Stateful vs. Stateless Functional Components in React

by Manjunath M 31 Oct 2017

Difficulty: Beginner Length: Long Languages: English

React Web Development

React is a popular JavaScript front-end library for building interactive user interfaces. React has a comparatively shallow learning curve, which is one of the reasons why it's getting all the attention lately.

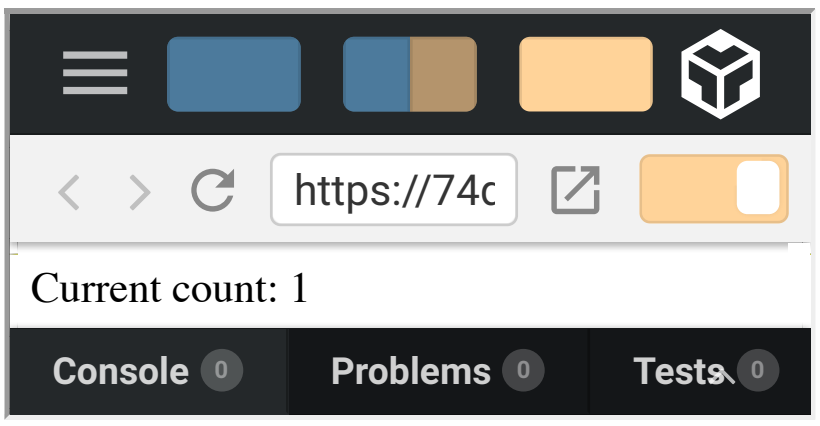
Although there many important concepts to be covered, components are undeniably the heart and soul of React. Having a good understanding of components should make your life easy as a React developer.

Prerequisites

This tutorial is intended for beginners who have started learning React and need a better overview of components. We will start with component basics and then move on to more challenging concepts such as component patterns and when to use those patterns. Different component classifications have been covered such as class

vs. functional components, stateful vs. stateless components, and container vs. presentational components.

Before getting started, I want to introduce you to the code snippet that we will be using in this tutorial. It is a simple counter built with React. I will be referring back to some parts of this example throughout the tutorial.



So let's get started.

What Are Components?

Components are self-sustaining, independent micro-entities that describe a part of your UI. An application's UI can be split up into smaller components where each component has its own code, structure, and API.

Facebook, for instance, has thousands of pieces of functionality interfaced together when you view their web application. Here is an interesting fact: Facebook comprises 30,000 components, and the number is growing. The component architecture allows you to think of each piece in isolation. Each component can update everything in its scope without being concerned about how it affects other components.

If we take Facebook's UI as an example, the search bar would be a good candidate for a component. Facebook's Newsfeed would make another component (or a component that hosts many sub-components). All the methods and AJAX calls concerned with the search bar would be within that component.

Components are also reusable. If you need the same component in multiple places,

that's easy. With the help of JSX syntax, you can declare your components wherever you want them to appear, and that's it.

