<https://www.freecodecamp.org/news/functional-setstate-is-the-future-of-react-374f30401b6b/>

**Functional setState is the future of React**

by Justice Mba

Update: I gave a follow up talk on this topic at React Rally. While this post is more about the “functional setState” pattern, the talk is more about understanding setState **deeply**

React has popularized functional programming in JavaScript. This has led to giant frameworks adopting the Component-based UI pattern that React uses. And now functional fever is spilling over into the web development ecosystem at large.

But the React team is far from relenting. They continue to dig deeper, discovering even more functional gems hidden in the legendary library.

So today I reveal to you a new functional gold buried in React, best kept React secret — **Functional setState!**

Okay, I just made up that name… and it’s not *entirely* new or a secret. No, not exactly. See, it’s a pattern built into React, that’s only known by few developers who’ve really dug in deep. And it never had a name. But now it does — **Functional setState!**

Going by [Dan Abramov](https://www.freecodecamp.org/news/functional-setstate-is-the-future-of-react-374f30401b6b/undefined)’s words in describing this pattern, **Functional setState** is a pattern where you

“Declare state changes separately from the component classes.”

Huh?

**Okay… what you already know**

React is a component based UI library. A component is basically a function that accept some properties and return a UI element.

function User(props) { return ( <div>A pretty user</div> );}

A component might need to have and manage its state. In that case, you usually write the component as a class. Then you have its state live in the class constructor function:

class User { constructor () { this.state = { score : 0 }; }

render () { return ( <div>This user scored {this.state.score}</div> ); }}

To manage the state, React provides a special method called setState(). You use it like this:

class User { ...

increaseScore () { this.setState({score : this.state.score + 1}); }

...}

Note how setState() works. You pass it ***an object*** containing part(s) of the state you want to update. In other words, the object you pass would have keys corresponding to the keys in the component state, then setState() updates or *sets* the state by merging the object to the state. Thus, “set-State”.

**What you probably didn’t know**

Remember how we said setState() works? Well, what if I told you that instead of passing an object, you could pass ***a function***?

Yes. setState() also accepts a function. The function accepts the *previous* state and *current* props of the component which it uses to calculate and return the next state. See it below:

this.setState(function (state, props) { return { score: state.score - 1 }});

Note that setState() is a function, and we are passing another function to it(functional programming… **functional setState**) . At first glance, this might seem ugly, too many steps just to set-state. Why will you ever want to do this?

**Why pass a function to setState?**

The thing is, [state updates may be asynchronous](https://facebook.github.io/react/docs/state-and-lifecycle.html#state-updates-may-be-asynchronous).

Think about [what happens when setState](https://facebook.github.io/react/docs/reconciliation.html)()[is called](https://facebook.github.io/react/docs/reconciliation.html). React will first merge the object you passed to setState() into the current state. Then it will start that *reconciliation*thing. It will create a new React Element tree (an object representation of your UI), diff the new tree against the old tree, figure out what has changed based on the object you passed to setState() , then finally update the DOM.

Whew! So much work! In fact, this is even an overly simplified summary. But trust in React!

React does not simply “set-state”.

Because of the amount of work involved, calling setState() might not *immediately*update your state.

React may batch multiple setState() calls into a single update for performance.

What does React mean by this?

First, “*multiple setState() calls”*could mean calling setState() inside a single function more than once, like this:

...

state = {score : 0};

// multiple setState() callsincreaseScoreBy3 () { this.setState({score : this.state.score + 1}); this.setState({score : this.state.score + 1}); this.setState({score : this.state.score + 1});}

...

Now when React, encounters “*multiple setState() calls”,*instead of doing that “set-state” **three whole times**, React will avoid that huge amount of work I described above and smartly say to itself: “No! I’m not going to climb this mountain three times, carrying and updating some slice of state on every single trip. No, I’d rather get a container, pack all these slices together, and do this update just once.” And that, my friends, is**batching**!

Remember that what you pass to setState() is a plain object. Now, assume anytime React encounters “*multiple setState() calls”,*it does the batching thing by extracting all the objects passed to each setState() call, merges them together to form a single object, then uses that single object to do setState() .

In JavaScript merging objects might look something like this:

const singleObject = Object.assign( {}, objectFromSetState1, objectFromSetState2, objectFromSetState3);

This pattern is known as **object composition.**

In JavaScript, the way “merging” or **composing** objects works is: if the three objects have the same keys, the value of the key of the *last object* passed to Object.assign() wins. For example:

const me = {name : "Justice"}, you = {name : "Your name"}, we = Object.assign({}, me, you);

we.name === "Your name"; //true

console.log(we); // {name : "Your name"}

Because you are the last object merged into we, the value of name in the you object — “Your name” — overrides the value of name in the me object. So “Your name” makes it into the we object… you win! :)

Thus, if you call setState() with an object multiple times — passing an object each time — React will **merge**. Or in other words, it will **compose** a new object out of the multiple objects we passed it. And if any of the objects contains the same key, the value of the key of the *last* object with same key is stored. Right?

