React for D3.js Users

Michael Freeman, University of Washington
@mf_viz

Objectives

Understand the **role of React** in a web visualization project

Learn the fundamental principles of **how React works**

Be able to use React to **scaffold D3.js visualizations**

Outline

Brief introductions

Workshop expectations and resources

Overview of React

Creating React components

Tracking application state

Combining D3 and React

Time pending: React tools overview

Outline

Brief introductions

Workshop expectations and resources

Overview of React

Creating React components

Tracking application state

Combining D3 and React

Time pending: React tools overview

Faculty member at UW iSchool

Courses I Teach

Intellectual Foundations of Informatics

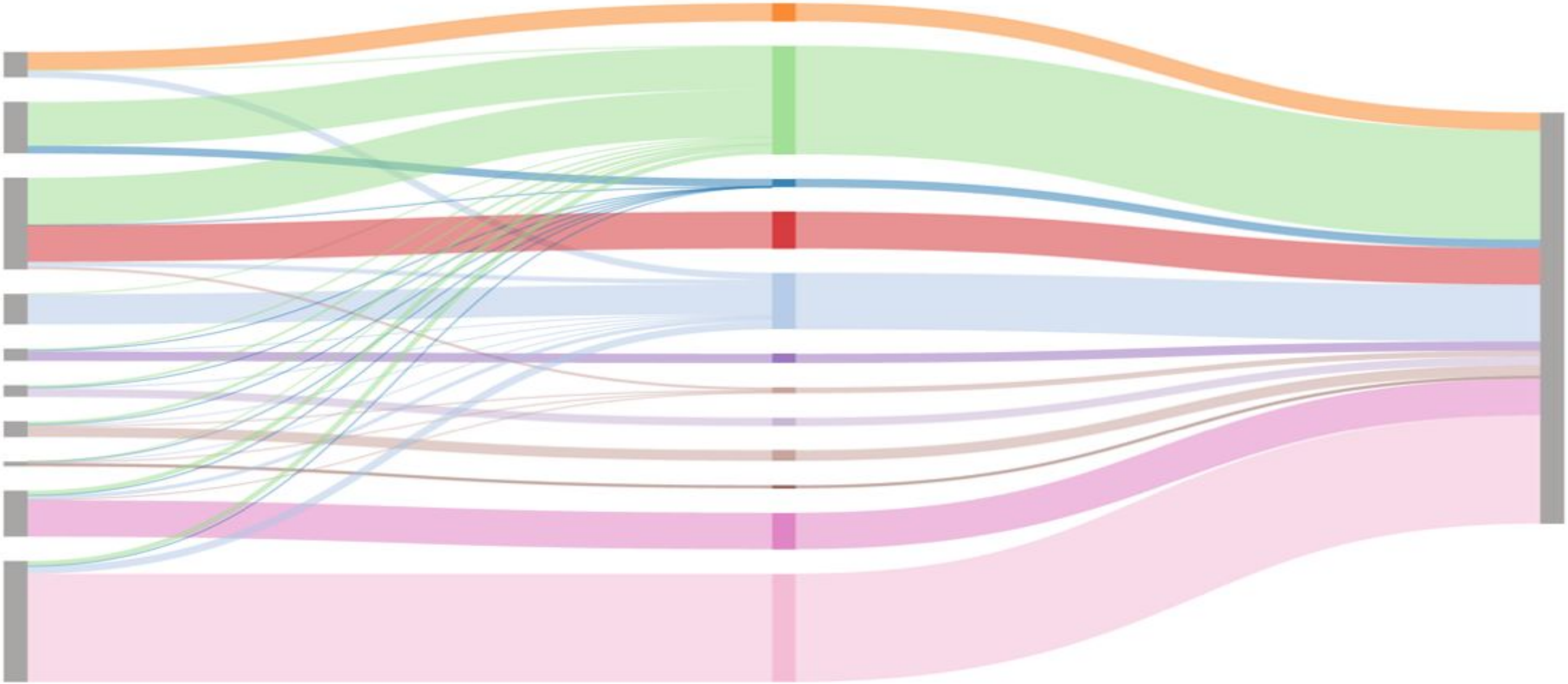
Technical Foundations of Informatics

Population Health Informatics

Client-side Web Development

Introduction to Data Science

Interactive Data Visualization



Institute for Health Metrics and Evaluation

My Background (link)

Briefly Introduce Yourselves

Name, profession, where you're from, etc.

Your technical and design background

Why you enrolled in this workshop?

Outline

Brief introductions

Workshop expectations and resources

Overview of React

Creating React components

Tracking application state

Combining D3 and React

Time pending: React tools overview

Process Expectations

Respect each others' time, intelligence, and experiences

Work collaboratively

Ask questions as you have them

Do you have any other expectations of me or one another?

Technical Expectations

What I expect of your background

- You are comfortable using D3 and underlying languages (JS, HTML, CSS)
- You are familiar with basic data manipulation in JS (`filter()`, `map()`, etc.)

