React JumpStart

Student Guide

Table of Contents

React JumpStart

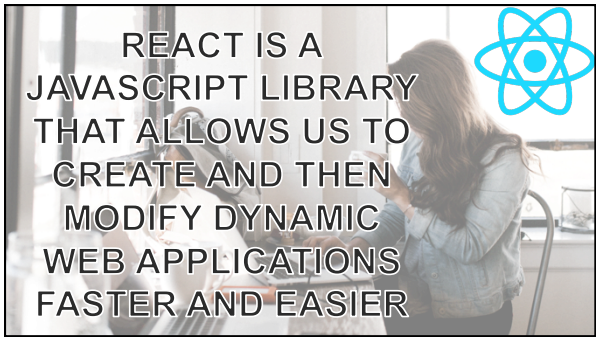
Hello, React

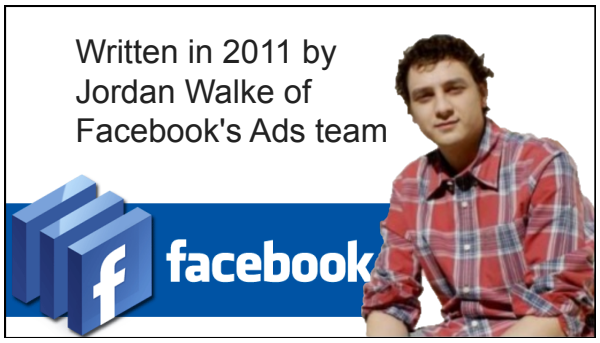
A gentle introduction to React. What is this thing anyway, and why should I learn it?

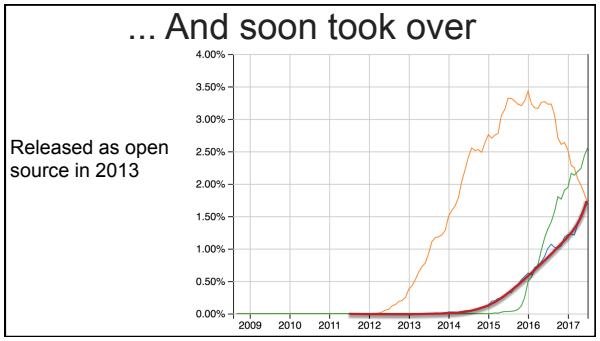
tl;dr

- What is React? It's a JavaScript library that helps you to create single page web apps faster and more abstractly
- The 3 design philosophies you need to know to understand React
- How React does things so darned fast!
- The secrets of React - What is really happening when an app runs...
- ... And how it was created

What is React?







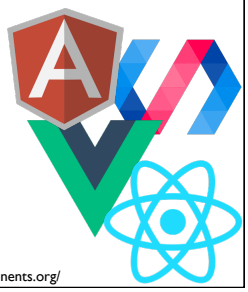
The top 3 design principles of React

1. Composition of components
2. Events -> state change -> re-render
3. One-way data flow

The web is going to components

- The web component spec is coming*
- We'll all be writing components in a few years

But for now...
- React
- Polymer
- Angular
- Vue.js

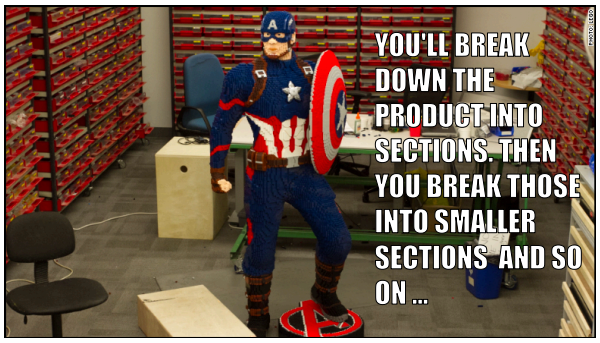


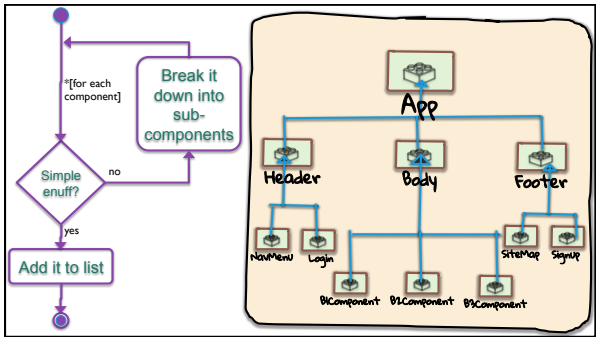
* Find more information on http://www.webcomponents.org/

With React, you'll no longer write pages; you'll write components

Each component is self-contained and encapsulated

- Even styles are local; CSS no longer cascades through components





In each component you define ...

- How the input parameters (props) should be displayed
- How to react to user interactions (events)

Components should be pure functions when possible.

- React will ... react ... to the data provided.
- The most predicable and debuggable components are those that are pure.

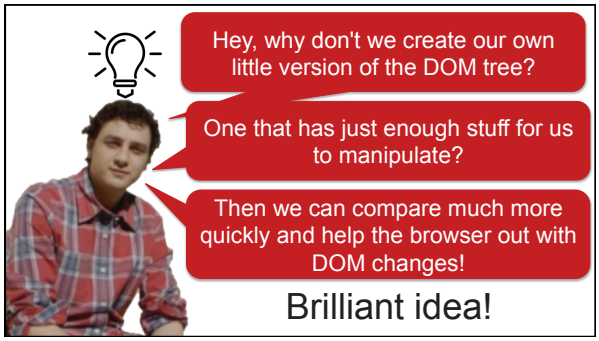
How is React so darned fast?

React is fast because of three things

1. The virtual DOM
2. One-way data flow
3. It is lightweight

Reason 1: React mimics the DOM

- JavaScript makes web pages dynamic by altering the DOM.
- But the real DOM is **extremely** heavy because it has to carry so much information in it.
- Because it is so heavy, it is very slow to manipulate.
- Just facts of life in the DOM manipulation world



Brilliant idea!

virtual DOM

- Internal algorithms compare the before and after pictures and determines the smallest amount of changes that can be made. And batches them. And then makes the changes in the DOM.
- Sound like a lot of overhead? Yep.
- But it ends up being so much faster!

- Because it diffs, it only makes changes to the part(s) of the page that have changed.
- Thus it only changes when internal state changes (Spoiler: React tracks 2 kinds of data:
  - Data that is sent into a component and does NOT change
  - Data that can change.
- Hint: Keep your components simple ... don't ever calculate diffs. Just redraw the whole component and let React use the virtual DOM to decide what should redraw and what won't need to.

Reason 2: Data only flows one way

- Data only flows down from parent to child. Never back up.
- Nothing ... not even form fields ... are bound 2-way.
- "So, how do I bind, then?"
- Patience, grasshopper. I promise I'll show you if you come back. It's not easy.

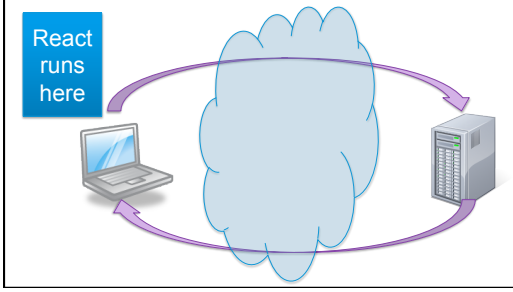
Reason 3: It is lightweight

- It is a library, not a framework
- No built-in Ajax capabilities
- Not opinionated about typing, threading, state management, routing, etc.
- You can add libraries as you like

How a React app works

And how it gets there

The web uses a thin-client architecture



How React works at runtime

1. User requests *any* url from our site.
2. Server responds with index.html
3. client requests bundle.js which is provided and contains all html, css, JavaScript files, libraries
4. Client runs

```
React.render(<App />, someNode)
```

5. This loads your main React component on the page.
6. It loads all subcomponents and so forth and so on.

Behind the scenes, React uses...

- webpack, Babel, npm, ugilfy, and a bunch of other tools
- to transpile, bundle, and minify all your HTML, CSS, and JS into bundle.js.
- All this is automated through scripts and npm.

- Bundle = put all the JS in one big file.
  - Reduces fetches and speeds us up
- Minify = Remove all bytes not needed to run.
  - Reduces size of download and speeds us up.
- Transpile ... Wait, what?

Transpile = converting one dialect of JavaScript into another

- Translating + Compiling = Transpiling
- And why do we need to transpile with React?

Because React uses a dialect called JSX

No browser in the world understands JSX, so it is transpiled by Babel/webpack/loaders

React demands that you have a very precise, very complex setup with literally hundreds of interdependent libraries having dozens of version numbers each.

