Understanding React

```
render()
ReactDOM.render(
  element,
  container,
  [callback]
)
```

At the most basic level Reacts uses the `Render` method to stamp out components to the DOM in the supplied container and return a reference to the component (or returns null for stateless components).

If the React element was previously rendered into container, this will perform an update on it and only mutate the DOM as necessary to reflect the latest React element.

If the optional callback is provided, it will be executed after the component is rendered or updated.

For example to render the following text inside the #app element:

```
<p>Hello World</p>
```

We use the following React code

```
<!DOCTYPE html>
<html>
  <head><html>title>Hello React</title>
    <meta charset="utf-8">
 </title>
  <body>
    <div id="app">
      <body><!-- my app renders here -->/div>
    <script src="react/build/react.js"></script>
    <script src="react/build/react-dom.js"></script>
    <script>
      ReactDOM.render(
        React.DOM.p(null, "Hello W<script src="react/build/react.js">Eleme
      );
    </script>
  </body>
</html>
```

> Note:
>
> ReactDOM.render() controls the contents of the container node you pass in. Any existing DOM elements inside are replaced when first called. Later calls use React's DOM diffing algorithm for efficient updates.
>
> ReactDOM.render() does not modify the container node (only modifies the children of the container). It may be possible to insert a component to an existing DOM node without overwriting the existing children.
>
> ReactDOM.render() currently returns a reference to the root ReactComponent instance. However, using this return value is legacy and should be avoided because future versions of React may render components asynchronously in some cases. If you need a reference to the root ReactComponent instance, the preferred solution is to attach a callback ref to the root element.

If we have common properties for the elements we create we can use the first parameter to `React.DOM.*` to pass the parameters we need to. To make the example more elaborate we can add attributes

```
const attributes = {
  id: "introduction"
}

ReactDOM.render(
React.DOM.p(attributes, "Hello World!"),
document.getElementById("app")
);
```

I’m old school, progressive enhancement, type of guy and when I don’t see content in a page I get worried. I don’t see any content in the page and wonder what will happen if Javascript times out or doesn’t load for any reason.

I know what you’ll tell me: It’s absurd to think that Javascript is disabled or will not load… but what happens if the user is in in an area of low wifi connectivity or where it’ll take so long that the user will choose not to engage with the application is just as bad as having Javascript disabled. We haven’t considered the possibility of the user disabling Javascript to get better performance or them being in a zone with poor wireless connectivity.

The content of the page is rendered client-side after all the React libraries have loaded. There are two things that worry me. OK, I’ll give you that this is a bare bones application built by someone who wants to learn React but it’s still troubling that there is no mention of accessibility anywhere in the React docs or in most conversations on the web about React.

The consensus seems to be that developers should work accessibility into their regular workflow but without guidance how to properly implement accessibility in React apps. What I don’t understand is why have a React Native accessibility checker but not one for React web?

React supports all data-* and aria-* attributes so we should be able to build the accessibility. My fear is that, without tools, the number of people who build accessible applications decreases. React native has an accessibility checking tool but it has not been ported to React.

And even in the best situation accessibility issues are not obvious or apparent unless you’re an expert. In the video below Marcy Sutton, an accessibility specialist covers some issues in React accessibility.

For more information see Client Rendered Accessibility by Marcy Sutton in Smashing Magazine.

Element versus components

So I've created my first React element, rendered it and complained about the potential access and accessibility issues present in such an approach we'll move on to creating React components. Again, I don't profess to be a React expert but someone who needs to know enough so that if asked I can provide a coherent answer.

Elements are hard to reuse and debug. That's where components come in. We can use React's `createClass` method or we can create classes directly.

An example using `React.createClass` looks like this

```
var Greeting = React.createClass({
  render: function() {
    return React.DOM.h1(null, "My name is " + this.props.name);
  }
});

ReactDOM.render(
  React.createElement(Greeting, {
    name: "Bob"
  }),
  document.getElementById("app")
);
```

The same component created as an ES6 class looks like this:

```
class Greeting extends React.Component {
  constructor(props) {
    super(props);
  }

  ReactDOM.render(
    <h1>name is  {this.props.name}</h1>);
  )
}

export default Greeting;
```

Server versus Client rendering

Does server-side rendering help with concerns like the ones I have? It might but I don't think it's a good solution. It moves part of the workload to the server but it still requires pushing the rendered content, the application state and any javascript library needed on the client… would love to see metrics of how server-side-rendering compares to client-side rendering across devices and applications

> In Server-Side Rendering with React, Node And Express

Links and Resources

- https://react.rocks/
- Look for react components in https://github.com/davidtheclark