Swizec Teller

React+d3.js

Build data visualizations with React and d3.js

React+d3.js

Build data visualizations with React and d3.js

Swizec Teller

This book is for sale at http://leanpub.com/reactd3js

This version was published on 2016-04-11

Tweet This Book!

Please help Swizec Teller by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I can’t wait to start reading React+d3.js!

The suggested hashtag for this book is #reactd3js.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#reactd3js

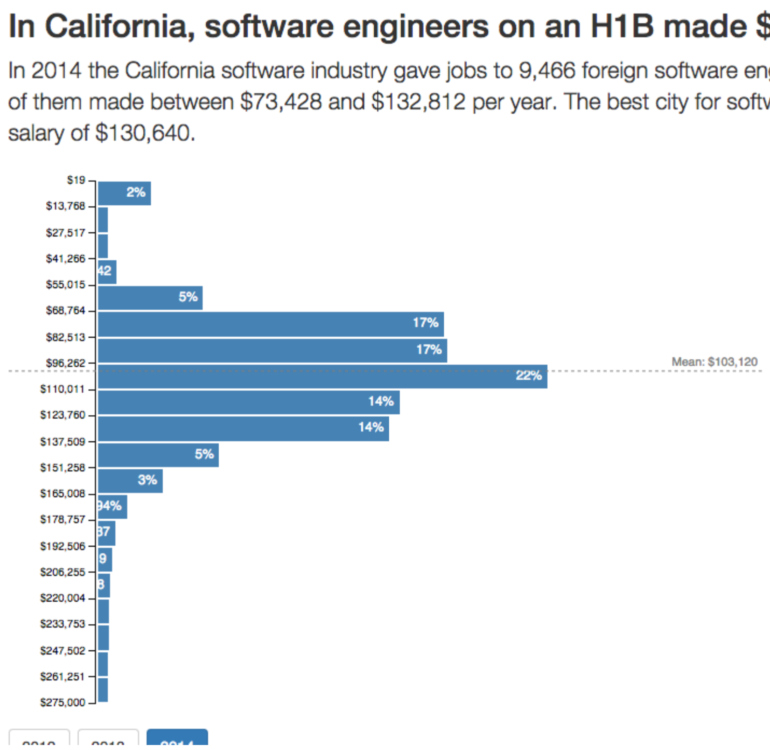
Also By Swizec Teller

Contents

Visualizing data with React and d3.js

Welcome to the main part of React+d3.js. I'm going to walk you through an example of building a visualization using React and d3.js.

We're going to build a subset of the code I used to visualize the salary distribution of H1B workers[1] in the United States software industry.



In California, software engineers on an H1B made $103,120/year in 2014

In 2014 the California software industry gave jobs to 9,466 foreign software engineers, 17% more than the year before. Most of them made between $73,428 and $132,812 per year. The best city for software engineers was Menlo Park with an average salary of $130,640.

H1B salary distribution for engineers in California

If you skipped the environment setup section, make sure you've installed the following dependencies:

- d3.js
- React
- Lodash

You should also have some way of running a static file server. I like having a simple node.js server that enables hot loading via Webpack.

[1]http://swizec.github.io/h1b-software-salaries/#2014-ca-engineer

We’re going to put all our code in a `src/` directory, and serve the compiled version out of `static/`. A `public/data/` directory is going to hold our data.

Before we begin, you should copy the dataset from the stub project you got with the book. It should be in the `public/data/` directory of your project.

JSX

We’re going to write our code in JSX, a JavaScript syntax extension that lets us treat XML-like data as normal code. You can use React without JSX, but I feel that it makes React’s full power easier to use.

The gist of JSX is that we can use any XML-like string just like it is part of JavaScript. No Mustache or messy string concatenation necessary. Your functions can return straight-up HTML, SVG, or XML.

For instance, the code that renders our whole application is going to look like this:

A basic Render

```
React.render(
    <H1BGraph url="data/h1bs.csv" />,
    document.querySelectorAll('.h1bgraph')[0]
);
```

Which compiles to:

JSX compile result

```
React.render(
    React.createElement(H1BGraph, {url: “data/h1bs.csv”}),
    document.querySelectorAll(‘.h1bgraph’)[0]
);
```

As you can see, HTML code translates to `React.createElement` calls with attributes translated into a property dictionary. The beauty of this approach is two-pronged: you can use React components as if they were HTML tags and HTML attributes can be anything.

