React JS : Props vs. State

props are passed into the component

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Welcome extends React.Component {  render() {  return <h1>Hello {this.props.name}</h1>;  }  }  const element = <Welcome name="Sara" />; |

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

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:

|  |  |
| --- | --- |
| 1  2  3 | function Welcome(props) {  return <h1>Hello {props.name}</h1>;  } |

Now the “props as arguments” comparison is even clearer.

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 Welcomeclass:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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

You used to be able to change props with setProps and replaceProps but these have been [deprecated](https://facebook.github.io/react/blog/2015/10/07/react-v0.14.html#new-deprecations-introduced-with-a-warning). 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.

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:

|  |  |
| --- | --- |
| 1  2  3 | function Welcome(props) {  return <h1>Hello {props.name}</h1>;  } |

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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | 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>);  }  } |

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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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.

state is changeable

Here’s updateCount again:

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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?

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

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.

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:

|  |  |
| --- | --- |
| 1  2  3  4 | // 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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) 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.

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

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

|  |  |
| --- | --- |
| 1  2  3 | this.state = {  count: 0,  }; |

1. The component renders, with “Clicked 0 times” as the button text

|  |  |
| --- | --- |
| 1 | Clicked {this.state.count} times |

1. The user clicks the button

*click!*

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

|  |  |
| --- | --- |
| 1 | onClick={() => this.updateCount()} |

1. updateCount calls setState with a call back to increase the counter from the previous state’s counter value

|  |  |
| --- | --- |
| 1  2  3 | this.setState((prevState, props) => {  return { count: prevState.count + 1 }  }); |

1. setState triggers a call to render

*React magic!*

1. The component renders, with “Clicked 1 times” as the button text

|  |  |
| --- | --- |
| 1 | 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.

=========================app1.html (Props and State)

<script>

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>);

}

}

React.render(<Button />, document.getElementById('app1'));

</script>

<div id=”app1”></div>

=========================

# props vs state

What's the exact difference between props and state?

It's fairly easy to understand how they work—especially when seen in context—but it's also a bit difficult to grasp them conceptually. It's confusing at first because they both have abstract terms and their values look the same, but they also have very different roles.

### Context

The main responsibility of a Component is to translate raw data into rich HTML. With that in mind, the props and the statetogether constitute the raw data that the HTML output derives from.

props + state is the input data for the render() function of a Component

### Common ground

Before separating props and state, let's also identify where they overlap.

* Both props and state are **plain JS objects**
* Both props and state changes trigger a **render update**
* Both props and state are **deterministic.** If your Component generates different outputs for the same combination of props and state then you're doing something wrong.

### Does this go inside props or state?

If a Component needs to alter one of its attributes at some point in time, that attribute should be part of its state, otherwise it should just be a prop for that Component.

#### props

props (short for properties) are a Component's **configuration,** its options if you may. They are received from above and **immutable** as far as the Component receiving them is concerned.

A Component cannot change its props, but it is responsible for putting together the props of its child Components.

#### state

The state starts with a default value when a Component mounts and then **suffers from mutations in time (mostly generated from user events).** It's a serializable\* representation of one point in time—a snapshot.

A Component manages its own state internally, but—besides setting an initial state—has no business fiddling with the state of its children. You could say the state is **private.**

\* We didn't say props are also serializable because it's pretty common to pass down callback functions through props.

#### Changing props and state

|  | **props** | **state** |
| --- | --- | --- |
| Can get initial value from parent Component? | Yes | Yes |
| Can be changed by parent Component? | Yes | No |
| Can set default values inside Component?\* | Yes | Yes |
| Can change inside Component? | No | Yes |
| Can set initial value for child Components? | Yes | Yes |
| Can change in child Components? | Yes | No |

\* Note that both props and state initial values received from parents override default values defined inside a Component.

### Should this Component have state?

state is optional. Since state increases complexity and reduces predictability, a Component without state is preferable. Even though you clearly can't do without state in an interactive app, you should avoid having too many Stateful Components.

#### Component types

* **Stateless Component** — Only props, no state. There's not much going on besides the render() function and all their logic revolves around the props they receive. This makes them very easy to follow (and test for that matter). We sometimes call these dumb-as-f\*ck Components (which [turns out](http://www.urbandictionary.com/define.php?term=dumb%20as%20fuck) to be the only way to misuse the F-word in the English language).
* **Stateful Component** — Both props and state. We also call these state managers. They are in charge of client-server communication (XHR, web sockets, etc.), processing data and responding to user events. These sort of logistics should be encapsulated in a moderate number of Stateful Components, while all visualization and formatting logic should move downstream into as many Stateless Components as possible.