MARCH 7, 2017

Functional Stateless Components in React

What are functional, stateless components?

React 0.14 introduced functional, stateless components as a way to define React components as a function, rather than as an ES2015 class or via `React.createClass`.

Prior to React 0.14, writing a presentational component (that is, one that just renders props, and doesn't have state), could be fairly verbose:

```
const Username = React.createClass({
  render() {
    return <p>The logged in user is: {this.props.username}</
  },
});
// OR:
class Username extends React.Component {
  render() {
    return <p>The logged in user is: {this.props.username}</
  }
}
```

React 0.14 introduced functional stateless components (or, from now on, FSCs), which lets you express the above more succinctly:

```
const Username = function(props) {
  return <p>The logged in user is: {props.username}</p>;
};
```

Which, via ES2015 arrow functions, destructuring and implicit returns can be cut down really nicely:

```
const Username = ({ username }) => <p>The logged in user is:
```

FSCs not only provide a cleaner syntax but also have some other benefits that I'd like to talk about today, along with a couple of gotchas and things to look out for.

It's also important to note that you can have stateless class components, and that in the future we might be able to have functional, *stateful* components. Tyler McGinnis' post on the different types of components does a great job of laying out all the different terminology.

I think the primary benefit of FSCs is simplicity, and to me they act as a visual signal: "this component is solely props in, rendered UI out". If I see a class component, I do have to scan through to see what lifecycle methods it may be using, and what callbacks it may have. If I see an FSC, I know it isn't doing anything fancy. There are definitely times I'll write a stateless class component so I can define callback methods as class properties (especially if I'm passing prop values into a callback prop), but I'll write FSCs to signal that "this is a very straightforward rendering component".

FSCs lead to simplicity and offer visual cues

Mark, who I asked to review this post, made a great point in his review that FSCs offer visual cues that a component is solely taking some props and rendering output. If you have a class component, you have to read through the code to see if it deals with state, has lifecycle hooks, and so on. FSCs by definition have to be simple and that can save you time as a developer.

If you do have a component that doesn't have any state, but needs to define lifecycle methods, or have many event handlers, you should still prefer class components, even if they don't use state, but for presentational components FSCs are a perfect fit.

The syntax of FSCs encourages stateless components

Stateless components (also known as presentational components) should make up the bulk of your React applications. As a general rule of thumb, the less stateful components your application has, the better. Stateless components are easier to test, because you never have to interact or set up state. You can pass them props and assert on their output, and never have to test user interactions. They will generally have fewer bugs in them; in my experience components that have and change state over time are where most bugs will occur.

It's hard to convert a FSC to a stateful component

Imagine you have a component that you think you might need to add state to. It's actually a fair bit of manual work to convert a FSC to a stateful component, regardless of if you're using ES2015 classes or

`React.createClass`. Although this used to really frustrate me, I've come to appreciate this because it makes you think about if you really want to add state to the component. When you have a class component with just a `render` method, it's trivial to add state, but with a FSC it needs to be converted. Making it harder to quickly add state to a component is a good thing; you should carefully consider if you really need to.

To be clear; there are times when you can convert a FSC to a stateful component with good reason, but make sure you have that reason first and you've fully thought it through.

FSCs are not bound to React

In the world of JavaScript a new framework comes and goes every day; we've all seen the satirical blog posts about frameworks and version numbers of frameworks. But a FSC is not tied to React at all, other than the fact that it uses the JSX syntax. If you wanted to switch to another framework, or one day React stopped being worked on, it would be easy for another framework to add support for FSCs and make the adoption path easy. There's no reason React will cease to exist - but in general I've found the less code in your app that's bound to a specific framework, the better.

FSCs are great for styling (particularly on smaller projects)

In smaller projects, or small hack days, I've found that I will often use FSCs to very quickly create components that are used purely for styling:

```
const MyBlueButton = props => {
  const styles = { background: 'blue', color: 'white' };

  return <button {...props} style={styles} />;
};
```

In the future, FSCs may be optimised for performance by React

In the release for React 0.14, it was noted that in the future there are potential optimisations that React can make for FSCs:

> *In the future, we’ll also be able to make performance optimizations specific to these components by avoiding unnecessary checks and memory allocations.*

Whilst this is still work that is on going, clearly the React team are heavily behind FSCs as the building blocks of your applications:

> *This pattern is designed to encourage the creation of these simple components that should comprise large portions of your apps.*

Not only should you use FSCs because the React team encourages it, but in a future release of React you may see good performance increases by doing so. **Note that currently there is no optimisations done on FSCs**. Whilst it is planned after the work on React Fiber, there is currently no difference in performance.

Event handlers and FSCs

It's a bit of a misconception that FSCs don't allow you to define event handlers. You can just define them in-line:

```
const SomeButton = props => {
  const onClick = e => (...)

  return <button onClick={onClick}>Click me!</button>
}
```

It's important to note that this isn't the most efficient way of doing this; every time the component is run to potentially be rerendered, the `onClick` function will be redefined. This is work that you might want to avoid - and in some performance critical applications you might see this make a small difference. You'll find many blog posts online saying you should never do this, but the reality is for most applications that the optimisations will not be noticed. You should be aware of this and know that in certain situations it might really hurt performance, but don't shy away from adding an event handler in an FSC because of it.

If you do really want to avoid this, you have two choices. You either need to turn the component into a full component, or you can pull the event handler out of the FSC (only if you don't want to refer to the component's `props`, which means this often isn't feasible):

```
const onClick = e => (...)

const SomeButton = props => {
  return <button onClick={onClick}>Click me!</button>
}
```

You can still define `propTypes` and `defaultProps` in FSCs

When using FSCs, you define `propTypes` and `defaultProps` on the function itself:

```
const Username = props => <p>...</p>;

Username.propTypes = {
  username: React.PropTypes.string.isRequired,
};

Username.defaultProps = {
  username: 'Jack',
};
```

Context in FSCs

Although you should generally be wary of context in React, as I blogged about recently, FSCs do still support context if you need them to. When using context, it's simply passed in as the second argument:

```
const Username = (props, context) => <p>Username is {context

FooComponent.contextTypes = {
  name: React.PropTypes.string.isRequired,
};
```

On the whole I advise mostly against context, for reasons documented in the above blog post, but it's good to know that if you need it, you are able to use it.

Conclusion

To conclude, I would actively encourage you to think about using FSCs for as much of your application as you can. They are cleaner, have the potential to be more performant as React develops, and encourage good patterns in your React codebase that will lead to a more maintainable application as it grows.

Huge thanks to Mark Erikson for taking time to review this blog post.

If you enjoyed this post, be sure to follow me on Twitter.

All blog posts