

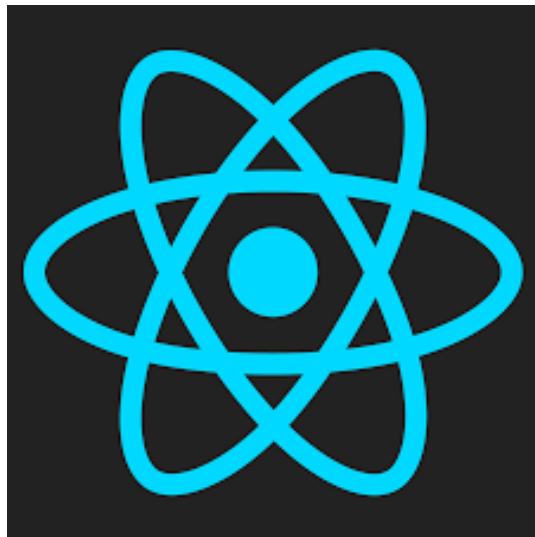


Osmel Mora

[Follow](#)Software developer | [https://twitter.com/osmel\\_mora](https://twitter.com/osmel_mora). Early adopter, passionate reader and learner. ...

Jul 18, 2016 · 6 min read

## How to handle state in React. The missing FAQ.



### Motivation

Recently there has been a lot of debate about how to manage the state in React. Some claim that `setState()` doesn't work as you might expect. That you have to externalize the state using a Flux-like state container and avoid completely the component's state.

On the other hand, there are people concerned that **these misconceptions could become a dogma**:

**MICHAEL JACKSON**


@mjackson

[Follow](#)

Nobody knows how to do anything without Redux anymore. It's ridiculous.

## Component-Based

Build encapsulated components that manage their own state, then compose them to make complex UIs.



