9 things every React.js beginner should know

24th January 2016

I've been using React.js for about 6 months now. In the grand scheme of things that's not very long at all, but in the ever-churning world of JavaScript frameworks, that just about qualifies you as a bearded elder! I've helped out a few people lately with React starter tips, so I thought it would be a good idea to write some of them up here to share more broadly. These are all either things that I wish I'd known when I started out, or things that really helped me 'get' React.

I'm going to assume that you know the absolute basics; if the words component, props, or state are unfamiliar to you, then you might want to read the official Getting started or Tutorial pages. I'm also going to use JSX, because it's a much more succinct and expressive syntax for writing components.

1. It's just a view library

Let's get the basics out of the way. React is not another MVC framework, or any other kind of framework. It's just a library for rendering your views. If you're coming from the MVC world, you need to realise that React is just the 'V', part of the equation, and you need to look elsewhere when it comes to defining your 'M' and 'C', otherwise you're going to end up with some really yucky React code. More on that later.

2. Keep your components small

This might seem like an obvious one, but it's worth calling out. Every good developer knows that small classes/modules/whatever are easier to understand, test, and maintain, and the same is true of React components. My mistake when starting out with React was under-estimating just how small my components should be. Of course, the exact size will depend upon many different factors (including you and your team's personal preference!), but in general my advice would be to make your components significantly smaller than you think they need to be. As an example, here is the component that shows my latest blog entries on the home page of my website:

```
const LatestPostsComponent = props => (
  <section>
    <div><h1>Latest posts</h1></div>
    <div>
      { props.posts.map(post => <PostPreview key={post.slug} post={post}/>) }
    </div>
  </section>
);
```

The component itself is a `<section>`, with only 2 `<div>`s inside it. The first one has a heading, and second one just maps over some data, rendering a `<PostPreview>` for each element. That last part, extracting the `<PostPreview>` as its own component, is the important bit. I think this is a good size for a component.

3. Write functional components

Previously there were two choices for defining React components, the first being `React.createClass()`:

```
const MyComponent = React.createClass({
  render: function() {
    return <div className={this.props.className}/>;
  }
});
```

... and the other being ES6 classes:

```
class MyComponent extends React.Component {
  render() {
    return <div className={this.props.className}/>;
  }
}
```

React 0.14 introduced a new syntax for defining components, as functions of props:

```
const MyComponent = props => (
  <div className={props.className}/>
);
```

This is by far my favourite method for defining React components. Apart from the more concise syntax, this approach can really help make it obvious when a component needs to be split up. Let's revisit the earlier example, and imagine that we hadn't split it up yet:

```
class LatestPostsComponent extends React.Component {
  render() {
    const postPreviews = renderPostPreviews();

    return (
      <section>
        <div><h1>Latest posts</h1></div>
        <div>
```

```
          { postPreviews }
        </div>
      </section>
    );
  }

  renderPostPreviews() {
    return this.props.posts.map(post => this.renderPostPreview(post));
  }

  renderPostPreview(post) {
    return (
      <article>
        <h3><a href={`/post/${post.slug}`}>{post.title}</a></h3>
        <time pubdate><em>{post.posted}</em></time>
        <div>
          <span>{post.blurb}</span>
          <a href={`/post/${post.slug}`}>Read more...</a>
        </div>
      </article>
    );
  }
}
```

This class isn't too bad, as far as classes go. We've already extracted a couple of methods out of the render method, to keep things small and well-named. And we've encapsulated the idea of rendering the latest post previews pretty well. Let's rewrite the same code using the functional syntax:

```
const LatestPostsComponent = props => {
  const postPreviews = renderPostPreviews(props.posts);

  return (
    <section>
      <div><h1>Latest posts</h1></div>
      <div>
        { postPreviews }
      </div>
    </section>
  );
};
```

```
const renderPostPreviews = posts => (
  posts.map(post => this.renderPostPreview(post))
);

const renderPostPreview = post => (
  <article>
    <h3><a href={`/post/${post.slug}`}>{post.title}</a></h3>
    <time pubdate><em>{post.posted}</em></time>
    <div>
      <span>{post.blurb}</span>
      <a href={`/post/${post.slug}`}>Read more...</a>
    </div>
  </article>
);
```

The code is mostly the same, other than that we now have bare functions, rather than methods in a class. However, for me, that makes a big difference. In the class-based example, my eyes see `class LatestPostsComponent {`, and naturally scan all the way down to the closing curly bracket, concluding "that's the end of the class, and so the end of the component". By comparison, when I read the functional component, I see `const LatestPostsComponent = props => {`, and scan only to the end of that function. "There's the end of the function, and so the end of the component", I think to myself. "But wait, what's all this other code sitting outside of my component, in the same module? Aha! It's another function that takes data and renders a view! I can extract that out into a component all of it's own!"