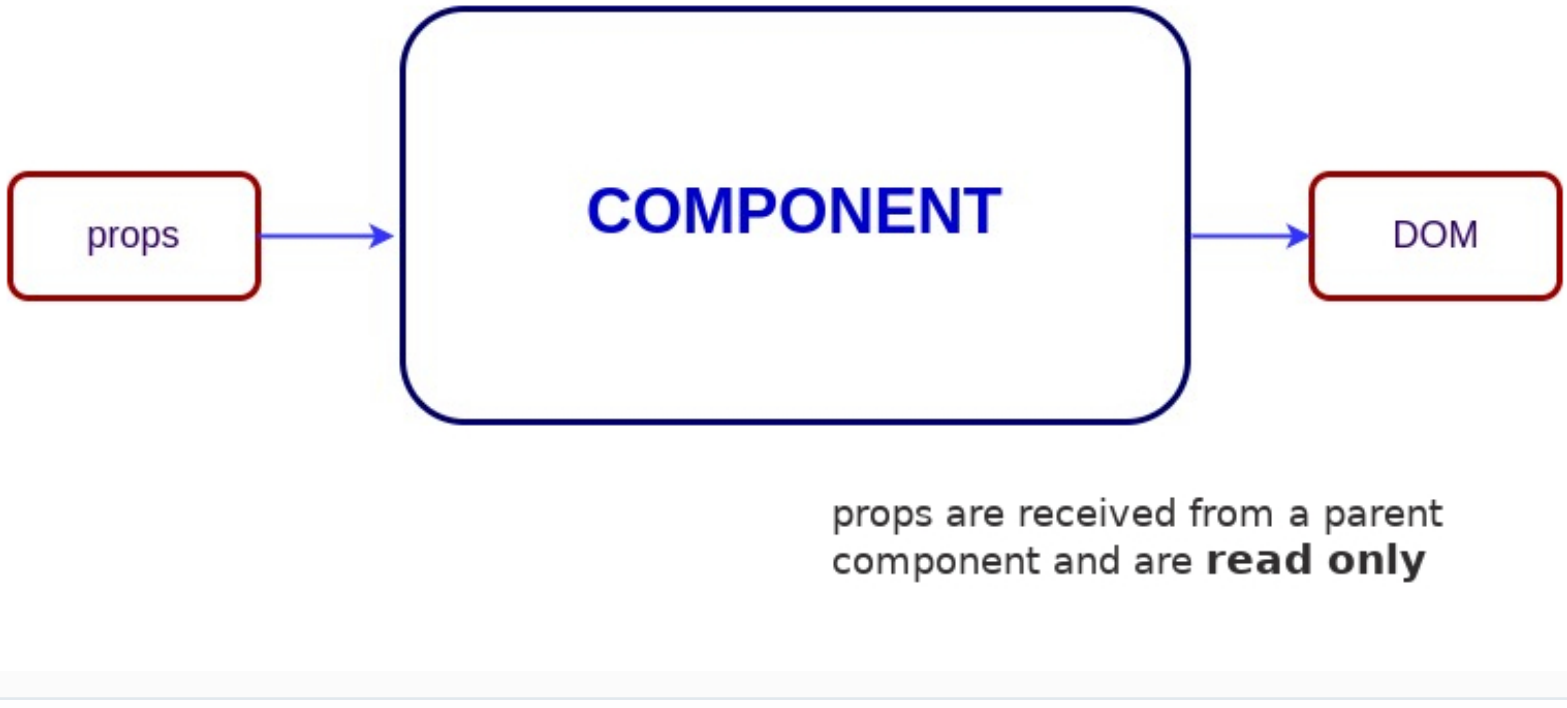
```
<div>
    Current count: {this.state.count}
    <hr />
    {/* Component reusability in action. */ }
    <Button sign = "+" count={this.state.count}
        updateCount = {this.handleCount.bind(this) }/>
    <Button sign = "-" count={this.state.count}
        updateCount = {this.handleCount.bind(this) }/>
</div>
```

Props and State

Components need data to work with. There are two different ways that you can combine components and data: either as **props** or **state**. props and state determine what a component renders and how it behaves. Let's start with props.

props

If components were plain JavaScript functions, then props would be the function input. Going by that analogy, a component accepts an input (what we call props), processes it, and then renders some JSX code.