If one small part of it is not configured properly, the system fails.

Wow! all these parts! Wouldn't it be nice to have a way to automate the creation/ scaffolding of the different parts?

Next chapter!

tl;dr

- What is React? It's a JavaScript library that helps you to create single page web apps faster and more abstractly
- The 3 design philosophies you need to know to understand React
- How React does things so darned fast!
- The secrets of React - What is really happening when an app runs...
- ... And how it was created

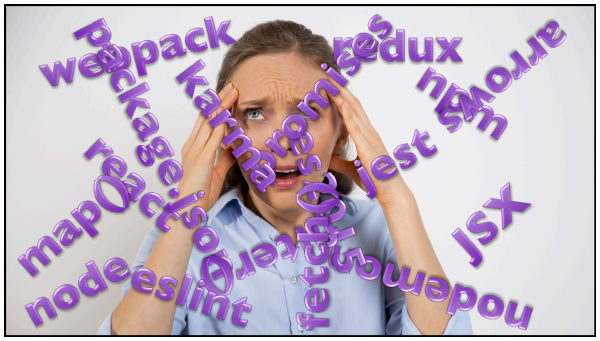
Create React App

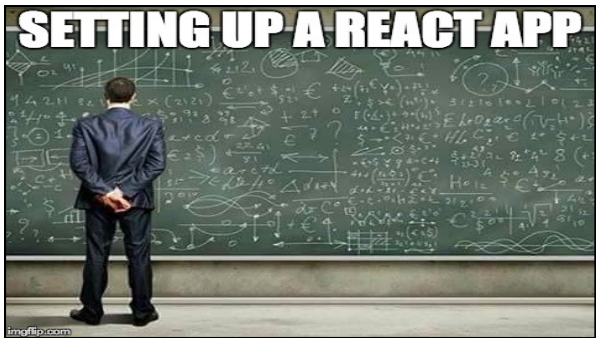
tl;dr

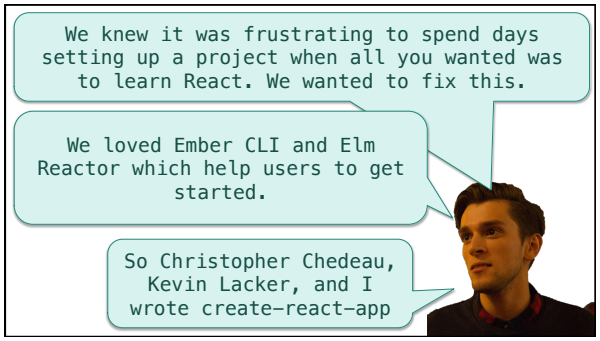
- React is very difficult to configure and brittle.
- So the team has provided us an incredibly simple tool called create-react-app that scaffolds the project
- Compile and run in watch mode with `npm run dev`
- To transpile, bundle, minify, lint and run tests for production, you `npm run build`

React has a lot of moving parts

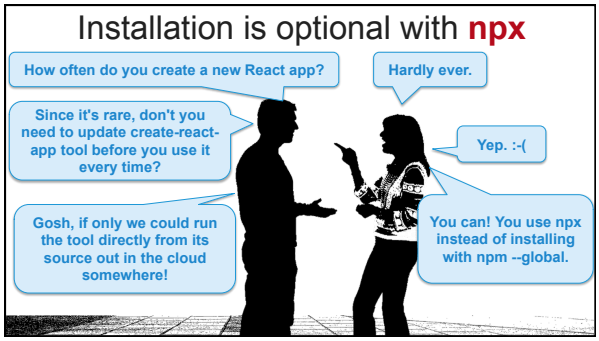
React uses JSX, which isn't JavaScript ...
So it has to be transpiled by Babel ...
which should be automated by webpack ...
which also minifies, bundles and tree-shakes ...
the CSS, HTML, and hundreds of libraries ...
which must be managed by npm ...
thus carefully configured in package.json ...
linted with eslint and tested with jest ...
using ES2015 modules and AMD requires ...
run through node processes ...







Installation is optional with **npx**



create-react-app makes your project

```
npx create-react-app hello-world
```

```
└─ react-dom@15.2.1

Success! Created hello-world at /Users/dan/hello-world.

Inside that directory, you can run several commands:
  * npm start: Starts the development server.
  * npm run build: Bundles the app into static files for production.
  * npm run eject: Removes this tool. If you do this, you can't go back!

We suggest that you begin by typing:
  cd /Users/dan/hello-world
  npm start

Happy hacking!
create-react-app)
```

Your project has a single build dependency and some node scripts.

package.json

```
{
  "name": "whatever-you-want",
  "dependencies": {
    "react": "^17.2.1",
    "react-dom": "^17.2.1"
  },
  "devDependencies": {
    "react-scripts": "1.1.0"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "eject": "react-scripts eject"
  }
}
```

Starting the Server

- Run npm start to launch the development server. The browser will open automatically with the created app's URL.

```
Compiled successfully!

The app is running at http://localhost:3000/
```

Then you're in watch mode

- When your code changes, webpack lints with eslint and transpiles with babel.

```
Failed to compile.

Error in ./src/App.js
Syntax error: /Users/dan/hello-world/src/App.js: Unexpected token (8:7)
   6 |   render() {
   7 |     return (
>  8 |       <<div className="App">
     |        ^
   9 |         <div className="App-header">
  10 |           <img src={logo} className="App-logo" alt="logo" />
  11 |           <h2>Welcome to React</h2>
 @ ./src/index.js 11:11-27
```

ESLint output shows in the console

- The lint rules are tuned for a React app.

```
Compiled with warnings.

Warning in ./src/App.js

/Users/dan/hello-world/src/App.js
  7:9  warning  'hello' is defined but never used  no-unused-vars

✖ 1 problem (0 errors, 1 warning)


You may use special comments to disable some warnings.
Use // eslint-disable-next-line to ignore the next line.
Use /* eslint-disable */ to ignore all warnings in a file.
```

"npm run build" to go to production

It is minified, bundled, content hashed (for caching).

```
hello-world) npm run build

> hello-world@0.0.1 build /Users/dan/hello-world
> react-scripts build

Successfully generated a bundle in the build folder!

You can now serve it with any static server, for example:
  cd build
  npm install -g http-server
  hs
  open http://localhost:8080

The bundle is optimized and ready to be deployed to production.
hello-world)
```

But what if I don't like the default settings?

- To create the config files and move the dependencies into your package.json, you simply ...

`npm run eject`

tl;dr

- React is very difficult to configure and brittle.
- So the team has provided us an incredibly simple tool called create-react-app that scaffolds the project
- Compile and run in watch mode with `npm run dev`
- To transpile, bundle, minify, lint and run tests for production, you `npm run build`

Resources for further study

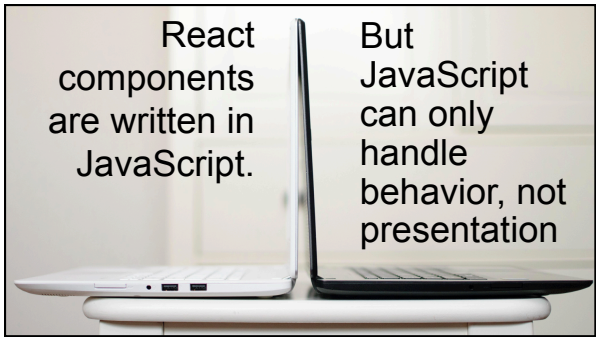
- Dan Abramov's blog post where these screen shots and quotes came from:
  - http://bit.ly/create-react-app-article

Stateless functional components

Less is more

tl;dr

- The 7 rules of JSX
- Everything it takes to make a component
- The two ways to get a component on a page



To help with the presentation, React introduces a new DSL* called JSX

* Domain-Specific Language

JSX

XML must be ...

- Well-formed
  - Conforms to rules
- Valid
  - Conforms to its DTD

Rules of well-formed XML

- All XML elements must have a closing tag.
- XML tags are case-sensitive.
- All XML elements must be properly nested.
- All XML documents must have a root element.
- Attribute values must always be quoted.

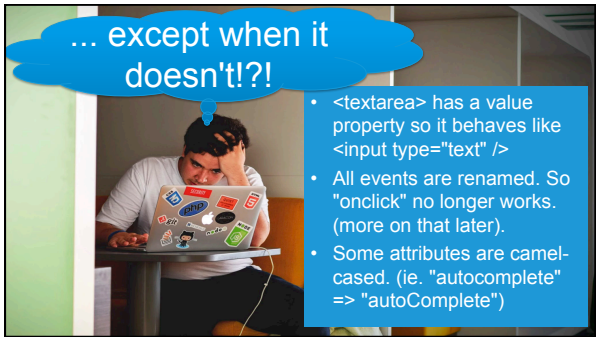
Alright, how about that valid thing?