Which is a very long-winded way of saying that functional components help us identify ways to follow point #2.

There are also optimisations coming to React in the future, which will make functional components more efficient than class-based ones. (**Update:** It turns out that the performance implications of functional components are much more complicated than I realised. I still recommend writing functional components by default, but if performance is a big concern, then you should read and understand this and this, and figure out what works best for you.)

It's important to note that functional components have a few 'limitations', which I consider to be their greatest strengths. The first is that a functional component cannot have a `ref` assigned to it. While a `ref` can be a convenient way for a component to 'look up' it's children and communicate with them, my feeling is that this is The Wrong Way to write React. `ref`s encourage a very imperative, almost jquery-like way of writing components, taking us away from the functional, one-way data flow philosophy for which we chose React in the first place!

The other big difference is that functional components cannot have state attached to them, which is also a huge advantage, because my next tip is to...

4. Write stateless components

I would have to say that by far, the vast, vast majority of pain that I've felt when writing React-based apps, has been caused by components that have too much state.

State makes components difficult to test

There's practically nothing easier to test than pure, data-in data-out functions, so why would we throw away the opportunity to define our components in such a way by adding state to them? When testing stateful components, now we need to get our components into "the right state" in order to test their behaviour. We also need to come up with all of the various combinations of state (which the component can mutate at any time), and props (which the component has no control over), and figure out which ones to test, and how to do so. When components are just functions of input props, testing is a lot easier. (More on testing later).

State makes components difficult to reason about

When reading the code for a very stateful component, you have to work a lot harder, mentally, to keep track of everything that's going on. Questions like "Has that state been initialised yet?", "What happens if I change this state here?", "How many other places change this state", "Is there a race condition on this state?", to mention a few, all become very common. Tracing through stateful components to figure out how they behave becomes very painful.

State makes it too easy to put business logic in the component

Searching through components to determine behaviour is not really something we should be doing anyway. Remember, React is a **view library**, so while *render* logic in the components is OK, *business* logic is a massive code smell. But when so much of your application's state is right there in the component, easily accessible by `this.state`, it can become really tempting to start putting things like calculations or validation into the component, where it does not belong. Revisiting my earlier point, this makes testing that much harder - you can't test render logic without the business logic getting in the way, and vice versa!

State makes it difficult to share information to other parts of the app

When a component owns a piece of state, it's easy to pass it further down the component hierarchy, but any other direction is tricky.

Of course, sometimes it makes sense for a particular component to totally own a particular piece of state. In which case, fine, go ahead use `this.setState`. It's a legitimate part of the React component API, and I don't want to give the impression that it should be off-limits. For example, if a user is typing into a field, it might not make sense to expose every keypress to the whole app, and so the field may track its own intermediate state until a blur event happens, whereupon the final input value is sent outwards to become state stored somewhere else. This scenario was mentioned to me by a colleague recently, and I think it's a great example.

Just be very conscious of every time you add state to a component. Once you start, it can become very easy to add 'just one more thing', and things get out of control before you know it!

5. Use Redux.js

In point #1, I said that React is just for views. The obvious question then is, "Where do I put all my state and logic?" I'm glad you asked!

You may already be aware of Flux, which is a style/pattern/architecture for designing web applications, most commonly ones that use React for rendering. There are several frameworks out there that implement the ideas of Flux, but without a doubt the one that I recommend is Redux.js*.

I'm thinking of writing a whole separate blog post on the features and merits of Redux, but for now I'll just recommend you read through the official tutorial at the above link, and provide this very brief description of how Redux works:

1. **Components** are given callback functions as props, which they call when a UI event happens
2. Those callbacks **create and dispatch actions** based on the event
3. **Reducers** process the **actions**, computing the new **state**
4. The new **state** of the whole application goes into a **single store**.
5. **Components** receive the new state as **props** and re-render themselves where needed.

Most of the above concepts are not unique to Redux, but Redux implements them in a very clean and simple way, with a tiny API. Having switched a decent sized project over from Alt.js to Redux, some of the biggest benefits I've found are:

- The reducers are pure functions, which simply do `oldState + action = newState`. Each reducer computes a separate piece of state, which is then all composed together to form the whole application. This makes all your business logic and state transitions *really* easy to test.
- The API is smaller, simpler, and better-documented. I've found it much quicker and easier to learn all of the concepts, and therefore much easier to understand the flow of actions and information in my own projects.
- If you use it the recommended way, only a very small number of components will depend upon Redux; all the other components just receive state and callbacks as props. This keeps the components very simple, and reduces framework lock-in.

