



TYLERMCGINNIS.COM

[Courses](#)[Blog](#)[Newsletter](#)[Reviews](#)[Log In](#)

# React Tutorial: A Comprehensive Guide to learning React.js in 2018

March 12 2018. 21 min read.

by [Tyler McGinnis](#) 

This post is part of our **React Fundamentals** course. And this emoji 🐶 is my favorite.

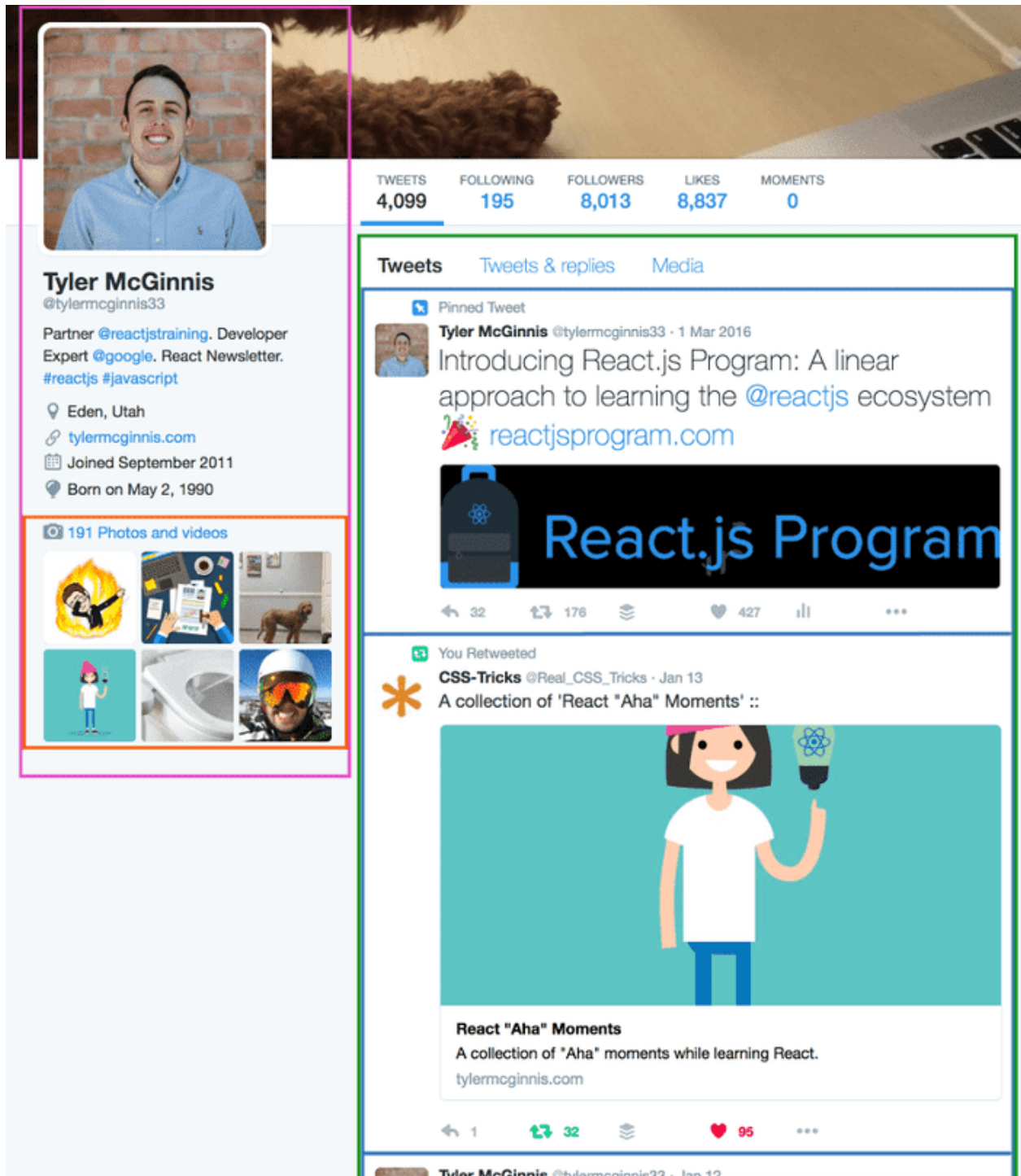
*This article was originally published in January of 2015 but was recently updated to React 16.3 and all the goodness it contains.*

## React.js Fundamentals:

Components are the building blocks of React. If you're coming from an Angular background, components are very similar to Directives. If you're coming from a different background, they're essentially widgets or modules. You can think of a component as a collection of HTML, CSS, JS, and some internal data specific to that component. I like to think of React components as the [Kolaches](#) of the web. They have everything you need, wrapped in a delicious composable bundle. These components are defined either in pure JavaScript or they can be defined in what the React team calls "JSX". If you decide to use JSX (which you most likely will, it's pretty standard — and it's what we'll use for this tutorial), you'll need some compile stage to convert your JSX to JavaScript, we'll get to this later.