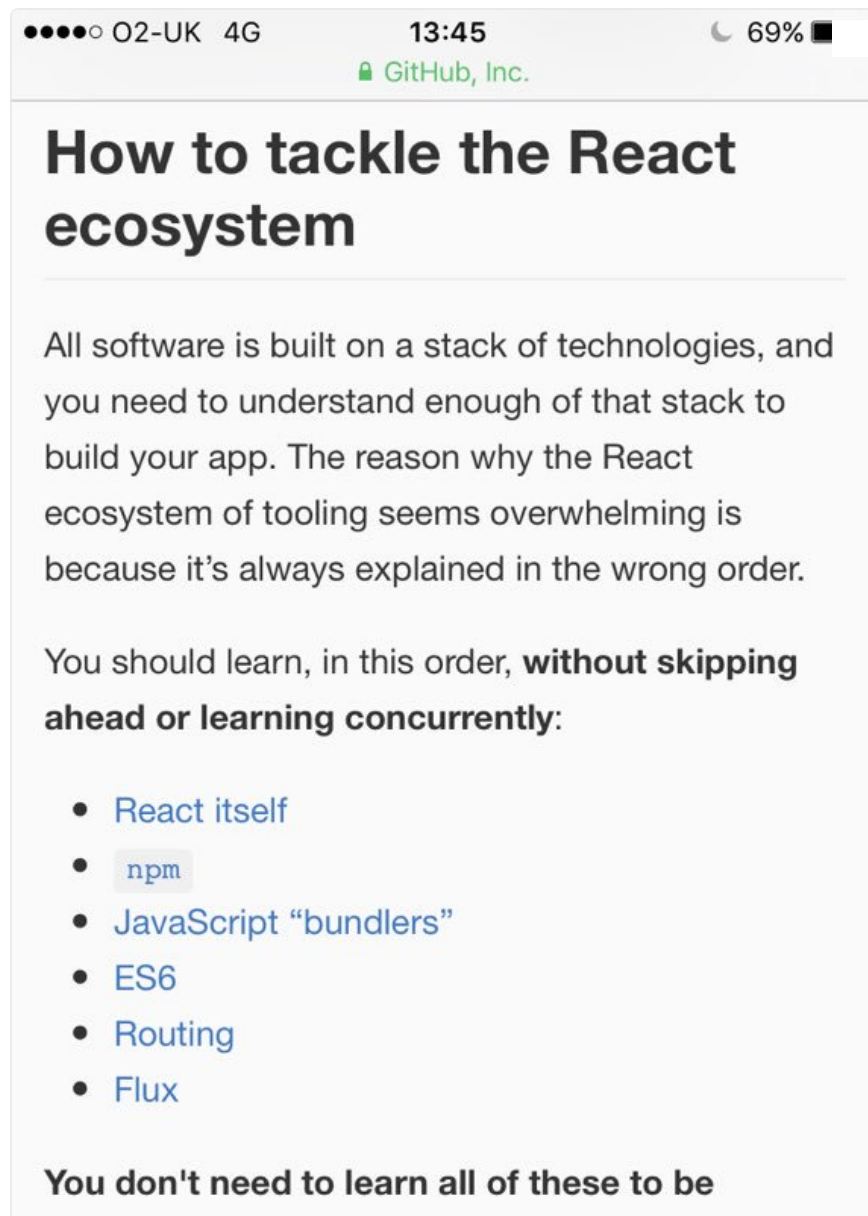
**Ryan Florence**  
@ryanflorence

Follow

Adding an external state container just for fun or because some tutorials tell you to do so, sounds like is not a good technical decision criteria.

To make myself clear, there is nothing wrong with Redux, MobX or whatever external state container you might think of. Actually they are pretty useful and come with a broad ecosystem to suit all your needs. But, the thing is: **you might not need them at all, or at least yet.**

If you are learning React from scratch, [Pete Hunt](#) coined the best advice you can get. Even the creator of Redux and React's core team member [Dan Abramov](#) recommends it:



That means you need to understand how to handle state in React way before thinking about Flux.

In case you are starting a real life application you can accomplish a lot without Flux. Dividing the components into two categories: containers and presentational. This way you'll gain in maintainability and reusability. Also, in case you need to introduce Flux later, the migration path will be cleaner. Giving you the option to make the decision at the last responsible moment.

Even if you are already using Flux, there are cases when you should use the component's state. Think of a component library that you

want to share across projects, where all the components are self-contained and independent. You really don't want to have a state container as a dependency here.

My point is:

- There are crossed opinions, misconceptions and lack of knowledge in the community about how to manage state in React.
- In order to embrace all the power React gives you, it is crucial a **solid understanding of how to handle state**.
- Don't add another layer of complexity to your application if you don't need it. Remember: **simplicity matters**.

The purpose of the following FAQ section is to mitigate the intricacies of handling state in React.

. . .

### *Frequently Asked Questions.*

## How does state work?

A React component is like a state machine that represents an user interface. Every user action potentially triggers a change in that state machine. Then, the new state is represented by a different React element.

React stores the component state in *this.state*. You can set the initial value of *this.state* in two different ways. Each one corresponding to the way you create the component:

```
// Using React.createClass
var Counter = React.createClass({
  getInitialState: function() {
    return {counter: 0};
  },
  ...
});
```

```
// Using ES6 classes
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {counter: 0};
  }
  ...
}
```

The component state can be changed calling:

```
this.setState(data, callback);
```

This method performs a shallow merge of *data* into *this.state* and re-renders the component. The *data* argument can be an object or a function that returns an object containing keys to update. The optional *callback*—if provided—will be called after the component finishes re-rendering. You'll rarely need to use this callback since React will take care of keeping your user interface up to date.

Let's take a look at an example:

```
1  class Counter extends React.Component {
2    constructor(props) {
3      super(props);
4      this.state = {count: 0};
5      this.incrementCounter = this.updateCounter.bind(this);
6      this.decrementCounter = this.updateCounter.bind(this);
7    }
8
9    render() {
10     return (
11       <div>
12         <div>{this.state.count}</div>
13         <input type='button' value='+' onClick={this.incrementCounter}/>
14         <input type='button' value='-' onClick={this.decrementCounter}/>
15       </div>
16     );
17   }
18 }
```

## What should I keep in React state?

[Dan Abramov](#) answered this question with one tweet:

```
function shouldIKeepSomethingInReactState() {  
  if (canICalculateItFromProps()) {  
    // Don't duplicate data from props in state.  
    // Calculate what you can in render() method.  
    return false;  
  }  
  if (!amIUsingItInRenderMethod()) {  
    // Don't keep something in the state  
    // if you don't use it for rendering.  
    // For example, API subscriptions are  
    // better off as custom private fields  
    // or variables in external modules.  
    return false;  
  }  
  // You can use React state for this!  
  return true;  
}
```

Basically he says don't keep state calculated from props, neither state that isn't used in the *render()* method. Let's exemplify:

```
// Don't duplicate data from props in state  
// Antipattern  
  
class Component extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {message: props.message};  
  }  
  
  render() {  
    return <div>{this.state.message}</div>;  
  }  
}
```