Well, there is no DTD, but it does follow rules ...

- All tags must be pre-defined. They're either ...
  1. Valid W3C-approved elements with proper attributes
  2. Valid pre-defined React components that you wrote
- Note: it uses casing to determine which is which.
  - W3C tags are lower-cased.
  - React components begin with an upper-case character

- Got compile errors in your JSX? Ask yourself "Is this strictly good W3C HTML?" And fix it where it is not standard.
- If you want to use a non-standard attribute, precede it with data-

JSX must conform to W3C standards ...



Quiz: What's wrong with this?

```
let class = "Programming 101";
let for = "because I want to graduate";
```

- Therefore, with JSX you can't go ...

```
<label class="big" for="firstname">First</label>
<input id="firstname" />
```

- Instead you have to ...

```
<label className="big" htmlFor="firstname">First</label>
<input id="firstname" />
```

- Warning in the console. Not a compile error.

Pop quiz!

On the next few pages, let's look at some problems and solutions

Quiz: What's wrong with this JSX?

BadJSX.js

Can't have >1 root element

```
<h1>Good shows</h1>
<ul>
 <li>Game of Thrones</li>
 <li>Big Bang Theory</li>
 <li>Mr. Robot</li>
</ul>
```

And how can we fix it?

BetterJSX.js

```
<>
<h1>Good shows</h1>
<ul>
<li>Game of Thrones</li>
<li>Big Bang Theory</li>
<li>Mr. Robot</li>
</ul>
</>
```

Wrap it in a or a React *fragment*

Quiz: What's wrong with this JSX?

BadJSX.js

```
<goodShows>
 <li>Game of Thrones</li>
 <li>Big Bang Theory</li>
 <li>Mr. Robot</li>
</goodShows>
```

That's not a valid element

And how can we fix it?

BetterJSX.js

```
<GoodShows>
 <li>Game of Thrones</li>
 <li>Big Bang Theory</li>
 <li>Mr. Robot</li>
</GoodShows>
```

Make the custom element properly cased

Quiz: What's wrong with this JSX?

BadJSX.js

```
<ul>
 <li><a href="got.com">Game of
Thrones</li></a>
 <li><a href="bbt.com">Big Bang
Theory</a></li>
 <li><a href="mrr.com">Mr.
Roboto</a></li>
</ul>
```

You can't overlap elements

And how can we fix it?

BetterJSX.js

```
<ul>
 <li>
  <a href="got.com">Game of
Thrones
  </a>
 </li>
 <li><a href="bbt.com">Big Bang
Theory</a></li>
 <li><a href="mrr.com">Mr.
Roboto</a></li>
</ul>
```

Nest them instead of overlapping them

Quiz: What's wrong with this JSX?

BadJSX.js

```
<>
<p>All geeks should watch these
shows:
<ul>
 <li>Game of Thrones</li>
 <li>Big Bang Theory</li>
</ul>
</>
```

Open tags must be closed

And how can we fix it?

BetterJSX.js

```
<p>All geeks should watch these
shows:</p>
<ul>
 <li>Game of Thrones</li>
 <li>Big Bang Theory</li>
</ul>
</>
```

Close the tag or make it self-closing

But wait! What browsers support JSX?

- All of them! (Or none of them?)
- Because JSX never lands in the browser.
- It is transpiled out and replaced with equivalent JavaScript

This transpiles to this

With_JSX.js

```
function Hello() {
 return (
  <div>
   <label htmlFor="n">Name</label>
   <input id="n" required
          value="foo.jpg" />
  </div>
 );
}
```

Without_JSX.js

```
function Hello() {
 return
React.createElement(
  "div",
  null,
  React.createElement(
   "label",
   { "htmlFor": "n" },
   "Name"
  ),
  React.createElement(
   "input",
   {id:"n", required:true,
     value: "foo.jpg" })
  );
}
```

To create a component

Here's all that is required

1. import React from 'react'
2. Create a function that returns JSX
3. Display that component inside a host.

1. import react

At the top of your component file ...

```
import React from 'react';
```

2. Create a function ...

- You create a function that returns JSX

```
function Person() {
  // You can do other things here.
  return <div>Hello world</div>
}
```

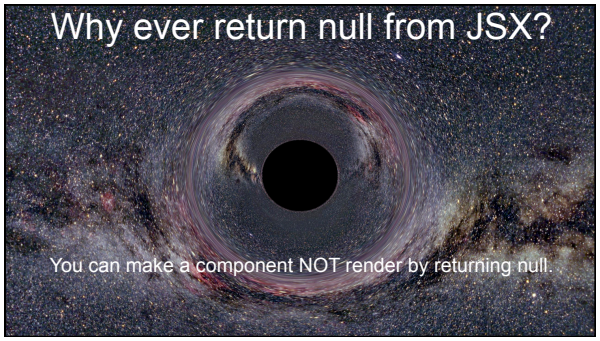
- Must start with an uppercase letter! Prefer Pascal-cased

... that returns JSX

- You must return
  1. JSX with a single root element
  2. An array of JSX elements
  3. A string
  4. null
- Anything else is an error

When you want a return and then have JSX starting on the next line, wrap them in parentheses or else Babel gets confused.

Why ever return null from JSX?



3. Host your component

You can host it inside another component or it can be the root component.

To host inside another component, put the name of this one in the parent's JSX

```
function OtherComponent() {
  return <div>
    <Person />
    <Person></Person>
  </div>
}
```

This would show two Person components.

To make this the top-level component, use ReactDOM.render()

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
ReactDOM.render(
  <Person></Person>
  , document.getElementById('root'));
```

This means that index.html must have an element with id="root".

One last thing ...

Later in the course we're going to be covering another way to write components that is more complex but it is also more capable.

tl;dr

- The 7 rules of JSX
- Everything it takes to make a component
- The two ways to get a component on a page

Styling components

tl;dr

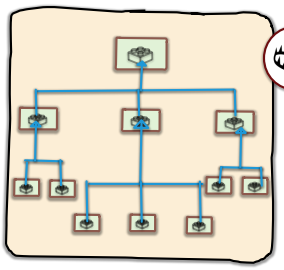
- The bad news: There are lots of methods to style and all of them feel kludgey in different ways
- The good news:
  - Import a CSS file to create global styles
  - JavaScript objects can live locally in a component
  - ... or we can import them from a .js file
  - Open source style libraries can be npm installed
  - All the above techniques can be used together

5 methods to style React components

1. Importing traditional CSS files
2. Inline JavaScript styles
3. Importing JavaScript modules
4. npm libraries
5. Hybrids of the above

Importing CSS files

```
import 'react-mdl/extra/material.css';
```



Note: no "from" in the import

- When you do this it flows DOWN only to inner components, but not up.

Then you're using it just like you do with regular CSS

```
<div className="big fancy">...</div>
<ul className="no-bullets">...</ul>
<input className="form-control" />
<img src="img.jpg" className="rounded" />
```

Remember, you can't use 'class'. You must use 'className'

Inline styling

Box.js

```
import React from 'react';

const divStyle = {
 margin: '40px',
 border: '5px solid pink'
};

export function Box() {
 const pStyle = {
  fontSize: '15px',
  textAlign: 'center'
 };
 return (
  <div style={divStyle}>
   <p style={pStyle}>
    Text goes here
   </p>
 </div>)
}
```

Importing JavaScript CSS files

styles.js

```
export const pStyle = {
 fontSize: '15px',
 textAlign: 'center'
};
// As many more styles as you like here
```

First, create a JavaScript file with as many styles as you like

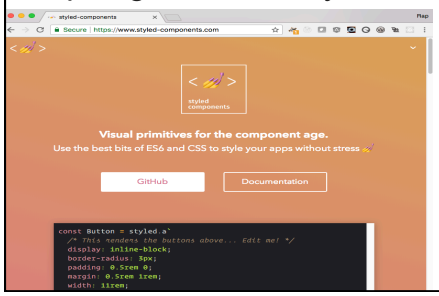
... then import the file and use the styles.

Box.js

```
import pStyle from './styles.js';
export function Box() {
 return (
  <div className="alert">
    <p style={pStyle}>
      Text goes here
    </p>
  </div>)
}
```

npm Libraries

https://github.com/styled-components



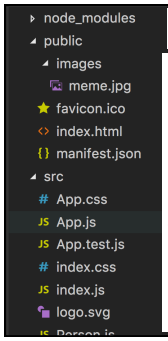

```
import React from 'react';
import styled from 'styled-components';

// Create a <Title> react component that renders an <h1>
const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: palevioletred;
`;

// Create a <Wrapper> react component that renders a <section>
const Wrapper = styled.section`
  padding: 4em;
  background: papayawhip;
`;

// Use them like any other React component – except they're styled!
<Wrapper>
  <Title>Hello World, this is my first styled component!</Title>
</Wrapper>
```

Images



- Static resources should be put under public. Including images!
- Say we have a folder under public called images. We put an image file in there:
- Then reference them relative to that public folder.