What makes React so convenient for building user interfaces is that data is either received from a component's parent component, or it's contained in the component itself. Before we jump into code, let's make sure we have a high level understanding of components.





Above we have a picture of my Twitter profile. If we were going to re-create this page in React, we would break different sections up into different components (highlighted). Notice that components can have nested components inside of them. We might name the left component (pink) the `UserInfo` component. Inside the `UserInfo` component we have another component (orange), that we could call the `UserImages` component. The way this parent/child relationship works is our `UserInfo` component, or the parent component, is where the 'state' of the data for both itself and the `UserImages` component (child component) lives. If we wanted to use any part of the parent component's data in the child component, which we do, we would pass that data to the child component as an attribute. In this example, we pass the `UserImages` component all of the images that the user has (which currently live in the `UserInfo` component). We'll get more into the details of the code in a bit,

but I want you to understand the bigger picture of what's happening here. This parent/child hierarchy makes managing our data relatively simple because we know exactly where our data lives and we shouldn't manipulate that data anywhere else.

The topics below are what I believe to be the fundamental aspects of React. If you understand all of them and their purposes, you'll be at a very good spot after reading this tutorial.

JSX – Allows us to write HTML like syntax which gets transformed to `lightweightJavaScript` objects.

Virtual DOM – A JavaScript representation of the actual DOM.

`React.Component` – The way in which you create a new component.

`render` (method) – Describes what the UI will look like for the particular component.

`ReactDOM.render` – Renders a React component to a DOM node.

`state` – The internal data store (object) of a component.

`constructor` (`this.state`) - The way in which you establish the initial state of a component.

`setState` – A helper method used for updating the state of a component and re-rendering the UI

`props` – The data which is passed to the child component from the parent component.

`propTypes` – Allows you to control the presence, or types of certain props passed to the child component.

`defaultProps` – Allows you to set default props for your component.

#### Component Lifecycle

- `componentDidMount` – Fired after the component mounted
- `componentWillUnmount` – Fired before the component will unmount
- `getDerivedStateFromProps` - Fired when the component mounts and whenever the props change. Used to update the state of a component when its props change

#### Events

- onClick
- onSubmit
- onChange

I know it seems like a lot, but you'll soon see how each piece is fundamental in building robust applications with React (and I also wasn't kidding when I said I wanted this to be a comprehensive guide).

At this point you should understand, on a very high level, how React works. Now, let's jump into some code.

## Creating your First Component (JSX, Virtual DOM, render, ReactDOM.render)

Let's go ahead and build our very first React component.

To create a React component, you'll use an ES6 class.

```
import React from 'react'
import ReactDOM from 'react-dom'

class HelloWorld extends React.Component {
  render() {
    return (
      <div>Hello World!</div>
    )
  }
}

ReactDOM.render(<HelloWorld />, document.getElementById('root'));
```

kyl0z8yv5 Edit on CodeSandbox

package.json index.html index.js < > ↺ https://kyl0z8yv5.co 🔗 🔍

```
1  import React from 'react'
2  import ReactDOM from 'react-dom'
3
4  class HelloWorld extends React.Component {
5    render() {
6      return (
7        <div>Hello World!</div>
8      )
9    }
10 }
11
12 ReactDOM.render(<HelloWorld />, document.getElementById('root'))
```

Console Problems Tests

Notice that the only method on our class is `render`. Every component is required to have a render method. The reason for that is render is describing the UI (user interface) for our component. So in this example the text that will show on the screen where this component is rendered is Hello World! Now let's look at what ReactDOM is doing. ReactDOM.render takes in two arguments. The first argument is the component you want to render, the second argument is the DOM node where you want to render the component. (Notice we're using ReactDOM.render and not React.render. This was a change made in React .14 to make React more modular. It makes sense when you think that React can render to more things than just a DOM element). In the example above we're telling React to take our HelloWorld component and render it to the element with an ID of `root`. Because of the parent/child relations of React we talked about earlier, you usually only have to use ReactDOM.render once in your application because by rendering the most parent component, all child components will be rendered as well.

