ReactJS: Props vs. State

Published on: November 27, 2016

Tags: reactjs and js

I've been using ReactJS with Redux at work recently, and I have quite a few questions about how it all fits together. I figured I'd start small, with just some React questions and see how it goes.

This "props vs. state" question is pretty common for new React devs - they look so similar, but are used differently. So what's going on there?

Props

What does "props" even mean?

To get the jargon out of the way, "props" is short for "properties" so nothing particularly fancy there.

Well, all right then. What makes props special?

`props` are passed into the component

Here's an example (code from the React Guide):

```
class Welcome extends React.Component {   render() {     return
<h1>Hello {this.props.name}</h1>;   } }  const element = <Welcome
name="Sara" />;
```

You can play with this on CodePen.

The line `<Welcome name="Sara" />` creates a property `name` with value `"Sara"`.

That sounds kinda like a function call...

Yep, the property is passed to the component, similar to how an argument is passed to a function. In fact, we could even rewrite the component to be simpler:

```
function Welcome(props) {   return <h1>Hello {props.name}</h1>; }
```

Now the "props as arguments" comparison is even clearer.

OK, so props "come from above."

Often, but not always. A component can also have default props, so if a prop isn't passed through it can still be set.

We can make the `name` property optional by adding `defaultProps` to the `Welcome`class:

```
class Welcome extends React.Component {   render() {     return <h1>Hello {this.props.name}</h1>;   } }  Welcome.defaultProps = { name: "world", };
```

If `Welcome` is called without a name it will simply render `<h1> Hello world</h1>`.

So `props` can come from the parent, or can be set by the component itself.

`props` should not change

What?! I've totally done that!

You used to be able to change `props` with `setProps` and `replaceProps` but these have been deprecated. During a component's life cycle `props` should not change (consider them immutable).
Fine, I won't change props any more.

Since `props` are passed in, and they cannot change, you can think of any React component that only uses `props` (and not `state`) as "pure," that is, it will always render the same output given the same input. This makes them really easy to test - win!

State

Like `props`, `state` holds information about the component. However, the kind of information and how it is handled is different.

By default, a component has no state. The `Welcome` component from above is stateless:

```
function Welcome(props) {   return <h1>Hello {props.name}</h1>; }
```

So when would you use state?

When a component needs to keep track of information between renderings the component *itself* can create, update, and use state.

We'll be working with a fairly simple component to see `state` working in action. We've got a button that keeps track of how many times you've clicked it.

Yawn...

I know, but here's the code:

```
class Button extends React.Component {   constructor() {
super();     this.state = {       count: 0,     };   }
updateCount() {     this.setState((prevState, props) => {
return { count: prevState.count + 1 }     });   }    render()
{     return (<button               onClick={() =>
this.updateCount()}             >               Clicked
{this.state.count} times             </button>);   } }
```

You can play with this code on CodePen.

Gah! There's so much there! What's going on?

So now we're working with `state` things are a bit more complicated. But we'll break it down to make it more understandable.

Our first real difference between `props` and `state` is that...

`state` is created in the component

Let's look at the `constructor` method:

```
constructor() {   super();   this.state = {     count: 0,
}; }
```

This is where `state` gets it's initial data. The inital data can be hard coded (as above), but it can also come from `props`.

Well that's just confusing.

It is, I know. But it makes sense - you can't change `props`, but it's pretty reasonable to want to do stuff to the data that a component receives. That's where state comes in.

Moving on brings us to our second difference...

`state` is changeable

Here's `updateCount` again:

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

We change the state to keep track of the total number of clicks. The important bit is `setState`. First off, notice that `setState` takes a function, that's becuase `setState`can run asynchronously. It needs to take a callback function rather than updating the state directly. You can see we have access to `prevState` within the callback, this will contain the previous state, even if the state has already been updated somewhere else. Pretty slick, huh?

But React goes one step better, `setState` updates the state object **and** re-renders the component automagically.

Boom!

Yeah, this is pretty great of React to do, no need for us to explicitly re-render or worry about anything. React will take care of it all!

`setState` warning one!

It is tempting to write `this.state.count = this.state.count + 1`. *Do not do this!* React cannot listen to the state getting updated in this way, so your component will not re-render. Always use `setState`.