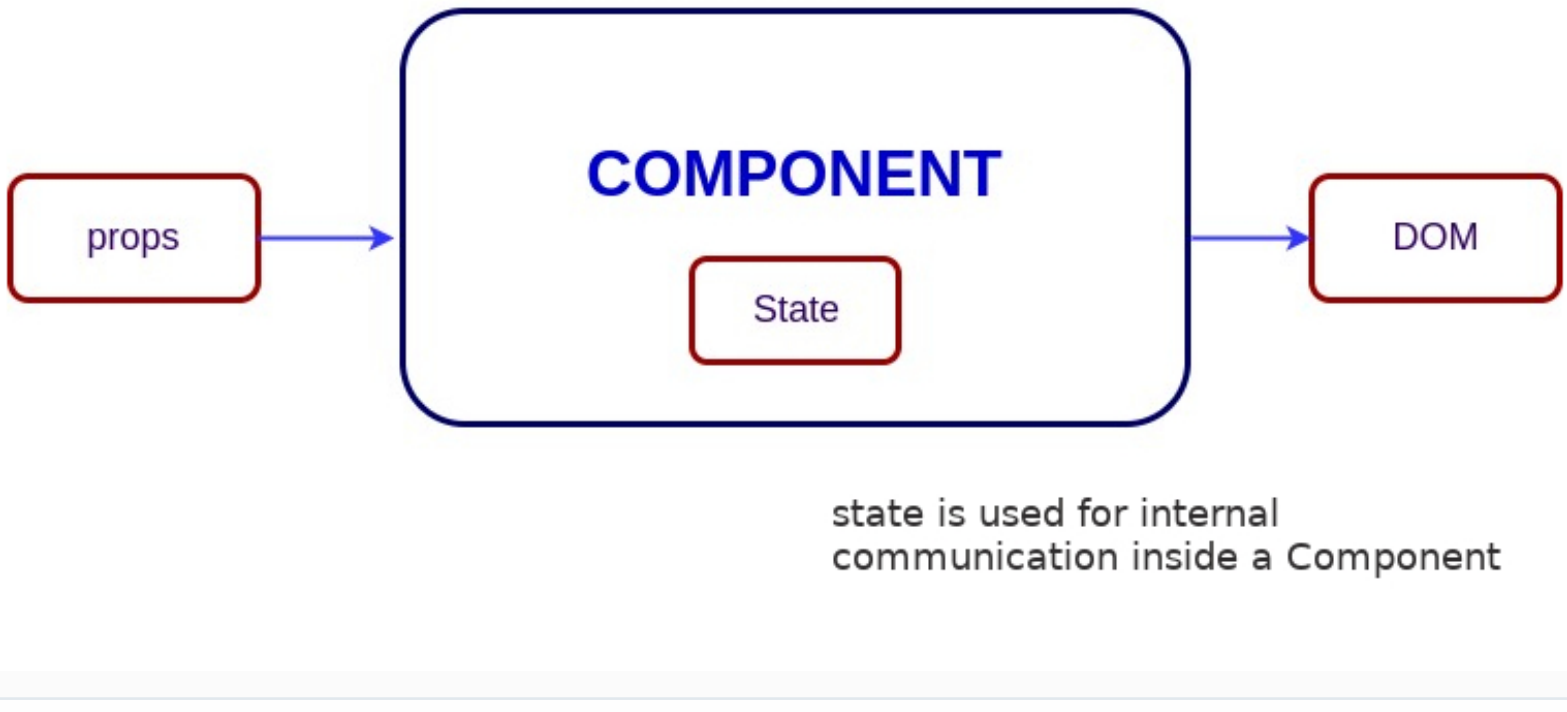


Although the data in props is accessible to a component, React philosophy is that props should be immutable and top-down. What this means is that a parent component can pass on whatever data it wants to its children as props, but the child component cannot modify its props. So, if you try to edit the props as I did below, you will get the "Cannot assign to read-only" TypeError.

```
const Button = (props) => {
    // props are read only
    props.count =21;
.
.
}
```

State

State, on the other hand, is an object that is owned by the component where it is declared. Its scope is limited to the current component. A component can initialize its state and update it whenever necessary. The state of the parent component usually ends up being props of the child component. When the state is passed out of the current scope, we refer to it as a prop.



