

React's Five Fingers of Death. Master these five concepts, then master React.



Sacha Greif

Follow

Jan 5, 2017 · 10 min read



Side effects of learning React include glowing red hands and being a bad ass (photo credit)

A few years ago, my friend Sean started telling me how this brand new front-end library called React was going to take over the web. At first I dismissed it as just another framework fad. But then I started hearing

about React more and more, to the point where I felt like ignoring it just wasn't an option anymore.

Maybe you're in the same position I was in: you've been hearing about React left and right, but actually sitting down and *learning* it feels like such a chore.

The good news is that you can boil everything you need to know about React down to **five key concepts**.

Now don't get me wrong, this doesn't mean I can turn you into a React master instantly. But at least you'll understand all the major concepts, if you do decide to jump in.

The five key concepts are:

1. **Components**
2. **JSX**
3. **Props & State**
4. **The Component API**
5. **Component Types**

Before we get started, note that I originally learned React through Wes Bos's courses, and have included a few affiliate links to them. Whenever possible, I've also included links to free resources.

Oh, and my friend Sean? He's since moved on to much more cutting-edge-ier things. After all, React is *so* 2015.

Concept #1: How React components work

The first thing you need to know about React is that it's all about **components**. Your React codebase is basically just one large pile of big components that call smaller components.


But what's a component, you ask? A perfect example of a component is the common `<select>` HTML element. Not only does it come with its own visual output (the grey box, text label, and downward arrow that

make up the element itself)—it also handles its own opening and closing logic.

Examples

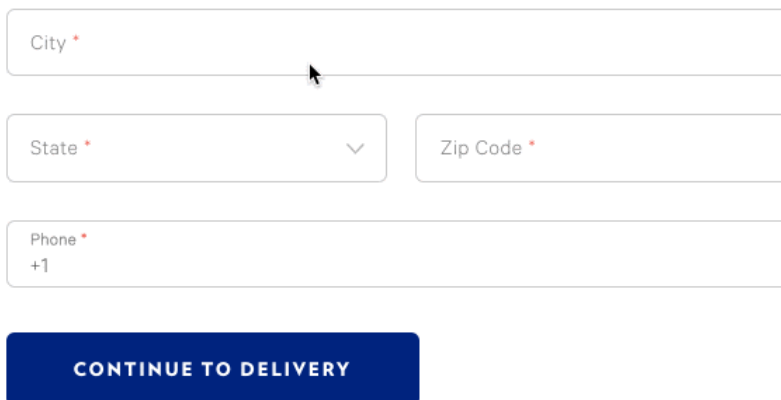
```
1 <!-- The second value will be selected initially -->
2 <select name="select">
3   <option value="value1">Value 1</option>
4   <option value="value2" selected>Value 2</option>
5   <option value="value3">Value 3</option>
6 </select>
```

Result



The classic `<select>`

Now imagine being able to build your own self-contained custom `<select>`, with its own style and behavior:



A fancier version of the good old `<select>`

Well, that's exactly what React lets you do. A React component is a single object that not only outputs HTML like a traditional template would, but also includes all the code needed to *control* that output.

In practice, the most common way to write React components is as an **ES6 class** containing a `render` method that returns HTML. (There's also a super-secret *functional* way, but you'll have to wait until concept #4 to learn about it):

```
class MyComponent extends React.Component {  
  
  render() {  
    return <p>Hello World!<p>;  
  }  
  
}
```

Concept #2: How JSX works

As you can see, the component approach means that both HTML and JavaScript code live in the same file. React's secret weapon to achieve this unholy alliance is the JSX language (where "X" stands for "XML").

JSX might seem awkward at first, but you get used to it pretty fast.

Yes, I know. We've all been taught to maintain a strong separation between HTML and JavaScript. But it turns out that relaxing these rules a bit can actually do wonders for your front-end productivity.

