React!

A javascript library for building user interfaces!

What you'll need:

```
- NodeJS
```

Tools you'll want:

```
- VS Code
```

Getting Started:

We're going to start with create react app : Create React App is an

---

Create an App!

```
npx create-react-app helloworld
cd helloworld
```

While there are a lot of files contained in here, just take a look at the package json and type the starting script:

```
npm start
```

And assuming you've got nothing else running on port 3000, go to: http://localhost:3000/

Delete the src!

There's too much to start with. Let's delete everything:

```
rm src/*
```

And create a new landing page: `index.js`

```
touch src/index.js
```

New SRC Materials:

Inside of index.js, type:

```
import React from "react";
import ReactDOM from "react-dom";

function HelloWorld() {
  return (
    <div>HELLO WORLD (or literally anything else you want to write here)
</div>
  );
}

ReactDOM.render(<HelloWorld />, document.querySelector("#root"));
```

What's going on??

Inside the `ReactDOM.render()` we've got a React element for the first parameter, and a DOM element for the second.

We have a component:

```
<HelloWorld />
```

This is a functional component (also known as a stateless function). There are class components, but those are deprecated.

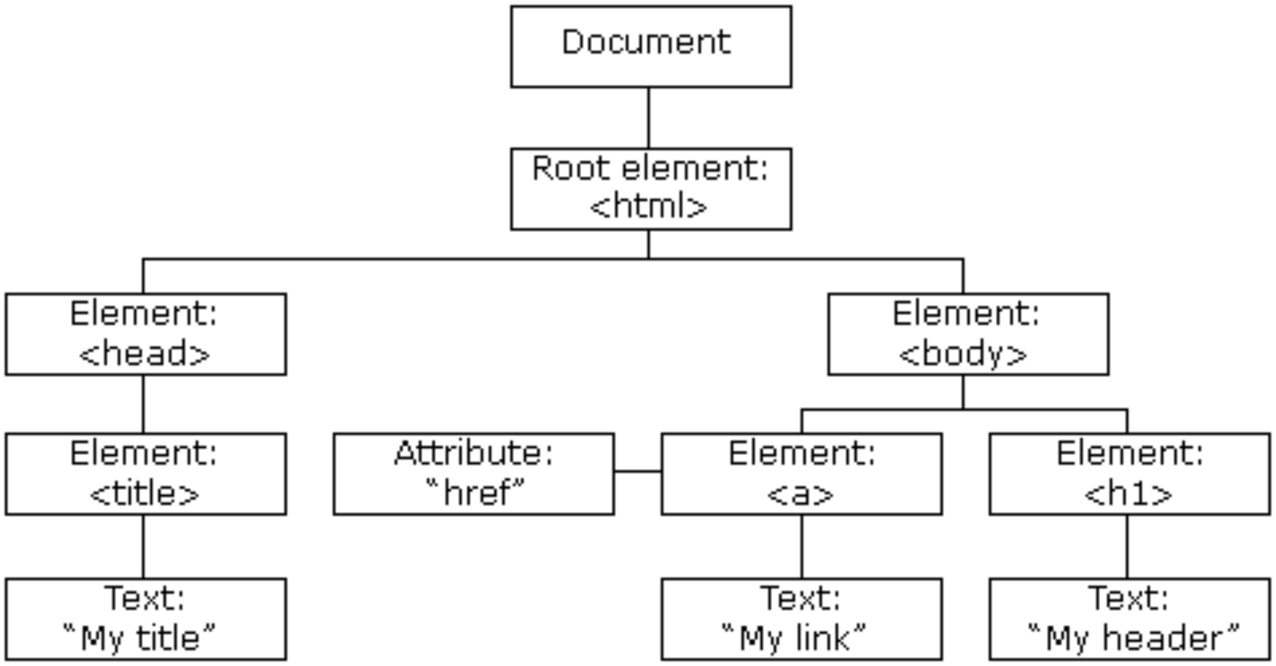
```
class HelloWorld extends React.Component {
  render() {
    return <div>Hello World!</div>;
  }
}
```

Below the component, we have:

```
document.qeuerySelector("#root");
```

React uses the virtual dom. This creates a component hierarchy, renders those components, and then inserts them into the DOM where you tell the function. In our case, we're inserting our `<HelloWorld/>` at `#root`.