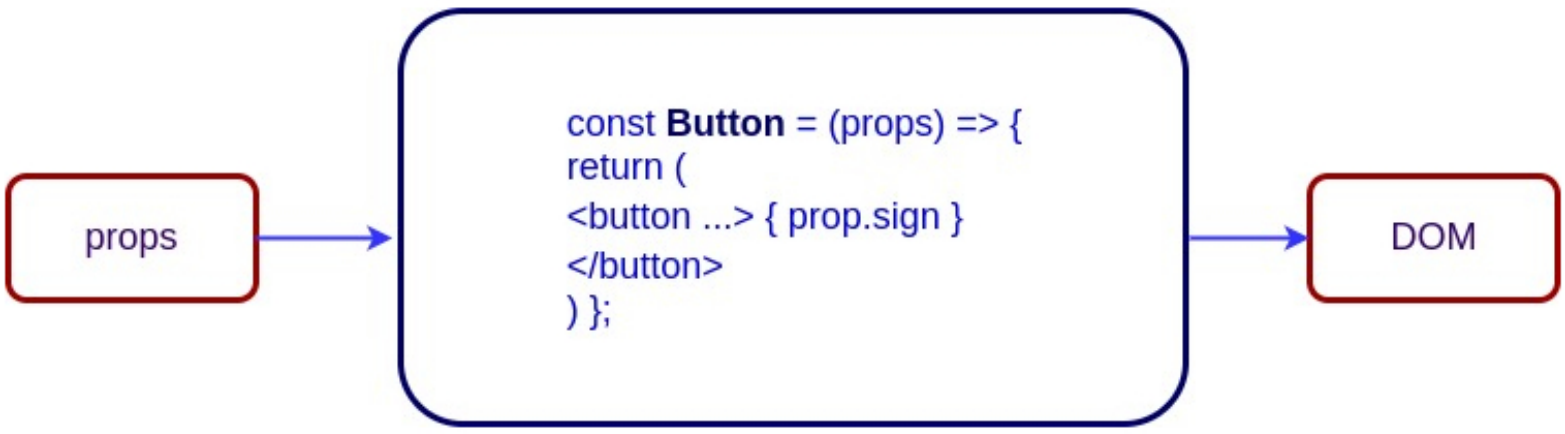
Now that we know the component basics, let's have a look at the basic classification of components.

Class Components vs. Functional Components

A React component can be of two types: either a class component or a functional component. The difference between the two is evident from their names.

Functional Components

Functional components are just JavaScript functions. They take in an optional input which, as I've mentioned earlier, is what we call props.

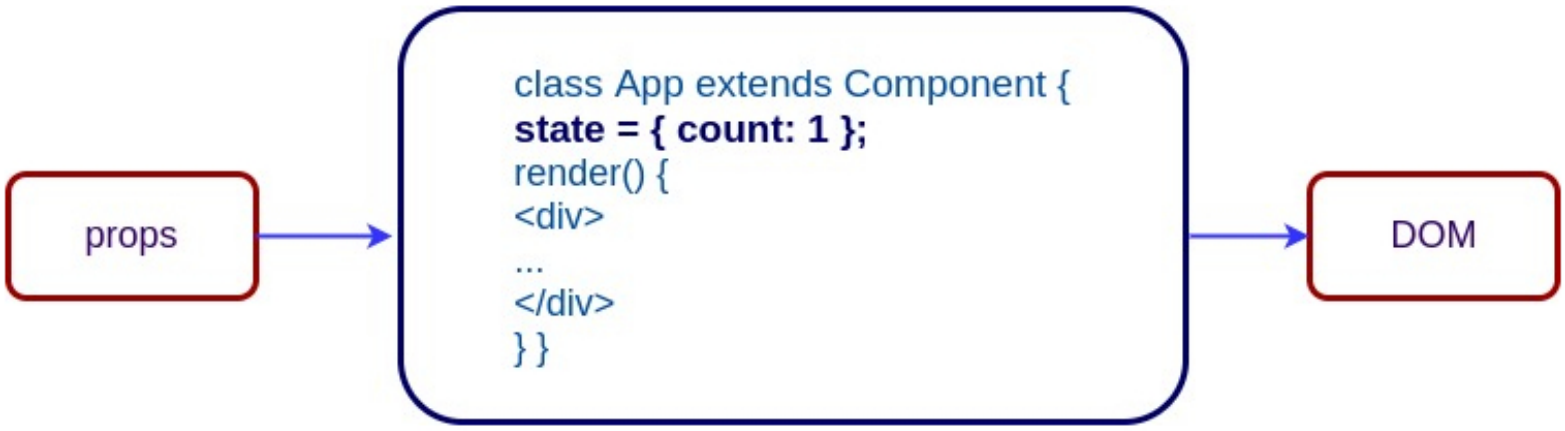


Some developers prefer to use the new ES6 arrow functions for defining components. Arrow functions are more compact and offer a concise syntax for writing function expressions. By using an arrow function, we can skip the use of two keywords, `function` and `return`, and a pair of curly brackets. With the new syntax, you can define a component in a single line like this.

```
1  const Hello = ({ name }) => (<div>Hello, {name}!</div>);
```

Class Components

Class components offer more features, and with more features comes more baggage. The primary reason to choose class components over functional components is that they can have `state`.