For example, since you now have the full power of JavaScript at your disposal, here's how you can display the current date by inserting a snippet of JavaScript in your HTML using `{...}` :

```
class MyComponent extends React.Component {  
  
  render() {  
    return <p>Today is: {new Date()}</p>;  
  }  
  
}
```

This also means that you'll use plain JavaScript for `if` statements or loops, rather than some kind of template-specific syntax. JavaScript's ternary operator comes in especially handy here:

```
class MyComponent extends React.Component {  
  
  render() {  
    return <p>Hello {this.props.someVar ? 'World' :  
    'Kitty'}</p>;  
  }  
  
}
```

And by the way, if you need to brush up on the newest points of JavaScript syntax, I recommend ES6 for Everyone by Wes Bos (if you like videos) or Practical ES6 by Nicolas Bevacqua (if you prefer reading).

Concept #3: How Props & State work

Maybe you've been wondering where the `this.props.someVar` variable above is coming from.

If you've ever written a line of HTML, you're probably familiar with HTML attributes like the `<a>` tag's `href`. In React, attributes are known as **props** (short for "properties"). Props are how components talk to each other.

```
class ParentComponent extends React.Component {  
  
  render() {  
    return <ChildComponent message="Hello World"/>;  
  }  
  
}  
  
class ChildComponent extends React.Component {  
  
  render() {  
    return <p>And then I said, "{this.props.message}"</p>;  
  }  
  
}
```

```
}
```

Because of this, React's data flow is **unidirectional**: data can only go from parent components to their children, not the other way around.

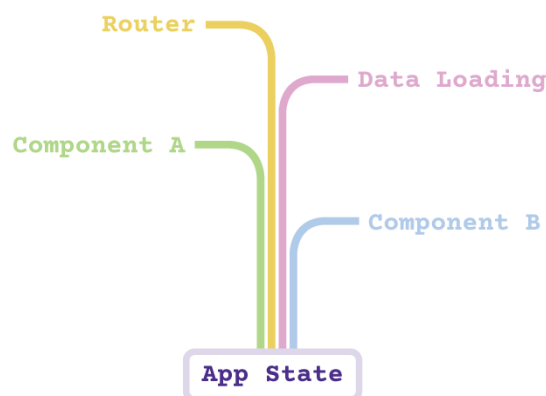
Sometimes though, a component needs to react to data that *doesn't* come from a parent component (such as user input for example). And this is where the **state** comes in.

A good metaphor to understand the difference between props and state would be the Etch-A-Sketch. Unlike things like the body color and dial position of the Etch-A-Sketch tablet (**props**), the drawing itself (**state**) is not an inherent property of the Etch-A-Sketch. It's just the temporary result of user input.



Pictured here: your typical React component

Note that a component's **state** can also be passed on to its own children as a **prop**. You can think of this as a big river flowing downhill, with the router, data layer, and various components each adding their own little stream of data to form the main app state.



Inside a component, state is managed using the `useState` function, which is often called inside an event handler:

```
class MyComponent extends React.Component {  
  
  handleClick = (e) => {  
    this.setState({clicked: true});  
  }  
  
  render() {  
    return <a href="#" onClick={this.handleClick}>Click  
me</a>;  
  }  
  
}
```

In practice, the vast majority of data in a React app will be a **prop**. It's only when you need to accept user input that you'll use **state** to handle the change.

Note that we're using a fat arrow here to take care of binding the `handleClick` handler. You can [learn more about this technique here](#).

Concept #4: How the Component API works

We've already mentioned `render` and `setState`, which are both part of a small set of component API methods. Another useful one is the `constructor`, which you can use to initialize your state and bind methods.

Apart from these three functions, React also provides a set of callbacks triggered at various points during the component's lifecycle (before loading, after loading, after unmounting, and so on). Unless you're doing some advanced React voodoo, you'll probably almost never need to worry about these.

If this section seems short, it's because learning React is actually much more about mastering programming and architectural concepts rather than learning a set of boring API methods. This is what makes it so refreshing!

Concept #5: How Component Types work

We've seen how to use classes to define a component:

```
class MyComponent extends React.Component {  
  
  render() {  
    return <p>Hello World!<p>;  
  }  
  
}
```

And we've also talked about the component methods supported by these classes. Now forget all about them! More and more, people are writing React components as **functional components**.