THE DOM!



JSX

Remember our:

```
function HelloWorld() {
  return (
    <div>HELLO WORLD (or literally anything else you want to write here)</div>
  );
}
```

The in line `HTML` looking code is our `JSX`.

JSX

JSX is really a syntactic sugar for React's `createElement()`:

```
React.createElement(component, props, ...children);
```

So, our Hello world function is:

```
React.createElement(
  "div",
  null,
  "HELLO WORLD (or literally anything else you want to write here)"
);
```

JSX

```
import React from "react";
import ReactDOM from "react-dom";

const HelloWorld = () =>
  React.createElement(
    "div",
    null,
    "HELLO WORLD (or literally anything else you want to write here)"
  );

ReactDOM.render(<HelloWorld />, document.querySelector("#root"));
```

JSX

Notice that the children parameter looks like it has the spread operator. That meants that the component can have any number of children (remember the DOM).

```
React.createElement(component, props, ...children);
```

So we can write:

```
import React from "react";
import ReactDOM from "react-dom";

const HelloWorld = () =>
  React.createElement(
    "div",
    null,
    "HELLO WORLD",
    "(or literally anything else",
    " you want to write here)"
  );

ReactDOM.render(<HelloWorld />, document.querySelector("#root"));
```

JSX: Nesting:

Just like HTML, you'll want to have components inside of components:

```
import React from "react";
import ReactDOM from "react-dom";

const GoodBye = () => {
  return <span>Goodbye cruel world!</span>;
};

function HelloWorld() {
  return (
    <div>
      HELLO WORLD (or literally anything else you want to write here)
      <GoodBye />
    </div>
  );
}

ReactDOM.render(<HelloWorld />, document.querySelector("#root"));
```

JSX: Nesting

You may be wondering why we didn't just add `<GoodBye\>` after the `<div>HELLO WORLD (or literally anything else you want to write here)</div>`:

```
import React from "react";
import ReactDOM from "react-dom";

const GoodBye = () => {
  return <span>Goodbye cruel world!</span>;
};

function HelloWorld() {
  return (
    <div>HELLO WORLD (or literally anything else you want to write here)
</div>
    <GoodBye />
  );
}

ReactDOM.render(<HelloWorld />, document.querySelector("#root"));
```

JSX Nesting:

The error received was:

```
./src/index.js
  Line 11:5:  Parsing error: Adjacent JSX elements must be wrapped in an
enclosing tag. Did you want a JSX fragment <span>...</span>?

   9 |   return (
  10 |     <div>HELLO WORLD (or literally anything else you want to write
here)</div>
> 11 |     <GoodBye />
     |     ^
  12 |   );
  13 | }
  14 |
```

When looking at this, it really makes not real sense at first glance. But let's try rewriting this with our `React.createElement()` function:

```
import React from "react";
import ReactDOM from "react-dom";

const GoodBye = () =>
  React.createElement("fragment", null, "Goodbye cruel world!");

const HelloWorld = () => ({
    React.createElement(
      "div",
      null,
      "HELLO WORLD (or literally anything else you want to write here)"
    )
     <GoodBye />
})

ReactDOM.render(<HelloWorld />, document.querySelector("#root"));
```

OR =>

```
import React from "react";
import ReactDOM from "react-dom";

const HelloWorld = () => {
  return React.createElement(
    "div",
    null,
    "HELLO WORLD (or literally anything else you want to write here)"
  );

  React.createElement("fragment", null, "Goodbye cruel world!");
};

ReactDOM.render(<HelloWorld />, document.querySelector("#root"));
```

OR =>

It may make a little more sense like:

```
import React from "react";
import ReactDOM from "react-dom";

const Bonjour = () => {
  return <span>Bonjour</span>;
};

const Monde = () => {
  return <span>Monde</span>;
};

const FrenchHelloWorld = () => {
    return (
        <Bonjour/>
        <Monde />
    )
}
  ReactDOM.render(<FrenchHelloWorld />, document.querySelector("#root"));
```

What's happening with BonjourMonde?

```
import React from "react";
import ReactDOM from "react-dom";

const Bonjour = () => {
  return <span>Bonjour</span>;
};

const Monde = () => {
  return <span>Monde</span>;
};

const FrenchHelloWorld = () => {
    return (
        React.createElement(Bonjour, null)
        React.createElement(Monde, null)
    )
}
  ReactDOM.render(<FrenchHelloWorld />, document.querySelector("#root"));
```

Javascript does not allow for you to pass back tuples, so you'll need to wrap all of your returned JSX in either a fragment, or some other tag.

