React state management: Flux and Redux

Learn how to handle application state in a centralized store with Redux, the most popular unidirectional data flow library!

WE'LL COVER THE FOLLOWING

We've been using a top-level component called `App` to manage our global application state. That works fine for a small application, but as we add more and more functionality it becomes very tedious to work with.

Together with React, Facebook released something called Flux. Flux is a methodology that helps you manage your global application state. Flux works fine, but has the downside that it uses events, which can lead to quite a bit of confusion.

Thankfully, Dan Abramov stepped in and created Redux. Redux has the same core concept as Flux, but works without events, is much easer to understand and now basically the standard for application state management.

> Note: None of the following concepts and technologies are **necessary** to build a production app with react. Many people use them to their advantage, but they have some downsides too. (which we'll examine)

The Concept

Remember the initial state of our `App` component? It looks like this:

```
state = {
  return {
    location: '',
    data: {},
```

```
    dates: [],
    temps: [],

    selected: {
      date: '',
      temp: null
    }
  };
};
```

The object we return from this function is our entire application state. At the first start of our application, our state thus looks like this:

```
{
  location: '',
  data: {},
  dates: [],
  temps: [],
  selected: {
    date: '',
    temp: null
  }
}
```

When users now change the location input field, the `location` field of our state changes:

```
{
  location: 'Vienna, Austria',
  data: {},
  dates: [],
  temps: [],
  selected: {
    date: '',
    temp: null
  }
}
```

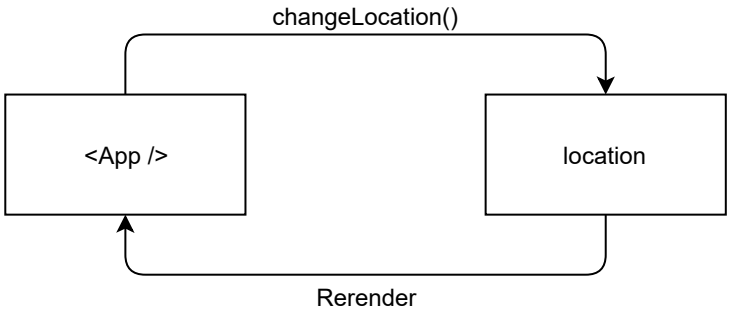
Instead of directly changing that location with `this.setState`, we'll call a function called `changeLocation` from our component. Redux will pick up that said function was called, do its magic and change the `location` field of our application state.

Now that the location is different and thus our application state has changed,

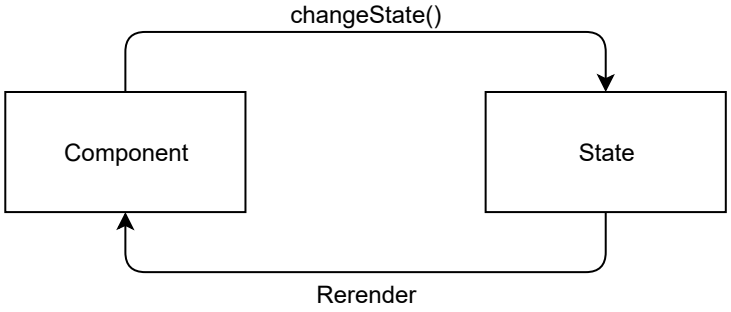
our main `<App />` component will automatically rerender with the new data! (just like with component state)

The big advantage of this approach is that the component no longer needs to know how exactly we save the location. We could be persisting it as a coordinate, we could save it without whitespace, but the component doesn't have to care about that—the component only calls `changeLocation` and that's it! The application state is thus decoupled from the individual components.

This cycle of state management thus looks like this:



If we put this into more general terms, we call a function which changes something in the application state which rerenders some component:



We'll now need to introduce some terminology before we can finally start implementing this. This function that we call to change the application state is called an "action" in Redux, and we "dispatch" the "action". Let's change the cycle one last time with the terminology: