WHY JSX?

Both server-side and client-side:

- functions that take state data and return HTML

Or at least, you SHOULD. Functions that take state data and modify the DOM can be broken down:

- Convert data to HTML
- Update page with HTML

JSX

- is this kind of function
- is written like HTML
- is still actually a function (transpiled)

JSX

JSX is a transpiled HTML-like syntax.

The output is a JS function that produces HTML.

HTML-like to make it easier to show the result

- All JSX tags can self-close (and must close)
- JSX tags require `className` instead of `class`
- JSX tags can take an object (not string) for `style`
  - If you use `style` - we won't
- In JSX anything inside `{}` is replaced with the evaluated JS results.
- Whitespace trims as much as possible

JSX EXAMPLE

JSX (`coolCat` is a JS variable that holds 'Maru')

```
<div className="demo">
  <span>{ 1 + 1 }</span>
  { coolCat }
</div>
```

Actual output:

```
<div class="demo"><span>2</span>Maru</div>
```

COMPOSITION

New JSX components are easily created:

- Often put inside `()` for clarity (not required)
- JSX is NOT a string - transpiles into a function
- JSX components have one top container element
- JSX components are MixedCase, not camelCase (by convention)
- Components can contain (call) other components

```
const MyComponent = (<div className="demo">Hi</div>);
const OtherComp =
  (<div> Check out my greeting: <MyComponent/> </div>);
```

```
<div>Check out my greeting: <div class="demo">Hi</div></div>
```

RENDERING AND THE VIRTUAL DOM

A defined component is an uncalled function.

Converting a component to HTML is "rendering"

A component can be rendered multiple times

React has a **virtual dom** - it keeps a lightweight copy of the DOM and renders changes to that.

- If it sees the new result is actually different, THEN it updates the real DOM
- Makes for faster changes
- Means you don't have to track if a render is required

VIRTUAL DOM

Because the VDOM tracks what **it thinks** the page is like...

- It is a BAD idea to change the DOM outside of React
- You can, but it's a source of bugs
- React may overwrite changes it doesn't know about

You can change outside of the area React manages

- React does not cover the whole page, just everything inside some root element

COMPONENTS: CLASSES VS FUNCTION

React Components can be defined as classes:

```
class MyComponent extends React.Component {
  render() {
    return ( <div>
      // ...
```

or as functions:

```
function MyComponent()
  return ( <div>
    // ...
```

Originally some actions required class-based components

In Feb 2019, they released "hooks": classes are no longer required.

WHICH DO WE DO, CLASSES OR HOOKS?

This is a hard decision:

- No time to do both in depth
- Lots of existing content uses class-based components
- Web world changes rapidly for new development
- ...but employers change dependencies slowly

A project can use both (but does require recent React version)

- We're teaching **function-based** because new development will likely use that

PROPS

Like HTML, React Components can be passed attributes, called "props"

The component gets them as arguments:

```
<MyComp name="Bao"/>
```

```
function MyComp(props) {
  return (<div>{props.name}</div>);
}
```

```
<div>Bao</div>
```

You can destructure like any object/function call:

```
function MyComp({ name }) {
  return (<div>{ name }</div>);
}
```

ABOUT PROPS

In HTML

- attributes must be strings
- properties have no value

In JSX, props can be ANY DATA (if in {})

```
<MyComp info={ [ 1, 2, 3 ] }/>
```

In JSX, properties should be set as boolean

```
<MyComp disabled={true}/>
```

JSX is often passed callback functions as props!

```
<MyComp onLogout={logoutCallback}/>
```

CHILDREN (TAG CONTENTS)

HTML tags have contents. To access JSX contents, use the special prop "children":

```
<MyComp>This is some content</MyComp>
```

```
const MyComp = ({ children }) => {
  return (
    <div>I heard: <b>{children}</b></div>
  );
};
```

```
<div>I heard: <b>This is some content</b></div>
```

COMPONENT FILES

JSX files can be `.js` or `.jsx`

- I require `.jsx` because it's valuable information

You CAN have multiple components per file

- A component is just a function, you can export it like any other function

BUT the convention is to have one component per file

- I **require** one component per file
- Name the file after the component (MixedCase, not camelCase)

COMPONENT STATE

Each component can have its own state

- class-based components did so as a state object
- function-based components use "hooks" - special closures

Either way, be careful in managing your state

- If the state doesn't belong to the component, it should be passed in as a prop
- Complex state is a source of bugs

COMPONENT STATE DEMO

```
import React from 'react';
import Counter from './Counter';

const App = () => ( <Counter start={1}/> );
export default App;
```

```
import React, { useState } from 'react';

const Counter = ({ start }) => {
  const [count, setCount] =  useState(start);
  return (
    <button
      onClick={ () => setCount(count+1) }
    >
      {count}
    </button>
  );
};
export default Counter;
```

PURE COMPONENTS

"Pure Functions" are functions that are not modified by, and do not modify, an outside state

- They return a value based only on the data passed in

"Pure Components" are the same:

- They return a value based only on the data passed in

```
const MyComp = ({ label, action }) => {
  return (<button onClick={action}>{label}</button>);
};
```

WHY WAS THAT GOOD?

Inline JS is bad, why is this good?

```
const MyComp = ({ label, action }) => {
  return (<button onClick={action}>{label}</button>);
};
```

- This is transpiled JSX
  - the output is NOT html with inline js

What's the value?

- Same as functions - this encapsulates responsibilities
- Change in one place

COMMON EARLY JSX MISTAKES

- Not using MixedCase for components
- Being too specific
  - Like functions, components should be reusable
  - components should not "know" about outer state
- Putting too much in one component
  - Like functions, break it down
  - one function, one purpose
  - one component can call others
- Expecting props to auto mean the same as HTML
- Putting too much logic in JSX
  - You should put in raw JS and import

APPLICATION STATE

In "vanilla" JS, app state is JS variables in memory

- Same in React
- Top-level component passes down to children
- Child components can pass down to deeper children

If too much state is passing too deep, you want application state management

- some basic using React Context
- or use an outside lib (Redux, etc)
- complex state management outside React-as-view

APPLICATION STATE DEMO

```
import React, { useState } from 'react';
import Counter from './Counter';
import TopN from '/.TopN';
import listOfStuff from './somelist'; // Not component

const App = () => {
  const [count, setCount] = useState(1);
  return ( <div>
    <Counter count={count} onCount={() => setCount(count+1)}/>
    <TopN showTop={count} list={listOfStuff}/>
  </div>);
};
export default App;
```

- Counter and TopN know **little** about each other
- Or even the context they are called in
- This is good practice - function or component