React & Data-Driven Documents

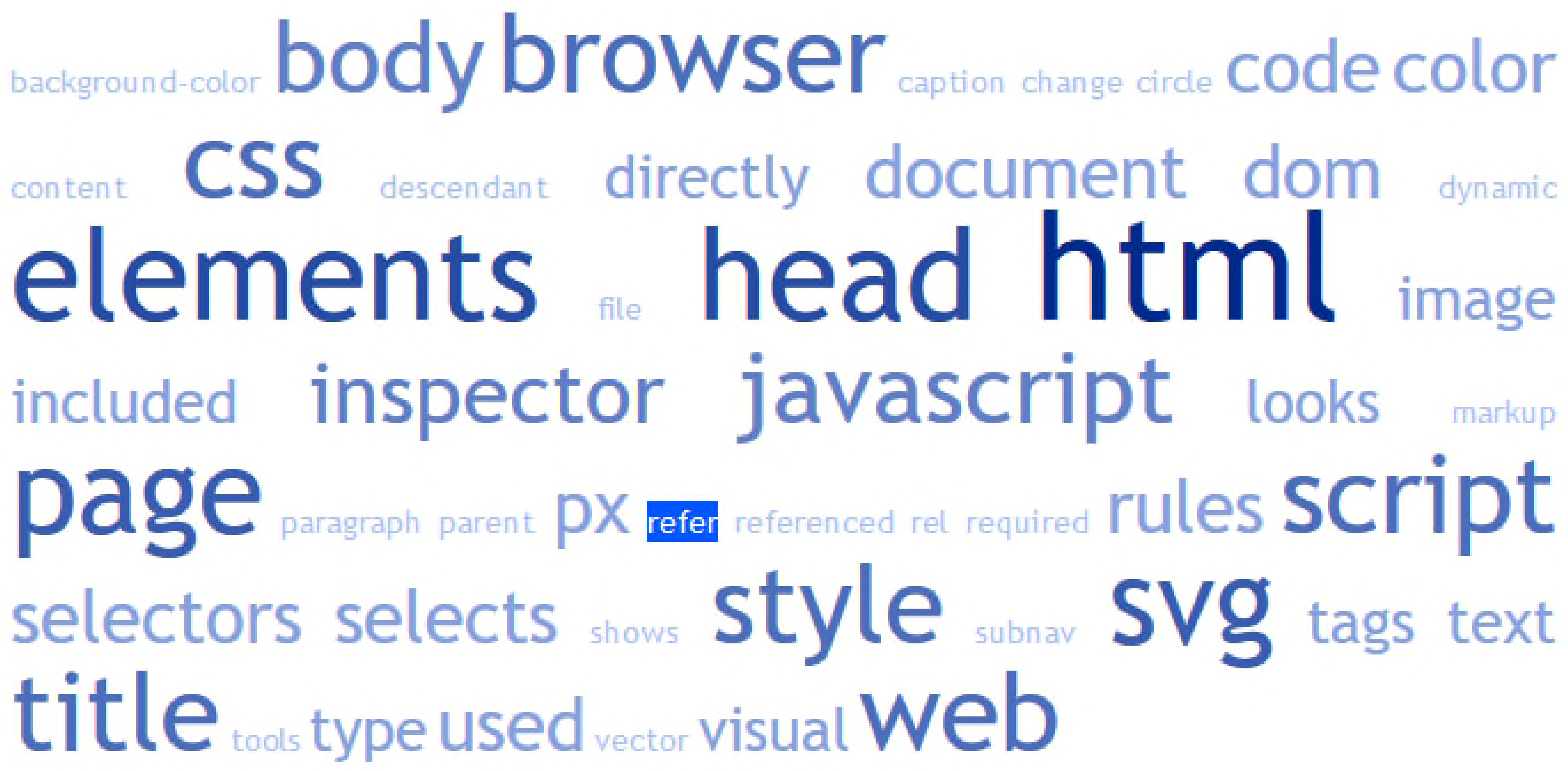
React Meetup Zürich, February 19th 2016

You Know React but what is D3.JS

- Data visualization tool written in Javascript
- D3 is Mike Bostock’s data visualization library. It’s being used by The New York Times as well as many other sites.
- D3 is built on top of common web standards like HTML, CSS, and SVG.
- D3 is the workhorse of data visualization on the web, and many charting libraries out there are based on it

D3 link

D3.JS and the web standards



http://alignedleft.com/tutorials/d3/fundamentals

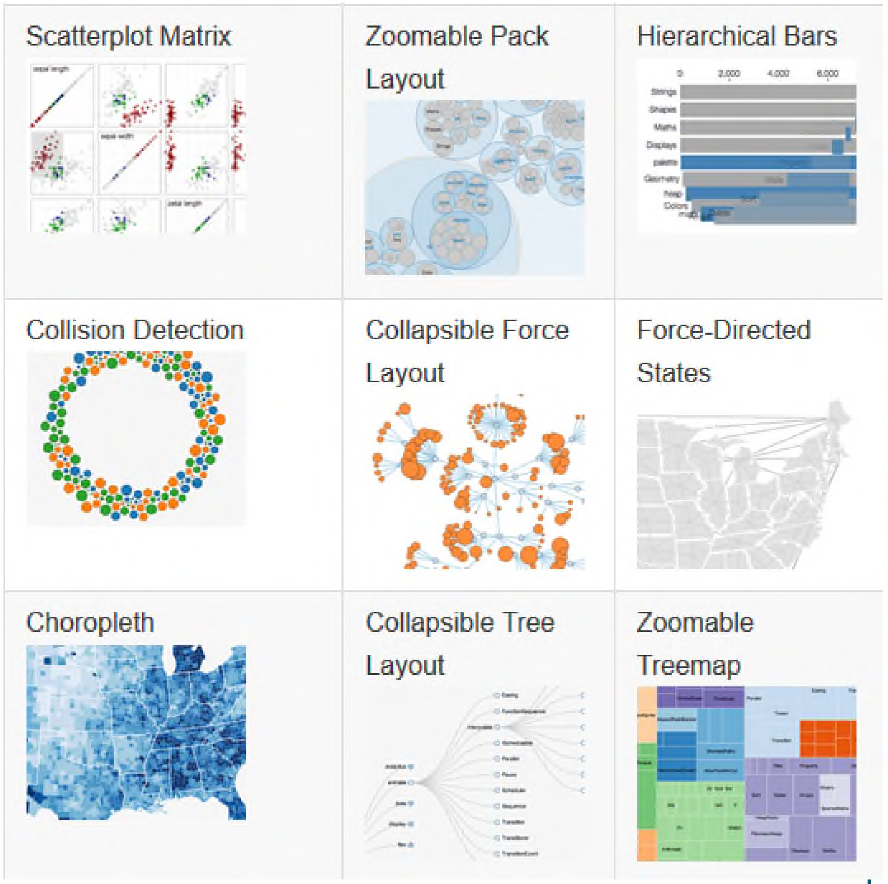
Why D3.js is great for Data Visualization

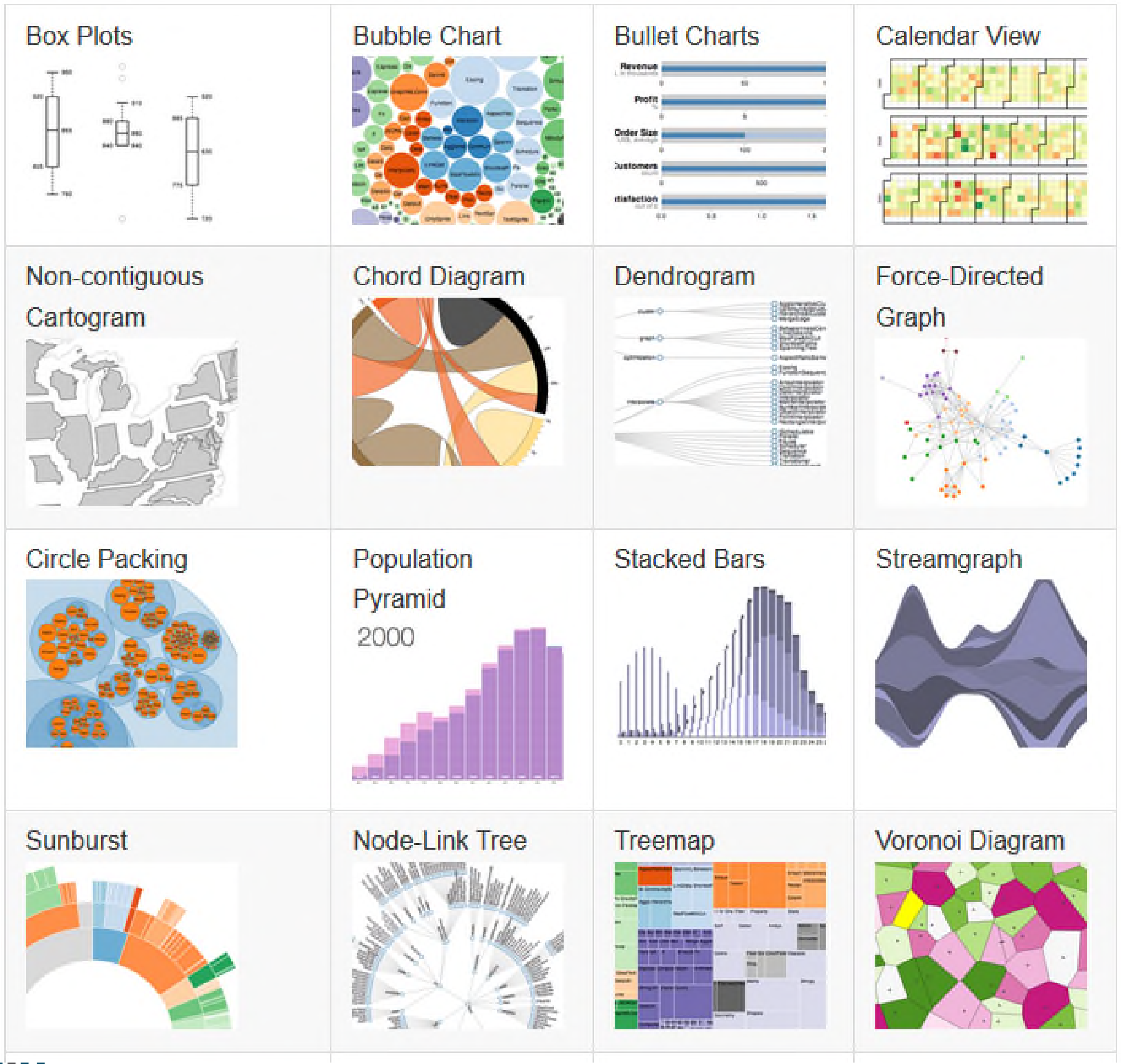
- It works on the web, in your browser !
- It is very flexible
- it works seamlessly with existing web technologies
- It can manipulate any part of the DOM

- Positioning :
  - Not a graphics library, nor a data processing library.
    It doesn't have pre-built charts that limit creativity.
  - It has tools that make the connection between data and graphics easy.
  - It sits right between the two, the perfect place for a library meant for data visualization.

D3.js Gallery

https://github.com/mbostock/d3/wiki/Gallery





Other charting library based on D3.js

- D3.js is a fairly low-level library.

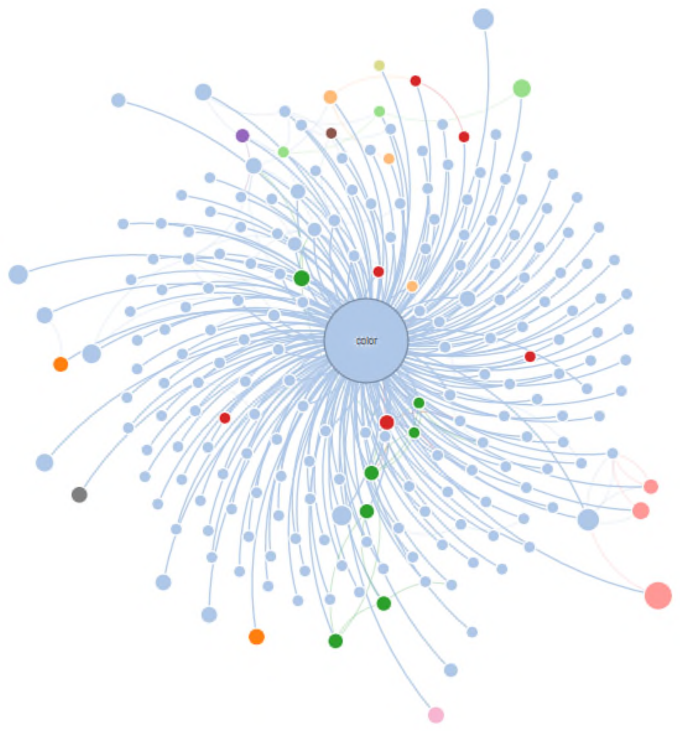
- You can’t just say “I have data; give me a barchart”.

  Well, you can, but it takes a few more lines of code than that.

  Once you get used to it though, d3.js is a joy to use

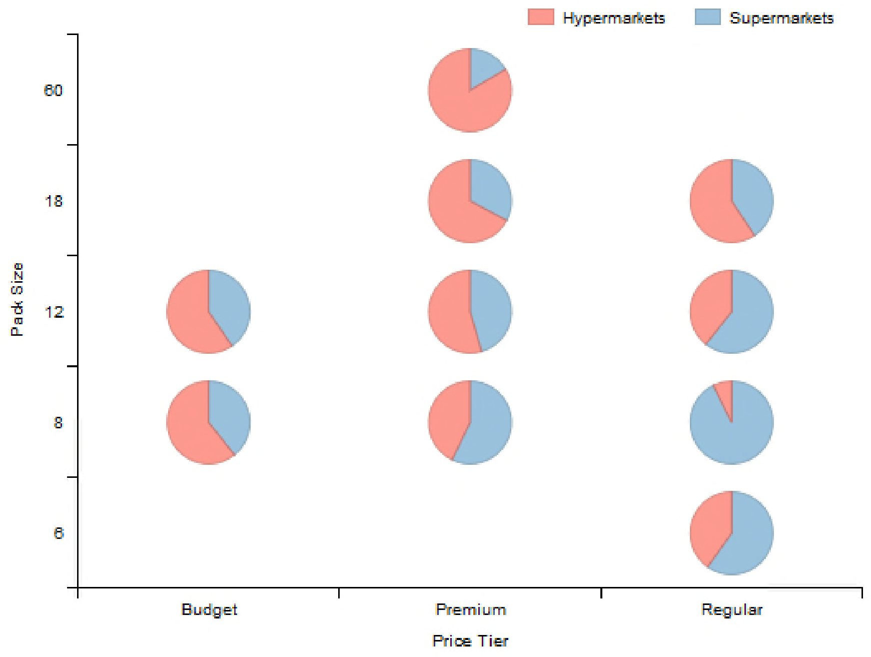
Examples of charting library based on D3.js