```
<img src="/images/meme.jpg" />
```

tl;dr

- The bad news: There are lots of methods to style and all of them feel kludgey in different ways
- The good news:
  - Import a CSS file to create global styles
  - JavaScript objects can live locally in a component
  - ... or we can import them from a .js file
  - Open source style libraries can be npm installed
  - All the above techniques can be used together

Events

How to respond to events in React

tl;dr

- JSX strips out the native HTML events and replaces them with their own.
- They're called synthetic events and they don't behave exactly like their native mirrors
- You create events on your own components by running a prop in the inner that was defined on the host.

React has created its own version of most events

- This is a place where React is very opinionated. :-(
- Normalized - to eliminate browser differences :-)

Some events that React can handle

- blur
- change
- click
- copy
- cut
- dbl-click
- focus
- keydown
- keypress
- keyup
- mousedown
- mouseenter
- mouseleave
- mousemove
- mouseout
- mouseover
- mouseup
- paste
- submit
- ...

... basically every event that is in a component's scope, React has an interface to.

Uppercase the first letter and precede it by *on*

```
<any onFoo={bar}></any>
```

Hey, React! When the user triggers the *foo* event, run the *bar* function.

Examples:

```
<button onClick={doIt}>
  Press me</button>
<img onMouseOver={count} />
<input onBlur={go}
       onKeyUp={run} />
```

But the native browser events are stripped out by React

- So if you write

```
<MyComponent onclick="alert('foo')" />
```

- Your onclick will be ignored.
- We are forced to use React's synthetic events

Mouse events

- onClick
- onDoubleClick
- onMouseDown
- onMouseEnter
- onMouseLeave
- onMouseMove
- onMouseOver
- onMouseUp

```
<button onClick={processOrder}>
Go</button>
<img src="..."
 onDoubleClick={viewProduct} />
<img src="..."
  onMouseOver={incrementCounter} />
```

Form events

- onFocus
- onBlur
- onChange
- onCut
- onCopy
- onPaste
- onSubmit
- onKeyDown
- onKeyUp
- onKeyPress

```
<input onFocus={checkAllFields}
  onBlur={checkAgain}
  onKeyUp={getSuggestions} />
```

- React adds these synthetic events to W3C elements (aka NOT your components!)
- Thus you can't have a, say, click event on your component. Just on the HTML elements inside your component.*

* Unless you create your own custom event. More on that later.

Even when Synthetic events appear to match their native counterparts, they're still different

- Examples: onchange fires only when the user commits a value (via blur for example) but onChange event fires on every keystroke.
- There's a native ondblclick but not a React onDblClick event. It is onDoubleClick. (sheesh!)
- And there are other peculiarities ...

There are unsupported events

- They usually fall in three categories

1. Window- and Browser-level events
   - beforePrint, hashChange, resize, message, DOMContentLoaded, beforeunload, load,
2. Experimental events
   - They eventually get support after they're mainstream
   - (eg. Device events, Touch events, pointer events are new-ish)
3. Events that just don't make sense to do
   - reset (for forms), wheel

The event object is reused!

- When an event fires, an event object is created. Then React creates a Synthetic event object. This is expensive.
- So to increase performance, the Synthetic event object is reused over and over.
- This would not be a good idea:

```
function handleClick(event) {
  let target = event.target;
  setTimeout(() => {
    console.log(target.name);
  }, 1000);
}
```

- Because one second later, the event.target object would point to a completely different object.
- event.persist() would cause it to save the value.

Passing values to the handler

Say you have an event handler function:

MyComponent.js

```
...
function addPerson(person) {
  console.log("Person was added", person);
  try {
   insertIntoDB(person);
   return true;
  } catch {
   return false;
  }
}
```

And you have some JSX:

MyComponent.js

```
let person = {};
return (
 <form onSubmit={addPerson}>
  <input value={person.first} />
  <input value={person.last} />
  <input type="submit" />
 </form>
);
```

How does the person object get sent to addPerson?!?

- When you specify an event handler, you're not running a function.
- You're registering a function to be run later.
- You must pass a function to the event

```
console.log(typeof addPerson);  // function
console.log(typeof addPerson(person));  // bool
```

So to pass a parameter, use an arrow function

MyComponent.js

```
let person = {};
return (
 <form onSubmit={() => addPerson(person)}>
  <input value={person.first} />
  <input value={person.last} />
  <input type="submit" />
 </form>
);
```

And to pass the event object ...

MyComponent.js

```
...
return (
 <button onDoubleClick={e => doStuff(e, obj1, obj2)}>
  Click me!
 </button>
);
```

- This works because when an event is triggered, the (synthetic) event object is passed into the registered function

How to create your own custom events

In the inner component...

InnerComponent.js

```
return <div>
 {/*
  Bunch of JSX here. The user interacts and some
  condition arises and we call raiseEvent()
 */}
</div>
function raiseEvent() {
  props.onCustomEvent();
}
```

Then in the host you can do this...

HostComponent.js

```
return <div>
 <InnerComponent onCustomEvent={doStuff} />
</div>

function doStuff() {
  // This is where you'd process the custom
  // event.
}
```

tl;dr

- JSX strips out the native HTML events and replaces them with their own.
- They're called synthetic events and they don't behave exactly like their native mirrors
- You create events on your own components by running a prop in the inner that was defined on the host.

Composition

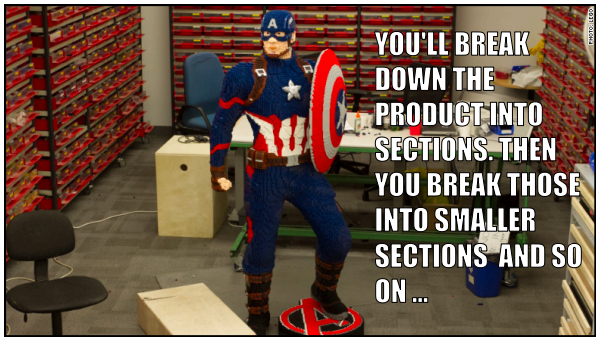
tl;dr

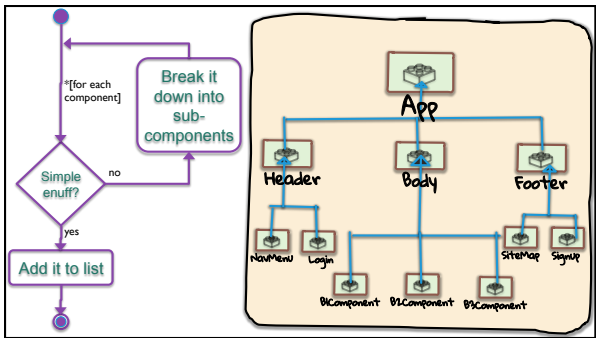
- React is centered around component composition
- To compose, we put the inner component's tag in the outer component's JSX
- To pass data down, write the values as attributes in the host and read them in *props* in the inner where they're immutable
- We cannot pass data back up but we can pass a function from the host to the inner where we pass params into that function
- To talk between other components, you pass data down from the lowest common ancestor and events up through that same ancestor

With React, you'll no longer write pages; you'll write components

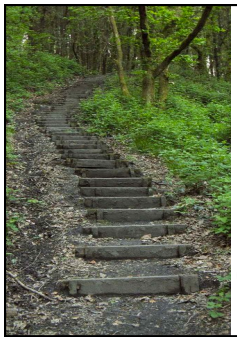
Each component is self-contained and encapsulated

- Even styles are local; CSS no longer cascades through components





How to compose



How to compose

1. Put a <tag> in the host component's JSX
2. If you want data to flow to the inner, use props
3. If you want data to flow up, bind an event

Remember: inner components must be imported into the host component

1. Add a tag

CompanyDirectory.js

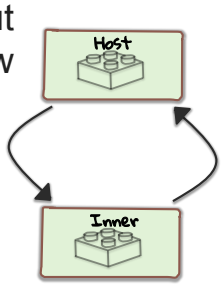
```
import {Person} from './Person';
function CompanyDirectory() {
return <div>
  <Person />
  </div>
}
```

Person.js

```
function Person() {
 return <section>
  <p>I am a person.</p>
 </section>
}
```

A word about data flow

We may want data to flow from host component to inner component and back up again.



props

props are basically the input parameters of a component.

How to pass data from host to inner

CompanyDirectory.js

```
function CompanyDirectory() {
 const [p1,p2] = get2Users();
 return <div>
  <Person hairColor="red" eyeColor="blue"
          first={p1.first} last={p1.last}
          imgSrc={p1.imgSrc} />
  <Person hairColor="brunette"
          eyeColor="brown" imgSrc="noImage.jpg"
          first={p2.first} last={p2.last} />
  </div>;
}
```

To read data in inner from host

Person.js

```
function Person(props) {
 const { first, last, hairColor, eyeColor } = props;

 return <section>
  <p>{first} has {hairColor} hair and {eyeColor} eyes.</p>
  <img src={props.imgSrc} alt={first + " " + last} />
  </section>
}
```

You always read the data with *props*.

Props are immutable (kind of)

- To add something to the props object is an error

```
TypeError: Cannot add property foo, object is
not extensible
```

- To reassign the value of a prop is an error. They're read-only.