JSX- Wrap it with a Fragment

```
import React from "react";
import ReactDOM from "react-dom";

const Bonjour = () => {
  return <span>Bonjour</span>;
};

const Monde = () => {
  return <span>Monde</span>;
};

const FrenchHelloWorld = () => {
  return (
    <div>
      <Bonjour />
      <Monde />
    </div>
  );
};
ReactDOM.render(<FrenchHelloWorld />, document.querySelector("#root"));
```

The above code will render to the DOM as:

```
<div id="root">
  <div>
    <span>Bonjour</span>
    <span>Monde</span>
  </div>
</div>
```

Keeping your DOM Clean

It's not always the case that you'll want to have your DOM be filled with a bunch of wrapper elements such as `<div>`s. As of React 16.2, it's possible to use a `fragment` which acts as a disappearing wrapper:

```
import React from "react";
import ReactDOM from "react-dom";

const Bonjour = () => {
  return <span>Bonjour</span>;
};

const Monde = () => {
  return <span>Monde</span>;
};

const FrenchHelloWorld = () => {
  return (
    <>
      <Bonjour />
      <Monde />
    </>
  );
};
ReactDOM.render(<FrenchHelloWorld />, document.querySelector("#root"));
```

Now our DOM looks like:

```
<div id="root">
  <span>Bonjour</span>
  <span>Monde</span>
</div>
;
```

Adding Javascript inside JSX:

Sometimes you'll either want or need (many opinions about that `need`) to run some javascript inside your JSX.

```
import React from "react";
import ReactDOM from "react-dom";

const Bonjour = () => {
  return <span>Bonjour</span>;
};

const name = "Jacques";

const FrenchHelloWorld = () => {
  return (
    <>
      <Bonjour />
      name
    </>
  );
};
ReactDOM.render(<FrenchHelloWorld />, document.querySelector("#root"));
```

Adding Javascript inside JSX:

You can access the javascript values by wrapping them within curly braces:

```
import React from "react";
import ReactDOM from "react-dom";
```

```

const Bonjour = () => {
  return <span>Bonjour</span>;
};

const name = " Jacques";

const FrenchHelloWorld = () => {
  return (
    <>
      <Bonjour />
      {name}
    </>
  );
};
ReactDOM.render(<FrenchHelloWorld />, document.querySelector("#root"));
```

Adding Javascript inside JSX:

Note: Whatever values you wrap within the curly braces must resolve to an actual value, otherwise, nothing will return:

```
import React from "react";
import ReactDOM from "react-dom";

const Bonjour = () => {
  return <span>Bonjour</span>;
};

const FrenchHelloWorld = () => {
  return (
    <>
      <Bonjour />
      {console.log("Wahtever")}
    </>
  );
};
ReactDOM.render(<FrenchHelloWorld />, document.querySelector("#root"));
```

A good rule of thumb is: If you can assign it to a variable, then it's likely able to be rendered in JSX.

Adding Javascript inside JSX: Conditionals

Suppose you wanted to have some material that displayed conditionally?

You can't write:

```
import React from "react";
import ReactDOM from "react-dom";

const Bonjour = () => {
  return <span>Bonjour</span>;
};

const Hello = () => {
  return <span>HELLO</span>;

}

const isFrench = true;

const FrenchHelloWorld = () => {
  return (
    <>
      { if(isFrench) {
          <Bonjour />
      } else {
          <Hello />
      }}
    </>
  );
};
ReactDOM.render(<FrenchHelloWorld />, document.querySelector("#root"));
```