`setState` warning two!

It might also be tempting to write something like this:

```
// DO NOT USE this.setState({   count: this.state.count + 1 });
```

Although this might look reasonable, doesn't throw errors, and you might find examples that use this syntax online, it is *wrong*. This does not take into account the asychronous nature that `setState` can use and might cause errors with out of sync state data.

Program as usual

And finally, `render`

```
render() {   return (<button              onClick={() =>
this.updateCount()}           >             Clicked
{this.state.count} times           </button>); }
```

`onClick={() => this.updateCount()}` means that when the button is clicked the `updateCount` method will be called. We need to use ES6's arrow function so `updateCount` will have access to this instance's `state`. The text rendered in the button is `Clicked {this.state.count} times`, which will use whatever `this.state.count` is at the time of rendering.

Phew! That was a lot! Can I have it one more time?

Sure thing, let's look at the whole flow:

1. The component is initialised and `state.count` is set to 0

   ```
   this.state = {     count: 0,   };
   ```

2. The component renders, with "Clicked 0 times" as the button text

   ```
   Clicked {this.state.count} times
   ```

3. The user clicks the button

   click!

4. `updateCount` is called, bound to this instance of the component

   ```
   onClick={() => this.updateCount()}
   ```

5. `updateCount` calls `setState` with a call back to increase the counter from the previous state's counter value

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

6. `setState` triggers a call to `render`

   React magic!

7. The component renders, with "Clicked 1 times" as the button text

   ```
   Clicked {this.state.count} times
   ```

Review

While `props` and `state` both hold information relating to the component, they are used differently and should be kept separate.

`props` contains information set by the parent component (although defaults can be set) and should not be changed.

`state` contains "private" information for the component to initialise, change, and use on it's own.

... props are a way of passing data from parent to child. ... State is reserved only for interactivity, that is, data that changes over time.

Understanding State and Props in React

I've been playing around with React and Redux recently and thought I would start writing articles on concepts which I've had to wrap my head around.

So unless you've been living in a cave for the past few years, you'll know that React is an awesome front-end library developed by the good folks at Facebook to make life easier for developers.

It's different to Angular or other frameworks as it is **purely** front-end (though see the comments below for a great clarification on this). With that said, it's extremely powerful.

One of the concepts I struggled to understand during my learning more about React was the interaction between State and Props. I figured that others may have had the same issue so here is my take on it.

PROPS

Let's start with props. This simply is shorthand for *properties*. Props are how components talk to each other. If you're at all familiar with React then you should know that props flow downwards from the parent component.

There is also the case that you can have default props so that props are set even if a parent component doesn't pass props down.

This is why people refer to React as having *uni-directional* data flow. This takes a bit of getting your head around and I'll probably blog on this later, but for now just remember: data flows from parent to child. Props are immutable (fancy word for it not changing)

So we're happy. Components receive data from the parent. All sorted, right?

Well, not quite. What happens when a component receives data from someone other than the parent? What if the user inputs data directly to the component?

Well, this is why we have state.

STATE

Props shouldn't change, so state steps up. Normally components don't have state and so are referred to as *stateless*. A component using state is known as *stateful*. Feel free to drop that little tidbit at parties and watch people edge away from you.

So state is used so that a component can keep track of information in between any renders that it does. When you *setState* it updates the state object and then re-renders the component. This is super cool because that means React takes care of the hard work and is blazingly fast.

As a little example of state, here is a snippet from a search bar (worth checking out this course if you want to learn more about React)

```
Class SearchBar extends Component {
 constructor(props) {
  super(props);
this.state = { term: '' };
 }
render() {
  return (
   <div className="search-bar">
   <input
   value={this.state.term}
```

```
   onChange={event => this.onInputChange(event.target.value)} />
   </div>
   );
 }
onInputChange(term) {
  this.setState({term});
  this.props.onSearchTermChange(term);
 }
}
```

SUMMARY

Props and State do similar things but are used in different ways. The majority of your components will probably be stateless.

Props are used to pass data from parent to child or by the component itself. They are immutable and thus will not be changed.

State is used for *mutable* data, or data that will change. This is particularly useful for user input. Think search bars for example. The user will type in data and this will update what they see.