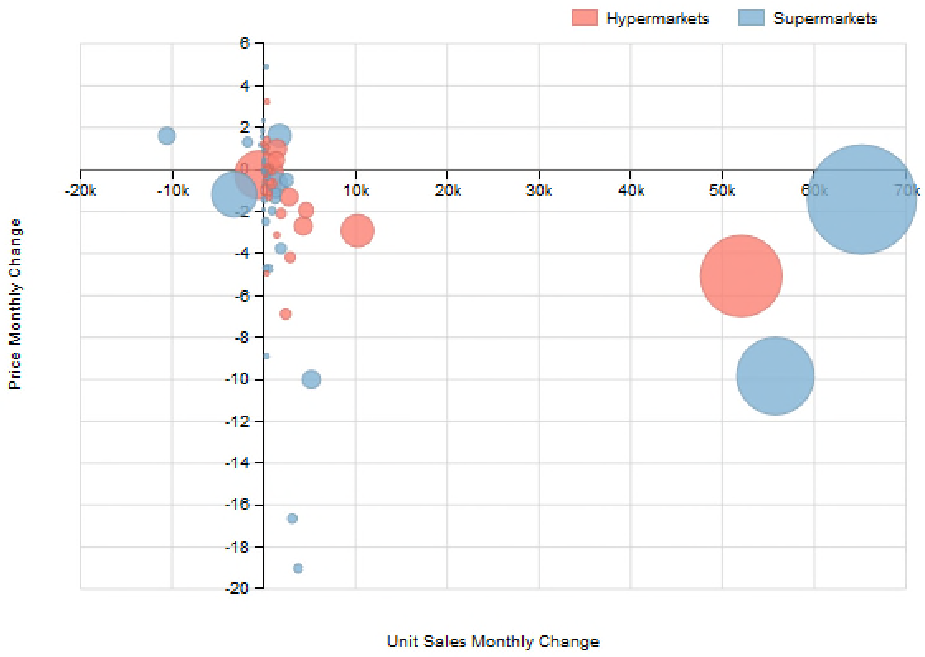
- Insigths.js http://ignacioola.github.io/insights/examples/basic.html



Examples of charting library based on D3.js

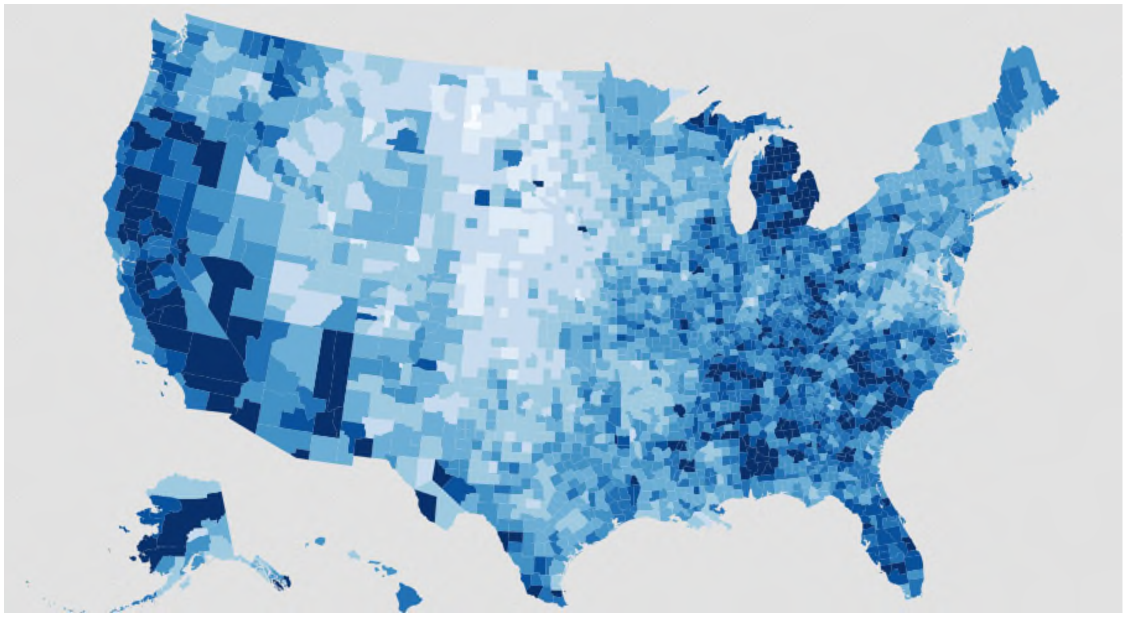
- Dimple http://dimplejs.org/index.html

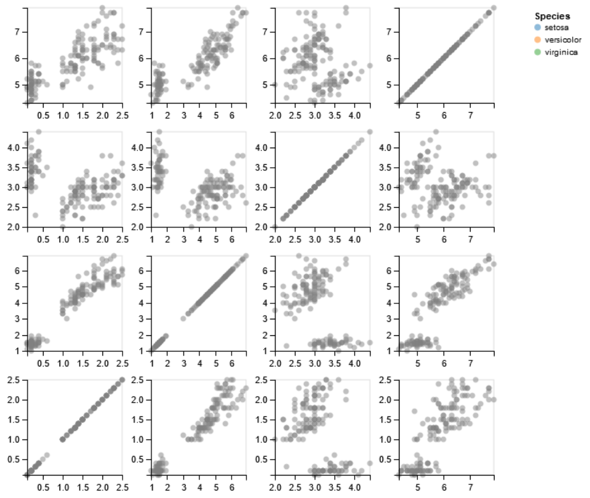




Examples of charting library based on D3.js

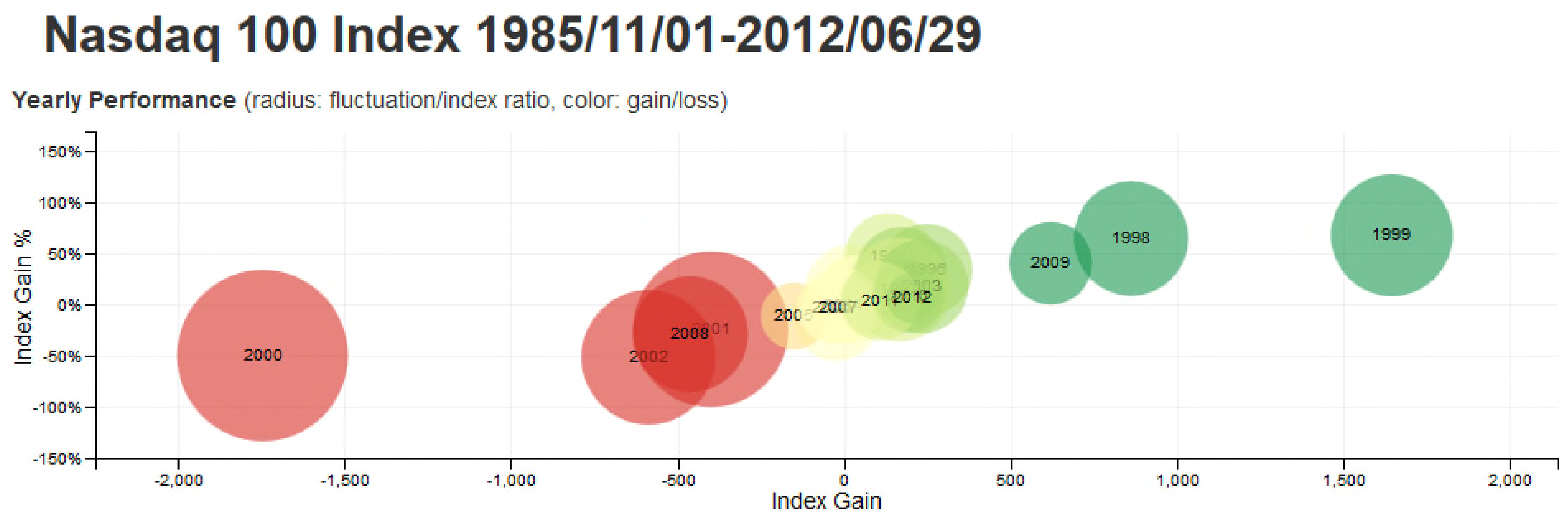
- Vega http://vega.github.io/vega-editor/index.html?spec=linking



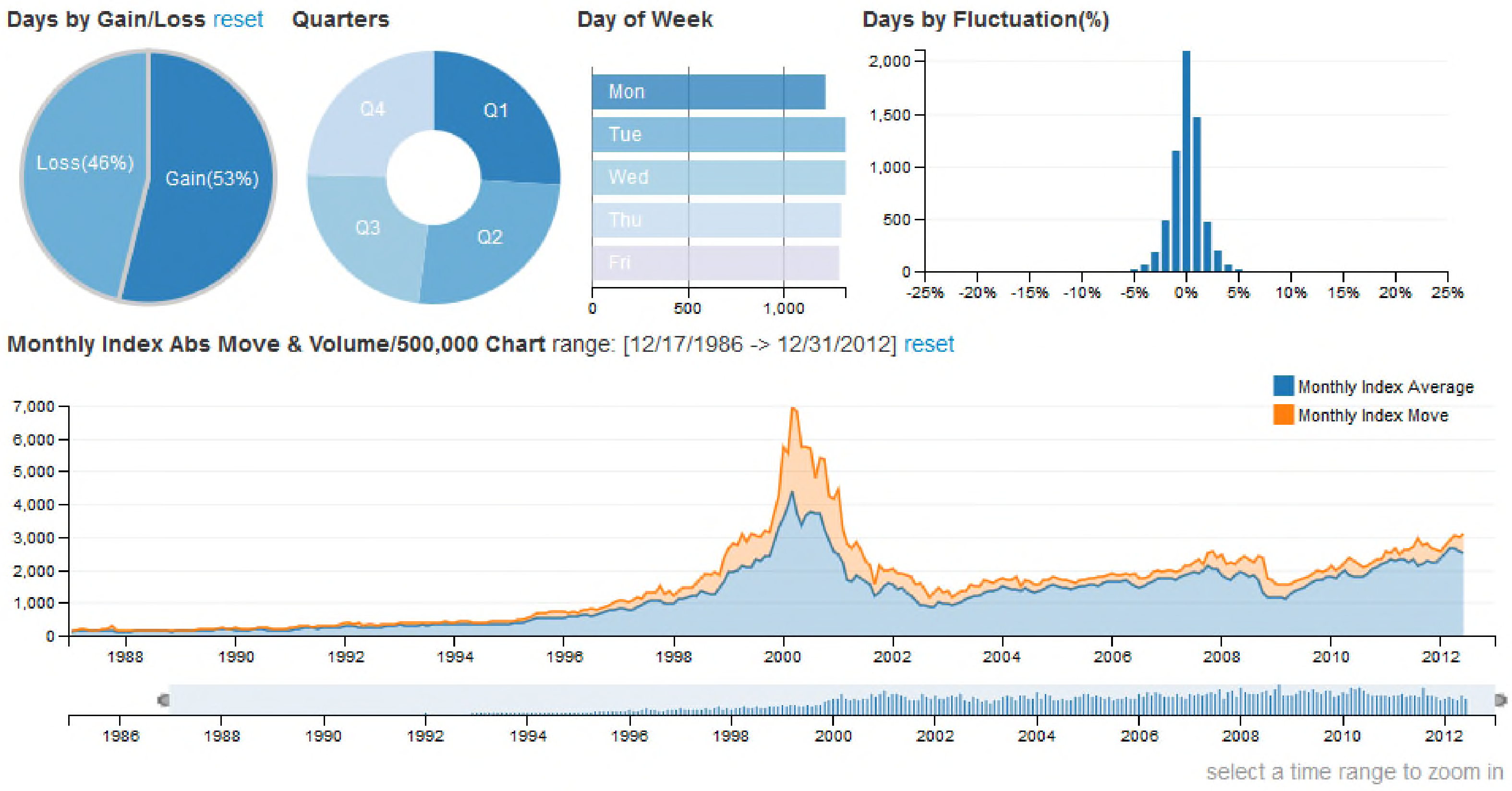


Examples of charting library based on D3.js

- dc.js http://dc-js.github.io/dc.js/

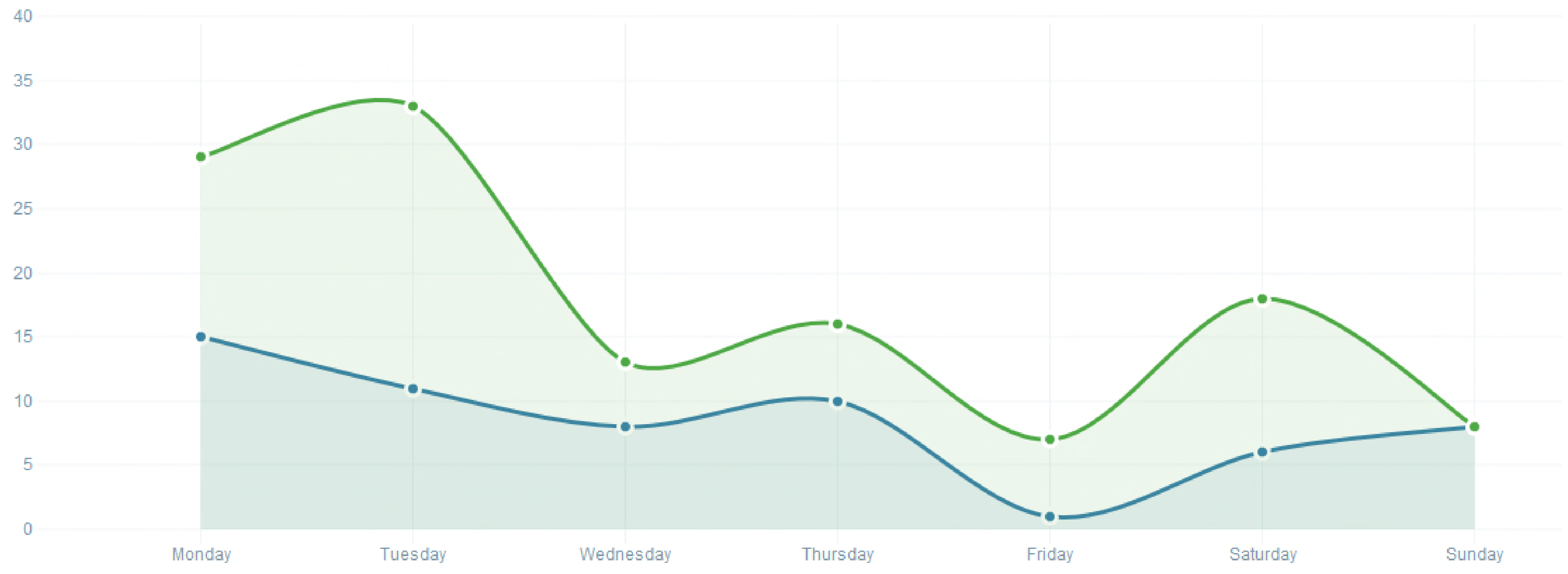


Examples of charting library based on D3.js



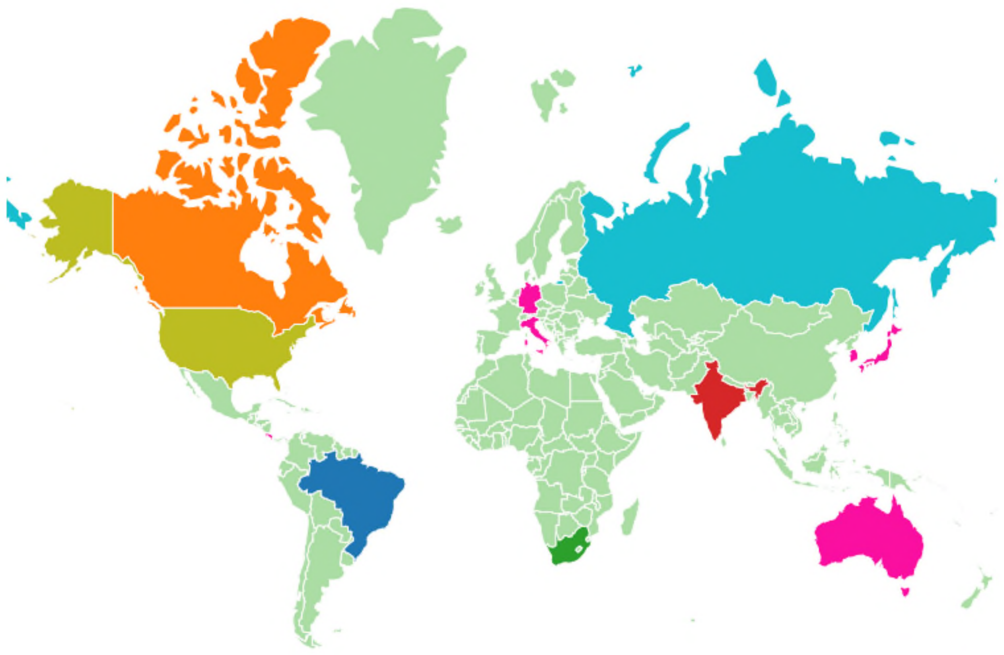
Examples of charting library based on D3.js