What will be covered in this workshop

- Foundational React skills
- *One* suggested structure for using D3 in a React application
- Building React applications with multiple, interactive, connected D3 visualizations

What won't be covered in this workshop

- D3 basics, a comparison of React + D3 approaches, using React in production

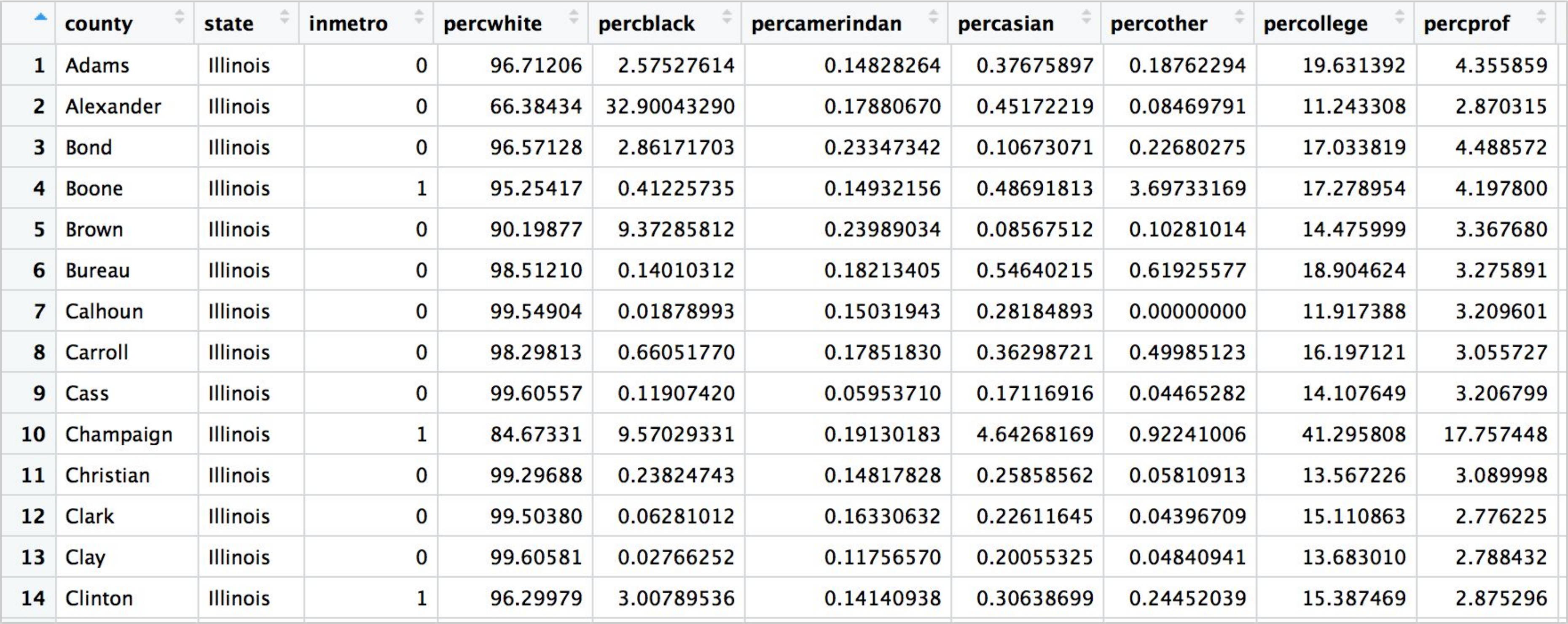
Format and Resources

Hands on workshop

Mix of lecture // demos // exercises // code-alongs (the balance will be up to you)

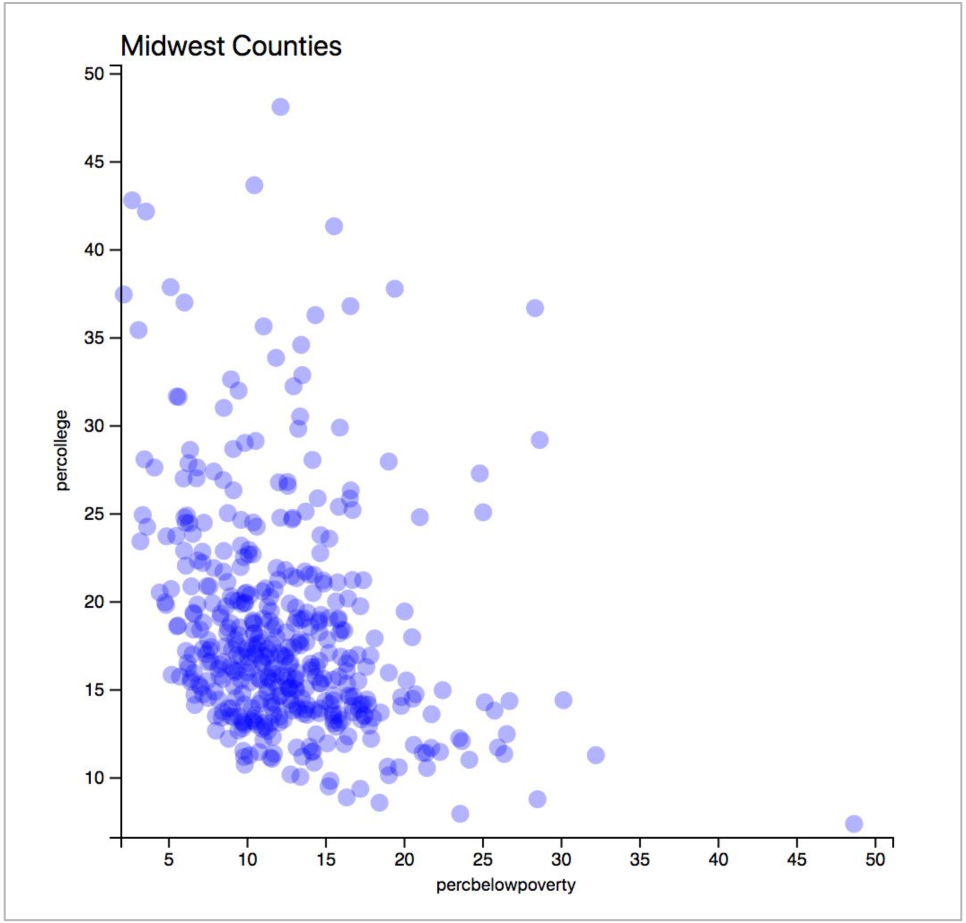
All resources are available online (and will remain there):