There are a few other libraries that complement Redux extremely well, which I also recommend you use:

- Immutable.js - Immutable data structures for JavaScript! Store your state in these, in order to make sure it isn't mutated where it shouldn't be, and to enforce reducer purity.
- redux-thunk - This is used for when your actions need to have a side effect other than updating the application state. For example, calling a REST API, or setting routes, or even dispatching other actions.
- reselect - Use this for composable, lazily-evaluated, views into your state. For example, for a particular component you might want to:
  - inject only the relevant part of the global state tree, rather than the whole thing
  - inject extra derived data, like totals or validation state, without putting it all in the store

You don't necessarily need all of these from day one. I would add Redux and Immutable.js as soon you have any state, reselect as soon you have derived state, and redux-thunk as soon as you have routing or asynchronous actions. Adding them sooner rather than later will save you the effort of retro-fitting them in later on.

* Depending on who you ask, Redux may or may not be 'true' Flux. Personally I feel that it's aligned well-enough with the core ideas to call it a Flux framework, but the argument is a semantic one anyway.

6. Always use propTypes

propTypes offer us a really easy way to add a bit more type safety to our components. They look like this:

```
const ListOfNumbers = props => (
  <ol className={props.className}>
    {
      props.numbers.map(number => (
        <li>{number}</li>)
      )
    }
  </ol>
);

ListOfNumbers.propTypes = {
  className: React.PropTypes.string.isRequired,
  numbers: React.PropTypes.arrayOf(React.PropTypes.number)
};
```

When in development (not production), if any component is not given a required prop, or is given the wrong type for one of its props, then React will log an error to let you know. This has several benefits:

- It can catch bugs early, by preventing silly mistakes

- If you use `isRequired`, then you don't need to check for `undefined` or `null` as often
- It acts as documentation, saving readers from having to search through a component to find all the props that it needs

The above list looks a bit like one you might see from someone advocating for static typing over dynamic typing. Personally, I usually prefer dynamic typing for the ease and speed of development it provides, but I find that propTypes add a lot of safety to my React components, without making things any more difficult. Frankly I see no reason *not* to use them all the time.

One final tip is to make your tests fail on any propType errors. The following is a bit of a blunt instrument, but it's simple and it works:

```
beforeAll(() => {
  console.error = error => {
    throw new Error(error);
  };
});
```

7. Use shallow rendering

Testing React components is still a bit of a tricky topic. Not because it's hard, but because it's still an evolving area, and no single approach has emerged as the 'best' one yet. At the moment, my go-to method is to use shallow rendering and prop assertions.

Shallow rendering is nice, because it allows you to render a single component completely, but without delving into any of its child components to render those. Instead, the resulting object will tell you things like the type and props of the children. This gives us good isolation, allowing testing of a single component at a time.

There are three types of component unit tests I find myself most commonly writing:

Render logic

Imagine a component that should conditionally display either an image, or a loading icon:

```
const Image = props => {
  if (props.loading) {
    return <LoadingIcon/>;
  }

  return <img src={props.src}/>;
};
```

We might test it like this:

```
describe('Image', () => {
  it('renders a loading icon when the image is loading', () => {
    const image = shallowRender(<Image loading={true}/>);

    expect(image.type).toEqual(LoadingIcon);
  });

  it('renders the image once it has loaded', () => {
    const image = shallowRender(<Image loading={false}
src="https://example.com/image.jpg"/>);

    expect(image.type).toEqual('img');
  });
});
```

Easy! I should point out that the API for shallow rendering is slightly more complicated than what I've shown. The `shallowRender` function used above is our own helper, which wraps the real API to make it easier to use.

