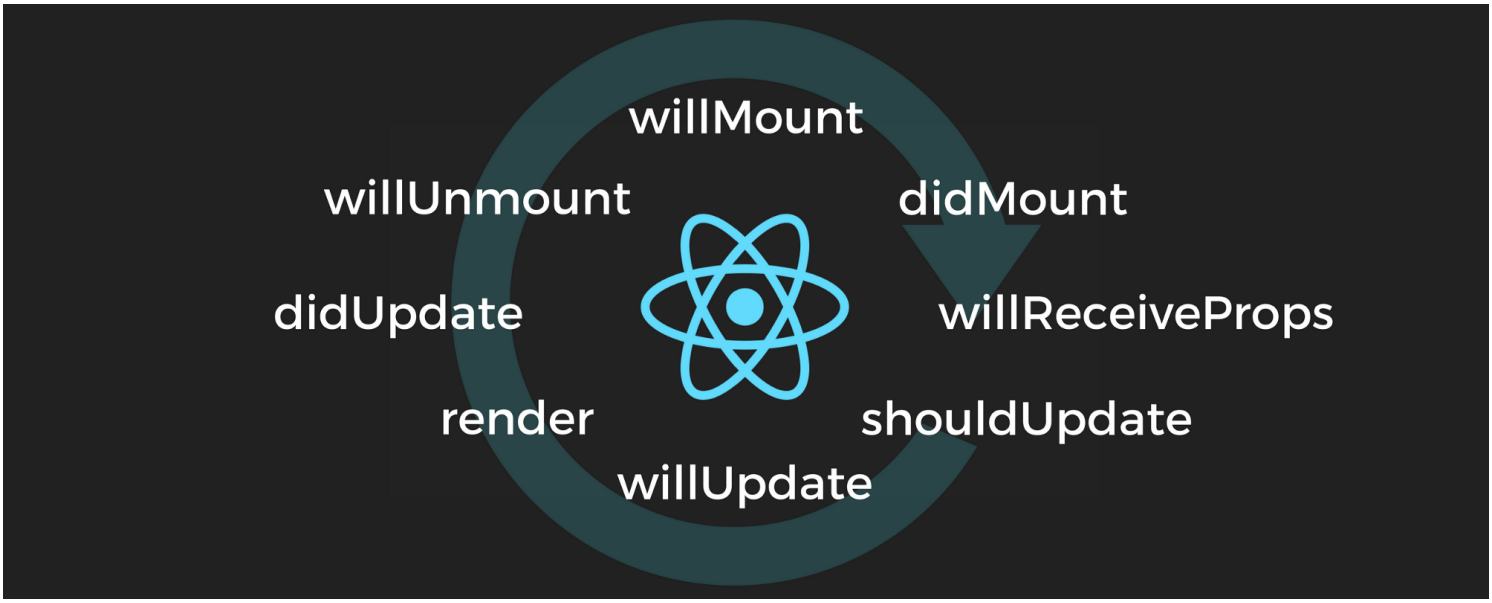
Scott Domes Follow
Lead Developer @ MuseFind.
Mar 28 · 7 min read

React Lifecycle Methods- how and when to use them



The above is the life of a React component, from birth (pre-mounting) and death (unmounting).

The beauty of React is the splitting of complicated UI's into little, bite-sized bits. Not only can we thus compartmentalize our app, we can also customize each compartment.

Through lifecycle methods, we can then control what happens when each tiny section of your UI renders, updates, thinks about re-rendering, and then disappears entirely.

Let's get started.

componentWillMount

Your component is going to appear on the screen very shortly. That chunky render function, with all its beautifully off-putting JSX, is about to be called. What do you want to do?

The answer is… probably not much. Sorry to start off slow, but componentWillMount is a bit of a dud.

The thing about componentWillMount is that there is no component to play with yet, so you can't do anything involving the DOM.

Also, nothing has changed since your component's constructor was called, which is where you should be setting up your component's default configuration anyway.

```
export default class Sidebar extends Component {
  tooltipsEnabled = true

  constructor(props) {
    super(props)
    this.state = {
      analyticsOpen: false,
      requirementsOpen: false,
      brandInfoOpen: false
    }
  }
```

Setting default state using a constructor.

Your component is in default position at this point. Almost everything should be taken care of by the rest of your component code, without the complication of an additional lifecycle method.

The exception is any setup that can only be done at runtime—namely, connecting to external API's. For example, if you use Firebase for your app, you'll need to get that set up as your app is first mounting.

But the key is that such configuration should be done at the highest level component of your app (the root component). That means 99% of your components should probably not use componentWillMount.

You may see people using componentWillMount to start AJAX calls to load data for your components. Don't do this. We'll get to that in the second.

Onto the next, much more useful method:

Most Common Use Case: App configuration in your root component**.**

Can call setState: Don't. Use default state instead.

componentDidMount

Now we're talking. Your component is out there, mounted and ready to be used. Now what?

Here is where you load in your data. I'll let Tyler McGinnis explain why:

> *You can't guarantee the AJAX request won't resolve before the component mounts. If it did, that would mean that you'd be trying to setState on an unmounted component, which not only won't work, but React will yell at you for. Doing AJAX in componentDidMount will guarantee that there's a component to update.*

You can read more of his answer here.

ComponentDidMount is also where you can do all the fun things you couldn't do when there was no component to play with. Here are some examples:

- draw on a <canvas> element that you just rendered
- initialize a masonry grid layout from a collection of elements
- add event listeners

Basically, here you want to do all the setup you couldn't do without a DOM, and start getting all the data you need.