- Hosted completed exercises and demos: mfviz.com/react-d3
- GitHub page with exercises and demos: github.com/mkfreeman/react-d3

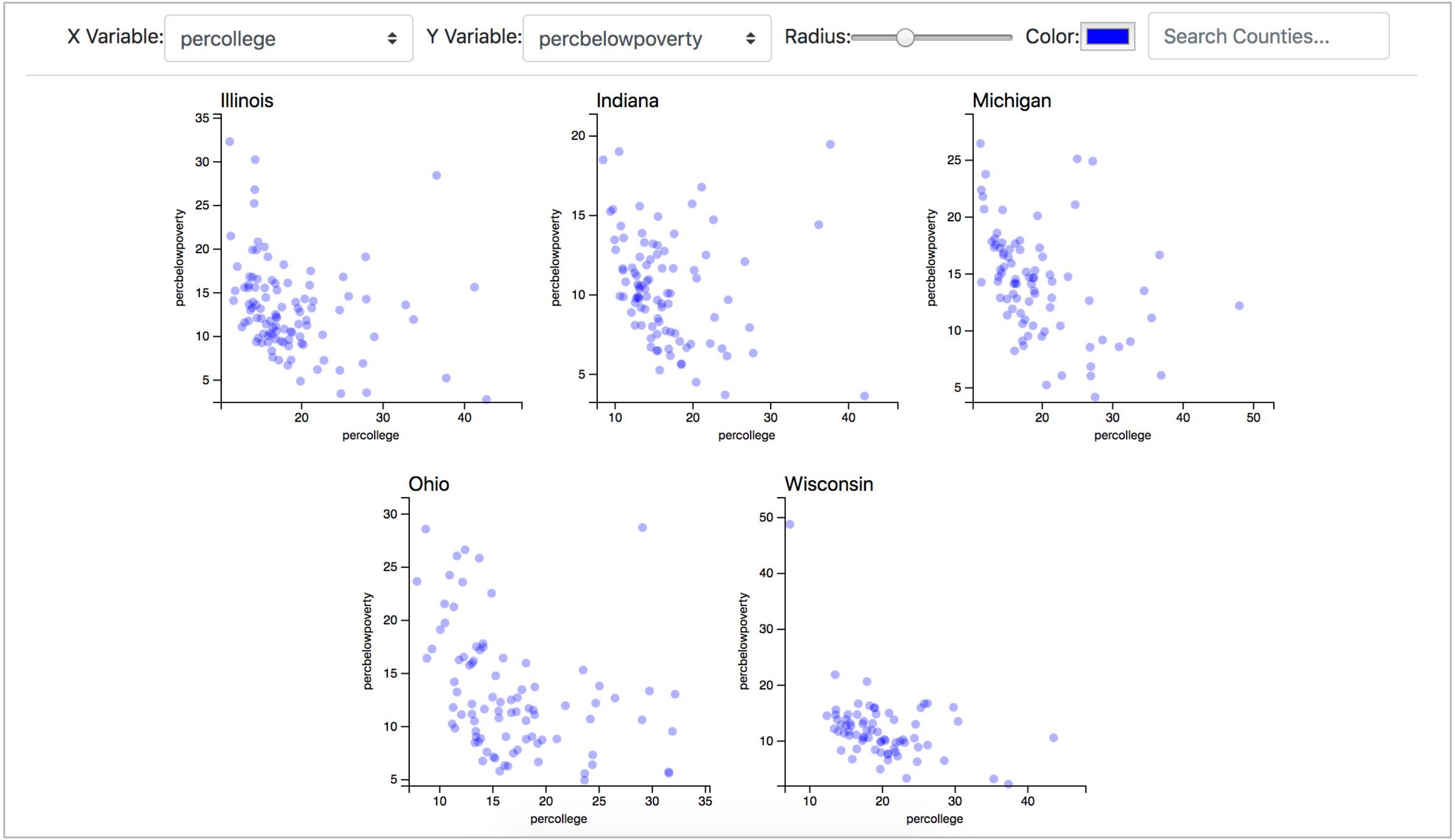


	county	state	inmetro	percwhite	percblack	percamerindan	percasian	percother	percollege	percprof
1	Adams	Illinois	0	96.71206	2.57527614	0.14828264	0.37675897	0.18762294	19.631392	4.355859
2	Alexander	Illinois	0	66.38434	32.90043290	0.17880670	0.45172219	0.08469791	11.243308	2.870315
3	Bond	Illinois	0	96.57128	2.86171703	0.23347342	0.10673071	0.22680275	17.033819	4.488572
4	Boone	Illinois	1	95.25417	0.41225735	0.14932156	0.48691813	3.69733169	17.278954	4.197800
5	Brown	Illinois	0	90.19877	9.37285812	0.23989034	0.08567512	0.10281014	14.475999	3.367680
6	Bureau	Illinois	0	98.51210	0.14010312	0.18213405	0.54640215	0.61925577	18.904624	3.275891
7	Calhoun	Illinois	0	99.54904	0.01878993	0.15031943	0.28184893	0.00000000	11.917388	3.209601
8	Carroll	Illinois	0	98.29813	0.66051770	0.17851830	0.36298721	0.49985123	16.197121	3.055727
9	Cass	Illinois	0	99.60557	0.11907420	0.05953710	0.17116916	0.04465282	14.107649	3.206799
10	Champaign	Illinois	1	84.67331	9.57029331	0.19130183	4.64268169	0.92241006	41.295808	17.757448
11	Christian	Illinois	0	99.29688	0.23824743	0.14817828	0.25858562	0.05810913	13.567226	3.089998
12	Clark	Illinois	0	99.50380	0.06281012	0.16330632	0.22611645	0.04396709	15.110863	2.776225
13	Clay	Illinois	0	99.60581	0.02766252	0.11756570	0.20055325	0.04840941	13.683010	2.788432
14	Clinton	Illinois	1	96.29979	3.00789536	0.14140938	0.30638699	0.24452039	15.387469	2.875296