Revisiting our `ListOfNumbers` component above, here is how we might test that the map is done correctly:

```
describe('ListOfNumbers', () => {
  it('renders an item for each provided number', () => {
    const listOfNumbers = shallowRender(<ListOfNumbers className="red" numbers={[3,
4, 5, 6]}/>);

    expect(listOfNumbers.props.children.length).toEqual(4);
  });
});
```

Prop transformations

In the last example, we dug into the children of the component being tested, to make sure that they were rendered correctly. We can extend this by asserting that not only are the children there, but that they were given the correct props. This is particulary useful when a component does some transformation on its props, before passing them on. For example, the following component takes CSS class names as an array of strings, and passes them down as a single, space-separated string:

```
const TextWithArrayOfClassNames = props => (
  <div>
    <p className={props.classNames.join(' ')}>
     {props.text}
    </p>
  </div>
);

describe('TextWithArrayOfClassNames', () => {
  it('turns the array into a space-separated string', () => {
    const text = 'Hello, world!';
    const classNames = ['red', 'bold', 'float-right'];
    const textWithArrayOfClassNames = shallowRender(<TextWithArrayOfClassNames
text={text} classNames={classNames}/>);

    const childClassNames =
textWithArrayOfClassNames.props.children.props.className;
    expect(childClassNames).toEqual('red bold float-right');
  });
});
```

One common criticism of this approach to testing is the proliferation of `props.children.props.children`... While it's not the prettiest code, personally I find that if I'm being annoyed by writing `props.children` too much in the one test, that's a sign that the component is too big, complex, or deeply nested, and should be split up.

The other thing I often hear is that your tests become too dependant on the component's internal implementation, so that changing your DOM structure slightly causes all of your tests to break. This is definitely a fair criticism, and a brittle test suite is the last thing that anyone wants. The best way to manage this is to (wait for it) keep your components small and simple, which should limit the number of tests that break due to any one component changing.

User interaction

Of course, components are not just for display, they're also interactive:

```
const RedInput = props => (
  <input className="red" onChange={props.onChange} />
)
```

Here's my favourite way to test these:

```
describe('RedInput', () => {
  it('passes the event to the given callback when the value changes', () => {
    const callback = jasmine.createSpy();
    const redInput = shallowRender(<RedInput onChange={callback}/>);

    redInput.props.onChange('an event!');
    expect(callback).toHaveBeenCalledWith('an event!');
  });
});
```

It's a bit of a trivial example, but hopefully you get the idea.

Integration testing

So far I've only covered unit testing components in isolation, but you're also going to want some higher level tests in order to ensure that your application connects up properly and actually works. I'm not going to go into too much detail here, but basically:

1. 1. assert on their HTML attributes or contents, or
   2. simulate DOM events and then assert on the side effects (DOM or route changes, AJAX calls, etc)

On TDD

In general, I don't use TDD when writing React components.

When working on a component, I often find myself churning its structure quite a bit, as I try to land on the simplest HTML and CSS that looks right in whatever browsers I need to support. And because my component unit testing approach tends to assert on the component structure, TDD would cause me to be constantly fixing my tests as I tweak the DOM, which seems like a waste of time.

The other factor to this is that the components should be so simple that the advantages of test-first are diminished. All of the complex logic and transformations are pulled out into action creators and reducers, which is where I can (and do) reap the benefits of TDD.

Which brings me to my final point about testing. In this whole section, I've been talking about testing the components, and that's because there's no special information needed for testing the rest of a Redux-based app. As a framework, Redux has very little 'magic' that goes on behind the scenes, which I find reduces the need for excessive mocking or other test boilerplate. Everything is just plain old functions (many of them pure), which is a real breath of fresh air when it comes to testing.

8. Use JSX, ES6, Babel, Webpack, and NPM

The only React-specific thing here is JSX. For me JSX is a no-brainer over manually calling `React.createElement`. The only disadvantage is that we add a small amount of build-time complexity, which is easily solved with Babel.

Once we've added Babel though, there's no reason not to go all out and use all the great ES6 features, like constants, arrow functions, default arguments, array and object destructuring, spread and rest operators, string interpolation, iterators and generators, a decent module system, etc. It really feels like JavaScript is beginning to 'grow up' as a language these days, as long as you're prepared to spend a little bit of time setting up the tools.

We round things out with Webpack for bundling our code, and NPM for package management, and we're now fully JavaScript buzzword compliant :)

9. Use the React and Redux dev tools

Speaking of tooling, the development tools for React and Redux are pretty awesome. The React dev tools let you inspect the rendered tree of React elements, which is incredibly useful for seeing how things actually look in the browser. The Redux dev tools are even more impressive, letting you see every action that has happened, the state changes that they caused, and even giving you the ability to travel back in time! You can add it either as a dev dependency, or as a browser extension.

You can also set up hot module replacement with webpack, so that your page updates as soon as you save your code - no browser refresh required. This makes the feedback loops a lot more efficient when tweaking your components and reducers.

That's it!

Hopefully that will give you a bit of a head-start on React, and help you avoid some of the more common mistakes. If you enjoyed this post, you might like to follow me on Twitter, or subscribe to my RSS feed.

Thanks for reading!