- tenXer http://tenxer.github.io/xcharts/ (example of animations)



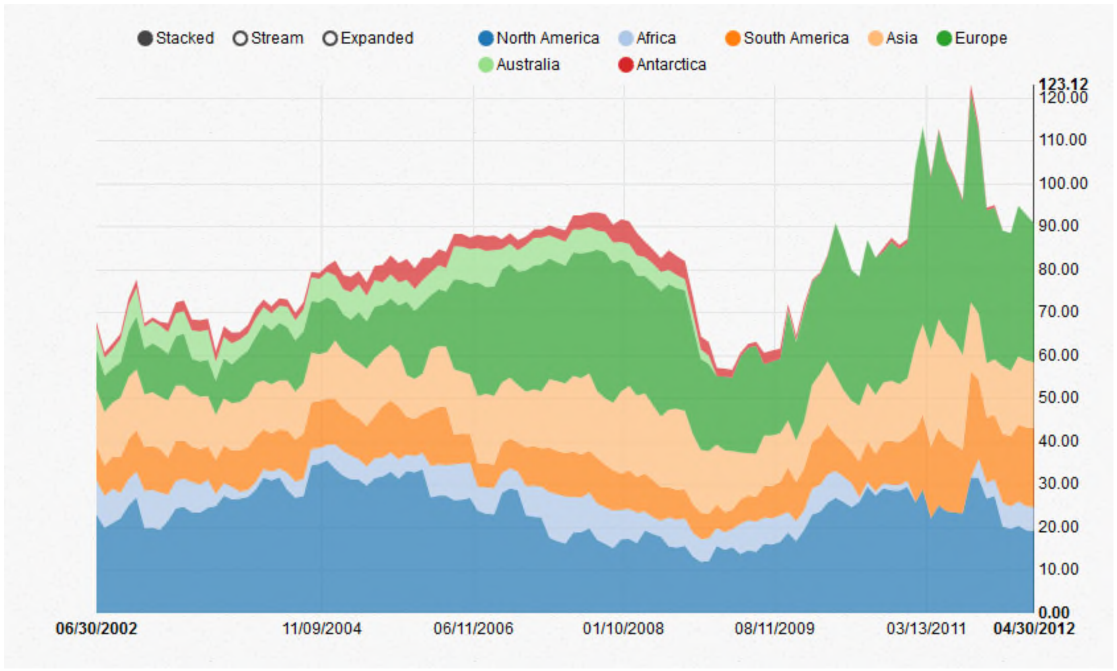
Examples of charting library based on D3.js

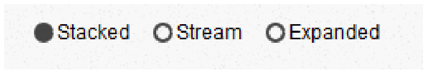
- DataMaps http://datamaps.github.io/



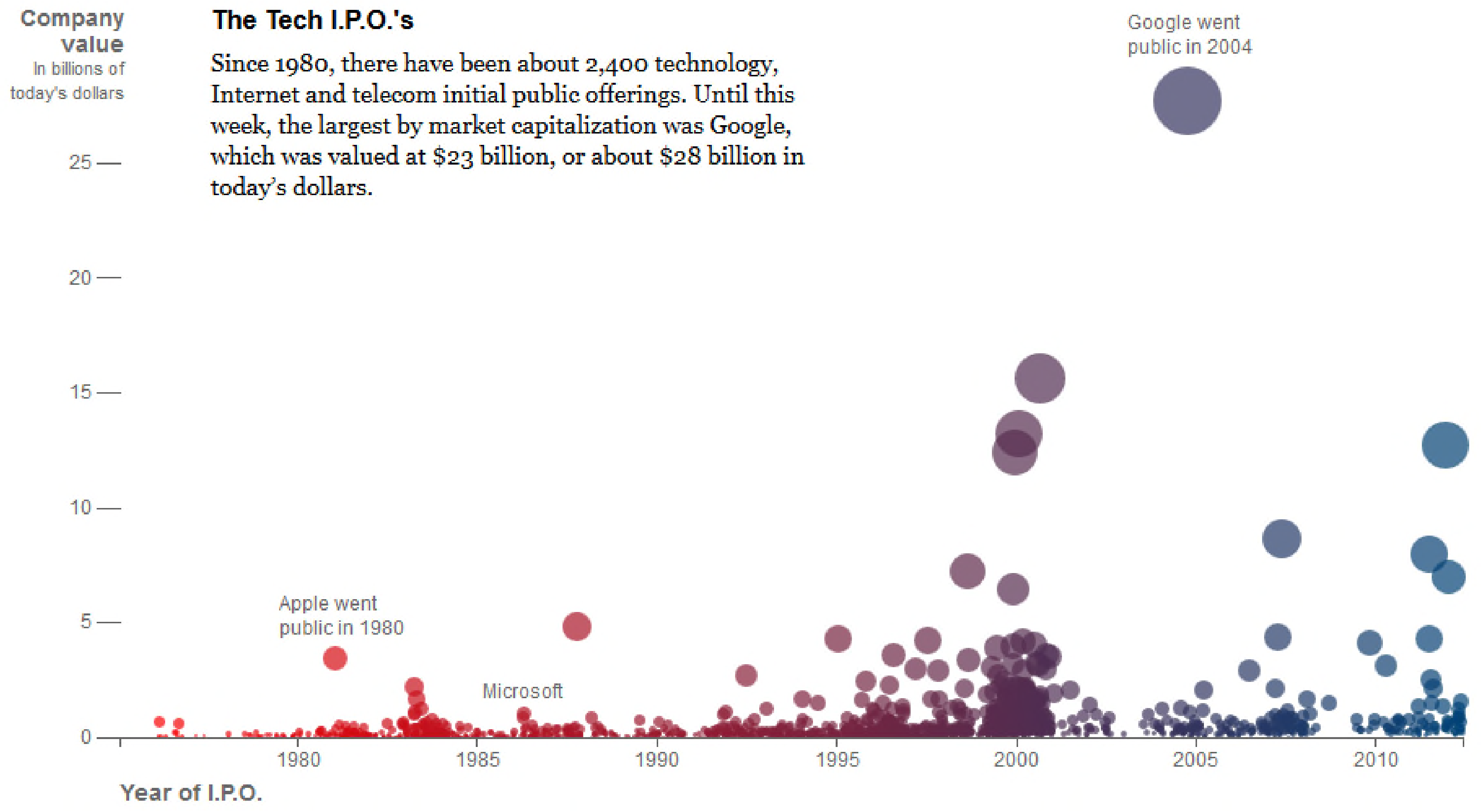
Examples of charting library based on D3.js

- nvd3 http://nvd3.org/examples/stackedArea.html (example of animations)





How the NYT is using D3.js

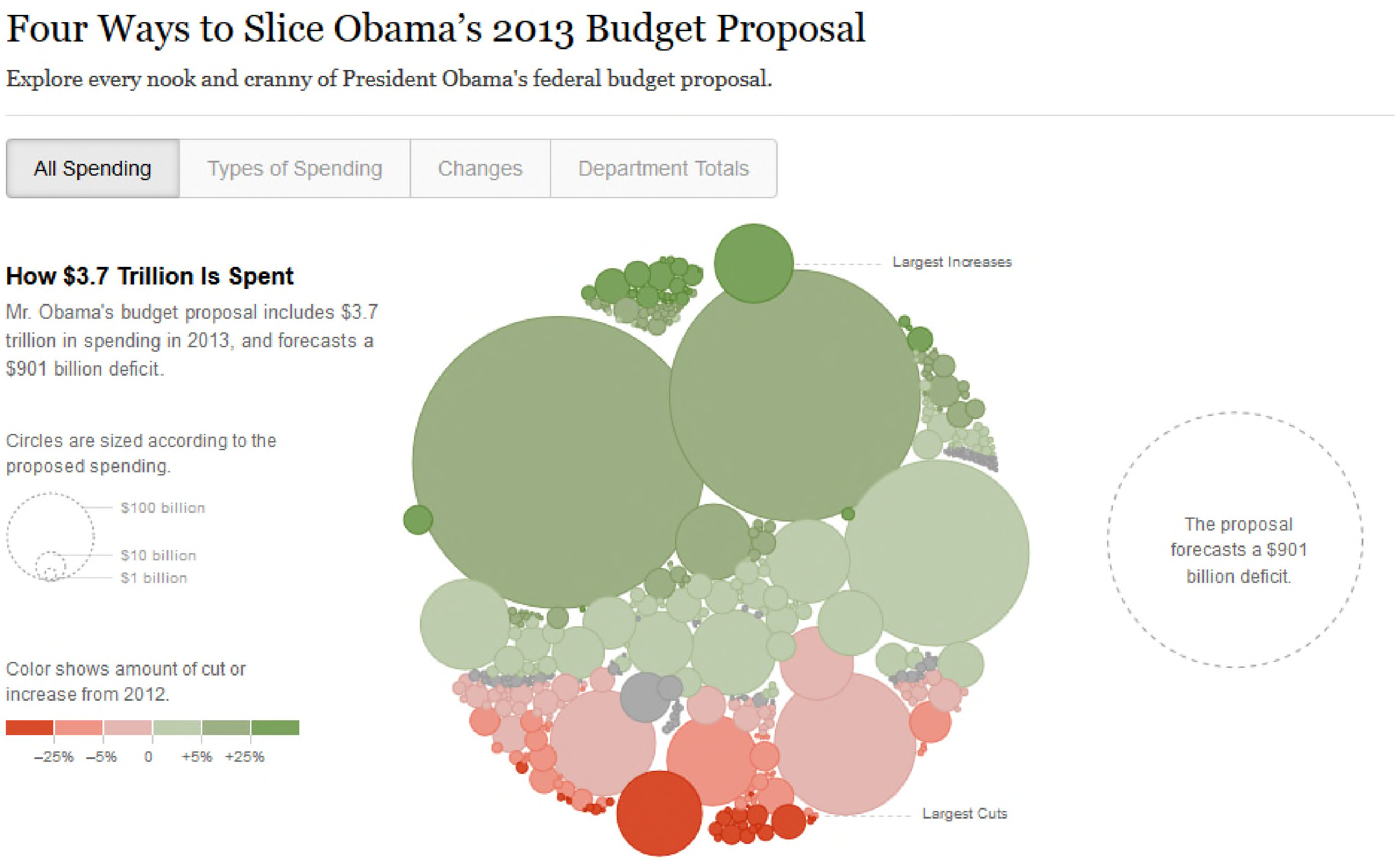


The Tech I.P.O.'s

Since 1980, there have been about 2,400 technology, Internet and telecom initial public offerings. Until this week, the largest by market capitalization was Google, which was valued at $23 billion, or about $28 billion in today's dollars.

I want to know more....

How the NYT is using D3.js



Four Ways to Slice Obama's 2013 Budget Proposal

Explore every nook and cranny of President Obama's federal budget proposal.

How $3.7 Trillion Is Spent

Mr. Obama's budget proposal includes $3.7 trillion in spending in 2013, and forecasts a $901 billion deficit.

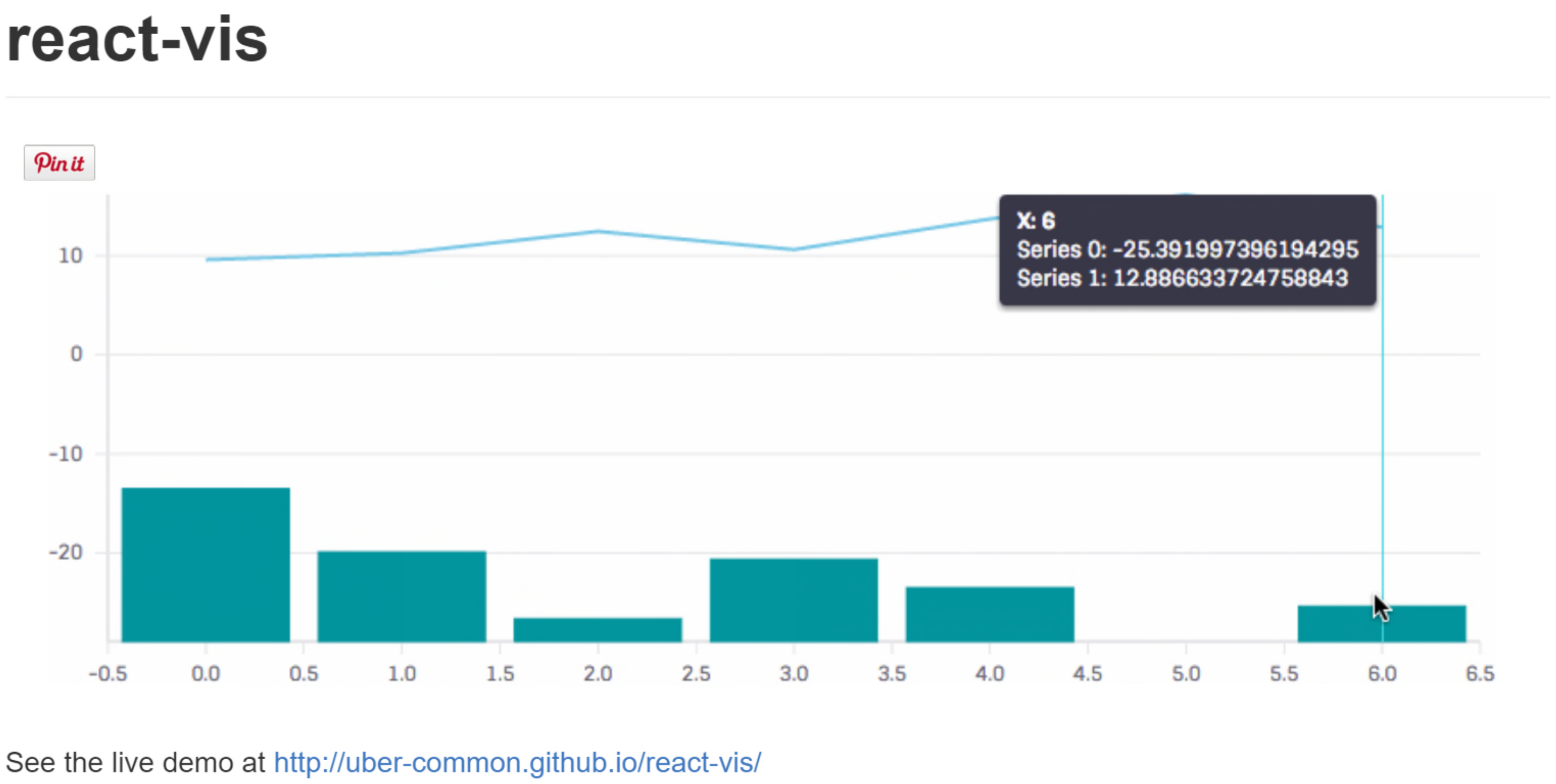
Circles are sized according to the proposed spending.

Color shows amount of cut or increase from 2012.

The proposal forecasts a $901 billion deficit.

[I want to know more....]

Latest D3 and React Implementation by Uber



react-vis

X: 6
Series 0: -25.391997396194295
Series 1: 12.886633724758843

See the live demo at http://uber-common.github.io/react-vis/

[I want to know more….]

More JavaScript charting libraries

- http://thenextweb.com/dd/2015/06/12/20-best-javascript-chart-libraries/

Reminder: Basic parts of a graph

- The scale

    The graph has to be "to scale". It has to have a coordinate system !

    YAxisScalingFactor = 1000/(double)largestNumber; (Bargraph)

    XAxisScalingFactor = 1000/(double)valuesToPlot.length; (Linegraph)

- The axes

    In order to be easy to read, axes need to be:

    - labelled
    - properly formated

- The data

    We have data coming in

    We transform it to something visual

https://www.google.com/finance

Reminder: What is the SVG format ?