```
TypeError: Cannot assign to read only property
'first' of object '#<Object>'
```

Full disclosure: if the prop is an object, the properties of that object can be changed, just not the prop itself

But what if I have data that needs to change? Like based on user input or Ajax calls or something?

- Well, that is not props. It's called state
- Coming soon to a lecture near you!

Passing data back up

From child to parent

3. Pass data up from inner to host

Something happened down here!

- For performance reasons, data cannot flow up.
- But we can pass a function to an inner to be invoked.
- If this function receives parameters, they will be seen in the host.

3 steps to get data to a host from an inner

In the host (parent)	In the inner (child)

1. Define a function
• Parameter(s) receive the data needed from the inner
`function funcName(p1, p2) {`
`  // Do stuff with p1 & p2`
`}`
2. Pass that function to the child as a prop
`<inner foo={funcName} />`
	3. Call the function, passing in the data
	`<any onClick={()=>props.foo(a,b)} >`

For example ...

- You have a task list component which is composed of a bunch of task components.

Q: Where is the list?
A: TaskList.js
Q: So where must the "delete" function live?
A: TaskList.js
Q: How will delete know which one to delete?
A: We can give it a task id.
Q: Where is the button to delete a task?
A: Task.js
So how does Task.js tell TaskList,js (the host) what is the task id?

Task.js

```
export function Task(props) {
 return (
  <li>
   <button
    onClick={()=>props.remove(props.id)}
   >✓</button>
   {props.text}
  </li>
 );
}
```

...And emit the event with the value you want to send up.

To send data from inner to host

TaskList.js

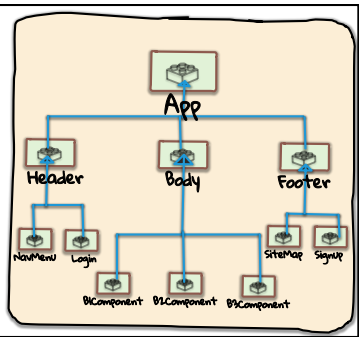
```
function TaskList {
 function delete(taskId){
  tasks=tasks.filter(t=>t.id!==taskId);
 };
 return (
  <div>
   <Task text="buy food" id=10
         remove={delete} />
  </div>
 );
}
```

When the remove event fires, run the delete() method

To read data in host from inner

Between siblings, cousins, and other distant relatives

- Just combine the parents and children techniques.
- Push up using events until you reach the lowest common ancestor, then push down using props



There are other ways to communicate

Global variables	Context (pass *this* down into a child)	refs (not a clean way. Almost non-used and confusing)	The cleanest way is to use a state container like Redux!

tl;dr

- React is centered around component composition
- To compose, we put the inner component's tag in the outer component's JSX
- To pass data down, write the values as attributes in the host and read them in *props* in the inner where they're immutable
- We cannot pass data back up but we can pass a function from the host to the inner where we pass params into that function
- To talk between other components, you pass data down from the lowest common ancestor and events up through that same ancestor

React Router

tl;dr

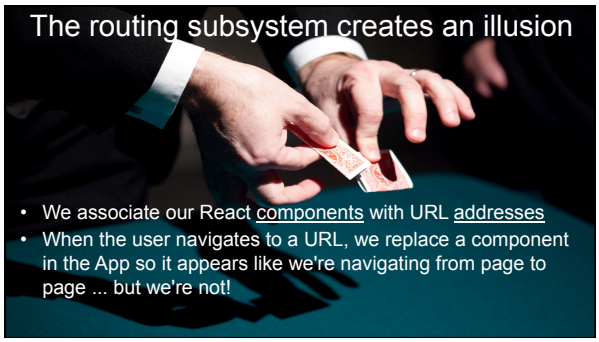
- Routing tricks the user into thinking they're navigating to pages when we're only swapping out components
- To route, you ...

1. Define the routing domain with a <Router>
2. Match URLs to components with <Route>s
3. Send the user to another route when ...
   - They type in a URL
   - They click on a <Link>
   - You push a URL onto the props.history stack
4. Read route parameters via props.match.params

React thinks in SPAs first

- But if there's only one page, how do we navigate from page to page?
- We don't!

The routing subsystem creates an illusion



- We associate our React components with URL addresses
- When the user navigates to a URL, we replace a component in the App so it appears like we're navigating from page to page ... but we're not!

Routing is that process of keeping the browser URL in sync with what's being rendered on the page

React Router lets you handle routing declaratively.

React Router is NOT from Facebook!

- It's from the community
- But it is the de facto router

We route through JSX

```
<Route path="/about" component={About}/>
```

Hey, wait! "<Route>" isn't a thing! What is it?

It's a React component!

So where did it come from?

```
npm install react-router-dom    //For web
// And then ...
import { Route } from 'react-router-dom';
```

We need to know these fundamental things to route:

1. How to define the domain of a router
2. How to create routes
3. How to enable users to get to a route
4. How to read route parameters out of the URL

1. How to define the domain of our router

Only <Route>s that are nested somewhere -- at any level -- inside a <Router> will be honored

So <Router>s are usually put around or directly inside the <App />

Let's see a few examples...

A Router **around** the App

index.js

```
ReactDOM.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
,
document.getElementById('mainDiv'));
```

A Router **inside** the App

App.js

```
function App {
return <div classname="app">
 <BrowserRouter>
  <section className="foo">
   {/* All ur code goes here */}
  </section>
 </BrowserRouter>
</div>;
}
```

Note: A <Router> can only have a single inner component in it.

The router

<BrowserRouter>	<HashRouter>
• us.com/someUrl • Prettier and should be the default	• us.com/#/someUrl • Uglier but works in older browsers

Unless you need to support old browsers, just use <BrowserRouter>

2. How to create the routes

If the URL is a match of "/foo", then cram the Bar component right here.

```
<Route path="/foo" component={Bar} />
```

Design the routes ...

URL	Component
/people	People
/people/123	Person
/teams	Team
/cart	ManageCart
/	Welcome
Anything else	FourOhFour

```
<Route path="/" component={Welcome} />
<Route path="people" component={People} />
<Route path="people/:personId" component={Person} />
<Route path="teams" component={Team} />
<Route path="cart" component={ManageCart} />
<Route component={FourOhFour} />
```

For example ...

App.js

```
<BrowserRouter>
...
 <Route path="/" component={Home}/>
 <Route path="/cat" component={Catalog}/>
 <Route path="/cart" component={Cart}/>
 <Route path="/cat/:id" component={Prod}/>
...
</BrowserRouter>
```

React Router uses *inclusive* routing

- All routes that match the path are included
- If the url were "/cat/123", we'd see Home, followed by Catalog, followed by Prod

App.js

```
<BrowserRouter>
...
 <Route path="/" component={Home}/>
 <Route path="/cat" component={Catalog}/>
 <Route path="/cart" component={Cart}/>
 <Route path="/cat/:id" component={Prod}/>
...
</BrowserRouter>
```

React router will look at the url, and scan for **all** Routes that match

http://us.com...	matched components
/	Home
/foo	Home
/cat	Home and Catalog
/cat/123	Home and Catalog and Prod

What if I want just one?

App.js

```
<BrowserRouter>
 <Switch>
  ...
  <Route exact path="/" component={Home}/>
  <Route exact path="/cat" component={Catalog}/>
  <Route path="/cart" component={Cart}/>
  <Route path="/cat/:id" component={Prod}/>
  <Route component={FourOhFour} />
 </Switch>
</BrowserRouter>
```

Add an *exact* attribute to each

Put them in a *Switch* element which says to match only the first one.

Properties of a Route component