Today's Dataset: demographics in Midwestern state counties (from R's ggplot2 package)



Where we'll start: comfortable with D3 basics (01-d3-basics-demo)



X Variable: percollege

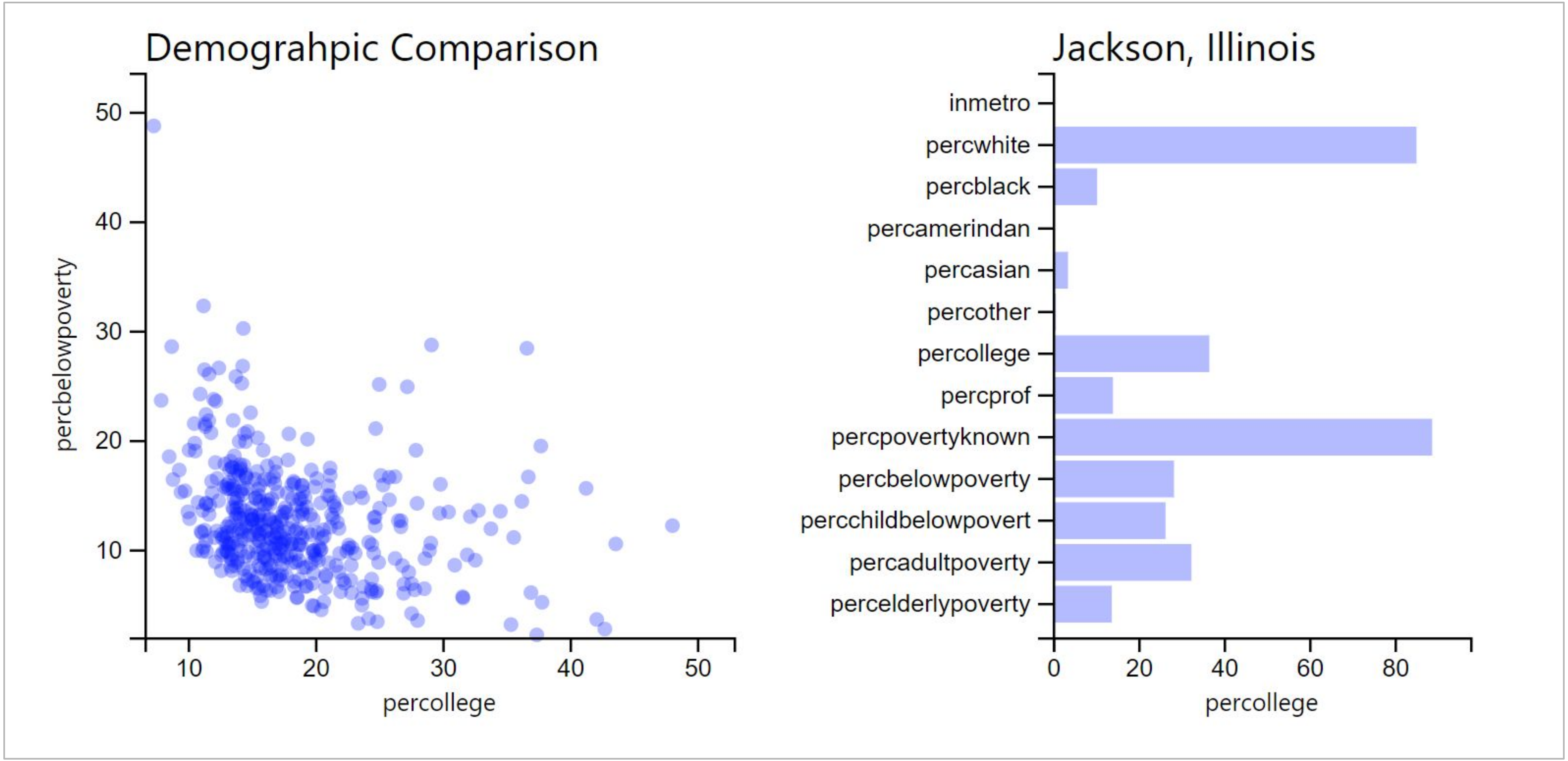
Y Variable: percbelowpoverty

Radius:

Color: 🟦

Search Counties...

Where we'll end: controlling D3 charts with React (09-small-multiples-exercise)



Where we'll end: Connect D3 charts in React (10-lifting-up-state-exercise/)

Outline

Brief introductions

Workshop expectations and resources

Overview of React

Creating React components

Tracking application state

Combining D3 and React

Time pending: React tools overview

What is React?

JavaScript library for building user interfaces

Build **reusable components** to use throughout your application (`<ScatterPlot />`)

Use a **one-directional data-flow** to pass information into components

React components will re-render based on their **lifecycle**

Commonly written in **JSX / ES6**

Why React?

Why structure your code with a framework?