- **Scalable Vector Graphics** (**SVG**) is an XML-based vector image format for two-dimensional graphics with support for interactivity and animation. The SVG specification is an open standard developed by the World Wide Web Consortium (W3C) since 1999.
- SVG images and their behaviors are defined in XML text files. This means that they can be searched, indexed, scripted, and compressed. As XML files, SVG images can be created and edited with any text editor, but are more often created with drawing software.
- All major modern web browsers—including Mozilla Firefox, Internet Explorer, Google Chrome, Opera, and Safari—have at least some degree of SVG rendering support.

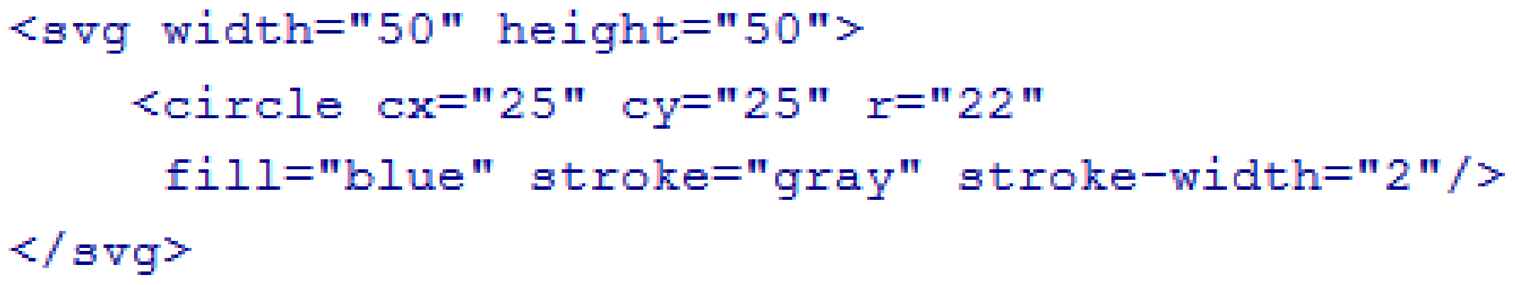
Source: Wikipedia (https://en.wikipedia.org/wiki/Scalable_Vector_Graphics)

Advantages of SVG

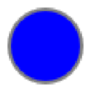
- Small file sizes that compress well
- Scales to any size without losing clarity (except very tiny sizes)
- Looks great on retina displays
- Design control like interactivity and filters
- SVGs are accessible to screen readers (with a little bit of work)
- There are plenty of SVG-based chart frameworks out there to help

D3.js can render Scalable Vector Graphics SVG

- D3 is at its best when rendering visuals as Scalable Vector Graphics.



```
<svg width="50" height="50">
    <circle cx="25" cy="25" r="22"
     fill="blue" stroke="gray" stroke-width="2"/>
</svg>
```



NOW IT'S TIME
FOR A DEMO

Look how the following map, airports, planes are represented using a very smooth animation. No-ready made charting library can give you this ;-)

[Link](#)   [Link2](#)

One more thing about SVG …

- Sparklines in SVG
- Demo + Code
- Make it dynamic with React.js

  Link

NOW REACT ENTERS THE SCENE

Why use D3 with React.JS

- Just like React, d3.js is **declarative**.

Tell what you want, not how you want it.

- This is where React comes in.

Using **React componentization**, once you've created a histogram component, you can always get a histogram with

<Histogram {...params} />.

Why use D3 with React.JS

- With React, you can make various graph and chart components build off the same data. It's ideal for Dashboards and Master/detail views.

  When your data changes, the whole visualization reacts.

  => Your graph changes.

  => The title changes.

  => The description changes.

  => Everything changes.

NOW IT'S TIME FOR A DEMO

Look how the following visualization changes as the user picks a subset of the data to look at.

What are we going at to look in details ?

- http://swizec.github.io/h1b-software-salaries/#2013-az-engineer
- https://github.com/Swizec

All the following examples are explained in more details in a book (also available as ebook) written by Swizec. The new version using ES6 is going to be published in the next 3-4 weeks (It should be available in beta at the time of publication of these slides). Price is fair (less than 30 USD so I encourage you to get the ebook if you are going to use more of D3 with React).

LET'S SEE HOW IT'S DONE

Under the hood explanations

Pre-requisites

- Code should re-compile when we change a file
- Page should update automatically when the code changes
- Dependencies and modules should be simple to manage
- Page shouldn’t lose state when loading new code
- Browser should report errors accurately in the right source files

=> Bundle with Webpack, compile with Babel

webpack
MODULE BUNDLER

- Webpack gives you the ability to organize code into modules and ‘require()’ what you need, much like Browserify. Unlike Browserify, Webpack comes with a sea of built-in features and a rich ecosystem of extensions called plugins.

- Webpack can solve two more annoyances
  - losing state when loading new code
  - accurately reporting errors

https://webpack.github.io/

BABEL

- Webpack isn’t the only tool mentioned earlier

- It needs a compiler

- Using Babel to compile the JSX and ES6 into ES5.

```
() => (<div>Hello there</div>)
```

```
(function () {
  return React.createElement(
    'div',
    null,
    'Hello there'
  );
});
```

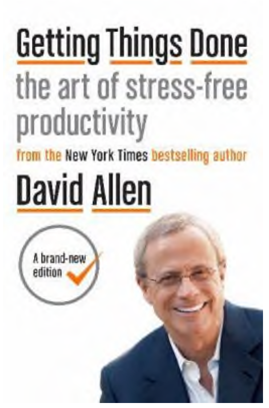
https://babeljs.io/repl/

Basic environment setup

Make sure you have node.js and npm installed then:

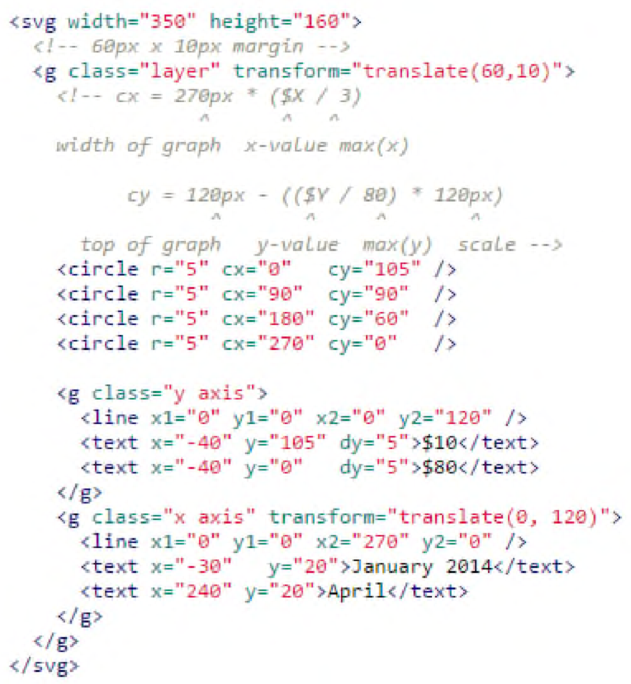
1. npm install webpack –g
2. git clone https://github.com/gaearon/react-transform-boilerplate.git
3. npm install --save d3 lodash autobind-decorator
4. npm install --save style-loader less less-loader css-loader

GETTING THINGS DONE


Getting Things Done
the art of stress-free productivity
from the New York Times bestselling author
David Allen
A brand-new edition

Thanks to David Allen for this title ;-)

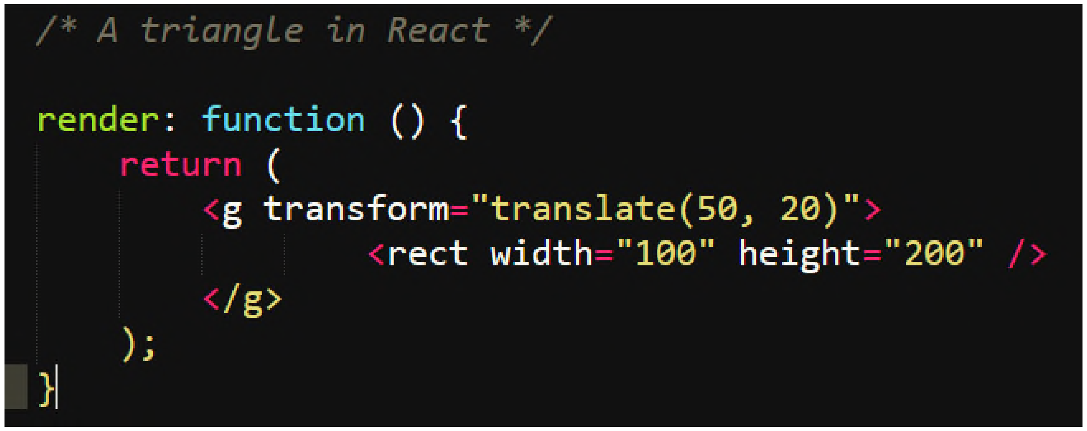
A pure SVG approach



```
<svg width="350" height="160">
  <!-- 60px x 10px margin -->
  <g class="layer" transform="translate(60,10)">
    <!-- cx = 270px * ($X / 3)
                ^      ^   ^
    width of graph  x-value max(x)

          cy = 120px - (($Y / 80) * 120px)
                 ^       ^     ^       ^
      top of graph   y-value  max(y)  scale -->
    <circle r="5" cx="0"   cy="105" />
    <circle r="5" cx="90"  cy="90"  />
    <circle r="5" cx="180" cy="60"  />
    <circle r="5" cx="270" cy="0"   />

    <g class="y axis">
      <line x1="0" y1="0" x2="0" y2="120" />
      <text x="-40" y="105" dy="5">$10</text>
      <text x="-40" y="0"   dy="5">$80</text>
    </g>
    <g class="x axis" transform="translate(0, 120)">
      <line x1="0" y1="0" x2="270" y2="0" />
      <text x="-30"   y="20">January 2014</text>
      <text x="240" y="20">April</text>
    </g>
  </g>
</svg>
```

The approach

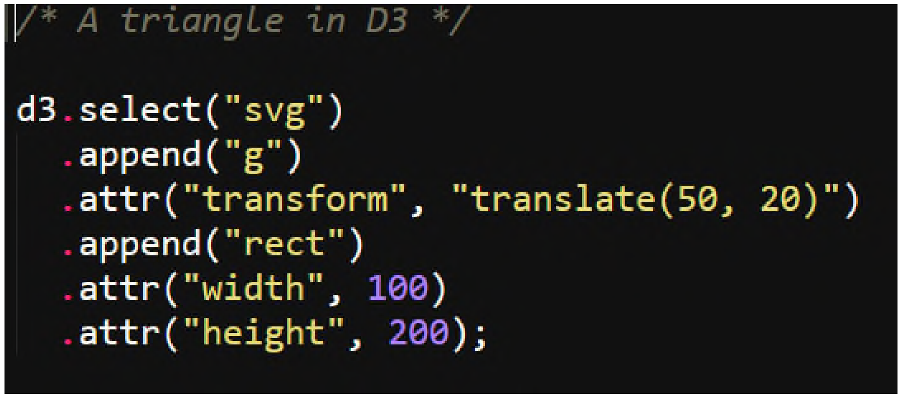
- Because SVG is an XML format that fits into the DOM, we can assemble it with React



```
/* A triangle in React */

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

The approach

- At first glance this looks cumbersome compared to traditional d3.js



```
/* A triangle in D3 */

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

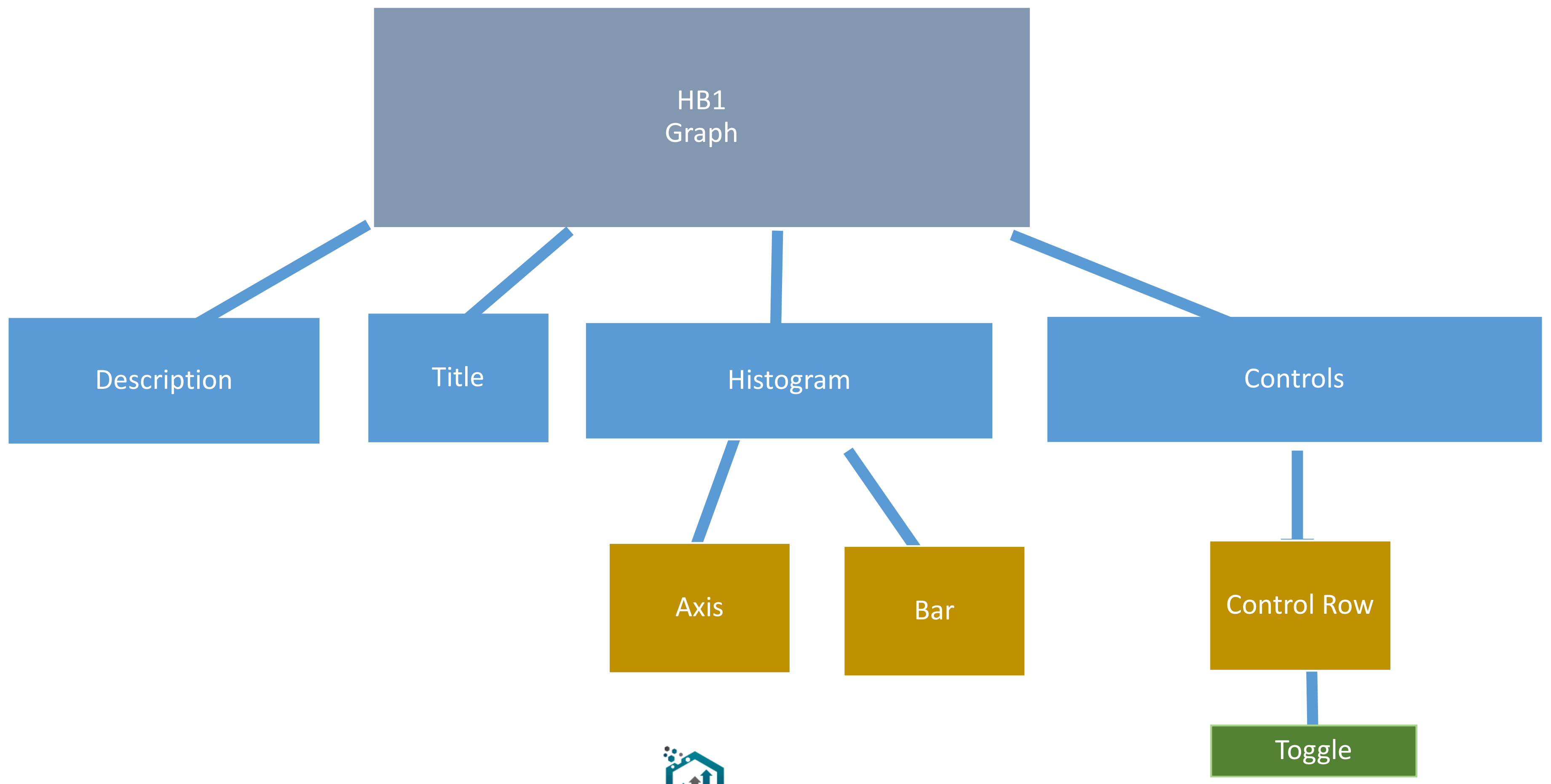
What’s the golden rule ?

Each library does what it is best at !