Now at this point you might feel a little weird throwing "HTML" into your JavaScript. Since you started learning web development, you've been told that you should keep your logic out of the view, AKA keep your JavaScript uncoupled from your HTML. This paradigm is strong, but it does have some weaknesses. I don't want to make this tutorial longer trying to convince you that this idea is a step in the right direction, so if this idea still bothers you you can check out [this link](#). As you learn more about React, this uneasiness should quickly subside. The "HTML" that you're writing in the render method isn't actually HTML but it's what React is calling "JSX". JSX simply allows us to write HTML like syntax which (eventually) gets transformed to lightweight JavaScript objects. React is then able to take these JavaScript objects and

from them form a “virtual DOM” or a JavaScript representation of the actual DOM. This creates a win/win situation where you get the accessibility of templates with the power of JavaScript.

Looking at the example below, this is what your JSX will eventually be compiled into.

```
class HelloWorld extends React.Component {  
  render() {  
    return React.createElement("div", null, "Hello World");  
  }  
}
```

*Now, you can forgo the JSX -> JS transform phase and write your React components like the code above, but as you can imagine, that would be rather tricky. I'm unaware of anyone who is not using JSX. For more information about what JSX compiles down to, check out [React Elements vs React Components](#)*

Up until this point we haven't really emphasized the importance of this new virtual DOM paradigm we're jumping into. The reason the React team went with this approach is because, since the virtual DOM is a JavaScript representation of the actual DOM, React can keep track of the difference between the current virtual DOM (computed after some data changes), with the previous virtual DOM (computed before some data changes). [React then isolates the changes between the old and new virtual DOM and then only updates the real DOM with the necessary changes.](#) In more layman's terms, because manipulating the actual DOM is slow, React is able to minimize manipulations to the actual DOM by keeping track of a virtual DOM and only updating the real DOM when necessary and with only the necessary changes. ([More info here](#)). Typically UI's have lots of state which makes managing state difficult. By re-rendering the virtual DOM every time any state change occurs, React makes it easier to think about what state your application is in. The process looks something like this,

**Some user event which changes the state of your app → Re-render virtual DOM -> Diff previous virtual DOM with new virtual DOM -> Only update real DOM with necessary changes.**

Because there's this transformation process from JSX to JS, you need to set up some sort of transformation phase as you're developing. In part 2 of this series I'll introduce Webpack and Babel for making this transformation.

It's time to take a look back at our “Most Important Parts of React” checklist and see where we're at now.

**JSX** – Allows us to write HTML like syntax which gets transformed to lightweight JavaScript objects.

**Virtual DOM** – A JavaScript representation of the actual DOM.

**React.Component** – The way in which you create a new component.

**render (method)** – Describes what the UI will look like for the particular component.

**ReactDOM.render** – Renders a React component to a DOM node.

**state** – The internal data store (object) of a component.

**constructor (this.state)** - The way in which you establish the initial state of a component.

**setState** – A helper method used for updating the state of a component and re-rendering the UI

**props** – The data which is passed to the child component from the parent component.

**propTypes** – Allows you to control the presence, or types of certain props passed to the child component.

**defaultProps** – Allows you to set default props for your component.

#### Component Lifecycle

- **componentDidMount** – Fired after the component mounted
- **componentWillUnmount** – Fired before the component will unmount
- **getDerivedStateFromProps** - Fired when the component mounts and whenever the props change. Used to update the state of a component when its props change

#### Events

- **onClick**
- **onSubmit**
- **onChange**

We're making good pace. Everything in bold is what we've already covered and you should at least be able to explain how those certain components fit into the React ecosystem.

### Adding State to your Component (state)

Next on the list is `state`. Earlier we talked about how managing user interfaces is difficult because they typically have lots of different states. This area is where React really starts to shine. Each component has the ability to manage its own state and pass its state down to child components if needed. Going back to the Twitter example from earlier, the `UserInfo` component (highlighted in pink above) is responsible for managing the state (or data) of the users information. If another component also needed this state/data but that state wasn't a direct child of the `UserInfo` component, then you would create another component that would be the direct parent of the `UserInfo` and the other component (or both components which required that state), then you would pass the state down as props into the child components. In other words, if you have a multi component hierarchy, a common parent component should manage the state and pass it down to its children components via props.

