Lista and Hooks

A React component undergoes three phases in its lifecycle: mounting, updating, and unmounting.

1. The *mounting phase* is when a new component is created and inserted into the DOM or, in other words, when the life of a component begins. This can only happen once, and is often called "initial render."

2. The *updating phase* is when the component updates or re-renders. This reaction is triggered when the props are updated or when the state is updated. This phase can occur multiple times, which is kind of the point of React.

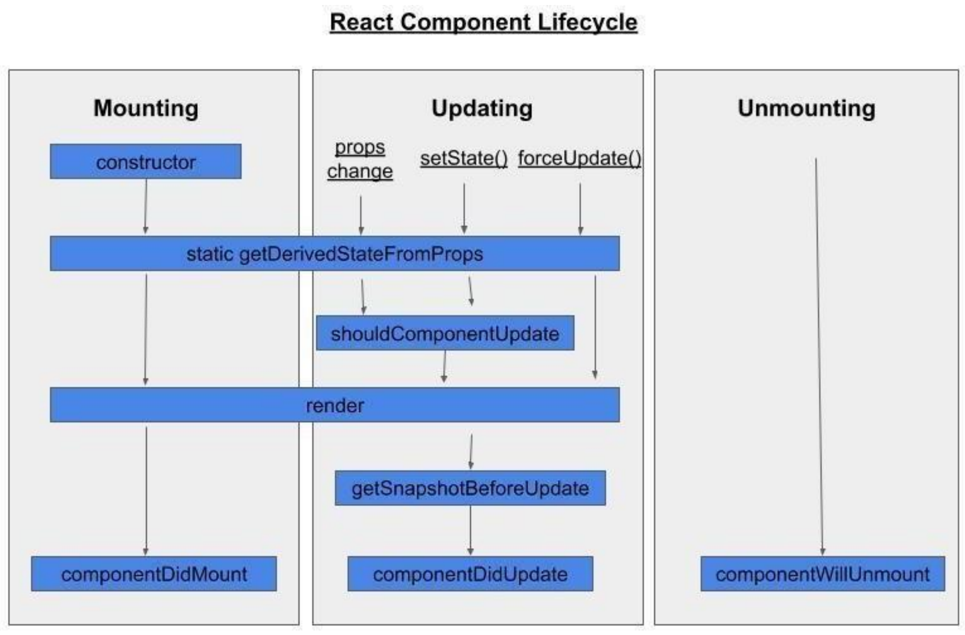
3. The last phase within a component's lifecycle is the *unmounting phase*, when the component is removed from the DOM.

In a class-based component, you can call different methods for each phase of the lifecycle (more on this below). These lifecycle methods are of course not applicable to functional components because they can only be written/contained within a class. However, React hooks give functional components the ability to use states.

Hooks have gaining popularity because they make working with React cleaner and often less verbose.

React lifecycle methods

Let's learn more about the methods that make up each of our three phases.



The mounting phase

In the mounting phase, a component is prepared for and actually inserted into the DOM. To get through this phase, four lifecycle methods are called: constructor, static getDerivedStateFromProps, render, and componentDidMount.

The constructor method

The constructor method is the very first method called during the mounting phase.

It's important to remember that you shouldn't add any side effects within this method (like sending an HTTP request) as it's intended to be pure.

This method is mostly used for initializing the state of the component and binding event-handler methods within the component. The constructor method is not necessarily required. If you don't intend to make your component stateful (or if that state doesn’t need to be initialized) or bind any method, then it’s not necessary to implement.

The constructor method is called when the component is initiated, but before it’s rendered. It’s also called with props as an argument. It's important you call the super(props) function with the props argument passed onto it within the constructor before any other steps are taken.

This will then initiate the constructor of React.Component (the parent of this class-based component) and it inherits the constructor method and other methods of a typical React component.

In the Counter component, you can see the component's state is initialized within the constructor method to keep track of the count state. The setCount method, which is an eventhandler attached to your button in this case, is bound within the constructor.

The static getDerivedStateFromProps method

Props and state are completely different concepts, and part of building your app intelligently is deciding which data goes where.

In many cases though, your component’s state will be *derivative* of its props. This is where the static getDerivedStateFromProps method comes in. This method allows you to modify the state value with any props value. It's most useful for changes in props over time, and we’ll learn later that it’s also useful in the update phase.

