**URL:** <https://scotch.io/tutorials/learning-react-getting-started-and-concepts>

**Learning React.js: Getting Started and Concepts**

[Ken Wheeler](https://pub.scotch.io/@) Oct 20, 2014 [Tutorials](https://scotch.io/tutorials) [Comments](https://scotch.io/tutorials/learning-react-getting-started-and-concepts#disqus_thread)

0

[Free Course](https://school.scotch.io/build-a-nodejs-website)

**[Build Your First Node.js Website](https://school.scotch.io/build-a-nodejs-website)**

[Node is a powerful tool to get](https://school.scotch.io/build-a-nodejs-website) **[JavaScript](https://school.scotch.io/build-a-nodejs-website)** [on the server. Use Node to build a great website.](https://school.scotch.io/build-a-nodejs-website)

[**# Introduction**](https://scotch.io/tutorials/learning-react-getting-started-and-concepts#introduction)

Today we are going to kick off the first installment in a new series of tutorials, **Learning React**, that will focus on becoming proficient and effective with Facebook’s React library. Before we start building anything meaningful, its important that we cover some base concepts first, so lets get this party started.

*Update 4/29/2016*: This article has been updated for React v15

**What is React?**

React is a UI library developed at Facebook to facilitate the creation of interactive, stateful & reusable UI components. It is used at Facebook in production, and Instagram.com is written entirely in React.

One of it’s unique selling points is that not only does it perform on the client side, but it can also be rendered server side, and they can work together inter-operably.

It also uses a concept called the Virtual DOM that selectively renders subtrees of nodes based upon state changes. It does the least amount of DOM manipulation possible in order to keep your components up to date.

**How does the Virtual DOM work?**

Imagine you had an object that you modeled around a person. It had every relevant property a person could possibly have, and mirrored the persons current state. This is basically what React does with the DOM.

Now think about if you took that object and made some changes. Added a mustache, some sweet biceps and Steve Buscemi eyes. In React-land, when we apply these changes, two things take place. First, React runs a “diffing” algorithm, which identifies what has changed. The second step is reconciliation, where it updates the DOM with the results of diff.

The way React works, rather than taking the real person and rebuilding them from the ground up, it would only change the face and the arms. This means that if you had text in an input and a render took place, as long as the input’s parent node wasn’t scheduled for reconciliation, the text would stay undisturbed.

Because React is using a fake DOM and not a real one, it also opens up a fun new possibility. We can render that fake DOM on the server, and boom, server side React views.

[**# Getting Started**](https://scotch.io/tutorials/learning-react-getting-started-and-concepts#getting-started)

Getting started with React is as simple as downloading their provided starter kit:

[React Starter Kit](https://facebook.github.io/react/downloads/react-15.0.1.zip)

You can also fork a JSFiddle they have provided:

[React JSFiddle](https://jsfiddle.net/reactjs/69z2wepo/)

**Page Setup**

When setting up your page, you want to include react.js, react-dom.js and JSXTransformer.js, and then write your component in a script node with type set to text/jsx:

<!DOCTYPE html>

<html>

<head>

<script src="build/react.js"></script>

<script src="build/react-dom.js"></script>

<script src="build/JSXTransformer.js"></script>

</head>

<body>

<div id="mount-point"></div>

<script type="text/jsx">

// React Code Goes Here

</script>

</body>

</html>

React DOM is supporting library and was introduced in recent version of React. The React DOM implementation was initially in the React library but become a stand alone library in v0.14

In React, components mount to an element, so in this example we can use the div mount-point as it’s parent container.

While this is the easiest way to get started, when its time to actually build something, its probably a good idea to use Browserify or webpack and set your components up in separate files.

[**# The Basics**](https://scotch.io/tutorials/learning-react-getting-started-and-concepts#the-basics)

React’s basic building blocks are called components. Lets write one:

<script type="text/jsx">

/\*\* @jsx React.DOM \*/

ReactDOM.render(

<h1>Hello, world!</h1>,

document.getElementById('myDiv')

);

</script>

If you haven’t seen one of these before, you are probably wondering what Javascript/HTML chimera sorcery is taking place right now.

**JSX**

It’s called JSX, and it is a Javascript XML syntax transform. This lets you write HTML-*ish* tags in your Javascript. I say HTML-*ish* because there are a couple gotchas. You are really just writing XML based object representations.

For regular html tags, the class attribute is className and the for attribute is htmlFor in JSX because these are reserved words in Javascript. More differences can be [reviewed here](http://facebook.github.io/react/docs/jsx-gotchas.html).

Now you don’t need to use JSX, here is what the syntax looks like without it:

/\*\* @jsx React.DOM \*/

React.render(

React.createElement('h1', null, 'Hello!'),

document.getElementById('myDiv')

);

You can read more about element support [here](http://facebook.github.io/react/docs/tags-and-attributes.html) if you’d like.