Most Common Use Case: Starting AJAX calls to load in data for your component.

Can call setState: Yes.

componentWillReceiveProps

Our component was doing just fine, when all of a sudden a stream of new props arrive to mess things up.

Perhaps some data that was loaded in by a parent component's componentDidMount finally arrived, and is being passed down.

Before our component does anything with the new props, componentWillReceiveProps is called, with the next props as the argument.

```
componentWillReceiveProps(nextProps) {
  if (parseInt(nextProps.modelId, 10) !== parseInt(this.props.modelId, 10)) {
    this.setState({ postsLoaded: false })
    this.contentLoaded = 0
  }
}
```

We are now in a fun place, where we have access to both the next props (via nextProps), and our current props (via this.props).

Here's what we should do:

1. check which props will change (big caveat with componentWillReceiveProps—sometimes it's called when nothing has changed; React just wants to check in)
2. If the props will change in a way that is significant, act on it

Here's an example. Let's say, as we alluded to above, that we have a canvas element. Let's say we're drawing a nice circle graphic on there based on this.props.percent.

Well, that looks nice.

When we receive new props, IF the percent has changed, we want to redraw the grid. Here's the code:

```
componentWillReceiveProps(nextProps) {
  if(this.props.percent !== nextProps.percent) {
    this.setUpCircle(nextProps.percent)
  }
}
```

One more caveat—componentWillReceiveProps is not called on initial render. I mean technically the component is receiving props, but there aren't any old props to compare to, so… doesn't count.

Most Common Use Case: Acting on particular prop changes to trigger state transitions.

Can call setState: Yes.

shouldComponentUpdate

Now our component is getting nervous.

We have new props. Typical React dogma says that when a component receives new props, or new state, it should update.

But our component is a little bit anxious and is going to ask permission first.

Here's what we get—a shouldComponentUpdate method, called with nextProps as the first argument, and nextState is the second:

```
shouldComponentUpdate(nextProps, nextState) {
  return this.props.engagement !== nextProps.engagement
    || nextState.input !== this.state.input
}
```

shouldComponentUpdate should always return a boolean—an answer to the question, "should I re-render?" Yes, little component, you should. The default is that it always returns true.

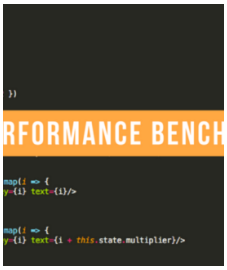
But if you're worried about wasted renders and other nonsense—shouldComponentUpdate is an awesome place to improve performance.

I wrote an article on using shouldComponentUpdate in this way—check it out:

How to Benchmark React Components: The Quick and Dirty Guide

A React Component works hard. As the user manipulates the state of the application, it may re-render 5, 10, 100 times...

engineering.musefind.com



In the article, we talk about having a table with many many fields. The problem was that when the table re-rendered, each field would also re-render, slowing things down.

ShouldComponentUpdate allows us to say: only update if the props you care about change.

But keep in mind that it can cause major problems if you set it and forget it, because your React component will not update normally. So use with caution.

Most Common Use Case: Controlling exactly when your component will re-render.

Can call setState: No.

componentWillUpdate

Wow, what a process. Now we've committed to updating. "Want me to do anything before I re-render?" our component asks. No, we say. Stop bothering us.

In the entire MuseFind codebase, we never use componentWillUpdate. Functionally, it's basically the same as componentWillReceiveProps, except you are not allowed to call this.setState.

If you were using shouldComponentUpdate AND needed to do something when props change, componentWillUpdate makes sense. But it's probably not going to give you a whole lot of additional utility.

Most Common Use Case: Used instead of componentWillReceiveProps on a component that also has shouldComponentUpdate (but no access to previous props).

Can call setState: No.

componentDidUpdate

Good job, little component.

Here we can do the same stuff we did in componentDidMount—reset our masonry layout, redraw our canvas, etc.

Wait- didn't we redraw our canvas in componentWillReceiveProps?

Yes, we did. Here's why: in componentDidUpdate, you don't know why it updated.

So if our component is receiving more props than those relevant to our canvas, we don't want to waste time redrawing the canvas every time it updates.

That doesn't mean componentDidUpdate isn't useful. To go back to our masonry layout example, we want to rearrange the grid after the DOM itself updates—so we use componentDidUpdate to do so.

```
componentDidUpdate() {
  this.createGrid()
}
```

Most Common Use Case: Updating the DOM in response to prop or state changes.

Can call setState: Yes.

componentWillUnmount

It's almost over.

Your component is going to go away. Maybe forever. It's very sad.

Before it goes, it asks if you have any last-minute requests.

Here you can cancel any outgoing network requests, or remove all event listeners associated with the component.

Basically, clean up anything to do that solely involves the component in question—when it's gone, it should be completely gone.

```
componentWillUnmount() {
  window.removeEventListener('resize', this.resizeListener)
}
```

Most Common Use Case: Cleaning up any leftover debris from your component.

Can call setState: No.

Conclusion

In an ideal world, we wouldn't use lifecycle methods. All our rendering issues would be controlled via state and props.

But it's not an ideal world, and sometimes you need to exact a little more control over how and when your component is updating.

Use these methods sparingly, and use them with care. I hope this article has been helpful in illuminating when and how to use lifecycle methods.

If this was any help to you, please recommend it to others by hitting the heart below, and follow me on Medium and Twitter for more React articles and tips.

And, as always, let me know in the comments if you agree/disagree with this approach!