A functional component is a function that takes a `props` object as argument, and returns a bunch of HTML. Almost like a traditional

template, with the key difference that you can still use whatever JavaScript code you need inside that function:

```
const myComponent = props => {  
  
  return <p>Hello {props.name}! Today is {new Date()}.</p>  
  
}
```

The consequence of using the functional component syntax is that you lose access to the component methods we just talked about. But it turns out that in practice that's perfectly fine, since the vast majority of your components probably won't need them.

By the way, one of these methods is `setState`, and this means functional components cannot have state. For that reason they're often referred to as **stateless** functional components.

Since functional components require much less boilerplate code, it makes sense to use them whenever possible. For this reason, most React apps contain a healthy mix of both syntaxes.

Note that there's also a third, legacy syntax using the `createClass` function. But anybody using it should be shamed and called names for daring to still be using coding patterns from 18 months ago:

```
var Greeting = React.createClass({  
  
  render: function() {  
    return <h1>Hello, {this.props.name}</h1>;  
  }  
  
});
```

Concept #6: How Component Roles work

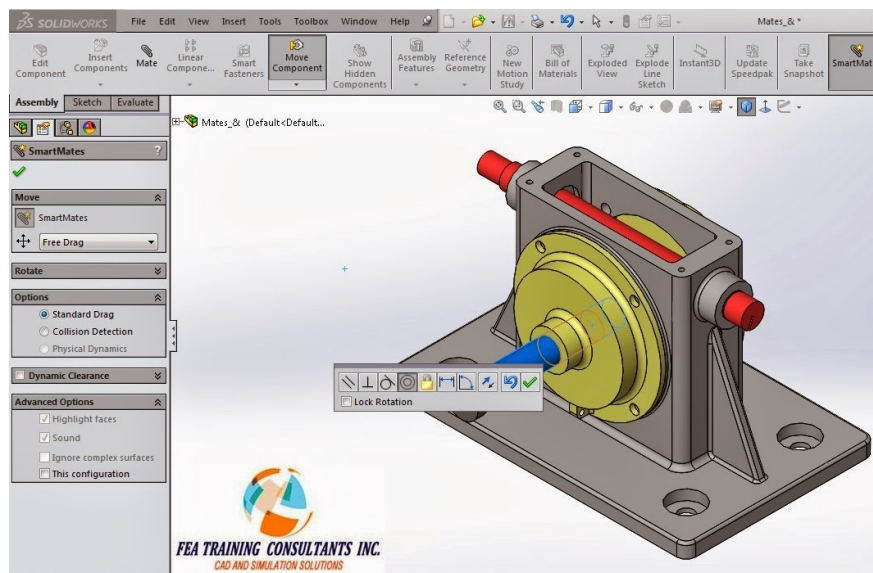
OK, I lied. There are actually six things, not five. But what can I say, the movie isn't called "Six Fingers Of Death." Although now that I think

about it, that sounds like it'd be a pretty cool movie, probably involving some kind of alien kung-fu master seeking revenge.

But back to the topic at hand. Now come the boring architectural concepts I was talking about. So if none of this makes sense feel free to come back once you've had a chance to play with React some more.

After using React for a while, people started seeing two distinct “flavors” of code appear in their components: one flavor was concerned with **UI** logic such as showing and hiding thing. And the other was all about **data** logic, such as loading data from your server.

This led to the distinction between **container** and **presentational** components (also sometimes known as “**smart**” and “**dumb**” components). Container components should handle your data, but—and this is the important part—**not your UI**. Presentational components are just the opposite.



Google image result for “smart component”. I have no clue what this is. Image credit

In other words, in the classic to-do list example, one component will load the data, and then pass that data on to a *different* component that will be in charge of outputting the actual HTML markup and handling local state changes.

This is very similar to the view/controller pattern you might be familiar with from your back-end developer days. (*'member Rails? 'member*

Django?)

The container/presentational distinction was popularized in [this blog post by Dan Abramov](#) (the creator of Redux), and I recommend checking it out if you want to dig deeper.

Higher-Order Components

Before we wrap things up, we should talk a bit about a type of container components known as **higher-order components** (often shortened as HoCs).