The state = {count: 1} syntax is part of the public class fields feature. More on this below.

There are two ways that you can create a class component. The traditional way is to use `React.createClass()`. ES6 introduced a syntax sugar that allows you to write classes that extend `React.Component`. However, both the methods are meant to do the same thing.

Class components can exist without state too. Here is an example of a class component that accepts an input props and renders JSX.

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

    render() {
        return(
            <div>
                Hello {props}
            </div>
        )
    }
}
```

We define a constructor method that accepts props as input. Inside the constructor,

we call **super()** to pass down whatever is being inherited from the parent class. Here are a few details that you might have missed.

First, the constructor is optional while defining a component. In the above case, the component doesn't have a state, and the constructor doesn't appear to do anything useful. `this.props` used inside the `render()` will work regardless of whether the constructor is defined or not. However, here's something from the official docs:

> *Class components should always call the base constructor with `props`.*

As a best practice, I will recommend using the constructor for all class components.

Secondly, if you're using a constructor, you need to call `super()`. This is not optional, and you will get the syntax error "*Missing super() call in constructor*" otherwise.

And my last point is about the use of `super()` vs. `super(props)`. `super(props)` should be used if you're going to call `this.props` inside the constructor. Otherwise, using `super()` alone is sufficient.

Stateful Components vs. Stateless Components

This is another popular way of classifying components. And the criteria for the classification is simple: the components that have state and the components that don't.

Stateful Components

Stateful components are always class components. As previously mentioned, stateful components have a state that gets initialized in the constructor.

```
// Here is an excerpt from the counter example
constructor(props) {
  super(props);
  this.state = { count: 0 };
}
```

We've created a state object and initialized it with a count of 0. There is an alternative syntax proposed to make this easier called class fields. It's not a part of the ECMAScript specification yet, but If you're using a Babel transpiler, this syntax should work out of the box.

```
class App extends Component {

  /*
  // Not required anymore
  constructor() {
      super();
      this.state = {
        count: 1
      }
  }
  */

  state = { count: 1 };

  handleCount(value) {
      this.setState((prevState) => ({count: prevState.count+value}));
  }

  render() {
    // omitted for brevity
  }

}
```

You can avoid using the constructor altogether with this new syntax.

We can now access the state within the class methods including `render()`. If you're going to use them inside `render()` to display the value of the current count, you need to place it inside curly brackets as follows:

```
render() {
return (
    Current count: {this.state.count}
    )
}
```

The `this` keyword here refers to the instance of the current component.

Initializing the state is not enough—we need to be able to update the state in order to create an interactive application. If you thought this would work, no, it won't.

```
//Wrong way

handleCount(value) {
    this.state.count = this.state.count +value;
}
```

React components are equipped with a method called setState for updating the state. setState accepts an object that contains the new state of the `count`.

```
// This works

handleCount(value) {
    this.setState({count: this.state.count+ value});
}
```

The `setState()` accepts an object as an input, and we increment the previous value of count by 1, which works as expected. However, there is a catch. When there are multiple setState calls that read a previous value of the state and write a new value into it, we might end up with a race condition. What that means is that the final results won't match up with the expected values.

Here is an example that should make it clear for you. Try this in the codesandbox snippet above.

```
// What is the expected output? Try it in the code sandbox.
handleCount(value) {
    this.setState({count: this.state.count+100});
    this.setState({count: this.state.count+value});
    this.setState({count: this.state.count-100});
}
```

We want the setState to increment the count by 100, then update it by 1, and then remove that 100 that was added earlier. If setState performs the state transition in the actual order, we will get the expected behavior. However, setState is asynchronous, and multiple setState calls might be batched together for better UI experience and performance. So the above code yields a behavior which is different

from what we expect.

Therefore, instead of directly passing an object, you can pass in an updater function that has the signature:

```
(prevState, props) => stateChange
```

prevState is a reference to the previous state and is guaranteed to be up to date. props refers to the component's props, and we don't need props to update the state here, so we can ignore that. Hence, we can use it for updating state and avoid the race condition.