**Components**

When using the ReactDOM’s render method above, our first argument is the component we want to render, and the second is the DOM node it should mount to. We can use the createClass method to create custom component classes. It takes an object of specifications as it’s argument. Lets create one below:

var MyComponent = React.createClass({

render: function(){

return (

<h1>Hello, world!</h1>

);

}

});

After creating a class we can render it to our document like so:

ReactDOM.render(

<MyComponent/>,

document.getElementById('myDiv')

);

Cool, huh?

**Props**

When we use our defined components, we can add attributes called props. These attributes are available in our component as this.props and can be used in our render method to render dynamic data:

var MyComponent = React.createClass({

render: function(){

return (

<h1>Hello, {this.props.name}!</h1>

);

}

});

ReactDOM.render(<MyComponent name="Handsome" />, document.getElementById('myDiv'));

[**# Specs, Lifecycle & State**](https://scotch.io/tutorials/learning-react-getting-started-and-concepts#specs-lifecycle-state)

The render method is the only required spec for creating a component, but there are several lifecycle methods & specs we can use that are mighty helpful when you actually want your component to do anything.

**Lifecycle Methods**

* **componentWillMount** – Invoked once, on both client & server before rendering occurs.
* **componentDidMount** – Invoked once, only on the client, after rendering occurs.
* **shouldComponentUpdate** – Return value determines whether component should update.
* **componentWillUnmount** – Invoked prior to unmounting component.

**Specs**

* **getInitialState** – Return value is the initial value for state.
* **getDefaultProps** – Sets fallback props values if props aren’t supplied.
* **mixins** – An array of objects, used to extend the current component’s functionality.

There are several more specs & lifecycle methods that you can read about [here](http://facebook.github.io/react/docs/component-specs.html).

**State**

Every component has a state object and a props object. State is set using the setState method. Calling setState triggers UI updates and is the bread and butter of React’s interactivity. If we want to set an initial state before any interaction occurs we can use the getInitialState method. Below, see how we can set our component’s state:

var MyComponent = React.createClass({

getInitialState: function(){

return {

count: 5

}

},

render: function(){

return (

<h1>{this.state.count}</h1>

)

}

});

[**# Events**](https://scotch.io/tutorials/learning-react-getting-started-and-concepts#events)

React also has a built in cross browser events system. The events are attached as properties of components and can trigger methods. Lets make our count increment below using events:

/\*\* @jsx React.DOM \*/

var Counter = React.createClass({

incrementCount: function(){

this.setState({

count: this.state.count + 1

});

},

getInitialState: function(){

return {

count: 0

}

},

render: function(){

return (

<div class="my-component">

<h1>Count: {this.state.count}</h1>

<button type="button" onClick={this.incrementCount}>Increment</button>

</div>

);

}

});

ReactDOM.render(<Counter/>, document.getElementById('mount-point'));

[**# Unidirectional Data Flow**](https://scotch.io/tutorials/learning-react-getting-started-and-concepts#unidirectional-data-flow)

In React, application data flows unidirectionally via the state and props objects, as opposed to the two-way binding of libraries like Angular. This means that, in a multi component heirachy, a common parent component should manage the state and pass it down the chain via props.

Your state should be updated using the setState method to ensure that a UI refresh will occur, if necessary. The resulting values should be passed down to child components using attributes that are accessible in said children via this.props.

See this example that shows this concept in practice:

/\*\* @jsx React.DOM \*/

var FilteredList = React.createClass({

filterList: function(event){

var updatedList = this.state.initialItems;

updatedList = updatedList.filter(function(item){

return item.toLowerCase().search(

event.target.value.toLowerCase()) !== -1;

});

this.setState({items: updatedList});

},

getInitialState: function(){

return {

initialItems: [

"Apples",

"Broccoli",

"Chicken",

"Duck",

"Eggs",

"Fish",

"Granola",

"Hash Browns"

],

items: []

}

},

componentWillMount: function(){

this.setState({items: this.state.initialItems})

},

render: function(){

return (

<div className="filter-list">

<input type="text" placeholder="Search" onChange={this.filterList}/>

<List items={this.state.items}/>

</div>

);

}

});

var List = React.createClass({

render: function(){

return (

<ul>

{

this.props.items.map(function(item) {

return <li key={item}>{item}</li>

})

}

</ul>

)

}

});

ReactDOM.render(<FilteredList/>, document.getElementById('mount-point'));

[**# Conclusion**](https://scotch.io/tutorials/learning-react-getting-started-and-concepts#conclusion)

Now that we have reviewed some React basics, take some time to check out the [React API](http://facebook.github.io/react/docs/top-level-api.html) and read up a bit on [JSX](http://facebook.github.io/react/docs/jsx-in-depth.html).

In the next installment of **Learning React** we will be integrating with Express to build an app that renders on the server side, as well as the client side, and uses socket.io to keep the two in sync.

Stay tuned!