React owns the DOM

calculates properties

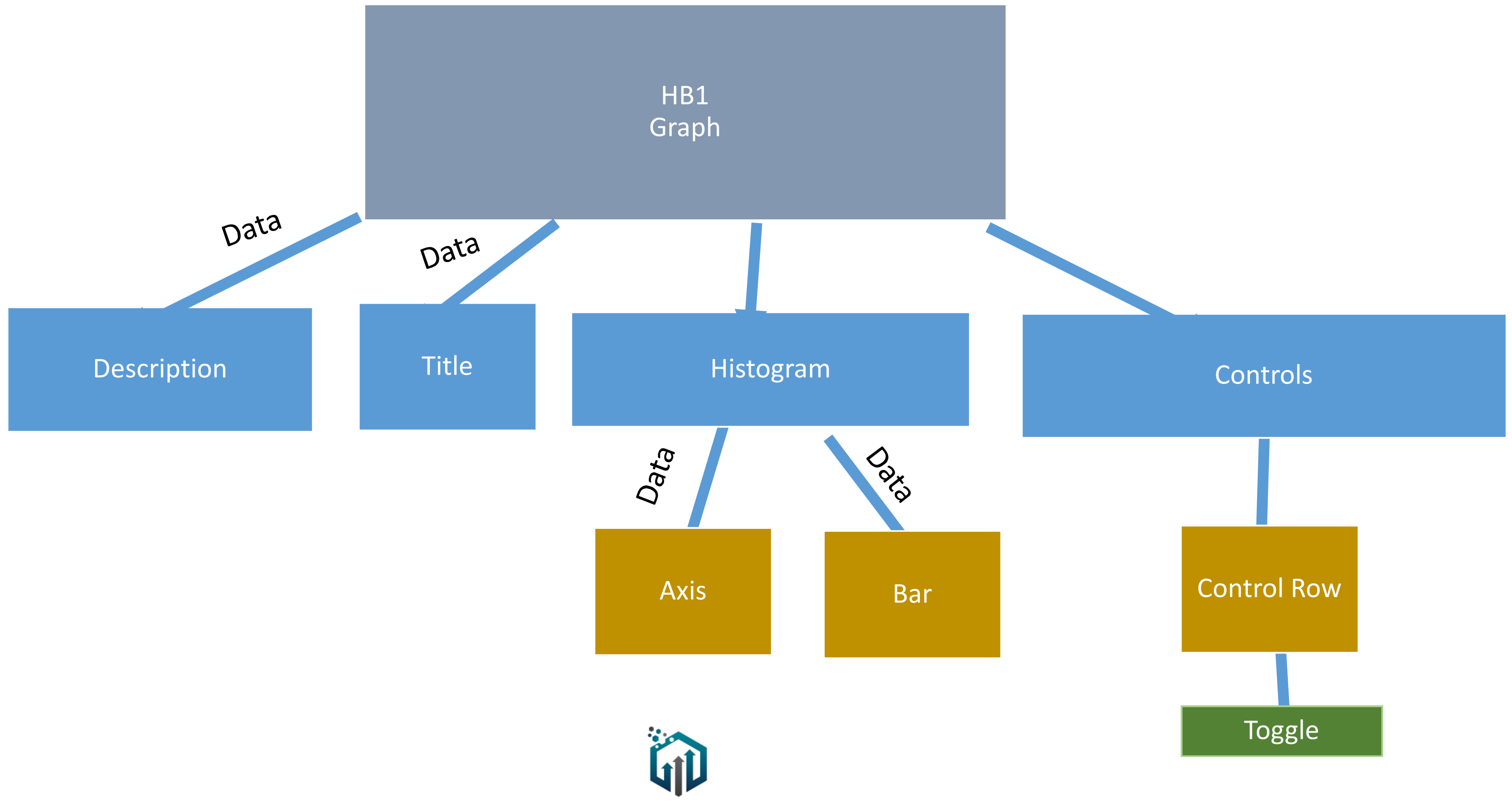
The basic architecture



The basic architecture

- 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

The basic architecture

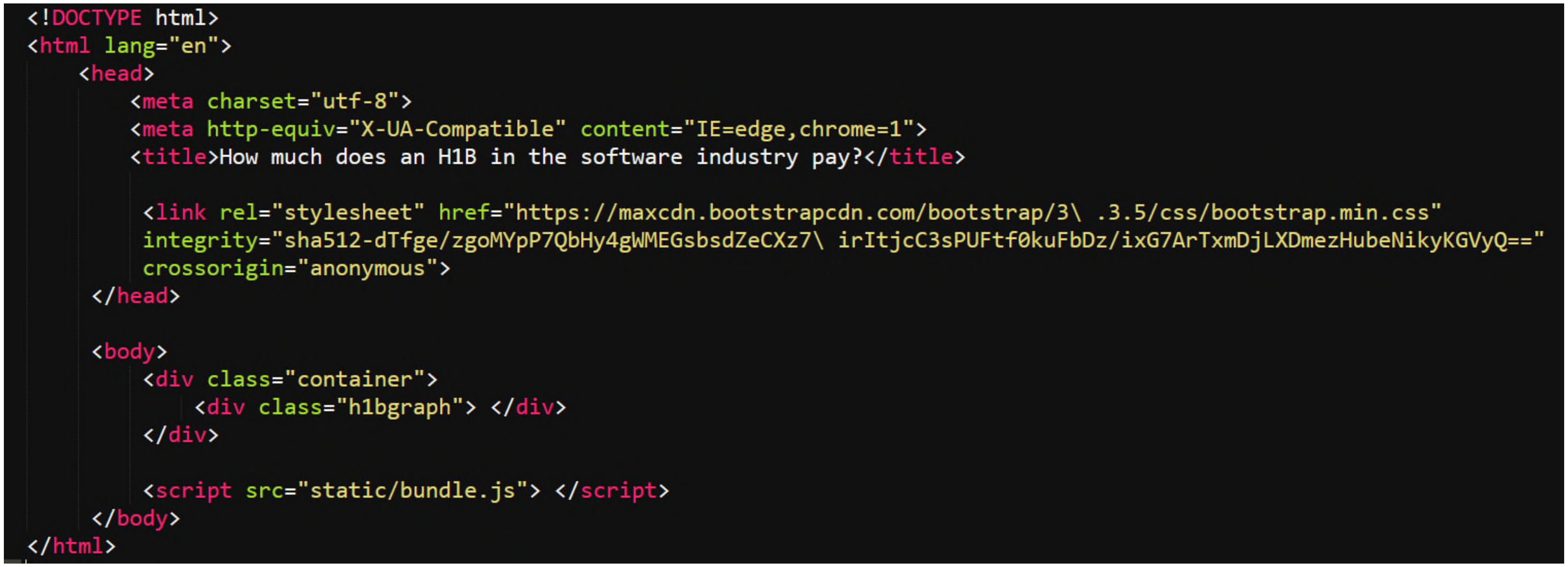


The architecture

- Having your components rely solely on their properties, is like having functions that rely just on their arguments.

- Given the same arguments, they always render the same output.

The HTML



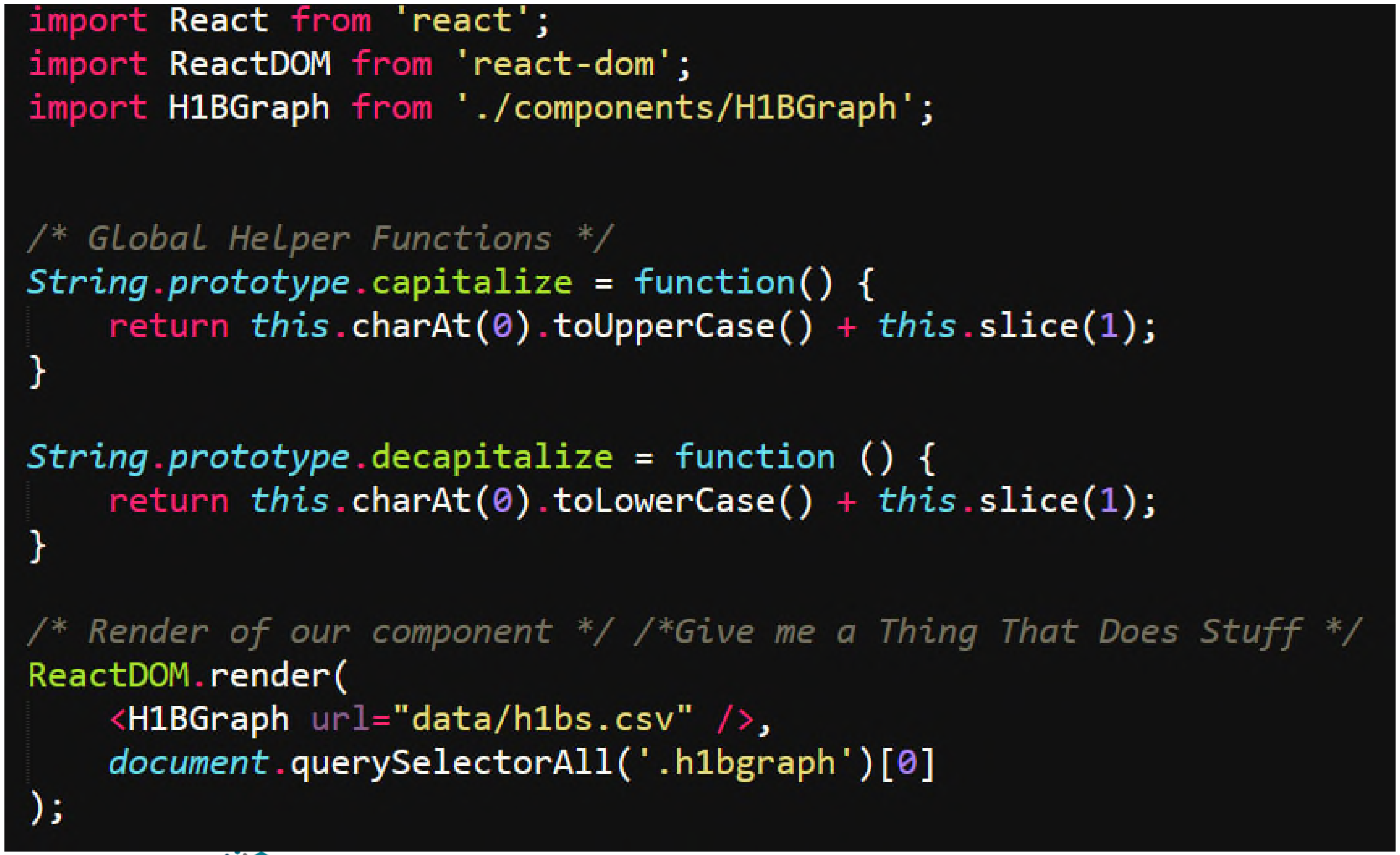
```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
        <title>How much does an H1B in the software industry pay?</title>

         <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>
```

The app

- As seen in the architecture part, we structure the app using components.
- Given the same arguments, they always render the same output.
- We have 2 main top-level components H1BGraph (UI) and Histogram (Charts)
- Single responsibility principle and separation of concerns

The React Main Component

- Located in src/index.jsx
- Creates a Bootstrap row with an <svg> element
- Renders the component itself



```
import React from 'react';
import ReactDOM from 'react-dom';
import H1BGraph from './components/H1BGraph';


/* Global Helper Functions */
String.prototype.capitalize = function() {
    return this.charAt(0).toUpperCase() + this.slice(1);
}

String.prototype.decapitalize = function () {
    return this.charAt(0).toLowerCase() + this.slice(1);
}

/* Render of our component */ /*Give me a Thing That Does Stuff */
ReactDOM.render(
    <H1BGraph url="data/h1bs.csv" />,
    document.querySelectorAll('.h1bgraph')[0]
);
```

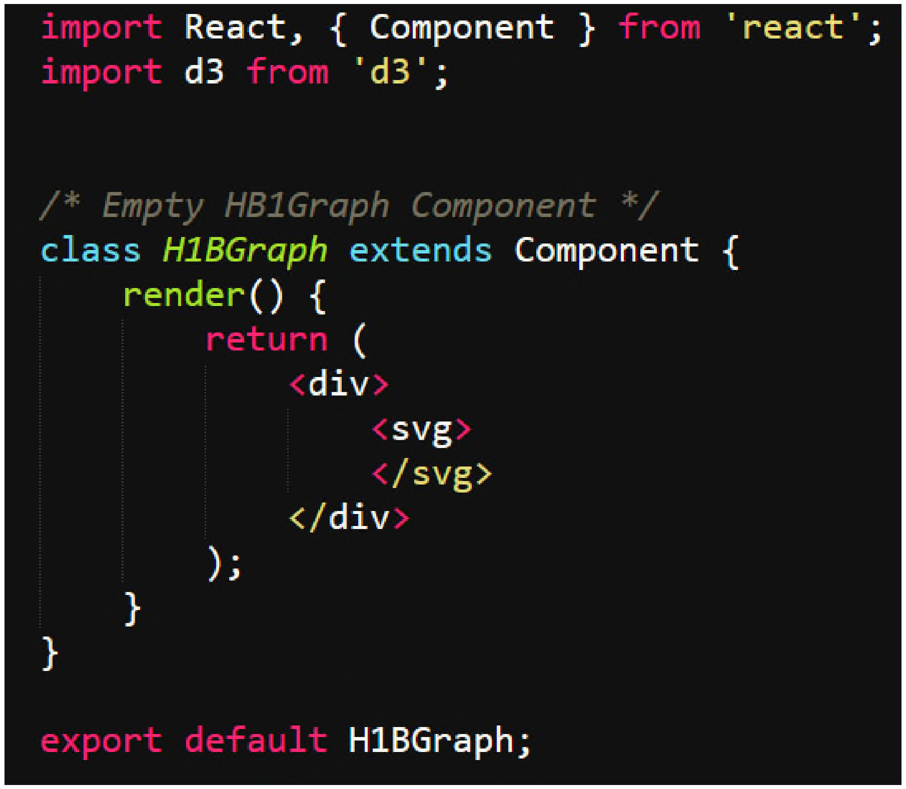
What do we get ?

- Your browser’s document inspector will show a blank SVG wrapped in some divs.
- Nothing fancy will be displayed at this stage ☹

Adding components

- All of our components are going to start the same way - some imports, a class with a render()
- method, and a default module export. That’s the minimum we need in <component>/index.jsx to
- define a component that doesn’t break our build.