```
// The right way

handleCount(value) {

  this.setState((prevState) => {
    count: prevState.count +1
  });
}
```

The `setState()` rerenders the component, and you have a working stateful component.

Stateless Components

You can use either a function or a class for creating stateless components. But unless you need to use a lifecycle hook in your components, you should go for stateless functional components. There are a lot of benefits if you decide to use stateless functional components here; they are easy to write, understand, and test, and you can avoid the `this` keyword altogether. However, as of React v16, there are no performance benefits from using stateless functional components over class components.

The downside is that you can't have lifecycle hooks. The lifecycle method `ShouldComponentUpdate()` is often used to optimize performance and to manually control what gets rerendered. You can't use that with functional components yet. Refs are also not supported.

Container Components vs. Presentational Components

This is another pattern that is very useful while writing components. The benefit of this approach is that the behavior logic is separated from the presentational logic.

Presentational Components

Presentational components are coupled with the view or how things look. These components accept props from their container counterpart and render them. Everything that has to do with describing the UI should go here.

Presentational components are reusable and should stay decoupled from the behavioral layer. A presentational component receives the data and callbacks exclusively via props and when an event occurs, like a button being pressed, it performs a callback to the container component via props to invoke an event handling method.

Functional components should be your first choice for writing presentational components unless a state is required. If a presentational component requires a state, it should be concerned with the UI state and not actual data. The presentational component doesn't interact with the Redux store or make API calls.

Container Components

Container components will deal with the behavioral part. A container component tells the presentational component what should be rendered using props. It shouldn't contain limited DOM markups and styles. If you're using Redux, a container component contains the code that dispatches an action to a store. Alternatively, this is the place where you should place your API calls and store the result into the component's state.

The usual structure is that there is a container component at the top that passes down the data to its child presentational components as props. This works for smaller projects; however, when the project gets bigger and you have a lot of intermediate components that just accept props and pass them on to child

components, this will get nasty and hard to maintain. When this happens, it's better to create a container component unique to the leaf component, and this will ease the burden on the intermediate components.

So What Is a PureComponent?

You will get to hear the term pure component very often in React circles, and then there is `React.PureComponent`. When you're new to React, all this might sound a bit confusing. A component is said to be pure if it is guaranteed to return the same result given the same props and state. A functional component is a good example of a pure component because, given an input, you know what will be rendered.

```
const HelloWorld = ({name}) => (
 <div>{`Hi ${name}`}</div>
);
```

Class components can be pure too as long as their props and state are immutable. If you have a component with a 'deep' immutable set of props and state, React API has something called `PureComponent`. `React.PureComponent` is similar to `React.Component`, but it implements the `ShouldComponentUpdate()` method a bit differently. `ShouldComponentUpdate()` is invoked before something is rerendered. The default behaviour is that it returns true so that any change to the state or the props rerenders the component.