Let's take a look at an example component using it's own internal state.

```
class HelloUser extends React.Component {  
  constructor(props) {  
    super(props)  
  
    this.state = {  
      username: 'tylermcginnis'  
    }  
  }  
  render() {  
    return (  
      <div>  
        Hello {this.state.username}  
      </div>  
    )  
  }  
}
```



l9pjn25v09 [Edit on CodeSandbox](#)

package.json index.html Hello.js <https://l9pjn25v09.cr>

```
1  import React from 'react';           Hello tylermcginnis
2  import ReactDOM from 'react-dom';
3
4  class HelloUser extends React.Component {
5    constructor(props) {
6      super(props)
7
8      this.state = {
9        username: 'tylermcginnis'
10     }
11   }
12   render() {
13     return (
14       <div>
15         Hello {this.state.username}
16       </div>
```

Console Problems Tests

We've introduced some new syntax with this example. The first one you'll notice is the constructor method. From the definition above, the constructor method is "The way in which you set the state of a component". In other terms, any data you put on `this.state` inside of the constructor will be part of that component's state. In the code above we're telling our component that we want it to keep track of a `username`. This `username` can now be used inside our component by doing `{this.state.username}`, which is exactly what we do in our render method.

The last thing to talk about with state is that our component needs the ability to modify its own internal state. We do this with a method called **setState**. Remember earlier when we talked about the re-rendering of the virtual dom whenever the data changes?

**Signal to notify our app some data has changed → Re-render virtual DOM -> Diff previous virtual DOM with new virtual DOM -> Only update real DOM with necessary changes.**

That "signal to notify our app some data has changed" is actually just `setState`. Whenever `setState` is called, the virtual DOM re-renders, the diff algorithm runs, and the real DOM is updated with the necessary changes.

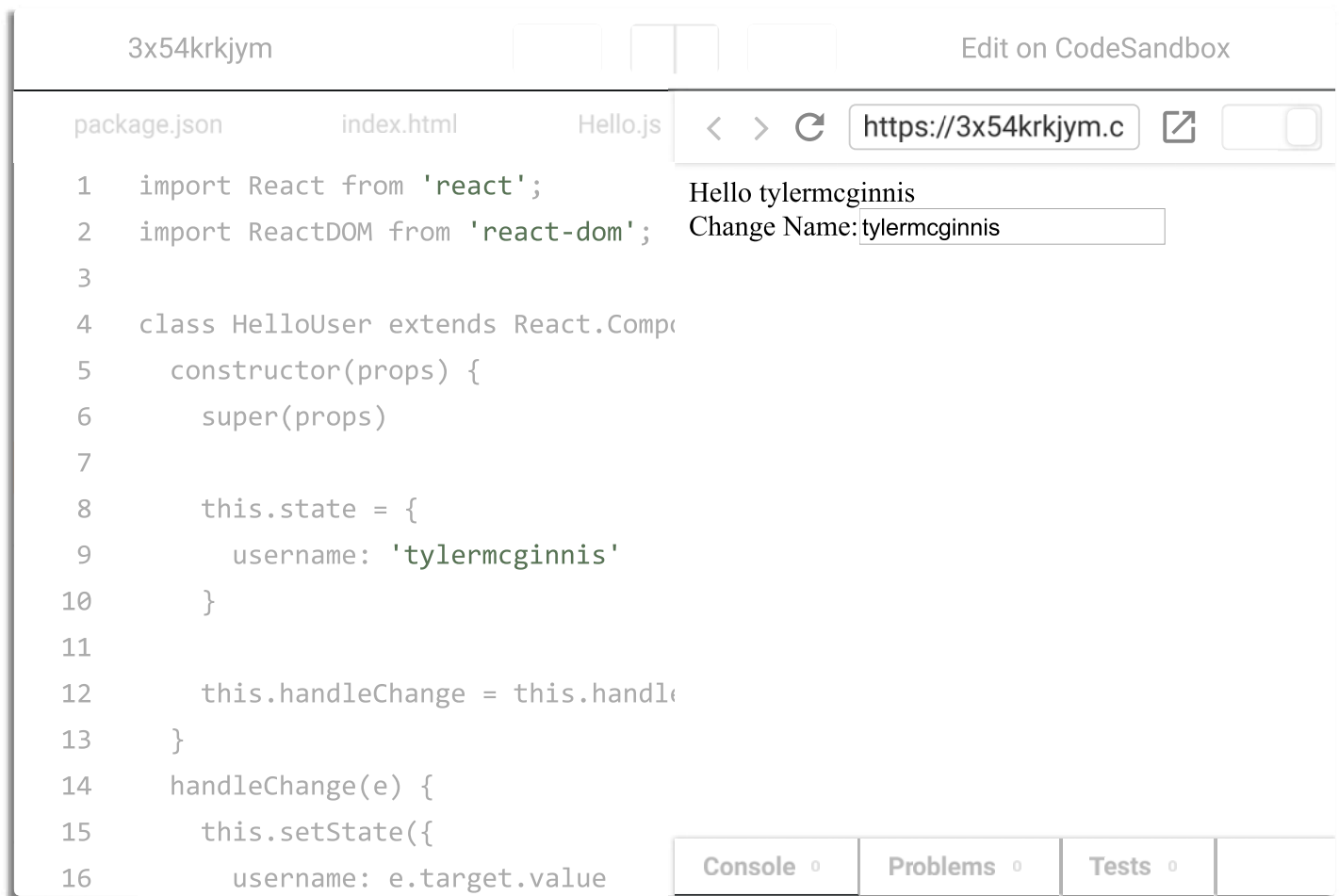
As a sidenote, when we introduce `setState` in the code below, we're also going to introduce a few events that are on our list. Two birds, one stone.