```
<Route path (component|render|children) [exact] />
```

path	A string which will be matched against the URL
component	The component to place here
render	A function that returns the JSX to render
children	A function that renders if the route does not match
exact	Don't honor partial matches. It must match exactly

3. How to enable users to navigate to a route

How can a user request a page?

At run time, a user can ...

1. Type a url
2. Click a link
3. Get pushed there in a Component class

1) They can type in a URL

- When the user types in the URL, the browser has no choice but to request a resource from the server.
- The server will have been configured to serve the root page at any address.
- Routing will then will intercept that and route him to the proper component.

2) They can click on a link

- Write your links like this:

```
<Link to="/people">All people</Link>
<Link to="/people/{p.id}">This Person</Link>
<Link to="/teams">Teams</Link>
<Link to="/cart">My cart</Link>

<Link to="/someLink">Text to display</Link>
```

3) They can get pushed in the class

```
function Foo() {
 return <div>
  {someCondition && <Redirect to="/other" />}
 </div>;
}
```

If you can use JSX, the <Redirect> component is the simplest option

3) They can get pushed in the class

- To send a user programmatically
- props has a history which wraps the browser's history object
- so just go
- props.history.push('/WhereYouWantThemToGo');

If you do not have access to JSX, push() can send them to another route.

We normally send props in JSX like this:

```
<Foo prop1={val1} prop2={val2} />
```

But if we're using the history API, how do we get props into it?

Pass a second argument to to it like this:

```
props.history.push("/SomeRoute", {
 prop1:val1,
 prop2:val2
});
```

The data ends up in props.location.state in the component.

4. How to read route parameters out of the URL

When a Route is matched, the component is given a match object which contains ...

- match.url - String. The url (eg. "/person/1234")
- match.path - String. The Route path (eg. "/person/:personId")
- match.isExact - boolean. true=exact. false=partial
- match.params - object.

Route parameters

- props.match is given to all components who were visited by the Router.
- The parameters will be found in props.match.params.foo

```
▼ {path: "/PickSeats/:showingId", ur
    isExact: true
  ▼ params:
      showingId: "11"
    ▶ __proto__: Object
    path: "/PickSeats/:showingId"
    url: "/PickSeats/11"
```

App.js

```
<BrowserRouter>
 <Route path="/cat/:id" component={Prod}/>
</BrowserRouter>
```

http://us.com/cat/1234

Prod.js

```
function Prod(props) {
 console.log(props.match.params.id);
 return <SomeJSX />
}
```

Match the thing on the left with its purpose on the right

A. <Switch>	1. Conditionally inserts a component here
B. <Link>	2. Forwards the user to another Route
C. <Route>	3. Forces an exclusive match
D. <Router>	4. Shows the boundaries of the router
E. <Redirect>	5. Takes the place of an <a>

tl;dr

- Routing tricks the user into thinking they're navigating to pages when we're only swapping out components
- To route, you ...

1. Define the routing domain with a <Router>
2. Match URLs to components with <Route>s
3. Send the user to another route when ...
   - They type in a URL
   - They click on a <Link>
   - You push a URL onto the props.history stack
4. Read route parameters via props.match.params

Expressions

Where React comes alive!

tl;dr

- If intelligent processing of JSX is needed as opposed to static JSX declarations, we include JavaScript expressions inside curly braces ("{" and "}")
- These allow us to run JavaScript inside our JSX to ...
  - conditionally display elements
  - display multiple elements by iterating an array
- If what you're trying to do is too complex, you can always call a function that outputs JSX

Say we're reading a list of people ...

people.json

```
{
"name": { "first": "maëlia", "last": "dupuis" },
"email": "maëlia.dupuis@example.com",
"cell": "06-76-31-32-56",
"picture": { "large": "md65.jpg"}
},
{
"name": { "first": "susanne", "last": "scott" },
"email": "susanne.scott@example.com",
"cell": "081-007-7340",
},
{
"name": { "first": "babür", "last": "çörekçi" },
"email": "babür.çörekçi@example.com",
"cell": "(743)-870-9450",
"picture": { "large": "bc65.jpg" }
}
```

... and displaying it

ListPeople.js

```
export function ListPeople(props) {
  const people = props.people;
  return (
    <section>
      <ul>
        <li>
          <img src={p.img} />
          {p.first} {p.last}
        </li>
      </ul>
    </section>
  )
}
```

This code won't work.
How do you enumerate the list?
How do you conditionally display?

Expressions are JavaScript inside of JSX

- We will need to run some JavaScript inside of the JSX

Expressions are ... well ... JavaScript expressions

ListPeople.js

```
export function ListPeople(props) {
  const [p]= props.people;
  return (
    <section>
      <ul>
        <li>
          <img src={p.img} />
          {p.first} {p.last}
        </li>
      </ul>
    </section>
  )
}
```

You can add JavaScript to JSX if you put it in curly braces

Expressions must be a single JavaScript expression

- ... not a statement
- ... not a block
- ... not an assignment
- ... not a line of code

Generally, expressions are something you'd find on the right side of an "="

An expression evaluates to a single thing which is then substituted back into the JSX

Not allowed:

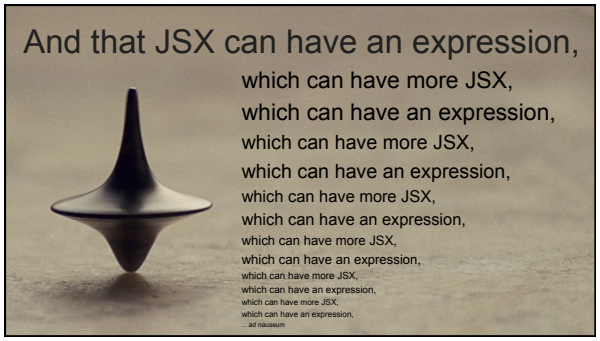
- if (foo === bar) doIt();
- while (foo === bar) doIt();
- function () { doIt(); }
- foo = bar
- expr1 ; expr2

- JSX expressions must be just that ... expressions as opposed to statements. One per set of curly braces. In them you can reference variables, use operators, and call functions (Thus, JSX can be used in methods other than the render method. Cool.

ListPeople.js

```
export function ListPeople(props) {
 const people = props.people;
 return (
   <section>
     { people.length ? <People /> : null }
     <ul>
       <li>
         <img src={p.img} />
         {p.first} {p.last}
       </li>
     </ul>
   </section>
 )
}
```

Expressions can contain JSX

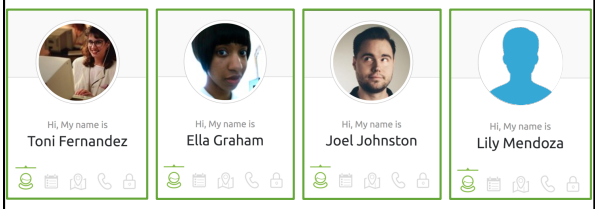


JSX is super useful for ...

- Conditional rendering
- Looping
- Calling functions

Conditional rendering

Say ListPeople.js has a photo



- If the user has a photo, show it. If not, show a generic placeholder image.

ListPeople.js

```
export function ListPeople(props) {
 const [p] = props.people;
 const ph = "/img/placeholder.jpg";
 return (
  <section>
   <img src={if (p.img) p.img else ph} />
   <div>
    <p>Hi, my name is</p>
    <p>{p.first} {p.last}</p>
   </div>
  </section>
  )
}
```

We can't do this

ListPeople.js

```
export function ListPeople(props) {
 const [p] = props.people;
 const ph = "/img/placeholder.jpg";
 return (
  <section>
   <img src={p.img ? p.img : ph} />
   <div>
    <p>Hi, my name is</p>
    <p>{p.first} {p.last}</p>
   </div>
  </section>
  )
}
```

But a ternary will work

ListPeople.js

```
export function ListPeople(props) {
 const [p] = props.people;
 const ph = "/img/placeholder.jpg";
 return (
  <section>
   <img src={p.img || ph} />
   <div>
    <p>Hi, my name is</p>
    <p>{p.first} {p.last}</p>
   </div>
  </section>
  )
}
```

Or short-circuiting will work

ListPeople.js

```
export function ListPeople(props) {
 const [p] = props.people;
 const ph = "/img/placeholder.jpg";
 return (
  <section>
   {p.img && <img src={p.img} />}
   <div>
    <p>Hi, my name is</p>
    <p>{p.first} {p.last}</p>
   </div>
  </section>
  )
}
```

Or short-circuiting will work

- If we don't have an image, just don't put anything in the JSX

Looping

Remember, this is an expression. A single statement.

- while is not a single statement
- for is not a single statement

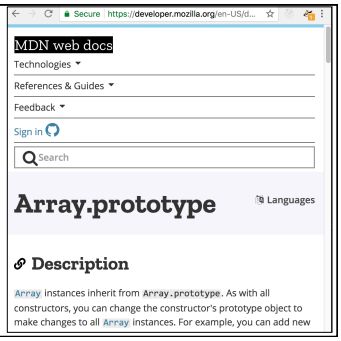
ListPeople.js

```
export function ListPeople(props) {
 return (
  <section>
   {for (let p of props.people)
    <Person person={p} />
   }
  </section>
 )
}
```

We can't do this

But there are a bunch of JavaScript Array.prototype.* methods that will iterate an array and operate on each thing

- concat()
- filter()
- flat()
- flatMap()
- join()
- slice()



Array.prototype.map will return a new array of elements, each having been transformed by the function.