You’ll see that anything from a simple value to a function or an object works equally well.

I’m not sure yet whether this is better than separate template files in Mustache or whatever. There are benefits to both approaches. I mean, would you let a designer write the HTML inside your JavaScript files? I wouldn’t, but it’s definitely better than manually +-ing strings or Angular’s approach of putting everything into HTML. Considerably better.

If you skipped the setup section and don’t have a JSX compilation system set up, you should do that now. You can also use the project stub you got with the book.

The basic approach

Because SVG is an XML format that fits into the DOM, we can assemble it with React. To draw a `100px` by `200px` rectangle inside a grouping element moved to `(50, 20)` we can do something like this:

A simple rectangle in React

```
render: function () {
    return (
        <g transform="translate(50, 20)">
            <rect width="100" height="200" />
        </g>
    );
}
```

If the parent component puts this inside an `<svg>` element, the user will see a rectangle. At first glance, this looks cumbersome compared to traditional d3.js. But look closely:

A simple rectangle in d3.js

```
d3.select("svg")
  .append("g")
  .attr("transform", "translate(50, 20)")
  .append("rect")
  .attr("width", 100)
  .attr("height", 200);
```

The d3.js approach outputs SVG as if by magic and looks cleaner because it's pure JavaScript. But it's a lot more typing and function calls for the same result.

Well, actually, the pure d3.js example is 10 characters shorter. But trust me, React is way cooler.

My point is that dealing with the DOM is not d3.js's strong suit, especially once you're drawing a few thousand elements and your visualization slows down to a leisurely stroll… if you're careful.

You always have to keep an eye on how many elements you're updating. React gives you all of that for free. Its primary purpose in life is knowing exactly which elements to update when some data changes.

We're going to follow this simple approach:

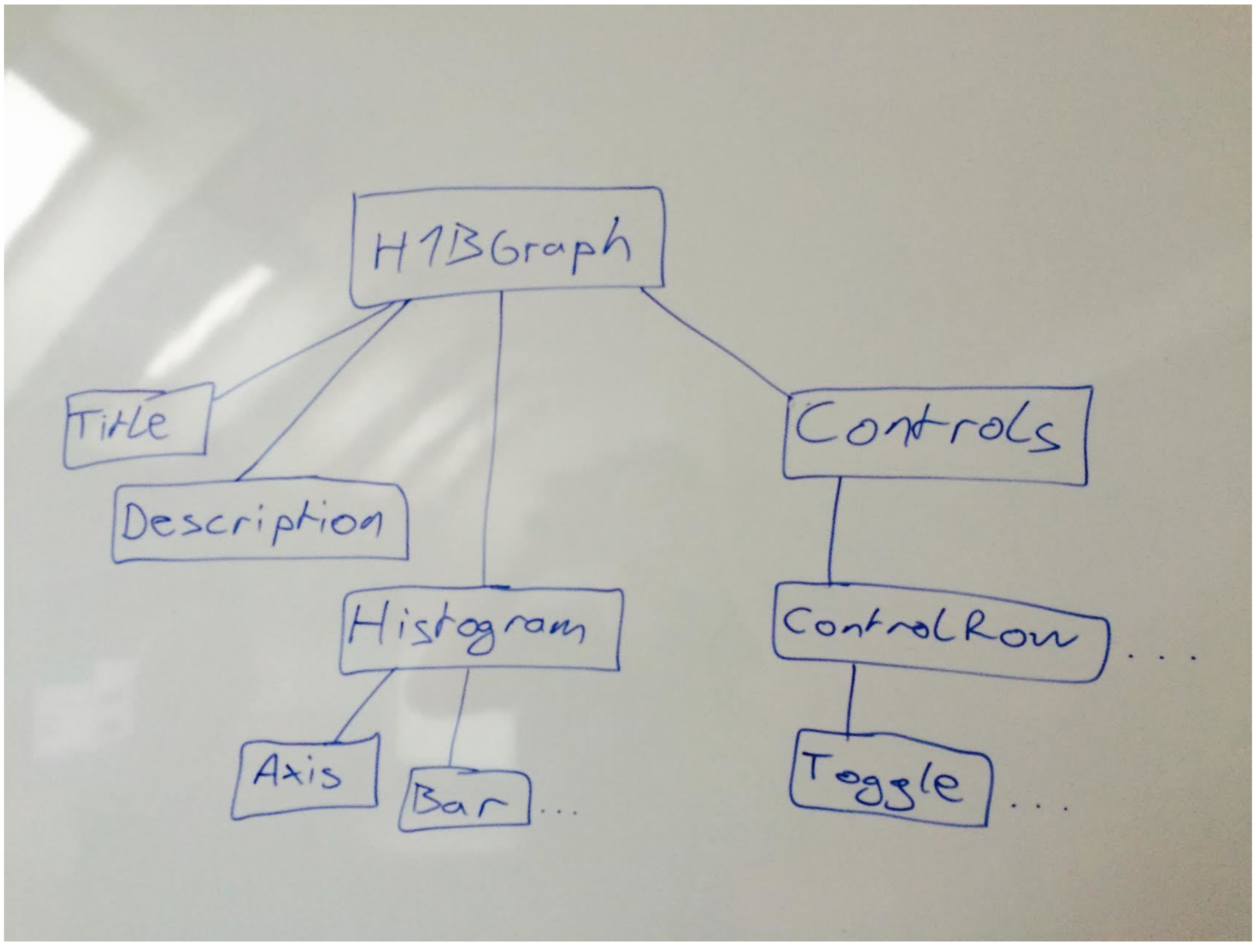
- React owns the DOM
- d3 calculates properties

