exercises:

- Read more about why Redux makes you a better JS developer
- Read more about Redux vs useReducer
- Read more about Redux vs MobX
  - Optional: Learn Redux and Redux with React

ORIGIN OF STATE

What makes all kinds of state the same is the nature of its transitions from one state to another state. However, the origin of state differs for frontend applications. State can origin within the client application (frontend) or from a remote server application (backend).

For instance, state that origins within the client application can be a boolean flag of for the status of an open/closed dialog component. The client application defines the initial state (e.g. closed dialog) and defines the state transitions + the actual possible states (e.g. boolean flag is set to false or true):

- Open/Closed state for Dialog, Dropdown, Popover and DatePicker components.
- Selected item in a Dropdown component.
- Filter/Sort state of a Table component.
- Text in an InputField component.

In contrast, if state origins from a remote server application, the initial state and the transitions may be defined in the client application -- e.g. the initial state is `null` but once data arrives from an API the state is set to the actual `data` -- but the possible state coming from the backend application isn't foreseeable for the client application.

- List of users coming from a remote API.
- Currently signed in user coming from a remote API.

Why do we need to know about this at all? Managing state that origins within the client application tends to be easier to manage than managing state coming from a backend application. The former, managing state that origins from the client application, can be achieved with all three strategies we have learned about:

- (1) useState and useReducer
- (2) useState/useReducer with useContext
- (3) Redux, MobX and various other state management libraries

The latter, managing state that origins from the server application, tends to be more complex. It doesn't only come with no data (e.g. `null`) or actual filled data states, but also with additional states for error and progress. In addition, it's a repetitive process to set up all these states with your chosen strategy and it's a real pain once you consider advanced topics like caching and stale state. It comes with lots of pain points.

That's where another technology comes into play: GraphQL.

REACT STATE: GRAPHQL

GraphQL is not strictly related to state. GraphQL is an alternative to REST for client-server communication. However, with the right GraphQL library in place for your React application, managing state that origins from a server application becomes much simpler.

For instance, Apollo Client is one of these GraphQL client libraries. It can be used to read and write data from and to a remote GraphQL API via GraphQL queries and mutations. For instance, using a query to read data with Apollo within a React component may look the following way:

```
import React from 'react';
import gql from 'graphql-tag';
import { Query } from 'react-apollo';

const GET_CURRENT_USER = gql`
  {
    viewer {
      login
      name
    }
  }
`;

const Profile = () => (
  <Query query={GET_CURRENT_USER}>
    {({ data, loading }) => {
      if (data.viewer) {
        return null;
      }

      if (loading) {
        return <div>Loading ...</div>;
      }

      return (
        <div>
          {data.viewer.name} {data.viewer.login}
        </div>
      );
    }}
  </Query>
);

export default Profile;
```

Even though GraphQL is just used to define the GraphQL query, the Query component from Apollo Client makes sure to give you all the states necessary to represent the whole data fetching process in the UI. In this case, it gives you `data` and a `loading` state, but you can also access `error` state and more. There is no need to write all the state transitions yourself, you just leave it to the Apollo Client GraphQL library.

Also caching is taken care of in advanced GraphQL Client library. There are multiple advanced features which help you to avoid stale data and avoid unnecessary data fetching procedures, because the data is already there and cached for you.

Now, knowing about state that origins in client and server applications, it may be the best solution to differentiate between both origins by splitting up the responsibilities the following way:

- client origin state management solutions
  - useState/useReducer + useContext/Redux/MobX
- server origin state management solutions
  - GraphQL + powerful GraphQL library

For many React applications, I strongly believe it would make state management a breeze if just GraphQL and a powerful GraphQL client library would be used to accommodate the server originated state. What's left is the UI state which can be easily managed by React's Hooks. There is even no strong need for Redux anymore.

Exercises:

- Learn GraphQL with React

REACT STATE: THIS.STATE AND SETSTATE (LEGACY)

If you are not using React Class Components but only React Function Components, you don't need to read any further here. If you are still using React Class Components, then either

- migrate them to React Function Components for enabling React Hooks
- deal with state management in React Class Components the old-school way

The following example shows you how to manage state in React Class Components:

```
class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      value: '',
    };
  }

  onChange = event => {
    this.setState({ value: event.target.value });
  };

  render() {
    return (
      <div>
        <h1>Hello React ES6 Class Component!</h1>

        <input
          value={this.state.value}
          type="text"
          onChange={this.onChange}
        />

        <p>{this.state.value}</p>
      </div>
    );
  }
}
```

Either way, you can manage state in Class Components and Function Components. However, only React Hooks in React Function Components enable you to use more modern and powerful state management solutions in React. Combining React Hooks with GraphQL may be the ultimate combination for taming the state in React.

Show Comments

KEEP READING ABOUT REACT

REACT GLOBAL STATE WITHOUT REDUX

The article is a short tutorial on how to achieve global state in React without Redux. Creating a global state in React is one of the first signs that you may need Redux (or another state management...

REACT STATE WITHOUT CONSTRUCTOR

The article is a short tutorial on how to have state in React without a constructor in a class component and how to have state in React without a class component at all. It may be a great refresher on...

THE ROAD TO REACT

TAKE PART

PORTFOLIO

Online Courses

Open Source

Tutorials

ABOUT

About me

What I use

How to work with me

How to support me

Contact Me Privacy & Terms