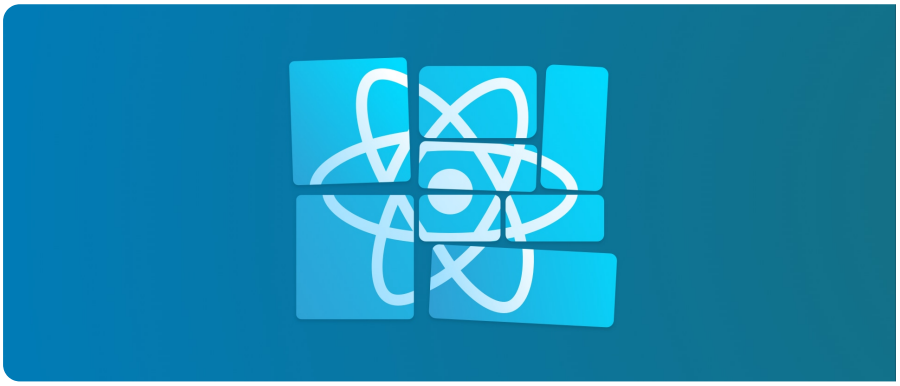
Strategies for Writing Maintainable Components in React



David Leger

Sep 10, 2019

In this article I'll discuss when and how React components should be broken down into smaller pieces.

Reduce boilerplate with functional components

In a typical React codebase, you don't have to look far to find monster components. These should be broken down into smaller pieces, but it's unclear how that affects readability for class components. The extra boilerplate can often be distracting.

Using functional components instead of class components reduces the amount of boilerplate needed to create a new component.

Class component (6 lines)

```
import React, { Component } from 'react';

export default class MyComponent extends Component {
  render() {
    return <h1>Hello!</h1>;
  }
}
```

Functional component (2 lines)

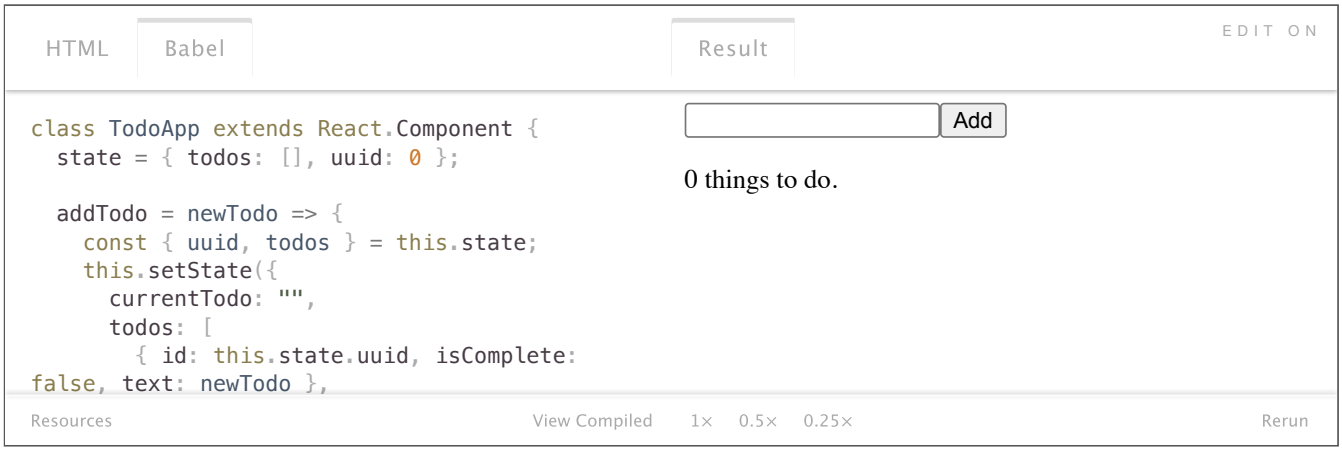
```
import React from 'react';

export default const MyComponent = () =><h1>Hello!</h1>;
```

This is a somewhat minor detail, but having less boilerplate for creating components means that it's easier to create many smaller components. A smaller component is typically easier to read and maintain because its purpose is more concise.

Sub-render methods

A common approach to creating maintainable components is to break down the render method into several sub-render methods. This approach is decent. Splitting up the main UI pieces into sub-render methods keeps things clean, organized, and readable. However, this approach still has its flaws: a sub-render method is still in scope of the class and is at risk of being tightly coupled to it.

A simple todo app that uses sub-render methods



If at some point we decide to pull the sub-render methods out into their own component, it would be tough to do so because they need to be in the scope of TodoApp's state and methods. The easiest way to ensure sub-render methods are entirely decoupled from TodoApp is to make them separate components.