- **Separation of concerns**: abstract your visualization code from data management
- **Reusability**: create components that you can use throughout / across projects
- **Reliability**: other people have written more reliable code that triggers updates

Why use the React framework?

- **Popularity**: quickly evolving, large open source community
- **Data centric**: based on a philosophy of observing data updates

We'll continue to use D3 to manipulate the DOM and calculate element positions: React will take care of "everything else".

Outline

Brief introductions

Workshop expectations and resources

Overview of React

Creating React components

Tracking application state

Combining D3 and React

Time pending: React tools overview

Creating React Components: Overview

What we'll discuss:

- Using **classes** in JavaScript
- Loading the React library
- Writing **jsx** code
- Creating **React components** and passing in *properties*

```javascript
// Create a new class Pet
class Pet {

    // Constructor function is called when the class is instantiated
    constructor(name) {
        this.name = name;
    }
}

// Create a new class Dog by extending the class Pet
class Dog extends Pet {

    constructor(name, sound) {
        super(name); // call parent class constructor function
        this.sound = sound;
    }

    // Add a new method `bark` to the class
    bark() {
        console.log(`${this.sound}! My name is ${this.name}`);
    }
}

// Create an instance of a dog
let myDog = new Dog("Mocha", "Woof Woof!");
```

Constructing Classes in JavaScript

See: 02-class-demo

```html
<!-- Read in React scripts -->
<script src="https://unpkg.com/react@16/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>


<!-- Read in Babel Script to transpile your jsx code to syntax interpretable by your browser -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.26.0/babel.min.js"></script>
```

A simple set-up: load library + babel (the are more sophisticated approaches)

```
// A regular JavaScript constant
const name = "Mike";


// Use JSX to store an HTML element in a JavaScript variable
const element = (
    <h1 className="large">
        Hello, {name}!
    </h1>
);


// Render your element to the DOM in your `root` element
ReactDOM.render(
    element,
    document.getElementById('root')
);
```

```
<!-- Load your script **as jsx** in your index.html file -->

<script type="text/jsx" src="js/main.js"></script>
```

JSX: Finally, writing HTML in your JavaScript (and JavaScript in your HTML fragments!)

```
// Create a component that represents someone's biography
class Bio extends React.Component {

    // React components have a render method to describe how to draw them on the DOM
    render() {

        return (

            <div>
                <h1>Arthur</h1>
                <p>Hello, my name is Arthur and I am an Aardvark</p>
            </div>

        )
    }
}

// Render the Bio component in the `root` element
ReactDOM.render(
    <Bio />,
    document.getElementById('root')
);
```

What if we want these to be dynamic?

Creating React components: extending the React.Component class to render HTML

```
// Create a component that represents someone's biography
class Bio extends React.Component {

    // React components have a render method to describe how to draw them on the DOM
    render() {

        return (

            <div>
                <h1>Arthur</h1>
                <p>Hello, my name is {this.props.name} and I am an {this.props.description}</p>
            </div>

        )
    }
}

// Render the Bio component in the `root` element
ReactDOM.render(
    <Bio name="Arthur" description="aardvark" />,
    document.getElementById('root')
);
```

Creating React components: pass in **props**

See: 03-component-demo

04-react-intro-exercise

Outline

Brief introductions

Workshop expectations and resources

Overview of React

Creating React components

Tracking application state

Combining D3 and React

Time pending: React tools overview

Component Lifecycle

React components execute specific ***lifecycle methods*** at given times, including:

- **Mounting**: when a component is ***added*** to the DOM
- **Updating**: when a component's ***state or props*** are changed
- **Unmounting**: when a component is ***removed*** from the DOM

At each event, *multiple functions* are invoked, most importantly, the **render function** (see docs, and blog)

When your ***props or state*** change, your component will **re-render** automatically!

Component State

Anything that *changes* you should track in the **state** of an application

You should set an initial state in the `constructor()` method:

```
// Add a constructor method to set the state
constructor(props) {
    super(props);
    this.state = {
        search: ""
    };
}
```

Then, add event handlers to change the state using the `setState()` method

```
<input onChange={(event) => this.setState({ search: event.target.value })} />
```

Props v.s. State

Properties represent the data that is **passed to a component** (can't be changed)

Component **state** represents something that may change about a component

State is reserved only for interactivity, that is, data that changes over time (link)

Do not set state directly*: instead, use: `this.setState({key:value})`

*Except in constructor function

See: 05-state-demo

06-state-exercise

Outline

Brief introductions

Workshop expectations and resources

Overview of React

Creating React components

Tracking application state

Combining D3 and React

Time pending: React tools overview

Reminder: We'll continue to use D3 to manipulate the DOM and calculate element positions: React will take care of "everything else".

React + D3: Who does what?

Manage data, keep track of settings (i.e., *what data* to visualize): **React**

Append static elements to the DOM (i.e., <g> elements that hold circles): **React**