Adding Javascript inside JSX: Conditionals

There are ways to get around this:

```
import React from "react";
import ReactDOM from "react-dom";

const Bonjour = () => {
  return <span>Bonjour</span>;
};

const Hello = () => {
  return <span>HELLO</span>;
};

const isFrench = true;

const FrenchHelloWorld = () => {
  return <>{isFrench ? <Bonjour /> : <Hello />}</>;
};
ReactDOM.render(<FrenchHelloWorld />, document.querySelector("#root"));
```

Adding Javascript inside JSX: Conditionals

Another method of using conditionals within your application, such as a `login` button is to make use of `short circuiting`:

```
import React from "react";
import ReactDOM from "react-dom";

const isLoggedIn = false;
const username = "reallyCoolGuy88";

const LogInButton = () => (
  <button onClick={() => alert("Log in, bozo")}> DO THE LOG IN PLEASE
</button>
);

const DisplayLogin = () => {
  return <> {isLoggedIn ? username : <LogInButton />} </>;
};
ReactDOM.render(<DisplayLogin />, document.querySelector("#root"));
```

PROPS

When creating our own `HTML` esque components, we need a way to pass data to them. HTML elements take in attributes. For React, we'll take in properties, or `props`.

Props work just like a function parameter (in fact, on the component side of things, that's exactly what they are). When using JSX, however, you pass the data, just as if you're passing something to an attribute.

Unlike parameters in functions, props are read only and cannot be mutated. The flow of data always comes from the parent to the child.

Props: Passing and Reading

```
import React from "react";
import ReactDOM from "react-dom";

const Hello = props => {
  return <span>HELLO {props.name}</span>;
};

const SayHelloAndPassAProp = () => {
  const somebodysName = "Matt";

  return <Hello name={somebodysName} />;
};
ReactDOM.render(<SayHelloAndPassAProp />, document.querySelector("#root"));
```

Props: Passing and Reading

Programatically, this allows for the ability to pass elements, both from the parent level to child elements:

```
const Hello = props => {
  return <span>HELLO {props.name}</span>;
};

const isFrench = Math.random() >= 0.5;

const SayHelloAndPassAProp = () => {
  const somebodysName = "Matt";

  return <Hello name={somebodysName} />;
};
ReactDOM.render(<SayHelloAndPassAProp />, document.querySelector("#root"));
```

Props: Passing and Reading Props

```
import React from "react";
import ReactDOM from "react-dom";

const Hello = props => {
  return <span>HELLO {props.name}</span>;
};

const Bonjour = ({ name }) => {
  return <span> Bonjour {name} </span>;
};

const isFrench = Math.random() >= 0.5;

const SayHelloAndPassAProp = () => {
  const somebodysName = "Matt";

  return isFrench ? (
    <Bonjour name={somebodysName} />
  ) : (
    <Hello name={somebodysName} />
  );
};
ReactDOM.render(<SayHelloAndPassAProp />, document.querySelector("#root"));
```

{" "} Notice that the parameter of bonjour has been destructured to name? That's because the passed props act as the keys and the passed data acts as the values in a key value pairing.{" "}

Props: Behind the scenes

Taking a deeper look at props (via the `React.createElement()` function):

```
import React from "react";
import ReactDOM from "react-dom";

const Hello = props => {
  return React.createElement("span", null, "Hello " + props.name);
};

const Bonjour = ({ name }) => {
  return React.createElement("span", null, "Bonjour " + name);
};

const isFrench = Math.random() >= 0.5;

const SayHelloAndPassAProp = () => {
  const somebodysName = "Matt";

  return isFrench
```

```
    ? React.createElement(Bonjour, { name: somebodysName }, null)
    : React.createElement(Hello, { name: somebodysName }, null);
};
ReactDOM.render(
  React.createElement(SayHelloAndPassAProp, null, null),
  document.querySelector("#root")
);
```

Props: Small Componenets

What's highly recommended is using JSX elements with props to keep your workspace clean:

```
import React from "react";
import ReactDOM from "react-dom";

const Hello = props => {
  return <span>HELLO {props.name}</span>;
};

const Bonjour = ({ name }) => {
  return <span> Bonjour {name} </span>;
};

const SomeGreeting = ({ languageBoolean, name }) =>
  languageBoolean ? <Bonjour name={name} /> : <Hello name={name} />;

const isFrench = Math.random() >= 0.5;

const SayHelloAndPassAProp = () => {
  const somebodysName = "Matt";

  return <SomeGreeting name={somebodysName} languageBoolean={isFrench} />;
};
ReactDOM.render(<SayHelloAndPassAProp />, document.querySelector("#root"));
```