```
const olderPeople = people.map(
 person => ({...person, age:person.age+10}));
```

So ... if you return JSX from map, you get a list of JSX elements

ListPeople.js

```
export function ListPeople(props) {
 return (
  <section>
   {props.people.map((p) => {
    return <Person person={p} />
   })}
   {/* Or more concisely... */}
   {props.people.map(
      p => <Person person={p} />)}
  </section>
 )
}
```

.map() will work great!

What happened to change this list?

from this ... ... to this

John Stamos
Bob Saget
Dave Coulier
Mary-Kate Olsen
Candace Cameron

1. △ "Bob" to "Dave"
2. △"Dave" to "Mary-Kate"
3. △ "Mary-Kate" to Candace
4. △"Candace" to "Bob"

John Stamos
Dave Coulier
Mary-Kate Olsen
Candace Cameron
Bob Saget

How can it keep track of what really happened?

How about now?

from this ... ... to this

102	John Stamos
334	Bob Saget
721	Dave Coulier
395	Mary-Kate Olsen
412	Candace Cameron

102	John Stamos
334	Dave Coulier
721	Mary-Kate Olsen
395	Candace Cameron
412	Bob Saget

It needs a unique ID for each VD element.

You should provide a unique key to each iterated JSX element

React uses this for the virtual DOM for performance. It matches the VD element to its DOM element by that ID.

Tip: Do not use index

- Using index is easy and it makes the warning message go away.

```
{people.map((person, index) =>
    <Person
       {...person}
       key={index}
       />
   )
}
```

- But when you do this, the keys aren't consistent from render to render.

Now, we've talked about conditionals and looping but we had a 3rd time when expressions are super useful ...

Calling functions

Calling functions

A function call is a single expression

- If the logic you want is too complex for a single expression, you can call a function
- Functions can be as complex as you like!

If it's too complex to be a single expression, but a whole function seems like overkill, remember that an iife is a single expression!

ListPeople.js

```
export function ListPeople(props) {
 return <section>
   {getSomePeople(props.people)}
  </section>
}
function getSomePeople(people) {
 const ppl = [];
 for (let p of people) {
  p.birthdate === today && ppl.push(p);
  if (p.name.city.startsWith("New"))
   ppl.push(p);
 }
 return ppl.map(p => <Person pers={p} />)
}
```

The function must return JSX

tl;dr

- If intelligent processing of JSX is needed as opposed to static JSX declarations, we include JavaScript expressions inside curly braces ("{" and "}")
- These allow us to run JavaScript inside our JSX to ...
  - conditionally display elements
  - display multiple elements by iterating an array
- If what you're trying to do is too complex, you can always call a function that outputs JSX

Managing State

tl;dr

- If a component needs to keep track of changing data, this can't be props, it is "state"
- You shouldn't change state directly. Call setState() because it always triggers a re-render
- Controlled components have input fields bound to state. When that occurs we must create event handlers to copy the values to state and then re-render
- Try to avoid having state whenever possible. When you must, use a state container and keep state as simple as possible

State is a POJSO* that represents the mutable data of your React component

*Plain old JavaScript object

State is not props!

state	props
• Data known by a component • Can be changed in the component • Lives 100% within this component	• Data known by a component • Once set, they don't change • Passed in to this component from its parent

Stateful components are classes

You must use a more complex form of the component

Functional component

```
function Foo(props) {
 return <div>
  {props.foo}
 </div>
}
```

Class-based component

```
import { Component } from 'react';
class Foo extends Component {
 render() {
  return <div>
   {this.props.foo}
  </div>
 }
}
```

This form does open some cool capabilities, though

- get, set
- methods
- properties
- this.*
- Makes Java devs happier!

Functional components	Stateful components

• Simpler! Thus easier to ...
  ○ Write
  ○ Test
  ○ Understand
  ○ Maintain
  ○ Extend
	• Class-based
	• Can use refs
	• Can tap into lifecycle events
	• Can have state
	• Can use forceUpdate()

Initializing a component's state

Initialize state in the constructor

Person.js

```
export class Person extends Component {
  constructor() {
    super()
    this.state = {first: "", last: ""};
  }
  render() { ... }
}
```

If you have a constructor, you have to call super() as the first thing in it.

It is very common to pass props into a child and then set initial state from those props.

Person.js

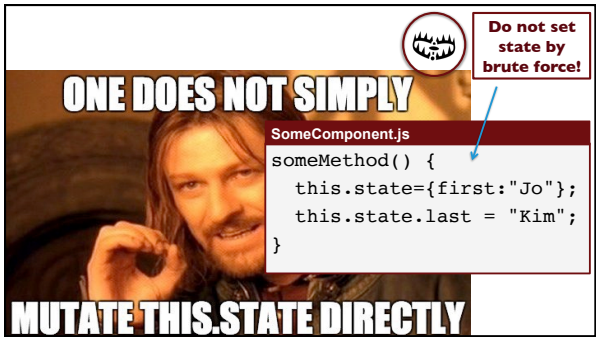
```
export class Person extends Component {
  constructor(props) {
    super(props)
    this.state = {first: props.first, last: props.last};
    // Or this.state = { ...props };
  }
  render() { ... }
}
```

if your constructor receives props, you have to pass those props to super.

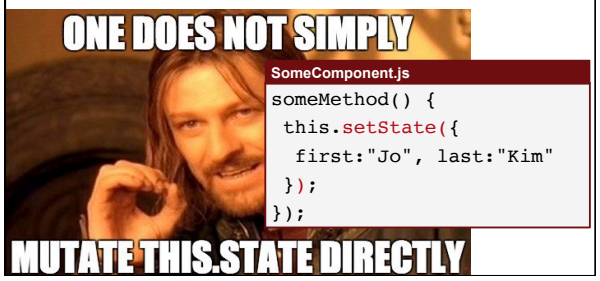
Be careful, though!

- React tries to be super-efficient. When a component is re-rendered React uses that object rather than disposing and re-instantiating it.
- The constructor is only called once on instantiation.
- So if the parent's props change, it will NOT re-set state in the child.

Changing state with setState()



setState() rerenders!



setState() is asynchronous

SomeComponent.js

```
someMethod() {
  console.log(this.state);
  this.setState({foo:"bar"});
  console.log(this.state);
}
```

These will be exactly the same!

Why? For performance; multiple setState() calls might not trigger multiple refreshes.

But you can make it synchronous

SomeComponent.js

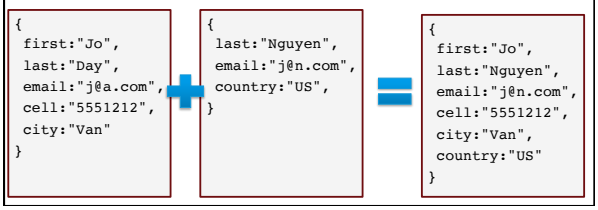
```
someMethod() {
  console.log(this.state);
  this.setState((ps,p) => ({foo:"bar"}));
  console.log(this.state);
}
```

The function can receive the previous state and props.

Pass in a function that returns the upserted state object.

setState upserts to the state

- If you have a huge state object with 10 keys but you only pass one key into setState, it adds that value (or updates it if it already exists).



It upserts? So how do you delete from state?

- For object properties:

SomeComponent.js

```
someMethod() {
 this.setState({first:undefined});
}
```

- For array elements

SomeComponent.js

```
removeFriend(friendToRemove) {
 this.setState({friends:this.state.filter(
  f => f !== friendToRemove)});
}
```

Forms in React

Say you have a React component that displays a person

ShowPeople.js

```
export class ShowPeople extends Component
 render() {
  {people.map(p => <ShowPerson
   first={p.first} last={p.last}
    eyes={p.eyes} email={p.email} />)}
 }
}
```

Is this props or state?

You also have an AddPerson component

AddPerson.js

```
export class AddPerson extends Component {
  render() {
    <form onSubmit={createPerson}>
      <input name="first" />
      <input name="last" />
    </form>
  }
  createPerson() { /* Create the person here */}
}
```

This is an 'uncontrolled component'

Is this props or state?

... and there's a ModifyPerson component that you pass data to.

ShowPeople.js

```
export class ShowPeople extends Component {
 render() {
  // Must pass values so they can be modified
  <ModifyPerson first={p.first}
    last={p.last} ... />
 }
 otherMethod() { /* Do stuff here */}
}
```

Is this props or state?

And inside of ModifyPerson, we use those values

ModifyPerson.js

```
export class ModifyPerson extends Component
 render() {
  <input value={this.props.first} />
  <input value={this.props.last} />
  <input value={this.props.email} />
 }
}
```

This cannot work b/c props can't change

Is this props or state?

So clearly we need both for a complete app

Props when being passed in
Then state when it is being changed.

ModifyPerson.js

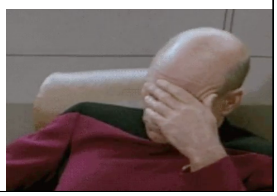
```
export class ModifyPerson extends Component {
 constructor(props) {
  super(props);
  this.state = props;
 }
 render() {
  <input value={this.state.first} />
  <input value={this.state.last} />
  <input value={this.state.email} />
 }
}
```

When form values are bound to state, it is a 'controlled component'

But we now have a new problem!