The method static getDerivedStateFromProps accepts two arguments: props and state, and returns an object, or null if no change is needed. These values are passed directly to the method, so there’s no need for it to have access to the instance of the class (or any other part of the class) and thus is considered a static method.

In the mounting phase, the getDerivedStateFromProps method is called after the constructor method and right before the component is rendered. This is the most rarely used method in this phase, but it’s still important to know so you can use it if needed.

Another way of looking at it is just providing a default for your component state based on props.

The render method

The render method is the only required method for a class-based React component. It’s called after the getDerivedStateFromProps method and actually renders or inserts the HTML to the DOM.

Typically, the render method returns the JSX which will eventually be rendered, but it can also return other values.

It's important to remember that the render method is meant to be pure. This means you can’t modify the state, have any direct interaction with the browser, or any other kind of side effect like sending an HTTP request. Just think of it as writing HTML, but of course as JSX.

The componentDidMount method

componentDidMount is the last lifecycle method called in the mounting phase. It’s called right after the component is rendered or mounted to the DOM.

With this method, you're allowed to add side effects like sending network requests or updating the component's state. Additionally, the **componentDidMount** method allows you to make subscriptions like subscribing to the Redux store. You can also call the this.setState method right away; however this will cause a re-render as it kicks in the update phase, since the state has changed.

You need to be careful with **componentDidMount** because it may cause unnecessary re-renders.

A common use case is waiting until a component renders to start an animation, or fetching data from an external source.

The updating phase

The Updating phase is triggered when component props or state change, and consists of the following methods: static
getDerivedFromProps, shouldComponentUpdate, render, getSnapshotBeforeUpdate, and **componentDidUpdate.**

The methods getDerivedFromProps and render are also part of the mounting phase. Since they’ve been covered previously, this section focuses on the other three methods.

The static getDerivedStateFromProps

In the update phase, the first lifecycle method called is **getDerivedStateFromProps**. This method is useful if you have updated props and you want to reflect that in the component's state.

For instance, a component’s state may depend on the value of its props.
With **getDerivedStateFromProps**, before the component was even re-rendered, its state can reflect those changes and can be shown (if applicable) to the newly updated component.

Remember, this method is rarely used and isn’t ideal for most situations.

The shouldComponentUpdate method

shouldComponentUpdate is another rarely used lifecycle method. It’s specifically intended for performance optimization, and basically lets you tell React when you *don't* need to re-render when a new state or props comes in. While it can help avoid re-renders, you shouldn’t rely on it to *prevent* re-renders since you might skip a necessary update and encounter bugs.

To prevent renders, you can opt in to logical rendering instead, or use a **PureComponent** which is recommended by React.

This method can accept nextProps and nextState as arguments, however, they’re optional, and you can declare it without the arguments. This method then returns a Boolean value. The Boolean value defines whether a re-render happens. The default value is true, where re-render happens in all cases whenever state or props changes.

Note that the shouldComponentUpdate method is ignored when forceUpdate() is invoked.

The getSnapshotBeforeUpdate method

The **getSnapshotBeforeUpdate method** gives you access to the previous props and state of the component before it's updated. This allows you to work or check on the previous values of the state or props. It's another method that's rarely used.

A good use case for this method is handling scroll positions in a chat app. When a new message comes in as the user is viewing old messages, it shouldn't push the old ones out of view.

getSnapshotBeforeUpdate is called after the render method, and before **componentDidUpdate**. If the getSnapshotBeforeUpdate method returns anything, it will be passed as a parameter for the **componentDidUpdate** method:

The componentDidUpdate Method

The **componentDidUpdate** method is the last lifecycle method invoked in the update phase. It allows you to create side effects like sending network requests or calling the this.setState method. It's important to remember that there should always be a way to avoid the setState (like some sort of logic), or it will result in an infinite loop of re-rendering.

This method can accept up to three parameters: prevProps, prevState, and snapshot (if you implement the getSnapshotBeforeUpdate method). **The unmounting phase**

The unmounting phase is the third and final phase of a React component. At this phase, the component is removed from the DOM. Unmounting only has one lifecycle method involved: **componentWillUnmount**.