This way we leverage both React and d3.js for their best functions and not much else.

The architecture

To make our lives easier, we're going to use a flow-down architecture where the entire application state is stored in one place. The architecture itself inspired by Flux, but the way we're structuring it uses less code and is easier to explain. The downside is that our version doesn't scale as well as Flux would.

If you don't know about Flux, don't worry; the explanations are self-contained. I only mention Flux to make your Googling easier and to give you an idea of how this approach compares.



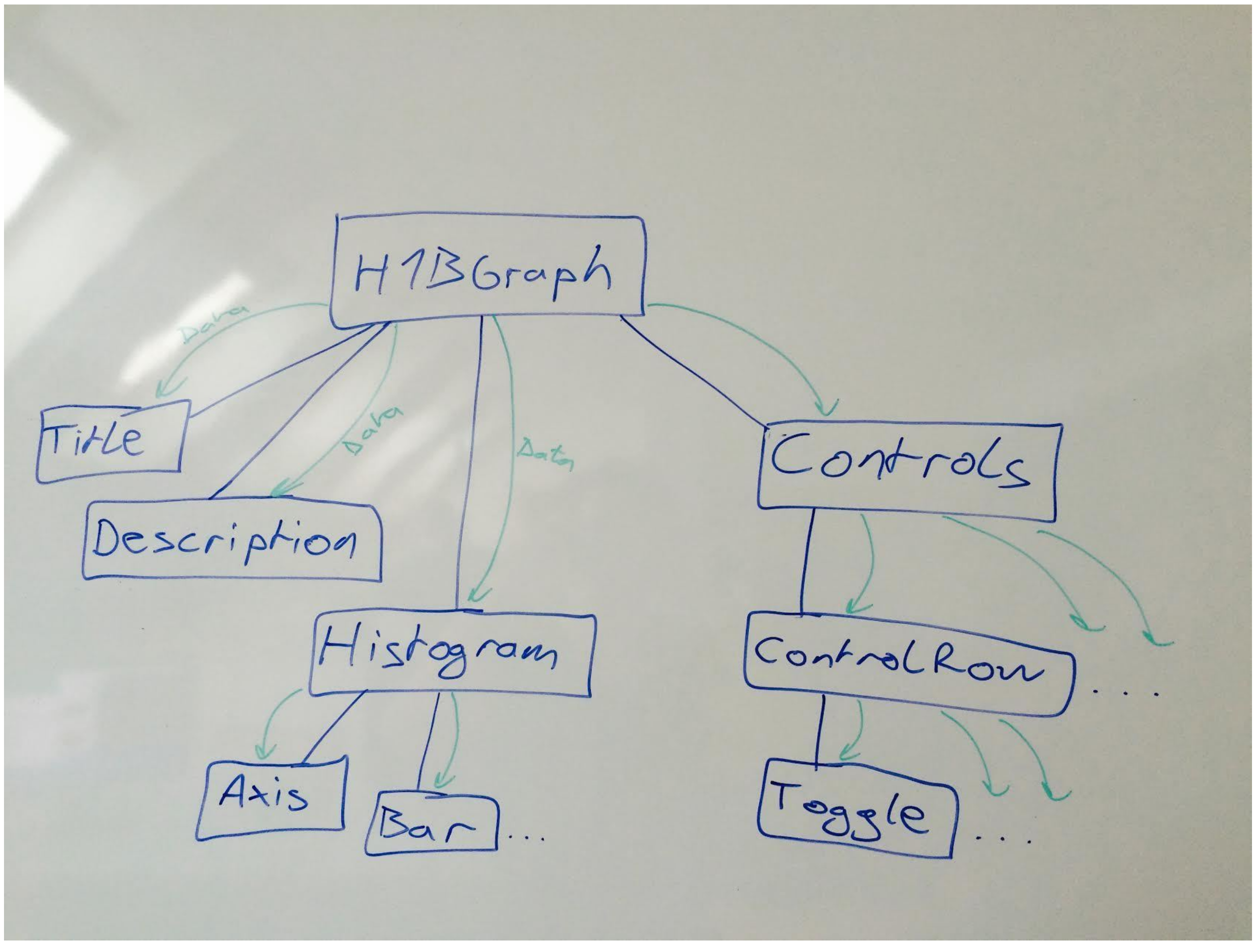
The basic architecture

The idea is this:

- The Main Component is the repository of truth
- Child components react to user events
- They announce changes up the chain of parents via callbacks
- The Main Component updates its truth

- The real changes flow back down the chain to update UI

This might look roundabout, but I promise, it's awesome. It's definitely better than worrying about parts of the UI going out of date with the rest of the app.



Data flows down

Having your components rely solely on their properties is like having functions that rely only on their arguments. This means that if given the same arguments, they *always* render the same output.

If you want to read about this in more detail, Google "isomorphic JavaScript". You could also search for "referential transparency" and "idempotent functions".

Either way, functional programming for HTML. Yay!

The HTML skeleton

We're building the whole interface with React, but we still need to begin with some HTML. It's going to take care of including files and giving our UI a container.

Make an index.html file that looks like this:

HTML skeleton

```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
        <title>How much does an H1B in the software industry pay?</title>

        <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3\
.3.5/css/bootstrap.min.css" integrity="sha512-dTfge/zgoMYpP7QbHy4gWMEGsbsdZeCXz7\
irItjcC3sPUFtf0kuFbDz/ixG7ArTxmDjLXDmezHubeNikyKGVyQ==" crossorigin="anonymous">
    </head>

    <body>
        <div class="container">