Compute visual layouts: **D3**

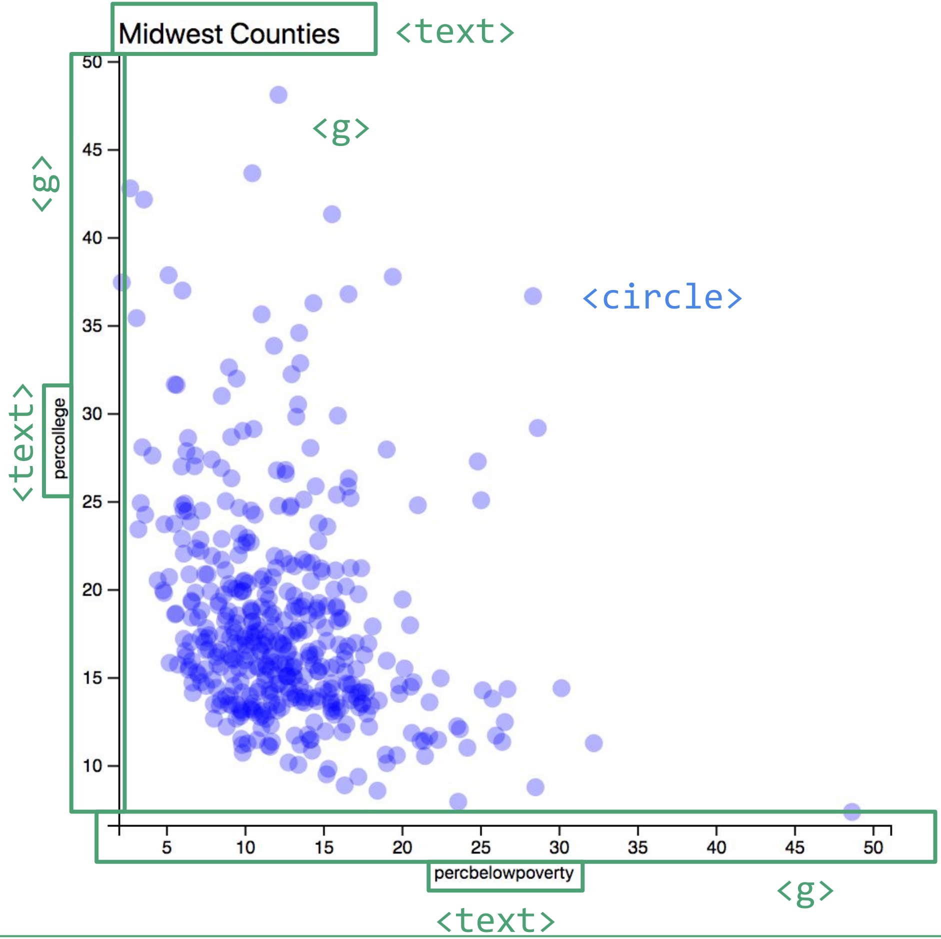
Render elements that are based on binding data to DOM (i.e., circles): **D3**

"Highly contested", may depend on type of app, team, data size, etc.

```
<ScatterPlot
     title={this.state.title}
     xTitle={this.state.xVar}
     yTitle={this.state.yVar}
     data={this.state.data}
     radius={this.state.radius}
     color={this.state.color}
/>
```

How does this component work?

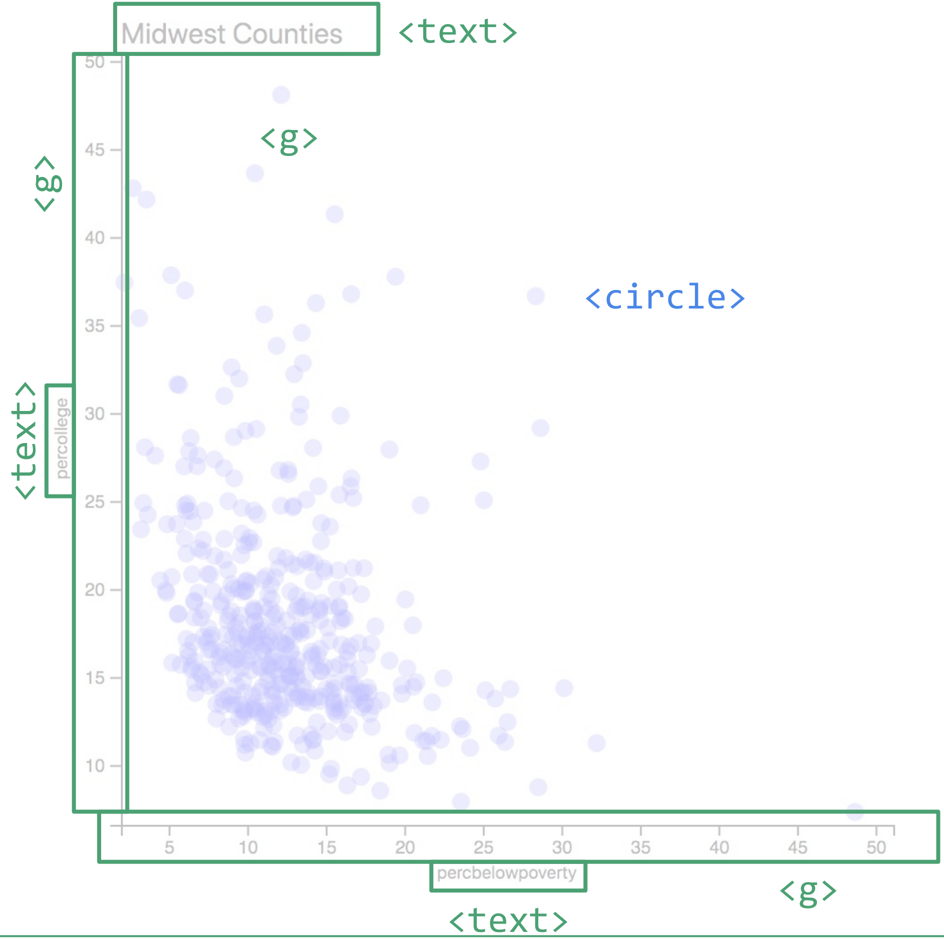
The vision: creating reusable visualization components



Added once (though may update)