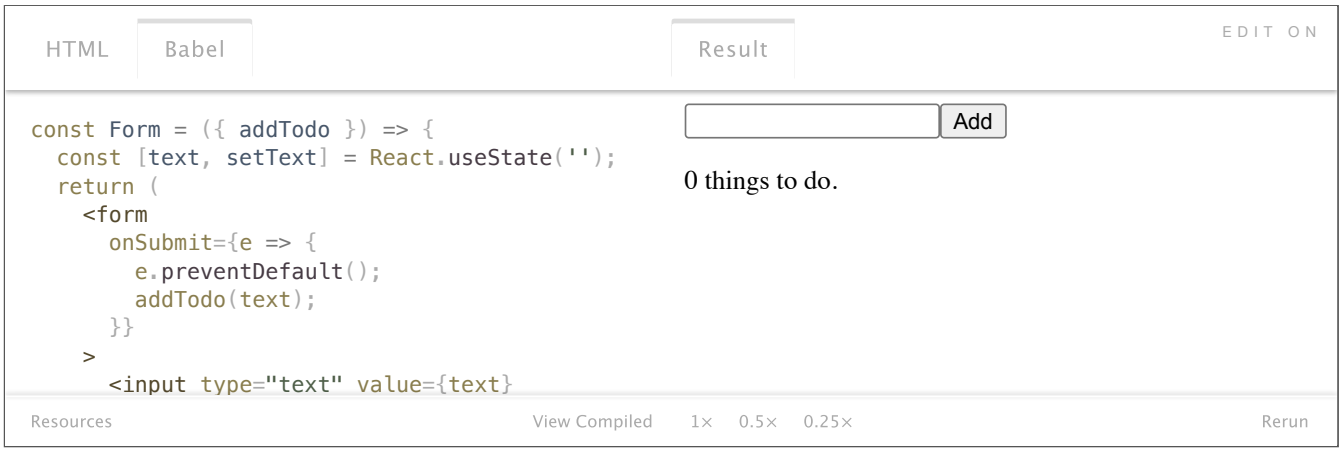
In a typical React codebase, you don't have to look far to find monster components. Our TodoApp, although simple, is already large and complex. It should be broken down into smaller pieces, but it's unclear whether creating a bunch of new components would improve readability or diminish it. The extra boilerplate of a new class can often be distracting. So it might be tempting to stick with the sub-render method approach, as it involves writing less code.

In the next section, we'll see that functional components are just as concise as sub-render methods while giving us all the benefits of class components.

Using many components

Using many components can seem counter-intuitive when making incremental changes to an app. It's much easier to add to an existing component rather than make a new one. This can lead to unnecessarily large components, though, which can be difficult to maintain. So it can be useful to invert our thinking and create many smaller components by default, to avoid inadvertently creating large, monolithic components.

As discussed in the previous section, functional components are better than class components when there are many components. Here's our `TodoApp` built with many functional components:



This is much more maintainable than using sub-render methods. Each subcomponent is fully decoupled from the main class. As the various parts of the UI get larger, they won't need to be retroactively decoupled from the parent and refactored into a standalone component.

The only modification that might happen as the subcomponents grow larger is they could be moved to their own file, just for better organization.

Guidelines for breaking down components

Breaking down components is usually a good idea, but it's still a balancing act. It's possible to get too granular with sub-components. Sometimes it makes sense to move a component into its own file, but

some components are more readable when they're in the same file as others.

Here are some guidelines to follow when deciding how to organize your components.

When should a component be broken down into subcomponents?

You may want to consider breaking a component into multiple subcomponents when one or more of the following conditions are true:

> - The component is large (I consider anything over 25 lines to be large).
> - At least one part of the component is reusable elsewhere.

When should a component have its own file?

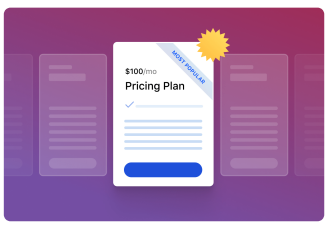
You may want to consider moving a component to its own file when one or more of the following conditions are true:

> - The component is being reused elsewhere (exported).
> - The component has several subcomponents.

This is just one strategy for writing maintainable React components. I find it works well in many cases, but the strategy that works best for each person, team, or codebase could look very different from this. That flexibility is what makes React so versatile. There are many ways to create the same UI in React—it's up to us as developers to choose the best approach for the problem at hand.React is a great UI library. Its flexibility makes it a great multitool for building almost any type of UI on the web. But this flexibility comes at the cost of guidance. With more flexibility, it's much easier to stray away from clean code, and it's even harder to identify exactly what clean code looks like.

RECENT POSTS

Add Value to Your Service by Joining a Marketplace



Comparing the Top 5 Pricing Models for

by Scott Fitzpatrick

Developer Apps

by Chris Tozzi