So in the next code sample, we're going to now have an input box that whenever someone types into it, it will automatically update our state and change the username.

```
class HelloUser extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      username: 'tylermcginnis'
    }

    this.handleChange = this.handleChange.bind(this)
  }
  handleChange (e) {
    this.setState({
      username: e.target.value
    })
  }
  render() {
    return (
      <div>
        Hello {this.state.username} <br />
        Change Name:
        <input
          type="text"
          value={this.state.username}
          onChange={this.handleChange}
        />
      </div>
    )
  }
}
```



Note we've introduced a few more things. The first thing is the `handleChange` method. This method is going to get called every time a user types in the input box. When `handleChange` is called, it's going to call `setState` to re-define our username with whatever was typed into the input box (`e.target.value`). Remember, whenever `setState` is called, React creates a new virtual DOM, does the diff, then updates the real DOM.

Now let's look at our render method. We've added a new line that contains an input field. The type of the input field is obviously going to be `text`. The value is going to be the value of our username which was originally defined in our `getInitialState` method and will be updated in the `handleChange` method. Notice there is a new attribute you've probably never seen before, `onChange`. `onChange` is a React thing and it will call whatever method you specify every time the value in the input box changes, in this case the method we specified was `handleChange`.

The process for the code above would go something like this.

**A user types into the input box → handleChange is invoked → the state of our component is set to a new value → React re-renders the virtual DOM → React Diff's the change → Real DOM is updated.**

Later on when we cover props, we'll see some more advanced use cases of handling state.

We're getting there! If you can't explain the items in bold below, go re-read that section. One tip on REALLY learning React, don't let passively reading this give you a false sense of security that you actually

know what's going on and can re-create what we're doing. Head over to CodeSandbox and try to recreate (or create your own) components without looking at what I've done. It's the only way you'll truly start learning how to build with React. This goes for this tutorial and the following to come.

**JSX** – Allows us to write HTML like syntax which gets transformed to lightweight JavaScript objects.

**Virtual DOM** – A JavaScript representation of the actual DOM.

**React.Component** – The way in which you create a new component.

**render (method)** – Describes what the UI will look like for the particular component.

**ReactDOM.render** – Renders a React component to a DOM node.

**state** – The internal data store (object) of a component.

**constructor (this.state)** - The way in which you establish the initial state of a component.

**setState** – A helper method used for updating the state of a component and re-rendering the UI

**props** – The data which is passed to the child component from the parent component.

**propTypes** – Allows you to control the presence, or types of certain props passed to the child component.

**defaultProps** – Allows you to set default props for your component.

#### Component Lifecycle

- **componentDidMount** – Fired after the component mounted
- **componentWillUnmount** – Fired before the component will unmount
- **getDerivedStateFromProps** - Fired when the component mounts and whenever the props change. Used to update the state of a component when its props change