Dynamically changing

Chart Anatomy: remember what we're looking at



Render with React
Added once (though may update)

Render with D3
Dynamically changing

Chart Anatomy: remember what we're looking at

```
<div className="chart-wrapper">
      <svg className="chart" width={this.props.width} height={this.props.height}>
            <g transform={`translate(${this.props.margin.left}, ${this.props.margin.top})`} />

            {/* Axes */}
            <g transform={`translate(${this.props.margin.left}, ${this.props.height - this.props.margin.bottom})`}></g>
            <g transform={`translate(${this.props.margin.left}, ${this.props.margin.top})`}></g>

            {/* Axis labels */}
            <text transform={`translate(${this.props.margin.left},15)`}>{this.props.title}</text>
            <text className="axis-label" transform={`translate(${this.props.margin.left + this.drawWidth / 2},
                  ${this.props.height - this.props.margin.bottom + 30})`}>{this.props.xTitle}</text>

            <text className="axis-label" transform={`translate(${this.props.margin.left - 30},
                  ${this.drawHeight / 2 + this.props.margin.top}) rotate(-90)`}>{this.props.yTitle}</text>
      </svg>
</div>
```

The HTML elements of a chart

The Trick: expose DOM elements so that D3 can manipulate them.

```
render() {
       return (
           <div className="chart-wrapper">
               <svg className="chart" width={this.props.width} height={this.props.height}>
                   <text transform={`translate(${this.props.margin.left},15)`}>{this.props.title}</text>
                   <g ref={(node) => { this.chartArea = node; }}
                       transform={`translate(${this.props.margin.left}, ${this.props.margin.top})`} />
...................................................


       )
}
```

`this.chartArea` will store a *reference* to the `<g>` element so that you can manipulate it

Expose DOM elements so you can manipulate them

```
// Whenever the component updates, select the <g> from the DOM, and use D3 to manipulate circles
    componentDidUpdate() {
       let circles = d3.select(this.chartArea).selectAll('circle').data(this.props.data);

       // Use the .enter() method to get your entering elements, and assign their positions
       circles.enter().append('circle')
           .merge(circles)
           ..........
   }

   render() {
       return (
           <div className="chart-wrapper">
               <svg className="chart" width={this.props.width} height={this.props.height}>
                   <text transform={`translate(${this.props.margin.left},15)`}>{this.props.title}</text>
                   <g ref={(node) => { this.chartArea = node; }}
                       transform={`translate(${this.props.margin.left}, ${this.props.margin.top})`} />
....................................................
       )
}
```

Expose DOM elements so you can manipulate them

See: 07-d3-demo/

Putting it all together

Each time your data updates, make the following changes:

- Update the scale functions (i.e., min and max values): `updateScales()`
- Re-render axes to reflect the new scales: `updateAxes()`
- Re-draw the new data with the new scales: `updatePoints()`

08-scatter-exercise

09-small-multiples-exercise

Connecting Charts

In a more advanced application, you'll want to create multiple connected charts

An event on one chart should be able to affect the data on another chart

How have you done this before?

How can we do this in React?

Lifting up state

The data flow in React is **one directiona**l

We can pass **functions as properties** to child components

Imagine a function that changes a *component's state*

```
updateXvar(d) {this.setState({ xVar: d})}
```

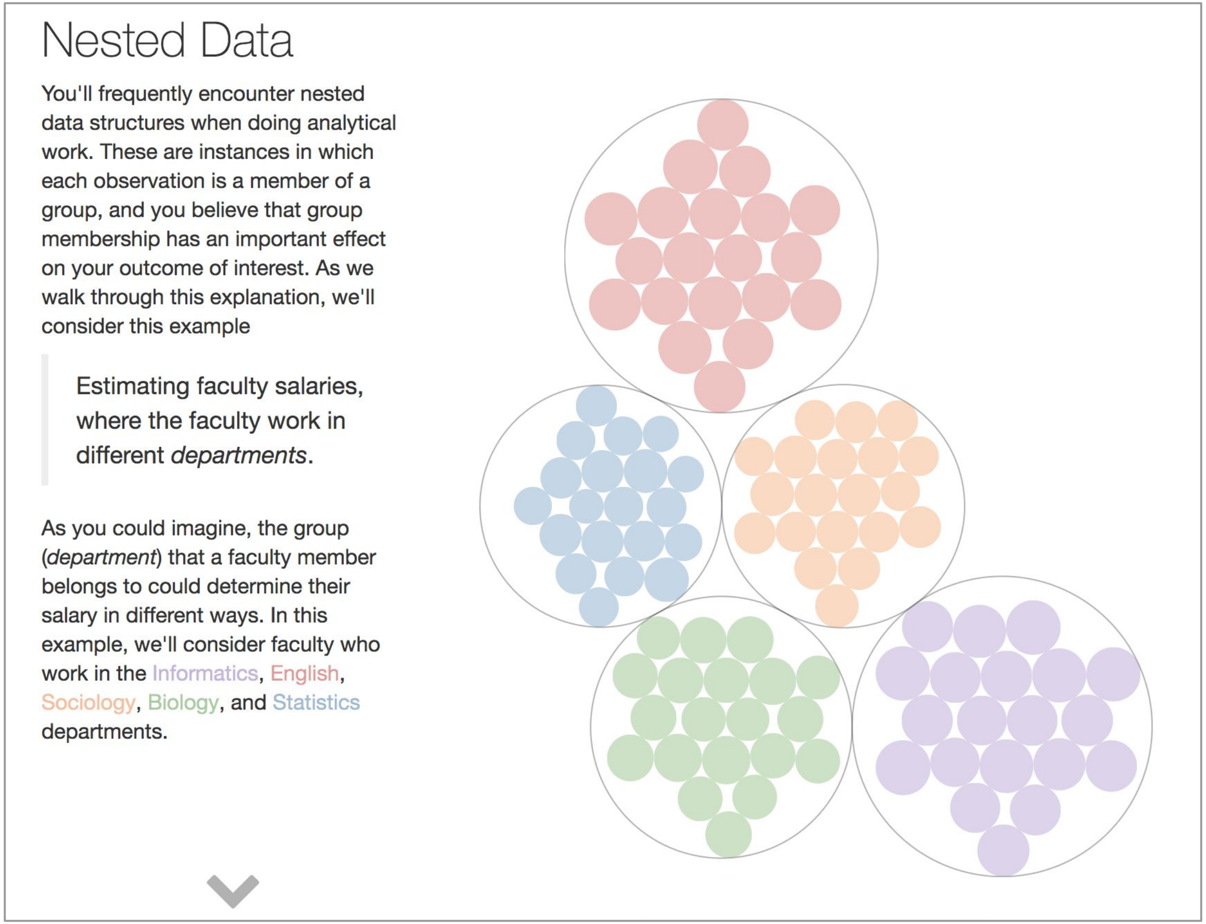
Pass this function to a child component *as a property*

```
<BarChart update={(d) => this.updateXvar(d)} />
```