The problem with the above example is it will only set the state when the component is first created. When new props arrive the state remains the same thus the user interface doesn't update. Then you

have to update the state in *componentWillReceiveProps()*, leading to **duplication of source of truth**. You can do better just by avoiding this situation:

```
// Better example

class Component extends React.Component {
  render() {
    return <div>{this.props.message}</div>;
  }
}
```

The same applies when you hold state based on props calculation:

```
// Don't hold state based on props calculation
// Antipattern

class Component extends React.Component {
  constructor(props) {
    super(props);
    this.state = {fullName: `${props.name}
${props.lastName}`};
  }

  render() {
    return <div>{this.state.fullName}</div>;
  }
}

// Better approach

class Component extends React.Component {
  render() {
    const {name, lastName} = this.props;
    return <div>`${name} ${lastName}`</div>;
  }
}
```

Although, there is nothing wrong with setting initial state based on props if you make it clear that is only seed data:

```
// Not an antipattern
```

```

class Component extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
    this.onClick = this.onClick.bind(this);
  }

  render() {
    return <div onClick={this.onClick}>
{this.state.count}</div>;
  }

  onClick() {
    this.setState({count: this.state.count + 1});
  }
}

```

Last but not least:

```

// Don't hold state that you don't use for rendering.
// Leads to unneeded re-renders and other inconsistencies.
// Antipattern

```

```

class Component extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 0};
  }

  render() {
    return <div>{this.state.count}</div>;
  }

  componentDidMount() {
    const interval = setInterval(() => (
      this.setState({count: this.state.count + 1})
    ), 1000);

```

```

    this.setState({interval});
  }

```

```

  componentWillUnmount() {
    clearInterval(this.state.interval);
  }
}

```

//Better approach

```

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

```



```
    this.state = {count: 0};
  }

  render() {
    return <div>{this.state.count}</div>;
  }

  componentDidMount() {
    this._interval = setInterval(() => (
      this.setState({count: this.state.count + 1})
    ), 1000);
  }

  componentWillUnmount() {
    clearInterval(this._interval);
  }
}
```

## Is it true that `setState()` is asynchronous?

The short answer: **Yes**.

Basically when you invoke `setState()` React **schedules an update**, computations are delayed until necessary. The React documentation is a little misleading about this:

`setState()` does not immediately mutate `this.state` but creates a pending state transition. Accessing `this.state` after calling this method can potentially return the existing value.

Apparently these two sentences are in contradiction. Let's experiment a little bit to see what happens:

```
class Component extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 0};
    this.onClick = this.onClick.bind(this);
  }

  render() {
    return <div onClick={this.onClick}>
{this.state.count}</div>;
  }

  onClick() {
    this.setState({count: this.state.count + 1});
    console.log(this.state.count);
  }
}
```

```
    }  
  }  
}
```

When you render this component and interact with it you will see that the values shown in the console are the values from the **previous state**. That's because **React owns the event and knows enough to batch the update**. The result: an asynchronous state mutation.

But, what happens if the event comes from an external source?

```
// Calling setState() twice in the same execution context is  
a bad // practice. It's used here for illustration purposes.  
Instead use // an atomic update in real code
```

```
class Component extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {count: 0};  
  }  
  
  render() {  
    return <div>{this.state.count}</div>;  
  }  
  
  componentDidMount() {  
    this._interval = setInterval(() => {  
      this.setState({count: this.state.count + 1});  
      console.log(this.state.count);  
      this.setState({count: this.state.count + 1});  
      console.log(this.state.count);  
    }, 1000);  
  }  
  
  componentWillUnmount() {  
    clearInterval(this._interval);  
  }  
}
```

Snap! Returns the existing value as the documentation suggests, even calling `setState()` twice in the same execution context. That's because **React doesn't know enough to batch the update and has to update the state as soon as possible**.

That's tricky, I suggest you that always treat `setState()` as asynchronous and you'll avoid trouble.

## I heard that under certain circumstances calling `setState()` doesn't trigger a re-render. Which are those circumstances?

1. When you call `setState()` within `componentWillMount()` or `componentWillReceiveProps()` it won't trigger any **additional re-render**. React batches the updates.
2. When you return **false** from `shouldComponentUpdate()`. This way the `render()` method is completely skipped along with `componentWillUpdate()` and `componentDidUpdate()`.

Note:

If you want to dig more into the component's lifecycle I already wrote a post on it.

. . .

## Conclusions

Properly handling state in React can be a challenge. I hope by now you have a clearer picture.

If you have a question that hasn't been answered feel free to ask in comments or via [Twitter](#). I'll be glad to receive your feedback and include the question in the FAQ section.

I hope this post has helped to deepen your knowledge about React. If so, please recommend it to reach out more people.