```
shouldComponentUpdate(nextProps, nextState) {
  return true;
}
```

However, with PureComponent, it performs a shallow comparison of objects. Shallow comparison means that you compare the immediate contents of the objects instead of recursively comparing all the key/value pairs of the object. So only the object references are compared, and if the state/props are mutated, this might not work as intended.

`React.PureComponent` is used for optimizing performance, and there is no reason why you should consider using it unless you encounter some sort of performance issue.

Final Thoughts

Stateless functional components are more elegant and usually are a good choice for building the presentational components. Because they are just functions, you won't have a hard time writing and understanding them, and moreover, they are dead easy to test.

It should be noted that stateless functional components don't have the upper hand in terms of optimization and performance because they don't have a `ShouldComponentUpdate()` hook. This might change in future versions of React, where functional components might be optimized for better performance. However, if you're not critical of the performance, you should stick to functional components for the view/presentation and stateful class components for the container.

Hopefully, this tutorial has given you a high-level overview of the component-based architecture and different component patterns in React. What are your thoughts on this? Share them through the comments.

Over the last couple of years, React has grown in popularity. In fact, we have a number of items in Envato Market that are available for purchase, review, implementation, and so on. If you’re looking for additional resources around React,

don’t hesitate to check them out.

Manjunath M

Kerala, India

Full-stack JavaScript developer. Rubyist by passion. Amateur musician. Favorite quote? "Being a jack of all trades doesn't mean you're a master at none." Follow me on GitHub for new projects and tutorials.

blizzerand

Subscribe below and we'll send you a weekly email summary of all new Code tutorials. Never miss out on learning about the next big thing.

Email Address

Update me weekly

Translations

Envato Tuts+ tutorials are translated into other languages by our community members—you can be involved too!

Translate this post

Powered by native

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Ed Mann • 3 months ago

Thanks! I've learned a ton about react with this article.

1 • Reply • Share ›

Jade Hayes • 3 months ago

I've been trying to learn react for a few days now and your article really made it clear to understand the basics and helped answer a lot of my questions. I'm using react for a project, do you have recommendations on where to learn more about fetch? Thank you!

1 • Reply • Share ›

Michael H • 7 months ago

Hey,
How do you get your following conclusion? How do you prove it? any benchmarks?
However, as of React v16, there are no performance benefits from using stateless functional components over class components.

1 • Reply • Share ›

Manjunath M ➚ Michael H • 7 months ago

Hi Michael, I haven't personally made any benchmark on this, but if you're interested, I will :) To start with, here's something from a two-year old official blog post:

> This pattern is designed to encourage the creation of these simple components that should comprise large portions of your apps. In the future, we'll also be able to make performance optimizations specific to these components by avoiding unnecessary checks and memory allocations.

This is from Dan Abramov last year.

And here is a good benchmark if you're interested.

see more

• Reply • Share ›

Matthew Browne ➚ Manjunath M • 3 months ago

This article is incorrect about React 16. Functional components were made faster than class components in version 16 - see https://github.com/reactjs/.... However, this is a micro-optimization that still pales in comparison with the performance gains you can get from using PureComponent (or your own efficient shouldComponentUpdate() logic). But of course PureComponent has to be used carefully - with immutable data objects - to make sure components still re-render when they should.

1 ^ | v • Reply • Share ›

Manjunath M ➚ Matthew Browne • 3 months ago

Thanks for the update. The docs need to be revised as you suggested on GIthub.

^ | v • Reply • Share ›

Matthew Browne ➚ Manjunath M • 3 months ago

Any plans to update the article? It could be misleading to people using React 16.

4 ^ | v • Reply • Share ›

Michael H ➚ Manjunath M • 7 months ago

functional components do not contains any states and do not need to handle them. In theory, they will be faster, it can be minor though. Say if you use redux to handle state, you do get performance benefits using functional components vs stateful components. According to this article, it can be as minor as 6%: https://medium.com/missive-..., but if facebook optimise it later or if you write the components the way shown in the article. You will get much faster app.

1 ^ | v • Reply • Share ›

Manjunath M ➚ Michael H • 7 months ago

The article cited above removes the *concept of Components* from the picture and instead calls the functions directly. That's pure JavaScript and not React way of doing it, I'd say. But indeed a good experiment.

However, I agree, the developers might optimize it in the future. I personally use stateless components wherever possible because of the ease of using them. For the apps I am working on right now, any difference in performance is negligible. If there is a constraint on performance later, I might shift to PureComponents. Just my thoughts.

see more

Reply • Share ›

Michael H ➚ Manjunath M • 7 months ago

no man, benchmarks are done for component, functional component and pure JavaScript:
with 6% and 45% performance benefits respectively for functional component and pure JavaScript function.

1 Reply • Share ›

Manjunath M ➚ Michael H • 7 months ago

I misread that part. 6% is actually good. I am just too curious to create my own benchmarks and see how they go. If I do, I will post them here. :)

Reply • Share ›

Michael H ➚ Manjunath M • 7 months ago

cool man.

Reply • Share ›

John • 2 months ago

https://medium.com/groww-en...

Reply • Share ›

ENVATO TUTS+

JOIN OUR COMMUNITY

HELP

envatotuts+

26,020 Tutorials

1,139 Courses

23,625 Translations

Follow Envato Tuts+