```
<input value={this.state.fname} />
```

- input.value is bound to state.
- When the user changes value by typing, the form immediately re-binds the value to what is in state
- End result: The user types and nothing appears to happen!



Here's a solution ... change state!

1. Add a change event handler to the input
2. In the handler, call setState() with the new value
3. Since setState() calls render, the component is re-drawn
4. The new value is now bound to the input
5. The user sees their changes!

ModifyPerson.js

```
export class ModifyPerson extends Component {
 ...
 render() {
  <input value={this.state.first}
    onChange={this.handleChange} />
 }
 handleChange(e) {
  this.setState(
   {first: e.target.value});
 }
}
```

Remember that events in React are "synthetic"

- They do not behave like native DOM events. They're better!
- The event object is normalized for all browsers
- The synthetic onChange event fires on every native keyup
- React allows the value property to be used with <select>s and <textarea>s
- For example ...

ChooseSchool.js

```
export class ChooseSchool extends Component {
 ...
 render() {
  <select value={this.state.college} onChange={this.ch}>
   <option value="10">Faber</option>
   <option value="27">South Harmon</option>
   <option value="51">Greendale</option>
   <option value="34">Wassamotta</option>
   <option value="86">UC Sunnydale</option>
  </select>
  <textarea value={this.state.description}
   onChange={this.ch}></textarea>
 }
}
```

So use checked and value

checked

```
<input type='radio' />
<input type='checkbox'/>
```

value

```
<input type='text' />
<input type='number' />
<input type='email' />
<input type='date' />
<input type='tel' />
```

- etc. etc.
- Literally everything else including ...

```
<select>
<textarea>
```

5 tips for handling state

1. If you can avoid state at all, do!

- Any state means that there are side-effects.
- The component is no longer a pure function.
- Understanding is harder
- Testing is harder
- Modification is harder
- Extension is harder

With Hooks, you can usually avoid state altogether!

2. Use a state manager

- State is the most complex thing in React.
- A state manager like Redux and MobX can greatly simplify your React components

3. Keep state as small as possible

- The smaller, the simpler
- Simpler is more abstract.

4. If you don't use something in the render method, it doesn't belong in state

- It can be just a class-scope variable.
- In other words, use this.whatever instead of this.state.whatever

5. Combine form inputHandlers

- There seems to be a pattern of very smart devs having a single do-it-all handler for forms.
- In this pattern there is one "handleChange" method and every onChange event uses it.

tl;dr

- If a component needs to keep track of changing data, this can't be props, it is "state"
- You shouldn't change state directly. Call setState() because it always triggers a re-render
- Controlled components have input fields bound to state. When that occurs we must create event handlers to copy the values to state and then re-render
- Try to avoid having state whenever possible. When you must, use a state container and keep state as simple as possible

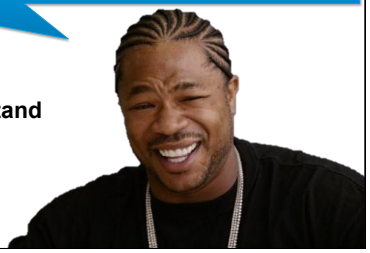
React Hooks

tl;dr

- We try to avoid using class-based components because they're more complex than SFCs
- Hooks add class-based component capability to SFCs yet retain most of their simplicity!
- useState() hooks allow us to manage state values
- useEffect() hooks enable asynchronous or impure activities
- useContext() hooks empower descendent components to reach back up to an ancestor to grab prop values in a controlled way

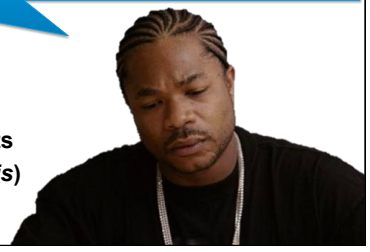
Stateless functional components are great!

Simple to understand
Pure
Testable



But SFCs don't support certain things.

State
Lifecycle events
Context (aka *this*)
Async



Hooks allow the best of both

- Hooks extend SFCs
- Simple like SFCs
- Handle state like class-based
- Warning! Hooks can't be used in class-based components

The 3 basic hooks

useState
useEffect
useContext

State setting hooks

useState

The useState() hook returns an array ...

1. A state object
2. A function to alter that object

```
const [state, setState] = useState(initialState);
```

A simple component

```
import React from "react";

export const FirstNameInput = () =>
 <input />;
```

A simple <input /> works okay because it is an uncontrolled component. But if we give it a value from props, it looks like it stops working

Make it class-based

```
import React, { Component } from "react";

export class FirstNameInput extends Component {
 constructor(props) {
  super(props);
  this.state.fName = props.fName;
 }

 onChange = e => {
  this.setState({
   fName: e.target.value
  });
 };

 render() {
  const { searchFieldValue } = this.state;
  return (
   <input
    onChange={this.onChange} value={this.state.fName}
   />
  );
 }
}
```

Now it functions, but it is complex.

With the useState hook ...

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

export const FirstNameInput = ({fName}) => {
 const [fName, setFName]= useState(fName);

 return (
  <input
    onChange={e => setFName(e.target.value)}
    value={fName}
  />
 );
};
```

From about 19 lines to about 8 lines.

Lazy initialState

- When re-rendering, the state persists.
- ie. React saves the last value and uses it again when re-drawing. It only gets the initialState on the first render.

Unlike setState(), the state object is completely replaced!

State is clobbered, not appended!

- If you want to add to state, you must use object spread:

```
setFoo({
 ...oldFoo,
 key: newValue,
 key2:newValue2
});
```

Side effect hooks

useEffect

Animations and timers are asynchronous

Sometimes we want to run impure functions

Ex: with the ComponentDidUpdate() lifecycle event.

But SFCs must be synchronous and pure!

useEffects can solve those problems for us.

ShowName.js

```
import React, {useEffect} from 'react';
import Foo from './Foo.js';
export const ShowName = props => {
 useEffect(Foo.asyncFunc(props.foo));
 return <><Foo /></>
};
```

- Non-blocking. Runs asynchronously
- Has access to the local variables that are in scope

Timing

- Since it can run async things, it runs after the render function paints but before the next repaint.
- If you need it to run before (like if a DOM change needs to happen before rendering), use useLayoutEffect instead.
- useLayoutEffect is exactly the same except in timing.

Optionally return a destructor

```
useEffect( () => {
 const aListener = createSomeListener();
 return () => aListener.closeListener();
});
```

Context hook

useContext

Have you ever wanted to pass state down through, like 10 levels?

useContext() to the rescue!

You set a value at a high level ...

App.js

```
import React from 'react';
import { SomeComponent } from './SomeComponent.js';

export const divisions = React.createContext({
 eastern:["Atlantic","Central","Southeast"],
 western:["Northwest","Pacific","Southwest"],
});

export App = () => (
 <SomeComponent></SomeComponent>
);
```

... and read it back at any lower level

Standings.js

```
import React, { useContext } from 'react';
import { divisions } from 'App';

const divisions = useContext(divisions);

export Standings = () => (
 <>
 <section>
  <h2>Eastern division standings</h2>
  {divisions["eastern"].map(d => <DivisionStanding div='d'></DivisionStanding>)}
 </section>
 <section>
  <h2>Western division standings</h2>
  {divisions["western"].map(d => <DivisionStanding div='d'></DivisionStanding>)}
 </section>
 </>
);
```

tl;dr

- We try to avoid using class-based components because they're more complex than SFCs
- Hooks add class-based component capability to SFCs yet retain most of their simplicity!
- useState() hooks allow us to manage state values
- useEffect() hooks enable asynchronous or impure activities
- useContext() hooks empower descendent components to reach back up to an ancestor to grab prop values in a controlled way