That means that, given our increaseScoreBy3 function above, the final result of the function will just be 1 instead of 3, because React did not *immediately* update the state in the order we called setState() . But first, React composed all the objects together, which results to this: {score : this.state.score + 1} , then only did “set-state” once — with the newly composed object. Something like this: User.setState({score : this.state.score + 1}.

To be super clear, passing object to setState() is not the problem here. The real problem is passing object to setState() when you want to calculate the next state from the previous state. So stop doing this. It’s not safe!

Because *this.props* and *this.state* may be updated asynchronously, you should not rely on their values for calculating the next state.

Here is a pen by [Sophia Shoemaker](https://www.freecodecamp.org/news/functional-setstate-is-the-future-of-react-374f30401b6b/undefined) that demos this problem. Play with it, and pay attention to both the bad and the good solutions in this pen:

**Functional setState to the rescue**

If you’ve not spent time playing with the pen above, I strongly recommend that you do, as it will help you grasp the core concept of this post.

While you were playing with the pen above, you no doubt saw that **functional setState**fixed our problem. But how, exactly?

Let’s consult the Oprah of React — Dan.

Note the answer he gave. When you do functional setState…

Updates will be queued and later executed in the order they were called.

So, when React encounters “*multiple functional setState() calls” ,*instead of merging objects together, (of course there are no objects to merge) React *queues*the functions “*in the order they were called.”*

After that, React goes on updating the state by calling each functions in the “queue”, passing them the *previous* state — that is, the state as it was *before* the first functional setState() call (if it’s the first functional setState() currently executing) or the state with the *latest* update from the *previous* functional setState() call in the queue.

Again, I think seeing some code would be great. This time though, we’re gonna fake everything. Know that this is not the real thing, but is instead just here to give you an *idea*of what React is doing.

Also, to make it less verbose, we’ll use ES6. You can always write the ES5 version later if you want.

First, let’s create a component class. Then, inside it, we’ll create a *fake*setState()method. Also, our component would have a increaseScoreBy3()method*,* which will do a multiple functional setState. Finally, we’ll instantiate the class, just as React would do.

class User{ state = {score : 0};

//let's fake setState setState(state, callback) { this.state = Object.assign({}, this.state, state); if (callback) callback(); }

// multiple functional setState call increaseScoreBy3 () { this.setState( (state) => ({score : state.score + 1}) ), this.setState( (state) => ({score : state.score + 1}) ), this.setState( (state) => ({score : state.score + 1}) ) }}

const Justice = new User();

Note that setState also accepts an optional second parameter — a callback function. If it’s present React calls it after updating the state.

Now when a user triggers increaseScoreBy3(), React queues up the multiple functional setState. We won’t fake that logic here, as our focus is on **what actually makes functional setState safe*.***But you can think of the result of that “queuing” process to be an array of functions, like this:

const updateQueue = [ (state) => ({score : state.score + 1}), (state) => ({score : state.score + 1}), (state) => ({score : state.score + 1})];

Finally, let’s fake the updating process:

// recursively update state in the orderfunction updateState(component, updateQueue) { if (updateQueue.length === 1) { return component.setState(updateQueue[0](component.state)); }

return component.setState( updateQueue[0](component.state), () => updateState( component, updateQueue.slice(1)) );}

updateState(Justice, updateQueue);

True, this is not as so sexy a code. I trust you could do better. But the key focus here is that every time React executes the functions from your **functional setState,** React updates your state by passing it a *fresh*copy of the updated state. That makes it possible for **functional setState** to **set state based on the previous state**.

Here I made a bin with the complete code. Tinker around it (possibly make it look sexier), just to get more sense of it.

[**FunctionalSetStateInAction**](http://jsbin.com/najewe/edit?js,console)  
[*A Play with the code in this bin will be fun. Remember! we’re just faking React to get the idea...*jsbin.com](http://jsbin.com/najewe/edit?js,console)

Play with it to grasp it fully. When you come back we’re gonna see what makes functional setState truly golden.

**The best-kept React secret**

So far, we’ve deeply explored why it’s safe to do multiple functional setStates in React. But we haven’t actually fulfilled the *complete* definition of functional setState: “Declare state changes separately from the component classes.”

Over the years, the logic of setting-state — that is, the functions or objects we pass to setState() — have always lived *inside*the component classes. This is more imperative than declarative.