            <div class="h1bgraph"></div>
        </div>

        <script src="static/bundle.js"></script>
    </body>
</html>
```

These 20 lines do everything we need. The `<head>` sets some meta properties that Bootstrap recommends and includes Bootstrap's stylesheet. This is a good approach for when you only need Bootstrap's default styles and don't want to change anything. We'll use `require()` statements to load our own stylesheets with Webpack.

The `<body>` tag creates a container and includes the JavaScript code. We didn't really need a `<div>` inside a `<div>` like that, but I like to avoid taking over the whole `.container` with React. This gives you more flexibility for adding dumb static content.

At the bottom, we load our compiled JavaScript from `static/bundle.js`. This is a virtual path created by our dev server, so it doesn't point to any actual files.

The Main Component

As mentioned before we're going to build everything off a central repository of truth - The Main Component. We'll call this component `H1BGraph` because it draws a graph of H1B data.

Very imaginative, I know.

This code is going to live in `src/index.jsx` and start by requiring React, Lodash, and d3.js:

Require the libraries

```
var React = require('react'),
    _ = require('lodash'),
    d3 = require('d3');
```

We're going to add more imports later.

We also need a basic React component called `H1BGraph`. It should look like this:

Blank component

```
var H1BGraph = React.createClass({
    render: function () {
        return (
            <div className="row">
                <div className="col-md-12">
                    <svg width="700" height="500">

                    </svg>
                </div>
            </div>
        );
    }
});
```

This creates a Bootstrap row with an `<svg>` element.

We also have to render our component into the page like this:

Render our component

```
React.render(
    <H1BGraph url="data/h1bs.csv" />,
    document.querySelectorAll('.h1bgraph')[0]
);
```

If all went well, your browser's document inspector will show a blank SVG wrapped in some divs. Don't forget to keep `npm start` running in the background.