Src/components/HB1Graph/index.jsx



```
import React, { Component } from 'react';
import d3 from 'd3';


/* Empty HB1Graph Component */
class H1BGraph extends Component {
    render() {
        return (
            <div>
                <svg>
                </svg>
            </div>
        );
    }
}

export default H1BGraph;
```

What’s next

1. We’ll make H1BGraph load our dataset

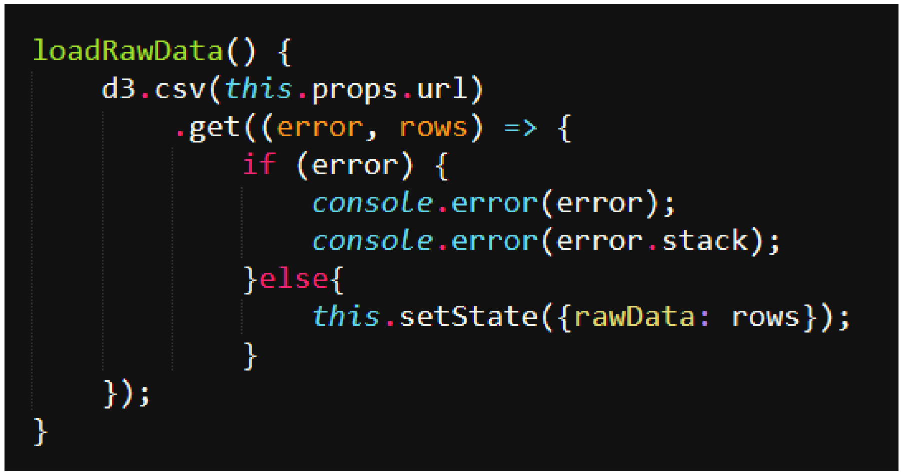
2. Build a Histogram component

3. Add dynamic meta text descriptions

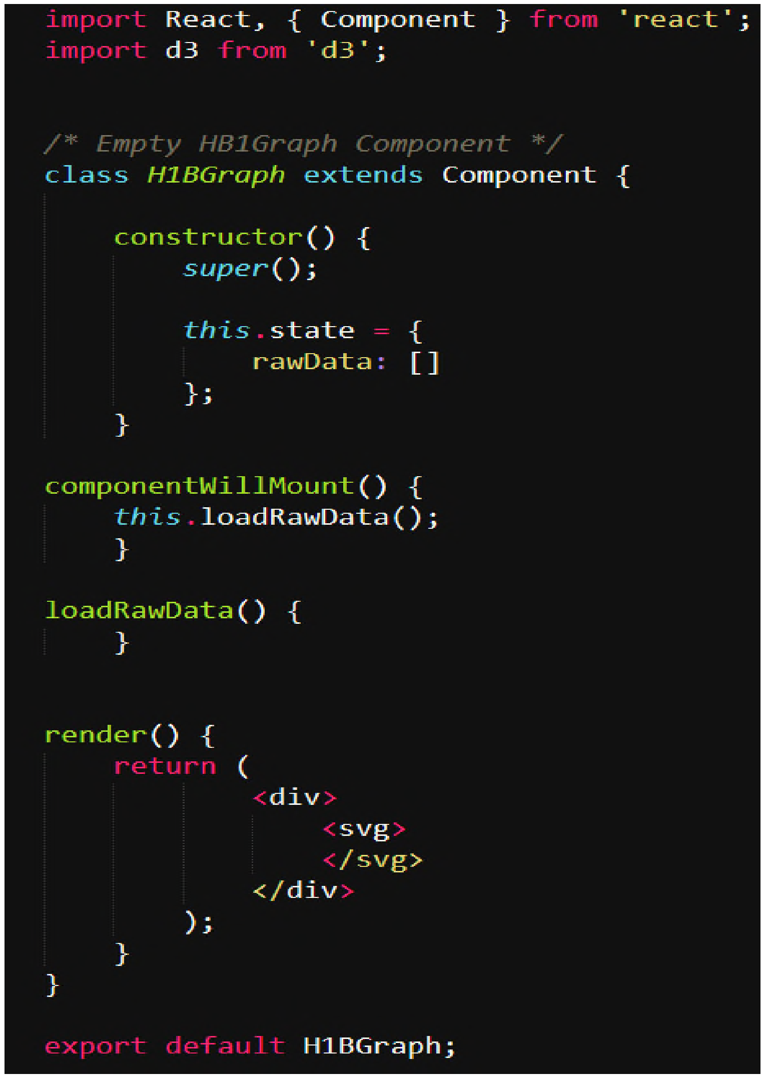
4. Add some data filtering

Loading Data

loadRowData function



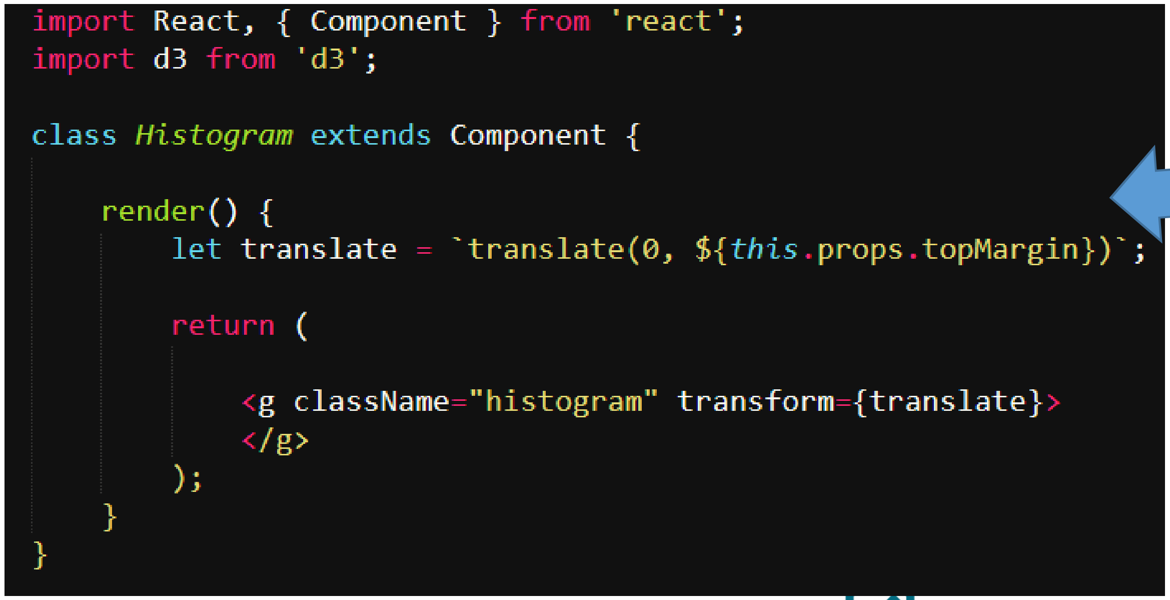
```
loadRawData() {
    d3.csv(this.props.url)
        .get((error, rows) => {
            if (error) {
                console.error(error);
                console.error(error.stack);
            }else{
                this.setState({rawData: rows});
            }
    });
}
```



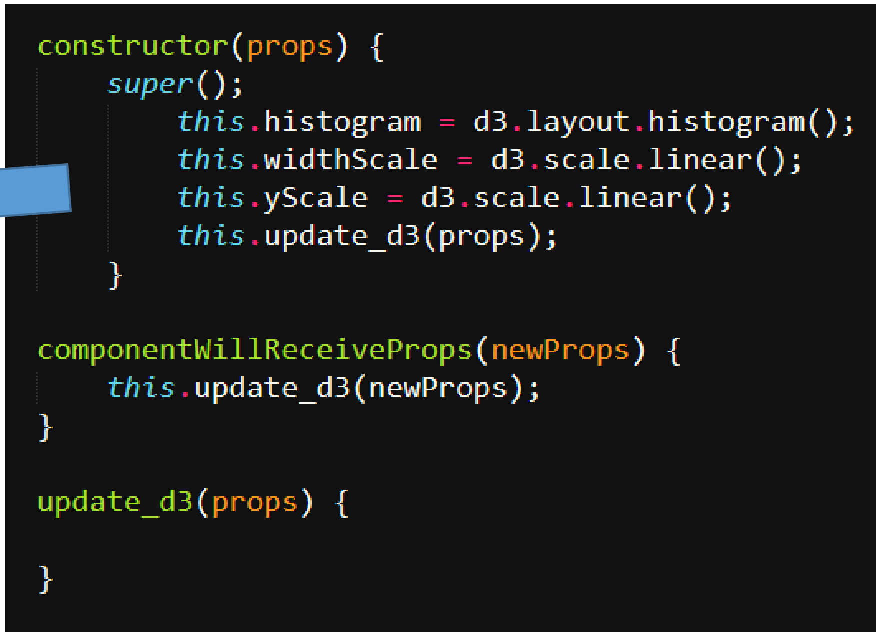
```
import React, { Component } from 'react';
import d3 from 'd3';


/* Empty HB1Graph Component */
class H1BGraph extends Component {

    constructor() {
        super();

        this.state = {
            rawData: []
        };
    }

componentWillMount() {
    this.loadRawData();
    }

loadRawData() {
    }


render() {
    return (
            <div>
                <svg>
                </svg>
            </div>
        );
    }
}

export default H1BGraph;
```

Drawing

- We add a blank component called Histogram.
- We add three methods to our Histogram component to manage d3.js:

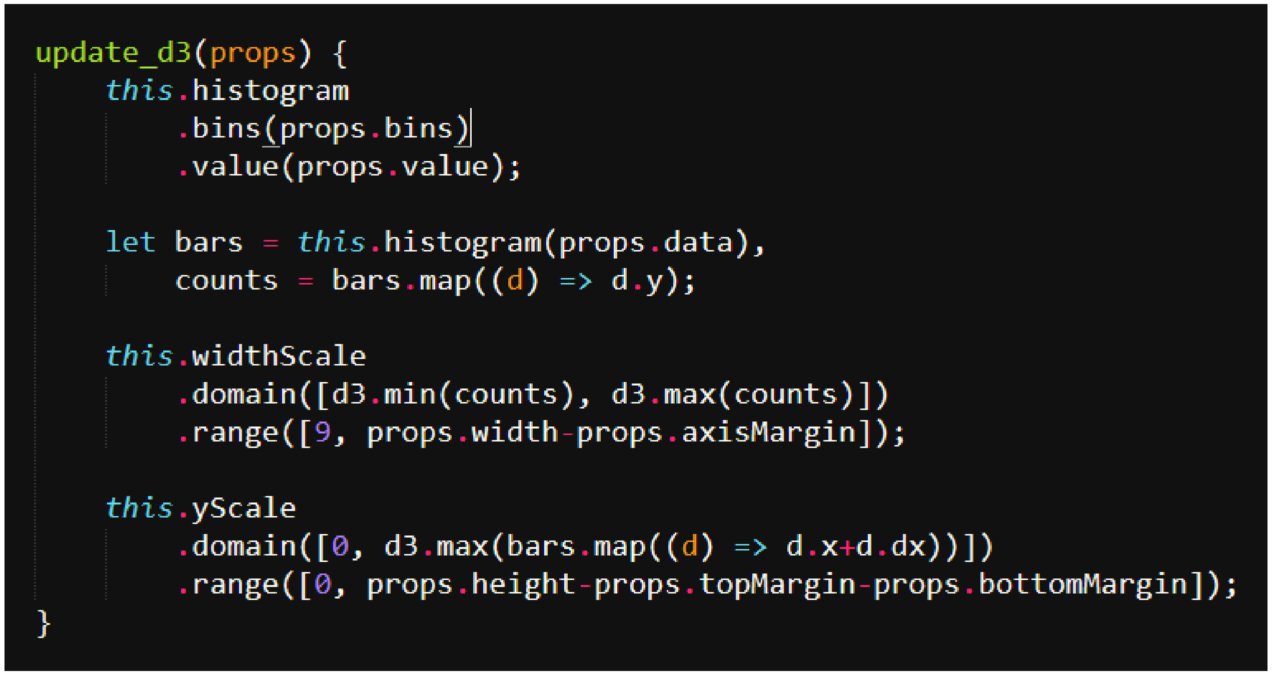


```
import React, { Component } from 'react';
import d3 from 'd3';

class Histogram extends Component {

    render() {
        let translate = `translate(0, ${this.props.topMargin})`;

        return (

            <g className="histogram" transform={translate}>
            </g>
        );
    }
}
```



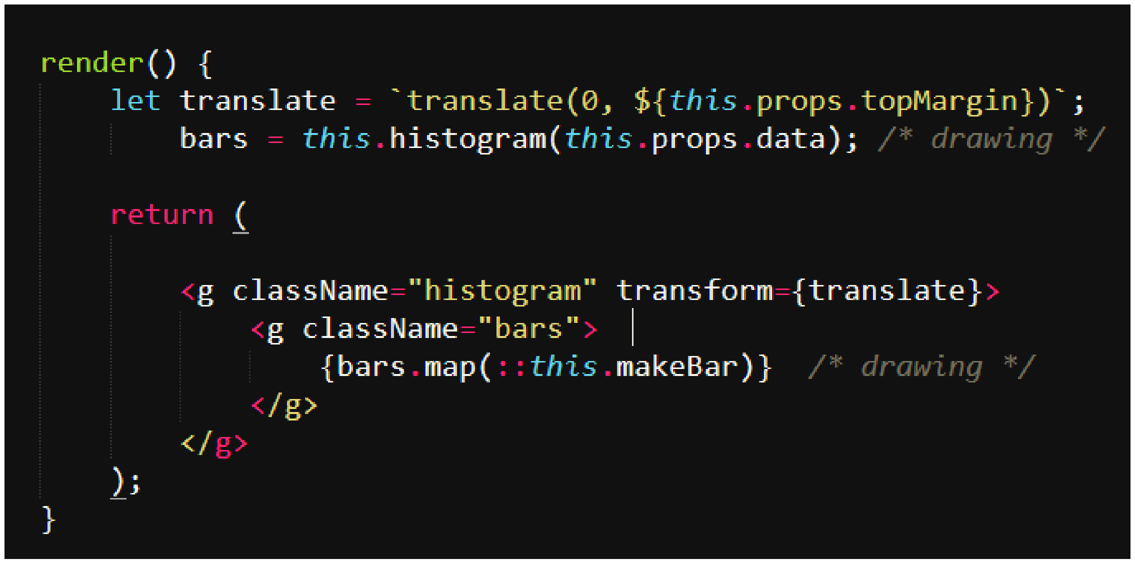
```
constructor(props) {
    super();
        this.histogram = d3.layout.histogram();
        this.widthScale = d3.scale.linear();
        this.yScale = d3.scale.linear();
        this.update_d3(props);
    }

componentWillReceiveProps(newProps) {
    this.update_d3(newProps);
}

update_d3(props) {

}
```

Drawing (count of bars and scales in D3)



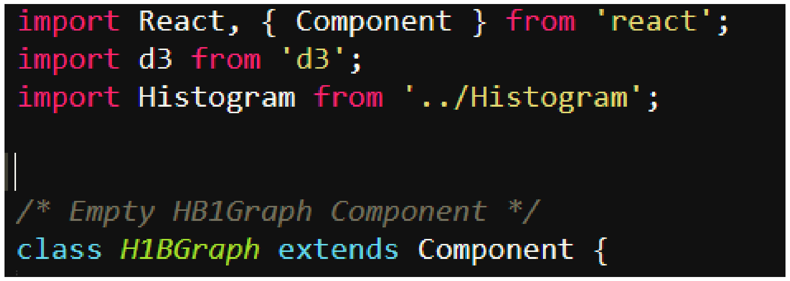
```
update_d3(props) {
    this.histogram
        .bins(props.bins)
        .value(props.value);

    let bars = this.histogram(props.data),
        counts = bars.map((d) => d.y);

    this.widthScale
        .domain([d3.min(counts), d3.max(counts)])
        .range([9, props.width-props.axisMargin]);

    this.yScale
        .domain([0, d3.max(bars.map((d) => d.x+d.dx))])
        .range([0, props.height-props.topMargin-props.bottomMargin]);
}
```

Drawing (rendering)

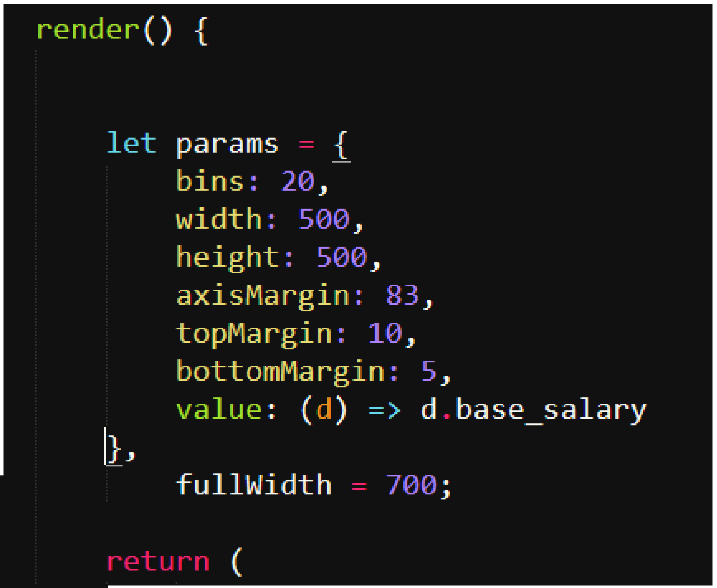


```
render() {
    let translate = `translate(0, ${this.props.topMargin})`;
        bars = this.histogram(this.props.data); /* drawing */

    return (

        <g className="histogram" transform={translate}>
            <g className="bars">
                {bars.map(::this.makeBar)}  /* drawing */
            </g>
        </g>
    );
}
```