#### Events

- **onClick**
- **onSubmit**
- **onChange**

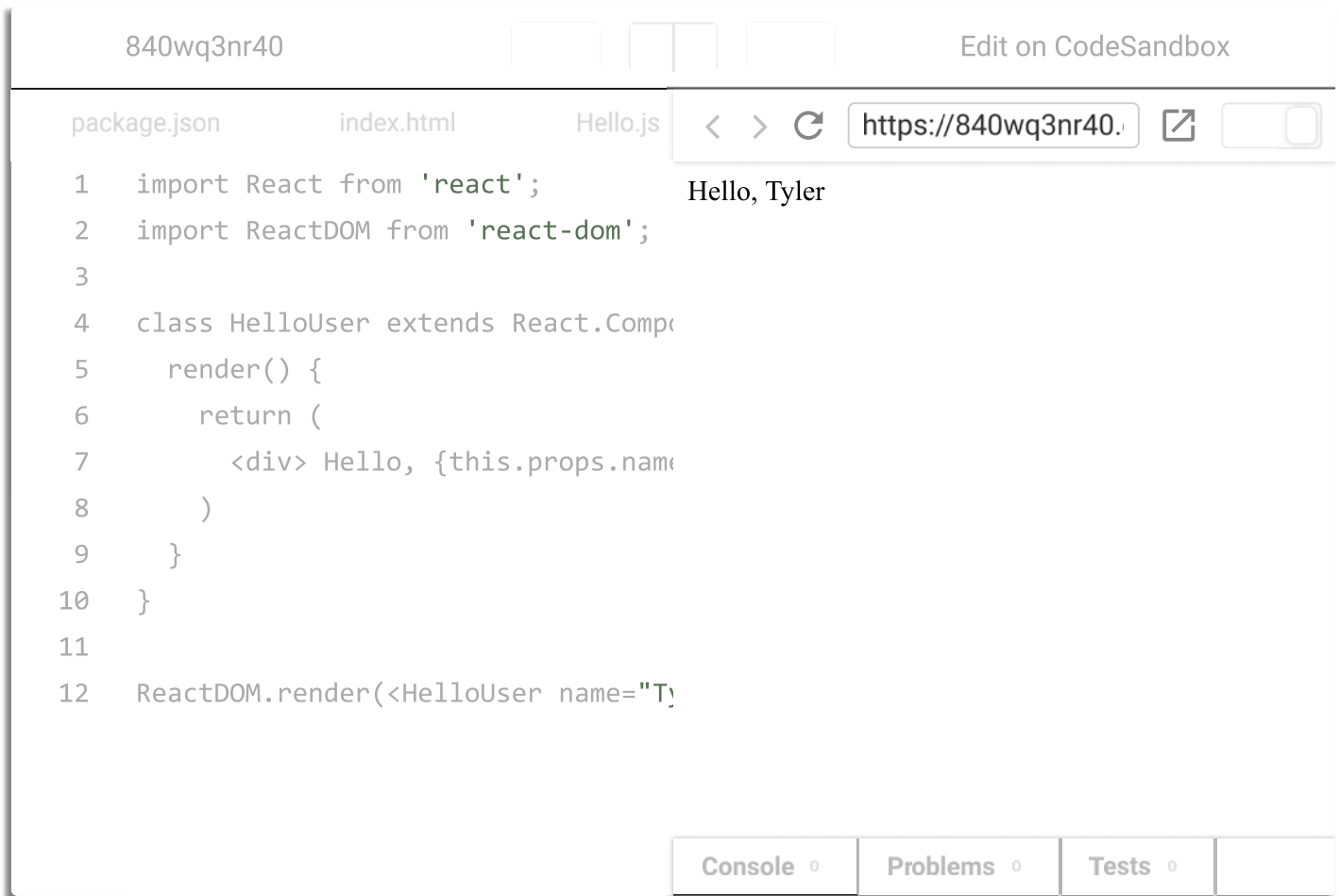
I

## Receiving State from Parent Component (props, propTypes, getDefaultProps)

We've talked about props a few times already since it's hard to really do much without them. By our definition above, props is the data which is passed to the child component from the parent component. This allows for our React architecture to stay pretty straight forward. Handle state in the highest most parent component which needs to use the specific data, and if you have a child component that also needs that data, pass that data down as props.

Here's a very basic example of using props.

```
class HelloUser extends React.Component {  
  render() {  
    return (  
      <div> Hello, {this.props.name}</div>  
    )  
  }  
}  
  
ReactDOM.render(<HelloUser name="Tyler"/>, document.getElementById('root'));
```



Notice on line 9 we have an attribute called name with a value of “Tyler”. Now in our component, we can use `{this.props.name}` to get “Tyler”.

Let’s look at a more advanced example. We’re going to have two components now. One parent, one child. The parent is going to keep track of the state and pass a part of that state down to the child as props. Let’s first take a look at that parent component.

Parent Component:

```
class FriendsContainer extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      name: 'Tyler McGinnis',
      friends: ['Jake Lingwall', 'Sarah Drasner', 'Merrick Christensen']
    }
  }
  render() {
    return (
```

```

    <div>
      <h3> Name: {this.state.name} </h3>
      <ShowList names={this.state.friends} />
    </div>
  )
}
}

```

There really isn't much going on in this component that we haven't seen before. We have an initial state and we pass part of that initial state to another component. The majority of the new code will come from this child component so let's take a closer look at that.