Well today, I present you with newly unearthed treasure — the **best-kept React secret:**

Thanks to [Dan Abramov](https://www.freecodecamp.org/news/functional-setstate-is-the-future-of-react-374f30401b6b/undefined)!

That is the power of functional setState. Declare your state update logic *outside*your component class. Then call it *inside*your component class.

// outside your component classfunction increaseScore (state, props) { return {score : state.score + 1}}

class User{ ...

// inside your component class handleIncreaseScore () { this.setState( increaseScore) }

...}

This is declarative! Your component class no longer cares *how* the state updates. It simply *declares* the *type*of update it desires.

To deeply appreciate this, think about those complex components that would usually have many state slices, updating each slice on different actions. And sometimes, each update function would require many lines of code. All of this logic would live *inside*your component. But not anymore!

Also, if you’re like me, I like keeping every module as short as possible, but now you feel like your module is getting too long. Now you have the power to extract all your state change logic to a different module, then import and use it in your component.

import {increaseScore} from "../stateChanges";

class User{ ...

// inside your component class handleIncreaseScore () { this.setState( increaseScore) }

...}

Now you can even reuse the increaseScore function in a *different* component. Just import it.

What else can you do with functional setState?

Make testing easy!

You can also pass **extra** arguments to calculate the next state (this one blew my mind… #funfunFunction).

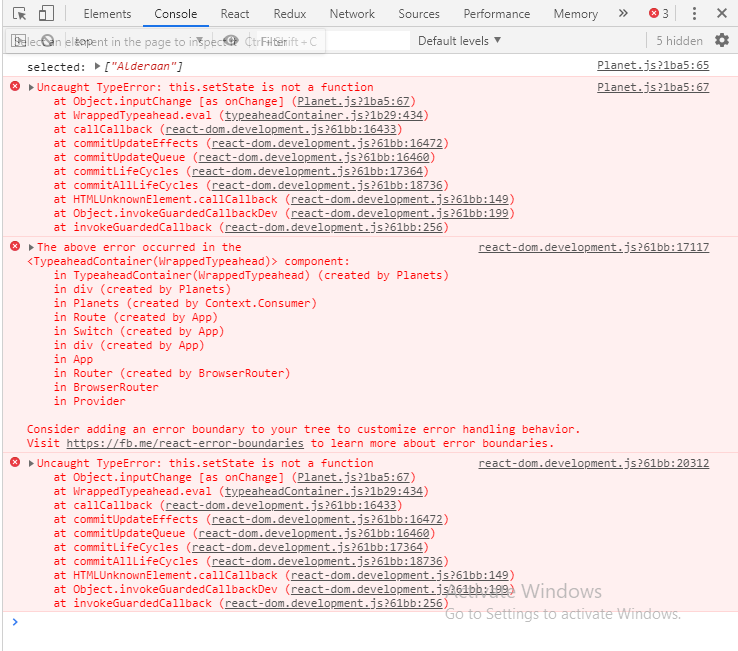
Expect even more in…

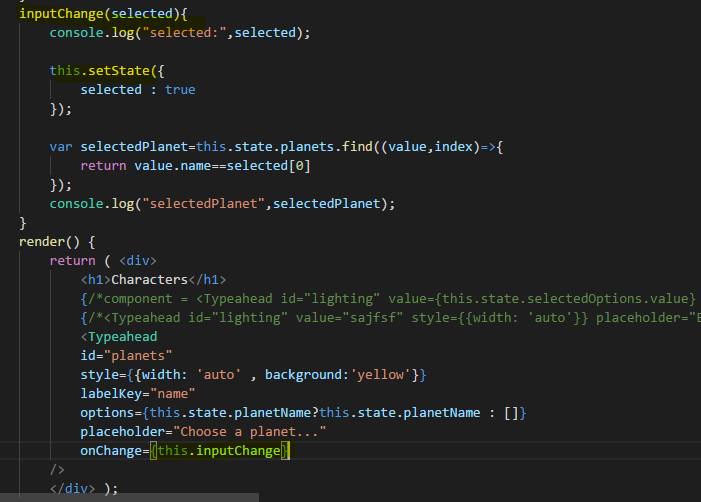
[**The Future of React**](https://github.com/reactjs/react-future/tree/master/07%20-%20Returning%20State)

For years now, the react team has been experimenting with how to best implement [stateful functions](https://github.com/reactjs/react-future/blob/master/07%20-%20Returning%20State/01%20-%20Stateful%20Functions.js).

Functional setState seems to be just the right answer to that (probably).

# Error : [React this.setState is not a function](https://stackoverflow.com/questions/31045716/react-this-setstate-is-not-a-function)



**Reason -** The callback is made in a different context. You need to bind to this in order to have access inside the callback: 

**Solution :**

**Use arrow function or bind the function with current context**

**OR**