A HoC is a component that you can **wrap around** another component to pass it special props, and it's typically created using a **higher-order component factory function**. Note that people commonly refer to the *function* itself as a "**HoC**", which might not be 100% correct technically, but isn't a big deal in practice.

As an example, calling React Router's `withRouter` factory function on `<MyComponent>` will wrap it in a new `<withRouter(MyComponent) />` component that passes the `Router` prop to the afore-mentioned `<MyComponent>` .

You can think of a HoC function as a golf caddie that follows their golfer around and hands them the club they need it. By themselves, the caddie can't actually *do* anything with the golf clubs. They're just there to give the golfer access to more tools.



"Hand me the Router prop, James!"

HoCs are a very powerful concept. For example, the Recompose library even lets you handle state changes through HoCs. In other words, you can now manage state without having to involve any ES6 class-based components.

With HoC composition becoming so common, it seems like React might be moving away from the ES6 class syntax and more towards a purely functional approach. Interesting times!

Recap

So let's recap what we've just learned:

- A React codebase is made up of components.
- These components are written using JSX.
- Data flows from parent to children, except when it comes to `state`, which originates inside a component.
- Components possess a small set of lifecycle and utility methods.
- Components can also be written as pure functions.
- You should keep data logic and UI logic in separate components.
- Higher-order components are a common pattern for giving a component access to new tools.

Believe it or not, we've just covered 90% of the knowledge used by a React developer on a daily basis. No matter how abstract or obscure the pattern, everything in React can always be boiled down to functions and props.

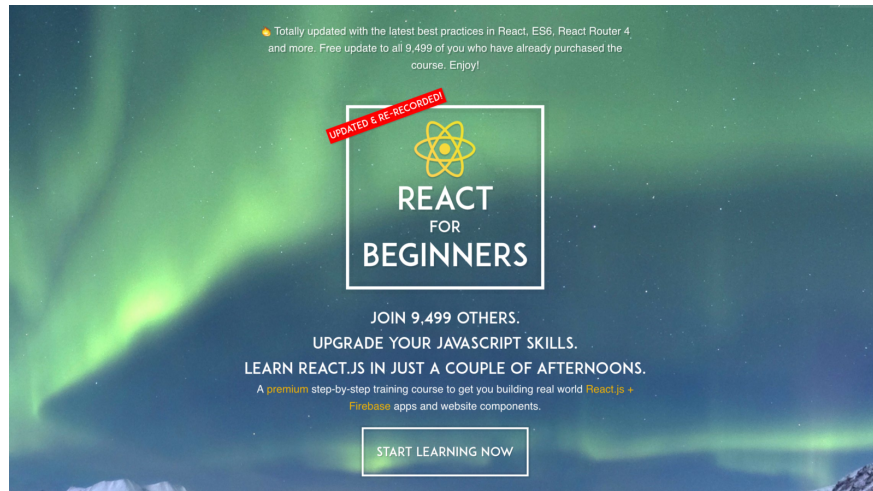
Once you truly understand this, React will stop being scary. You'll be able to see patterns in the code, understand new codebases at a glance, and only then will you be able to proudly proclaim:

"Pfff! React is so 2015!"

Going Further

If I've managed to convince you that React isn't so bad, you might want to take a stab at learning it properly. If so, I can't recommend the React

for Beginners video course enough. It's how I learned React myself, and it's actually just been updated to cover all the cool new stuff like functional stateless components:



Don't let the "artistic" background choice fool you: this is high-quality material

If you don't want your hard-earned dollars to finance the nefarious React lobby (I heard Dan Abramov is onto his third yacht), you can also learn for free by checking out [this huge list of React resources](#).

And if you need to put all this newly-acquired knowledge in practice by contributing to a cool React open-source project, check out [Telescope Nova](#). It's the easiest way to quickly create a full-stack React + GraphQL app, complete with user accounts, forms, and data loading out of the box. And did I mention we're looking for contributors?

Finally, if you've enjoyed this article, please share it and recommend it (that little green heart just below). And please [let me know on Twitter](#) what you'd like me to write about next!