Child Component:

```

class ShowList extends React.Component {
  render() {
    return (
      <div>
        <h3> Friends </h3>
        <ul>
          {this.props.names.map((friend) => <li>{friend}</li>)}
        </ul>
      </div>
    )
  }
}

```

Remember that the code that gets returned from our render method is a representation of what the real DOM should look like. If you're not familiar with **Array.prototype.map**, this code might look a little wonky. All map does is it creates a new array, calls our callback function on each item in the array, and fills the new array with the result of calling the callback function on each item. For example,

```

const friends = ['Jake Lingwall', 'Sarah Drasner', 'Merrick Christensen'];
const listItems = friends.map((friend) => {
  return "<li> " + friend + "</li>";
});

console.log(listItems);
// ["<li> Jake Lingwall</li>", "<li> Sarah Drasner</li>", "<li> Merrick Christensen</li>"]

```

The console.log above returns `["<li> Jake Lingwall</li>", "<li> Murphy Randall</li>", "<li> Merrick Christensen</li>"]` .

Notice all that happened was we made a new array and added `<li> </li>` to each item in the original array.

What's great about map is it fits perfectly into React (and it's built into JavaScript). So in our child component above, we're mapping over names, wrapping each name in a pair of `<li>` tags, and saving that to our `listItems` variable. Then, our render method returns an unordered list with all of our friends.

Let's look at one more example before we stop talking about props. It's important to understand that wherever the data lives, is the exact place you want to manipulate that data. This keeps it simple to reason about your data. All getter/setter method for a certain piece of data will always be in the same component where that data was defined. If you needed to manipulate some piece of data outside where the data lives, you'd pass the getter/setter method into that component as props. Let's take a look at an example like that.

```
class FriendsContainer extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      name: 'Tyler McGinnis',
      friends: [
        'Jake Lingwall',
        'Sarah Drasner',
        'Merrick Christensen'
      ],
    }


    this.addFriend = this.addFriend.bind(this)
  }
  addFriend(friend) {
    this.setState((state) => ({
      friends: state.friends.concat([friend])
    }))
  }
  render() {
    return (
      <div>
        <h3> Name: {this.state.name} </h3>
        <AddFriend addNew={this.addFriend} />
      </div>
    )
  }
}
```



```

        <ShowList names={this.state.friends} />
      </div>
    )
  }
}

```

 Notice that in our `addFriend` method we introduced a new way to invoke `setState`. Instead of passing it an object, we're passing it a function which is then passed `state`. Whenever you're setting the new state of your component based on the previous state (as we're doing with our `friends` array), you want to pass `setState` a function which takes in the current state and returns the data to merge with the new state.

```

class AddFriend extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      newFriend: ''
    }

    this.updateNewFriend = this.updateNewFriend.bind(this)
    this.handleAddNew = this.handleAddNew.bind(this)
  }
  updateNewFriend(e) {
    this.setState({
      newFriend: e.target.value
    })
  }
  handleAddNew() {
    this.props.addNew(this.state.newFriend)
    this.setState({
      newFriend: ''
    })
  }
  render() {
    return (
      <div>
        <input

```

```
        type="text"
        value={this.state.newFriend}
        onChange={this.updateNewFriend}
      />
      <button onClick={this.handleAddNew}> Add Friend </button>
    </div>
  )
}
```

```
class ShowList extends React.Component {
  render() {
    return (
      <div>
        <h3> Friends </h3>
        <ul>
          {this.props.names.map((friend) => {
            return <li> {friend} </li>
          })}
        </ul>
      </div>
    )
  }
}
```

9o3l3pr1np Edit on CodeSandbox

package.json index.html Hello.js

< > ↺

https://9o3l3pr1np.c

🔗

🔍

```

1  import React from 'react';
2  import ReactDOM from 'react-dom';
3
4  class AddFriend extends React.Component {
5    constructor(props) {
6      super(props)
7
8      this.state = {
9        newFriend: ''
10     }
11
12     this.updateNewFriend = this.updateNewFriend;
13     this.handleAddNew = this.handleAddNew;
14   }
15   updateNewFriend(e) {
16     this.setState({

```

**Name: Tyler McGinnis**

**Friends**

- Jake Lingwall
- Sarah Drasner
- Merrick Christensen

Console 1

Problems 0

Tests 0

You'll notice the code above is mostly the same as the previous example, except now we have the ability to add a name to our friends list. Notice how I created a new AddFriend component that manages the new friend we're going to add. The reason for this is because the parent component (FriendContainer) doesn't care about the new friend you're adding, it only cares about all of your friends as a whole (the friends array). However, because we're sticking with the rule of only manipulate your data from the component that cares about it, we've passed the addFriend method down into our AddFriend component as a prop and we call it with the new friend once the handleAddNew method is called.

At this point I recommend you try to recreate this same functionality on your own using the code above as a guidance once you've been stuck for 3-4 minutes.

Before we move on from props, I want to cover two more React features regarding props. They are **propTypes** and **defaultProps**. I won't go into too much detail here because both are pretty straight forward.

**prop-types** allow you to control the presence, or types of certain props passed to the child component. With propTypes you can specify that certain props are required or that certain props be a specific type.