Props: Dataflow and Communication

While dataflow with props is unidirectional, due to the nature of Javascript, there is a way for child elements to communicate with their parents via props (hint: think first class functions and scoping):

```
import React from "react";
import ReactDOM from "react-dom";
```

```

const Child = props => {
  const num1 = 23;
  const num2 = 42;
  const sum = num1 + num2;

  props.listen(`I CAN DO MATH! ${num1} + ${num2} is ${sum}`);

  return <div> MATH IS FUN!</div>;
};

const Parent = () => {
  const listenToTheChild = childWords => {
    console.log(`Passed back a variable!!: `, childWords);
  };

  return (
    <>
      <div> It's good to learn math! </div>
      <Child listen={listenToTheChild} />
    </>
  );
};

ReactDOM.render(<Parent />, document.querySelector("#root"));
```

Props: Dataflow and Communication

It looked as if the above was immediate, let's slow things down with a button:

```
import React from "react";
import ReactDOM from "react-dom";

const Child = props => {
  return <button onClick={props.listen}> Press me! (In the child) </button>;
};

const Parent = () => {
  const listenToTheChild = eventResponseFromChild => {
    console.log(
      `Passed back a event from the button press!!: `,
      eventResponseFromChild
    );
  };

  return (
    <>
      <div> It's good to learn math! </div>
      <Child listen={listenToTheChild} />
    </>
  );
};

ReactDOM.render(<Parent />, document.querySelector("#root"));
```

Children: Think of them!

JSX supports nested components just like HTML, but how does that work? Let's take a look at the `createElement` function again:

```
React.createElement(type, [props], [...children]);
```

Notice that the children have been spread. That means that ultimately, we can have as many child elements as we please (but we'll start with just one, for now).

Children: Think of them!

When the children are passed through the function, nothing is immediately done with them. For example:

```
import React from "react";
import ReactDOM from "react-dom";

const TextComponent = () => {
  return <div>I AM SOME TEXT</div>;
};

const ComponentWithChildren = () => {
  return (
    <>
      <TextComponent>
        <span> I am a child of Text Component!!</span>
      </TextComponent>
    </>
  );
};
```

```

ReactDOM.render(<ComponentWithChildren />, document.querySelector("#root"));
```

Children: Dealing with them

Because JSX elements are not the exact same things as HTML, we can't just write JSX like HTML without some extra steps.

To deal with child elements, we can treat them similarly to props:

```
import React from "react";
import ReactDOM from "react-dom";

const TextComponent = ({ children }) => {
  return (
    <>
      {children}
      <div>I AM SOME TEXT</div>
    </>
  );
};

const ComponentWithChildren = () => {
  return (
    <>
      <TextComponent>
        <span> I am a child of Text Component!!</span>
      </TextComponent>
    </>
  );
};

ReactDOM.render(<ComponentWithChildren />, document.querySelector("#root"));
```

Children: Dealing with Multiple:

When dealing with multiple children, you can see that they're passed as an array:

```
import React from "react";
import ReactDOM from "react-dom";

const TextComponent = ({ children }) => {
```

```jsx
  console.log("Children: ", children);

  return (
    <>
      {children}
      <div>I AM SOME TEXT</div>
    </>
  );
};

const ComponentWithChildren = () => {
  return (
    <>
      <TextComponent>
        <div> Oh hi! I am another child! </div>
        <span> I am a child of Text Component!!</span>
        <span> I'm also a child </span>
      </TextComponent>
    </>
  );
};

ReactDOM.render(<ComponentWithChildren />, document.querySelector("#root"));
```

Children: Dealing with Multiple:

You can access specific children by their element:

```jsx
import React from "react";
import ReactDOM from "react-dom";

const TextComponent = ({ children }) => {
  console.log("Children: ", children);

  return (
    <>
      {children[0]}
      {children[1]}
      <div>I AM SOME TEXT</div>
    </>
  );
};

const ComponentWithChildren = () => {
  return (
    <>
      <TextComponent>
        <div> Oh hi! I am another child! </div>
        <span> I am a child of Text Component!!</span>
        <span> I'm also a child </span>
      </TextComponent>
    </>
  );
};

ReactDOM.render(<ComponentWithChildren />, document.querySelector("#root"));
```

Children: Dealing with One or Multiple

Children are JSX objects. If need be you can work with what's inside of them (though, in all actuality, you probably shouldn't, or should do it elsewhere, but it's good to know if you're in a pinch):

```
import React from "react";
import ReactDOM from "react-dom";

const TextComponent = ({ children }) => {
  console.log("Children: ", children);

  return (
    <>
      {React.Children.map(children, child => {
        return child.props.children + "STUFF FROM INSIDE THE MAP";
      })}
      <div>I AM SOME TEXT</div>
    </>
  );
};

const ComponentWithChildren = () => {
  return (
    <>
      <TextComponent>
        <div> Oh hi! I am another child! </div>
        <span> I am a child of Text Component!!</span>
        <span> I'm also a child </span>
      </TextComponent>
    </>
  );
};

ReactDOM.render(<ComponentWithChildren />, document.querySelector("#root"));
```

What about Props WITH Children?

What happens if we pass in both props AND children? The solution is rather counterintuitive:

```
import React from "react";
import ReactDOM from "react-dom";

const TextComponent = props => {
  return (
    <>
      <span> {props.whatever} </span>
      {props.children}
      <div>I AM SOME TEXT</div>
    </>
  );
};

const ComponentWithChildren = () => {
  return (
    <>
      <TextComponent whatever="WORDS">
        <div> Oh hi! I am another child! </div>
        <span> I am a child of Text Component!!</span>
        <span> I'm also a child </span>
      </TextComponent>
    </>
  );
};

ReactDOM.render(<ComponentWithChildren />, document.querySelector("#root"));
```

What about Props WITH Children?

In the TextComponent javascript, we're only passing in one single object.

```
const TextComponent = props => {
  return (
    <>
      <span> {props.whatever} </span>
      {props.children}
      <div>I AM SOME TEXT</div>
    </>
  );
};
```

What about Props WITH Children?

So, ultimately, while we say "props", it's really just an object from which we can destructure any of our props, along with any of our children.

```
const TextComponent = ({ whatever, children }) => {
  return (
    <>
      <span> {whatever} </span>
      {children}
      <div>I AM SOME TEXT</div>
    </>
  );
};
```

Prop Types:

When you're coding JSX, you'll almost inevitably forget to pass a prop at some given point. What ends up happening is that your prop becomes undefined:

```
import React from "react";
import ReactDOM from "react-dom";

const TextComponent = ({ whatever }) => {
  return (
    <>
      <span> {whatever} </span>
      <div>I AM SOME TEXT</div>
    </>
  );
};
```

```

const ComponentWithChildren = () => {
  return (
    <>
      <TextComponent />
    </>
  );
};

ReactDOM.render(<ComponentWithChildren />, document.querySelector("#root"));
```

Here we forgot to pass `whatever` as a prop, and ultimately it ended up undefined.

Prop Types:

There are a number of ways to get around this, but the one that's build into react is `PropTypes`. These enforce that we're pulling in the proper props!

Depending on what version of react you're using, you may need to install prop-types with (using create-react-app, you should be fine):

```
npm i --save prop-types
```

Prop Types: Example

```
import React from "react";
import ReactDOM from "react-dom";
import PropTypes from "prop-types";

const TextComponent = ({ whatever }) => {
  return (
    <>
      <span> {whatever} </span>
      <div>I AM SOME TEXT</div>
    </>
  );
};

TextComponent.propTypes = {
  whatever: PropTypes.string.isRequired
};

```

```
const ComponentWithChildren = () => {
  return (
    <>
      <TextComponent />
    </>
  );
};

ReactDOM.render(<ComponentWithChildren />, document.querySelector("#root"));
```

Prop Types

The one thinig to note is that, while prop types will give you a warning in your developer console, it will only work if you remember to implement them.