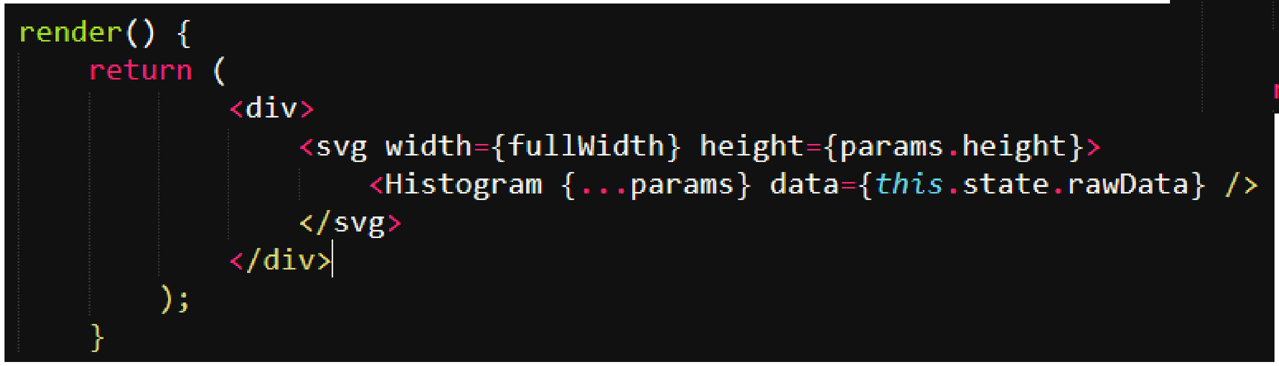
Adding Histogram component to the main component H1BGraph



```
import React, { Component } from 'react';
import d3 from 'd3';
import Histogram from '../Histogram';

/* Empty HB1Graph Component */
class H1BGraph extends Component {
```

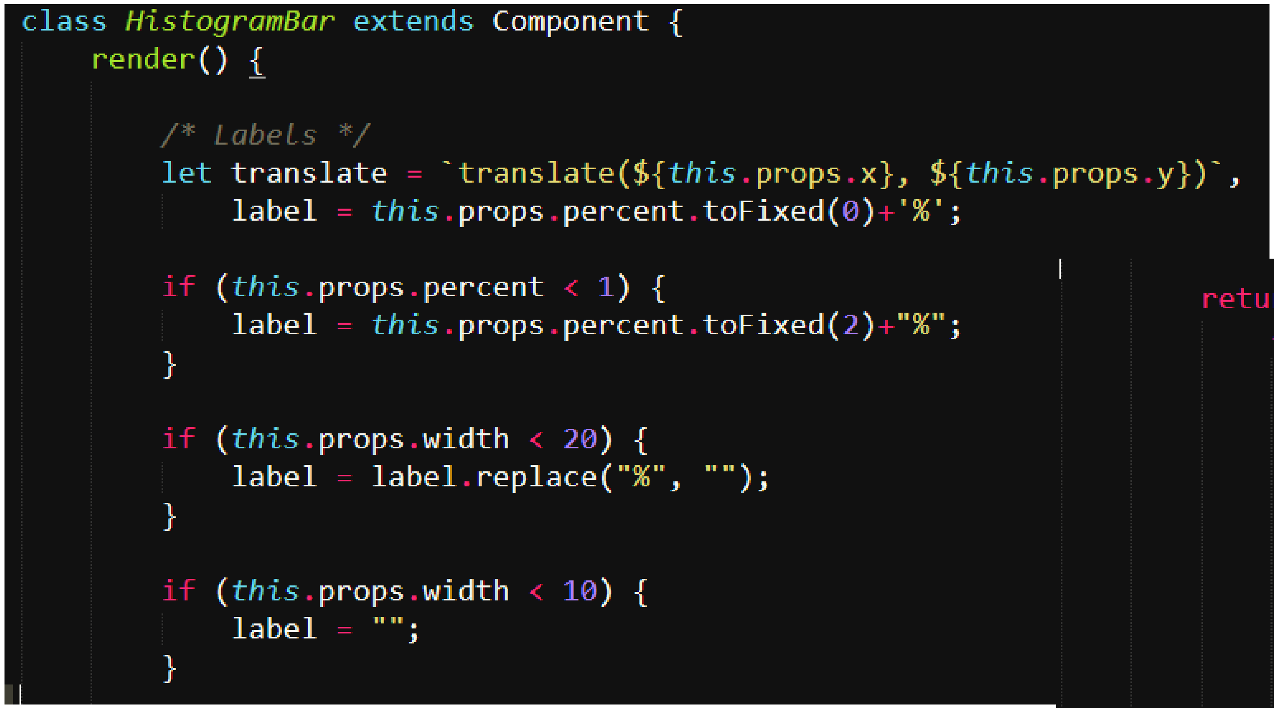


```
render() {


    let params = {
        bins: 20,
        width: 500,
        height: 500,
        axisMargin: 83,
        topMargin: 10,
        bottomMargin: 5,
        value: (d) => d.base_salary
    },
        fullWidth = 700;

    return (
```

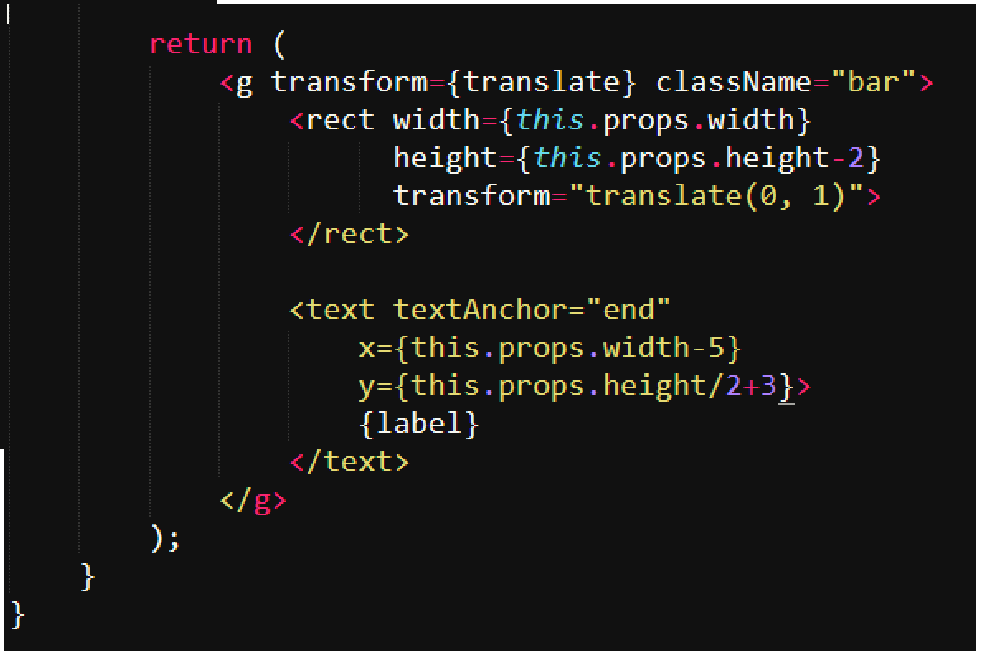


```
render() {
    return (
            <div>
                <svg width={fullWidth} height={params.height}>
                    <Histogram {...params} data={this.state.rawData} />
                </svg>
            </div>
        );
    }
```

Displaying the result



```
class HistogramBar extends Component {
    render() {

        /* Labels */
        let translate = `translate(${this.props.x}, ${this.props.y})`,
            label = this.props.percent.toFixed(0)+'%';

        if (this.props.percent < 1) {
            label = this.props.percent.toFixed(2)+"%";
        }

        if (this.props.width < 20) {
            label = label.replace("%", "");
        }

        if (this.props.width < 10) {
            label = "";
        }
```



```
        return (
            <g transform={translate} className="bar">
                <rect width={this.props.width}
                      height={this.props.height-2}
                      transform="translate(0, 1)">
                </rect>

                <text textAnchor="end"
                    x={this.props.width-5}
                    y={this.props.height/2+3}>
                    {label}
                </text>
            </g>
        );
    }
}
```

Gluing it together

- Adding Histogram to the main component
  - First we have to add drawers.jsx to our requires.
  - Then add the histogram component to our render method
  - And that’s it. You can render as many histograms as you want just by adding <drawers.Histogram /> to your output and setting the properties.

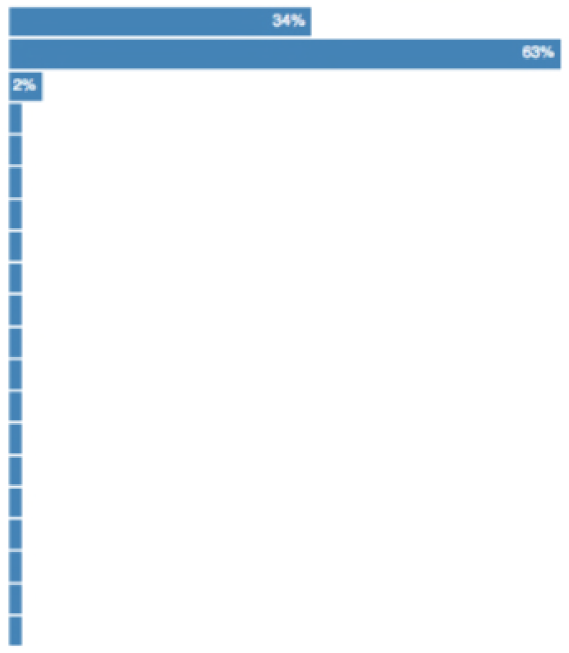
Require drawers.jsx

```
var React = require('react'),
    _ = require('lodash'),
    d3 = require('d3'),
    drawers = require('./drawers.jsx');

var H1BGraph = React.createClass({
```

```
        return (
            <div className="row">
                <div className="col-md-12">
                    <svg width={fullWidth} height={params.height}>
                        <drawers.Histogram {...params} data={this.state.rawData}\
/>
                    </svg>
                </div>
            </div>
        );
    }
```

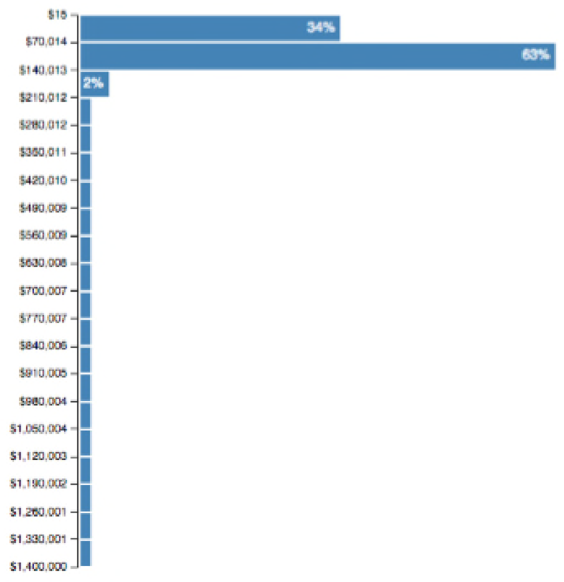
Results

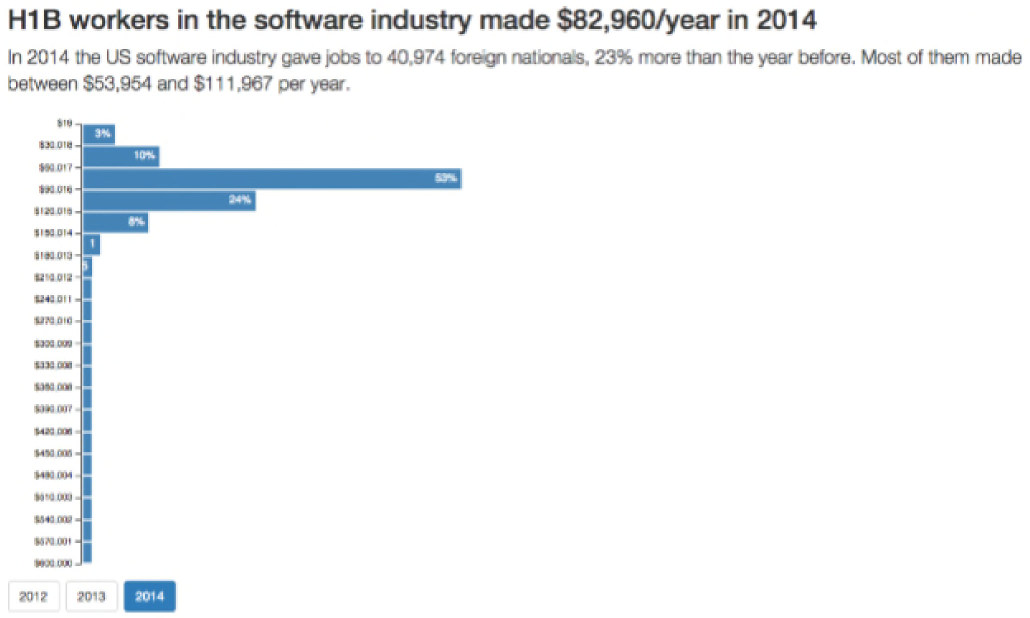
- The basic histogram looks like this



Next step – Adding an axis

- Adding an axis, a title and a description

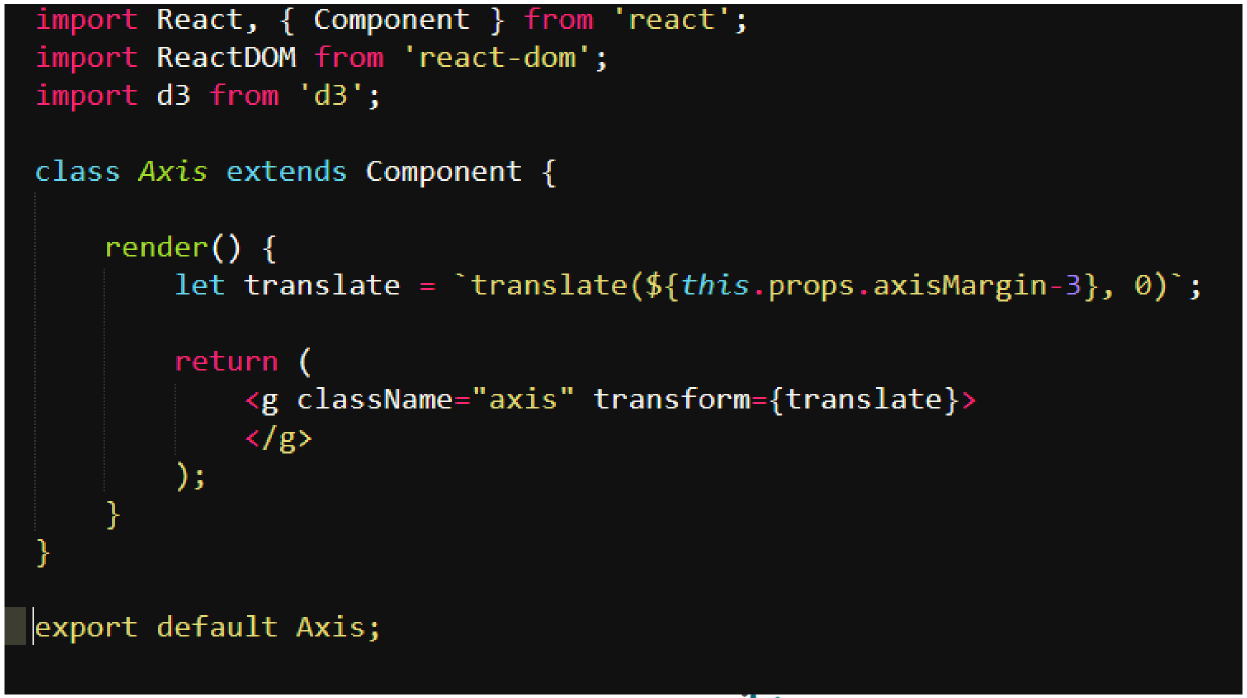



H1B workers in the software industry made $82,960/year in 2014

In 2014 the US software industry gave jobs to 40,974 foreign nationals, 23% more than the year before. Most of them made between $53,954 and $111,967 per year.