*As of React 15, PropTypes is no longer included with the React package. You'll need to install it separately by running `npm install prop-types`.*

**defaultProps** allow you to specify a default (or a backup) value for certain props just in case those props are never passed into the component.

I've modified our components from earlier to now, using propTypes, require that addFriend is a function and that it's passed into the AddFriend component. I've also, using defaultProps, specified that if no array of friends is given to the ShowList component, it will default to an empty array.

```
import React from 'react'
import PropTypes from 'prop-types'

class AddFriend extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      newFriend: ''
    }
  }

  updateNewFriend(e) {
    this.setState({
      newFriend: e.target.value
    })
  }

  handleAddNew() {
    this.props.addNew(this.state.newFriend)
    this.setState({
      newFriend: ''
    })
  }

  render() {
    return (
      <div>
        <input type="text" value={this.state.newFriend} onChange={this.updateNewFriend} />
        <button onClick={this.handleAddNew}> Add Friend </button>
      </div>
    )
  }
}
```

```
}  
  
AddFriend.propTypes: {  
  addNew: PropTypes.func.isRequired  
}
```

```
class ShowList extends React.Component {  
  render() {  
    return (  
      <div>  
        <h3> Friends </h3>  
        <ul>  
          {this.props.names.map((friend) => {  
            return <li> {friend} </li>  
          })}  
        </ul>  
      </div>  
    )  
  }  
}  
  
ShowList.defaultProps = {  
  names: []  
}
```

Alright, we're on the last stretch for this first tutorial. Let's take a look at our guide and see what we have left.

**JSX** – Allows us to write HTML like syntax which gets transformed to lightweight JavaScript objects.

**Virtual DOM** – A JavaScript representation of the actual DOM.

**React.Component** – The way in which you create a new component.

**render (method)** – Describes what the UI will look like for the particular component.

**ReactDOM.render** – Renders a React component to a DOM node.

**state** – The internal data store (object) of a component.

**constructor (this.state)** - The way in which you establish the initial state of a component.

**setState** – A helper method used for updating the state of a component and re-rendering the UI

**props** – The data which is passed to the child component from the parent component.

**prop-types** – Allows you to control the presence, or types of certain props passed to the child component.

**defaultProps** – Allows you to set default props for your component.

#### Component Lifecycle

- **componentDidMount** – Fired after the component mounted
- **componentWillUnmount** – Fired before the component will unmount
- **getDerivedStateFromProps** - Fired when the component mounts and whenever the props change. Used to update the state of a component when its props change

#### Events

- **onClick**
- **onSubmit**
- **onChange**

We're so close!

## Component Lifecycle

Each component you make will have its own lifecycle events that are useful for various things. For example, if we wanted to make an ajax request on the initial render and fetch some data, where would we do that? Or, if we wanted to run some logic whenever our props changed, how would we do that? The different lifecycle events are the answers to both of those. Let's break them down.

```
class App extends React.Component {  
  constructor(props) {  
    super(props)
```

```
    this.state = {
      name: 'Tyler McGinnis'
    }
  }
  componentDidMount(){
    // Invoked once the component is mounted to the DOM
    // Good for making AJAX requests
  }
  static getDerivedStateFromProps(nextProps, prevState) {
    // The object you return from this function will
    // be merged with the current state.
  }
  componentWillUnmount(){
    // Called IMMEDIATELY before a component is unmounted
    // Good for cleaning up listeners
  }
  render() {
    return (
      <div>
        Hello, {this.state.name}
      </div>
    )
  }
}
```

**componentDidMount** - Invoked once after the initial render. Because the component has already been invoked when this method is invoked, you have access to the virtual DOM if you need it. You do that by calling **this.getDOMNode()**. So this is the lifecycle event where you'll be making your AJAX requests to fetch some data.\*

**componentWillUnmount** - This life cycle method is invoked immediately before a component is unmounted from the DOM. This is where you can do necessary clean up.

**getDerivedStateFromProps** - Sometimes you'll need to update the state of your component based on the props that are being passed in. This is the lifecycle method in which you'd do that. It'll be passed the props and the state, and the object you return will be merged with the current state.

Well, if you stuck with me until this point, great job. I hope this tutorial was beneficial to you and you now feel at least mildly comfortable with React.

*For a much more in depth look at the fundamentals of React, check out our [React Fundamental course](#).*

Liked this post? Share it 



TYLERMCGINNIS.COM

[Courses](#)[Blog](#)[Newsletter](#)[Reviews](#)[Podcast](#)[Jobs](#)[Gear](#)[Log In](#)

## The Socials

## The Newsletter