Use the property to have the component *do whatever it should* in an event

```
ele.on('mouseover', (d) => { this.props.update(d.label)})
```

10-lifting-up-state-exercise/



Nested Data

You'll frequently encounter nested data structures when doing analytical work. These are instances in which each observation is a member of a group, and you believe that group membership has an important effect on your outcome of interest. As we walk through this explanation, we'll consider this example

> Estimating faculty salaries, where the faculty work in different *departments*.

As you could imagine, the group (*department*) that a faculty member belongs to could determine their salary in different ways. In this example, we'll consider faculty who work in the Informatics, English, Sociology, Biology, and Statistics departments.

React + D3 in action (link)

Outline

Brief introductions

Workshop expectations and resources

Overview of React

Creating React components

Tracking application state

Combining D3 and React

Time pending: React tools overview

Node + NPM

Node is a *JavaScript Runtime* that allows us to run JavaScript on a server

NPM is a package manager (think of it like `install.packages` or `pip install`)

We'll use these tools to **install packages** for using react

More info on installation: see this course book

Documenting projects and tracking packagers

As projects scale, they'll need to be documented and shared

This commonly involves describing your *dependencies* in a **package.json** file

```
{
  "name": "Your project",
  "version": "1.0.0",
  "description": "A project with an example package.json",
  "author": "Michael Freeman",
  "license": "ISC",
  "dependencies": {
    "lodash": "^4.17.4"
    },
}
```

You can install the packages listed by running `npm install` in the directory

React Tools

Our current approach (using Babel.js) is a bit limited:

- Introduces a slight delay for the client
- Doesn't leverage modern JavaScript project structure

Need a way to **compile JavaScript** for projects

Lots of different ways to do this (Gulp, Browserify, Webpack)

create-react-app provides a default configuration + processes optimized for React

```
# Globally install the create-react-app command-line utility: only do this once!
npm install -g create-react-app

# Create a project called my-app (in your current directory)
create-react-app my-app

# Change your directory to my-app
cd my-app/

# Start running a local server with your project: visible at localhost:3000
npm start
```

Using Create React App

```
# Clone the repository, as usual
git clone URL/TO/my-app.git

# Change your directory to my-app, as usual
cd my-app/

# Install packages listed in package.json, as usual
npm install

# Start running a local server with your project: visible at localhost:3000
npm start
```

If a project has already been created with create-react-app

Importing Functions

Because we're using the create-react-app scaffolding, we can import functions directly into our JavaScript file

Import functions from a package:

```
# Install in your project from the terminal
npm install --save lodash

// Load a utility into your script, then use it as you like
import uniq from 'lodash'
```

```
// In our utility.js file, write a function that converts feet to meters
function feetToMeters(feet) {
    return feet / 3.28084
}

// Write another function metersToFeet
function metersToFeet(meters) {
    return meters * 3.28084
}

// Export each *named* function
export {feetToMeters, metersToFeet} // named exports
```

Exporting your own (named) functions to use in other scripts

Importing your own (named) functions from other scripts

```
// Importing options for a named export from Utility.js

// Option 1 -------- import named exports by name
import {feetToMeters, metersToFeet} from './Utility';
let meters = feetToMeters(30); // use function


// Option 2 -------- import named exports by name AS a shorter name
import feetToMeters as f2m from './Utility'
let meters = f2m(30); // use function


// Option 3 -------- import all named exports with a prefix
// Import our own components
import * as Utilities from './Utility';
let meters = Utilities.feetToMeters(30); // use function
```

```
# Install the gh-pages package and save it
npm install gh-pages --save

# Edit your package.json {see next slide}

# Build and deploy your project to the gh-pages branch
npm run deploy
```

Deploying to your gh-pages branch

```
{
    "name": "page",
    "homepage": "http://YOUR-DOMAIN/project-name",
    "dependencies": {
        "gh-pages": "^1.1.0",
        "react": "^16.2.0",
    },
    "scripts": {
        "start": "react-scripts start",
        "deploy": "npm run build&&gh-pages -d build"
    }
}
```

Indicate where your project will be hosted

Add the `deploy` script to npm, enabling command line functionality

Deploying to your gh-pages branch

Thank you!

Michael Freeman, University of Washington

@mf_viz