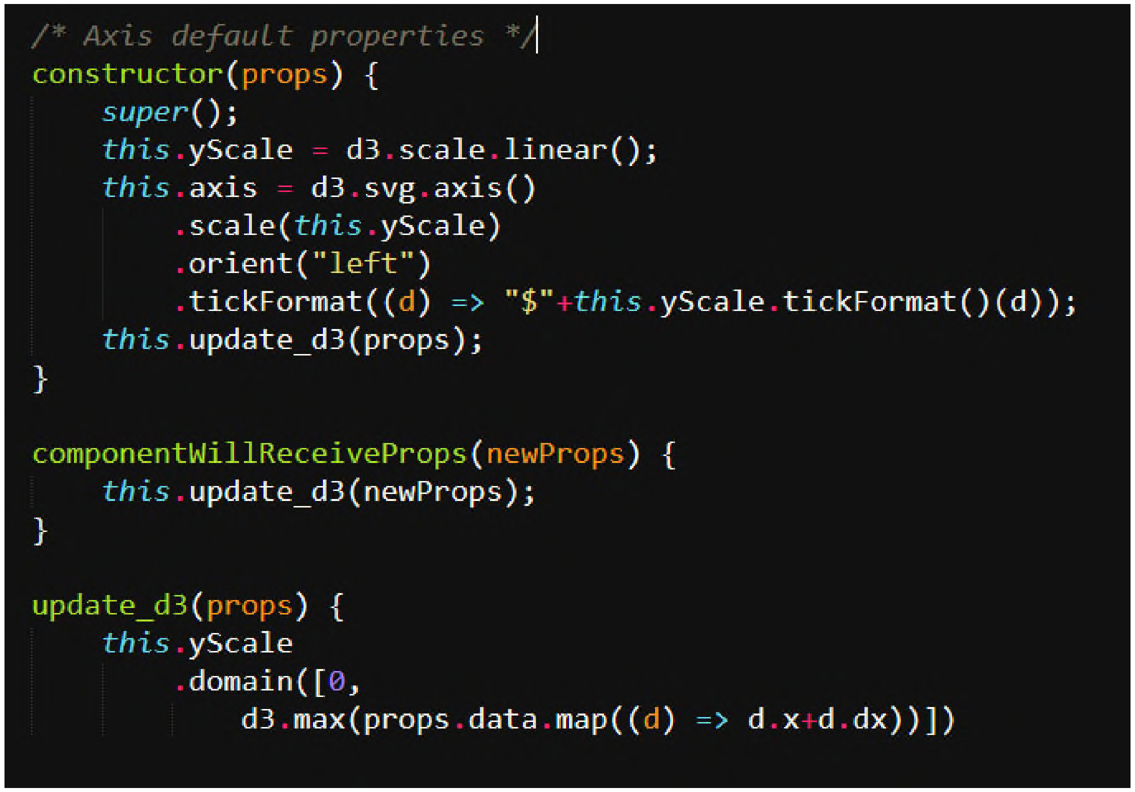
The wrapper in details (Axis.jsx)

- **Wrapping a pure-d3 element in React**



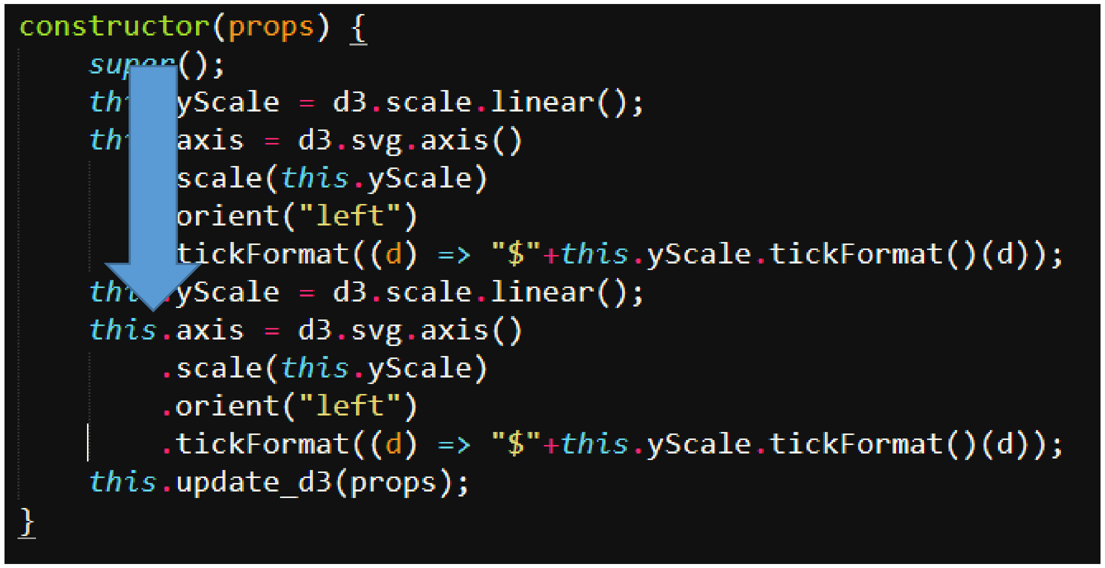
```jsx
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
import d3 from 'd3';

class Axis extends Component {

    render() {
        let translate = `translate(${this.props.axisMargin-3}, 0)`;

        return (
            <g className="axis" transform={translate}>
            </g>
        );
    }
}

export default Axis;
```

Update Axis properties



```
/* Axis default properties */
constructor(props) {
    super();
    this.yScale = d3.scale.linear();
    this.axis = d3.svg.axis()
        .scale(this.yScale)
        .orient("left")
        .tickFormat((d) => "$"+this.yScale.tickFormat()(d));
    this.update_d3(props);
}

componentWillReceiveProps(newProps) {
    this.update_d3(newProps);
}

update_d3(props) {
    this.yScale
        .domain([0,
            d3.max(props.data.map((d) => d.x+d.dx))])
```

Update Axis State



```
constructor(props) {
    super();
    this.yScale = d3.scale.linear();
    this.axis = d3.svg.axis()
        .scale(this.yScale)
        .orient("left")
        .tickFormat((d) => "$"+this.yScale.tickFormat()(d));
    this.yScale = d3.scale.linear();
    this.axis = d3.svg.axis()
        .scale(this.yScale)
        .orient("left")
        .tickFormat((d) => "$"+this.yScale.tickFormat()(d));
    this.update_d3(props);
}
```

Then add Axis to the Histogram component with **import Axis from './Axis';**

The dirty trick for embedding d3 renders

- We hook into the componentDidUpdate and componentDidMount callbacks with a renderAxis method. This ensures renderAxis gets called every time our component has to re-render.
- In renderAxis we use ReactDOM.findDOMNode() to find this component's DOM node. We feed the node into d3.select(), which is how node selection is done in d3 land, then .call() the axis. This lets axis do its thing and add a bunch of SVG elements inside our node in the shape of an axis.
- As a result, we re-render the axis from scratch on every update. This is inefficient because it goes around React's fancy tree diffing algorithms, but it works great.

Next step – Reacting to data changes

Adding a filter by year and user controls to define it

- Our approach will follow the approach of having events flow up the hierarchy and data updates flowing down.
- When a user clicks a button, it calls a function on its parent.

Adding user controls

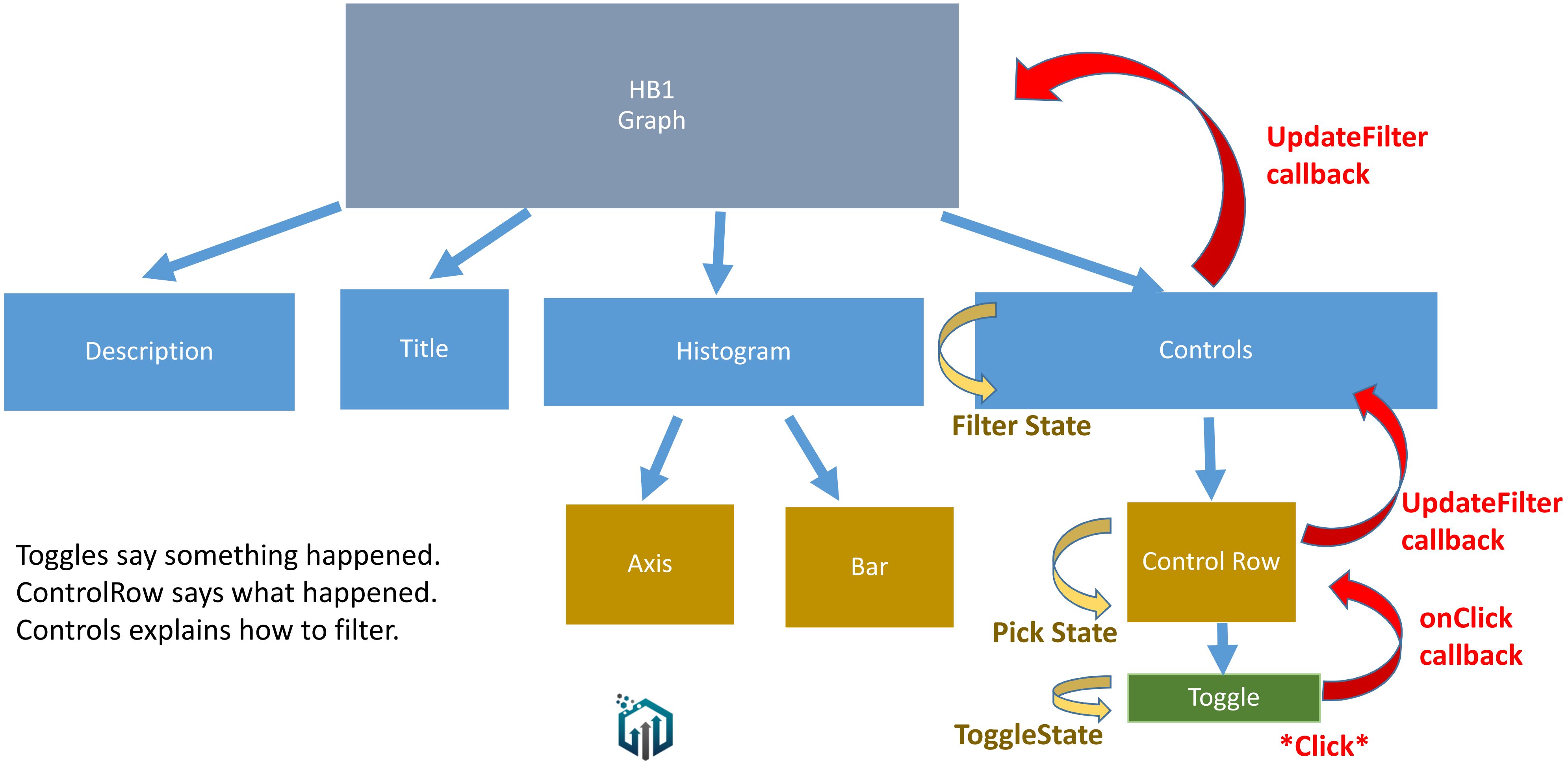
- Controls, which holds the controls together
- ControlRow, which is a row of buttons
- Toggle, which is a toggleable button

Principle

- User events are going to flow up the component hierarchy via callbacks
- Main component will change its data state
- React will propagate data changes back down through component prop

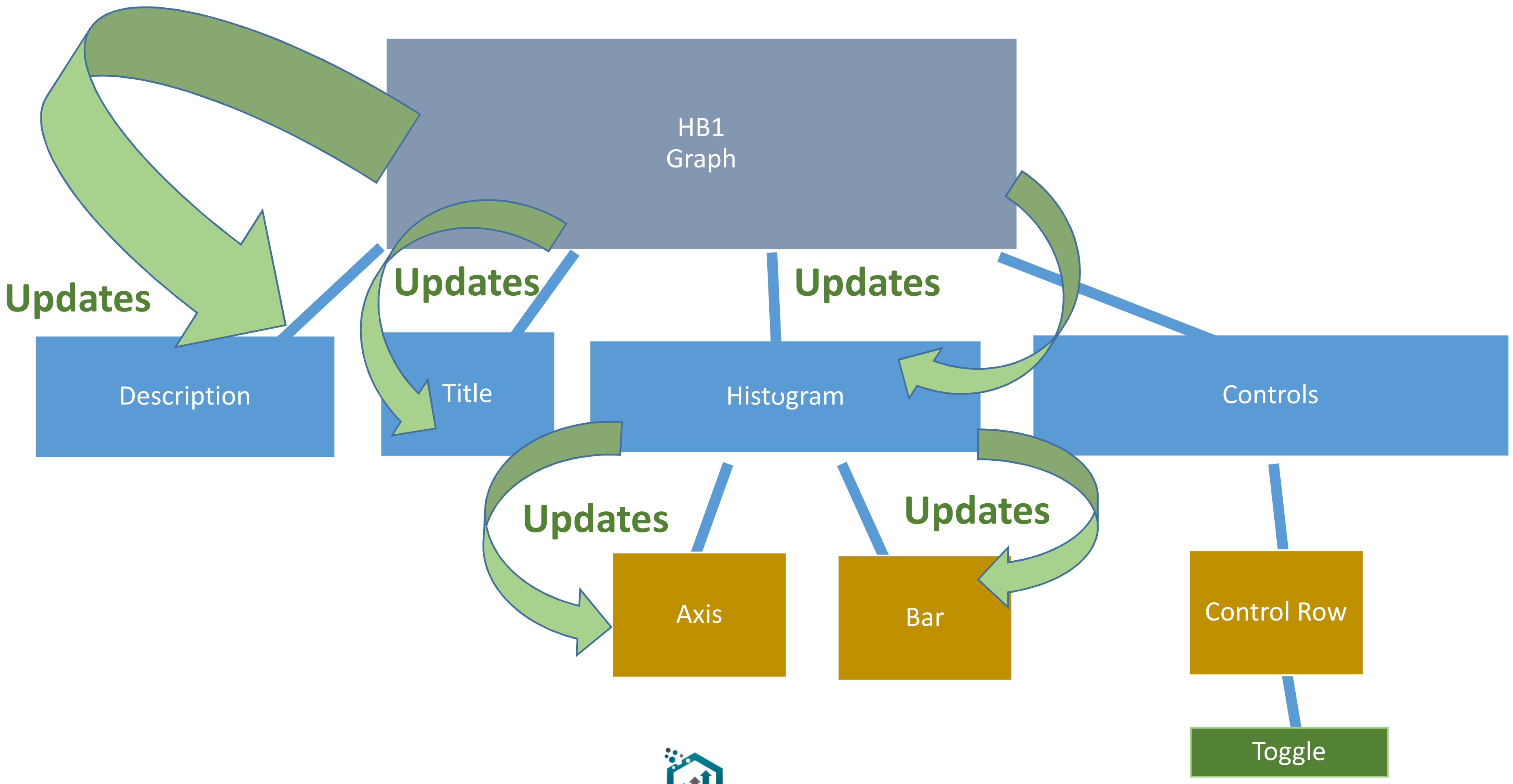
Architecture in details

Events flow up through callbacks



Toggles say something happened.
ControlRow says what happened.
Controls explains how to filter.

Architecture in details - Updated data flows back down



A few details on Toggle

We return a <Toggle> component with some properties:

- label for the visible label
- name for the button’s name property
- key is something React needs to tell similar components apart; it has to be unique
- value for the button’s initial value state
- onClick for the click callback

The benefit of this isolation of components

- A lot of work went into that and we’ve got plenty of moving parts. But remember, all it takes to add more filters is adding another

- <ControlRow /> to Controls, writing a filter function, and making sure it’s included in updateDataFilter.

- That’s it. A minute of typing to let users filter by almost anything.

Final Word

Now you know how to make React and d3.js work together.

You know much more about making reusable visualizations than you did an hour ago.

Any Questions ?