The componentWillUnmount Method

componentWillUnmount is invoked right before the component is unmounted or removed from the DOM. It's meant for any necessary clean up of the component, like unsubscribing to any subscriptions (i.e., Redux) or canceling any network requests. Once this method is done executing, the component will be destroyed.

React Hooks and the component lifecycle

Versions of React before 16.8 consider two kinds of components based on statefulness: the classbased stateful component, and the stateless functional components (often referred to as a "dumb component"). But with the release of React 16.8, Hooks were introduced and empowered developers to access state from functional components, instead of writing an entire class. With this change, building components became easier and less verbose.

Hooks known as default hooks come with React, and you're also able to create your own custom hook. A custom hook is just a function that starts with use, like useStore, or useWhatever.

The two most common default hooks are useState and useEffect. The useState hook gives state to the functional component, and useEffect allows you to add side effects within it (like after initial render), which aren’t allowed within the function's main body. You can also act upon updates on the state with useEffect.

React has released more default hooks, but useState and useEffect are the ones you should be most familiar with. Let’s take a look at how they work and compare them to the component lifecycle we covered above.

useState

The useState hook is used to store state for a functional component. This hook accepts one parameter: initialState, which will be set as the initial stateful value, and returns two values: the stateful value, and the update function to update the stateful value. The update function accepts one argument, newState, which replaces the existing stateful value.

Converting this to a functional component with useState eliminates a lot of code and makes things cleaner and shorter. Here's what the above component looks like with the useState hook. If you’ve read a React tutorial over the past 2 years, chances are you’ve seen some syntax like this.

With useState(), whatever you put in the parenthesis is the default state.

The simplicity and clarity of these functional components with Hooks popularized their use among developers who prefer using functional components rather than traditional, class-based ones.

But what if your class-based component’s state object has plenty of items like this:

You might be tempted at first to use one useState hook, and build some sort of dictionary to hold all of this state. But React allows you to have multiple useState hooks within a functional component. So instead of one single object, you can set each item of the state object into its own state:

With a lot of different “types” of state this can obviously get verbose, so exercise judgment for larger components.

useEffect

As with the render() method of a class-based component, the main body of a functional component should be kept pure. With the **useEffect hook**, you're able to create side effects while maintaining the component's purity. Within this hook, you can send network requests, make subscriptions, and change the state value.

The **useEffect** hook accepts a function as an argument, where you write all your side effects. This function is invoked after every browser paint and before any new renders (this will depend on the dependency array, which is explained in the next paragraph). This function can return another function called the clean-up function, which can be used to clean up the side effects (i.e. when the component is destroyed) like unsubscribing to a store. It’s kind of a mash up of several of the methods explained in the previous section.

This Hook accepts a second argument called the dependency array, which is an array of dependencies like state or props values, which the **useEffect** uses as reference to only run when these values change. If the dependency array is empty, then the **useEffect** will only run once, after the first paint.

The dependency array is optional, so if it's not defined, **useEffect** will fire first when the component is first mounted, and then on every re-render.

This component's **useEffect** will only run once, since the dependency array is empty. Within the input function, the subscription to the Redux store is invoked, which returns an unsubscribe function. This unsubscribe function is returned, which serves as the clean-up function.

You might have noticed that the **useEffect** from the above component has similarities to **componentDidMount.** It holds the side effects, and only runs once, when the component is mounted. The only difference is that it's invoked after the first browser paint, whereas **componentDidMount** doesn't wait for that. It's also important to note that the clean-up function can be compared to the **componentWillUnmount**, as this function is invoked when the component was destroyed. So again, **useEffect()** is a sort of hybrid in this sense.

The **useEffect** hook works similarly to the three lifecycle methods: **componentDidMount, componentDidUpdate, and componentWillUnmount.** The **componentDidMount** and **componentWillUnmount** were discussed above, but what about **componentDidUpdate?**

If you add dependencies in the dependency array, the function passed into the **useEffect** hook will run every time the dependencies, like a piece of stateful data, changes. This behavior of the **useEffect** hook is comparable to the **componentDidUpdate** method since it's invoked on every state/props change. The difference is that you can specifically choose what state/props you want **useEffect** to depend on, rather than the **componentDidUpdate** which acts upon every state or props change.