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 is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

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

Why programmers work at night

Contents

Introduction

Hi, I wrote this book for you as an experiment. The theory we're testing is that technical content works better as a short e-book than as a long blog post.

You see, the issue with technical blogging is that posts live and die by their length. As soon as you surpass 700 words, everything falls apart. Readers wander off to a different tab in search of other material, never to be seen or commented from again.

This sucks for me as a writer, but more importantly, it sucks for you as a reader. You're not going to learn something if you don't finish reading.

So here we both are.

I'm counting on you to take some time away from the internet, to truly sit down and read. I'm counting on you to follow the examples. I'm counting on you to *learn*.

You're counting on me to have invested more time, effort, and care in crafting this than I would invested in a blog post. I have.

I've tried to keep this book as concise as possible. iA Writer estimates it will take you about an hour to read React+d3.js in its entirety, but playing with the examples could add some time to that.

Why you should read React+d3.js

After an hour with React+d3.js, you'll know how to make React and d3.js play together. You'll know how to create composable data visualizations. You're going to understand *why* that's a good idea, and you will have the tools to build your own library of reusable visualization parts.

Ultimately, you're going to understand whether React and d3.js fit the needs of your project.

What you need to know

I'm going to assume you already know how to code and that you're great with JavaScript. Many books have been written to teach the basics of JavaScript; this is not one of them.

I'm also going to assume some knowledge of d3.js. Since it isn't a widely-used library, I'm still going to explain the specific bits that we use. If you want to learn d3.js in depth, you should read my book, Data Visualization with d3.js[1].

React is still a new kid on the block, so I'm going to assume you're not quite as familiar. We're not going to talk about all the details, but you'll be fine, even if this is your first time looking at React.

How to read this book

Relax. Breathe. You're here to learn. I'm here to teach. I promise Twitter will wait and so will Facebook.

Just you and some fun code. To get the most out of this material I suggest two things:

1. Try the example code yourself. Don't just copy-paste; type it and execute it. Execute it frequently. If something doesn't fit together, look at the full working project on Github here[2], or check out the zip files that came with the book.
2. If you already know something, skip that section. You're awesome. Thanks for making this easier for me.

React+d3.js is heavily based on code samples. They look like this:

```
var foo = 'bar';
```

Added code looks like this:

```
var foo = 'bar';
foo += 'this is added';
```

Removed code looks like this:

```
var foo = 'bar';
~~foo += 'this is added';~~
```

ES5 and ES6 versions

This book comes in two versions: ES5 and ES6. They are functionally the same, but use different versions of JavaScript for their code samples. The ES6 version also has a cleaner, but more complicated, file structure.

You can read either version depending on which version of JavaScript you're more comfortable with.

As of November 30th, 2015, the ES6 version is delayed by a few days, but you'll get it soon.

[2] https://github.com/Swizec/h1b-software-salaries

Why React and d3.js

React is Facebook's and Instagram's approach to writing modern JavaScript front-ends. It encourages building an app out of small, re-usable components. Each component is self-contained and only needs to know how to render a small bit of the interface.

The catch is that many frameworks have attempted this: everything from Angular to Backbone and jQuery plugins. But where jQuery plugins quickly become messy, Angular depends too much on HTML structure, and Backbone needs a lot of boilerplate, React has found a sweet spot.

I have found it a joy to use. Using React was the first time I have ever been able to move a piece of HTML without having to change any JavaScript.

D3.js is Mike Bostock's infamous data visualization library. It's being used by The New York Times along with many other sites. It is the workhorse of data visualization on the web, and many charting libraries out there are based on it.

But 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.

Just like React, d3.js is declarative. You tell it *what* you want, instead of *how* you want it. It gives you access straight to the SVG so you can manipulate your lines and rectangles at will. The issue is that d3.js isn't that great if all you want are charts.

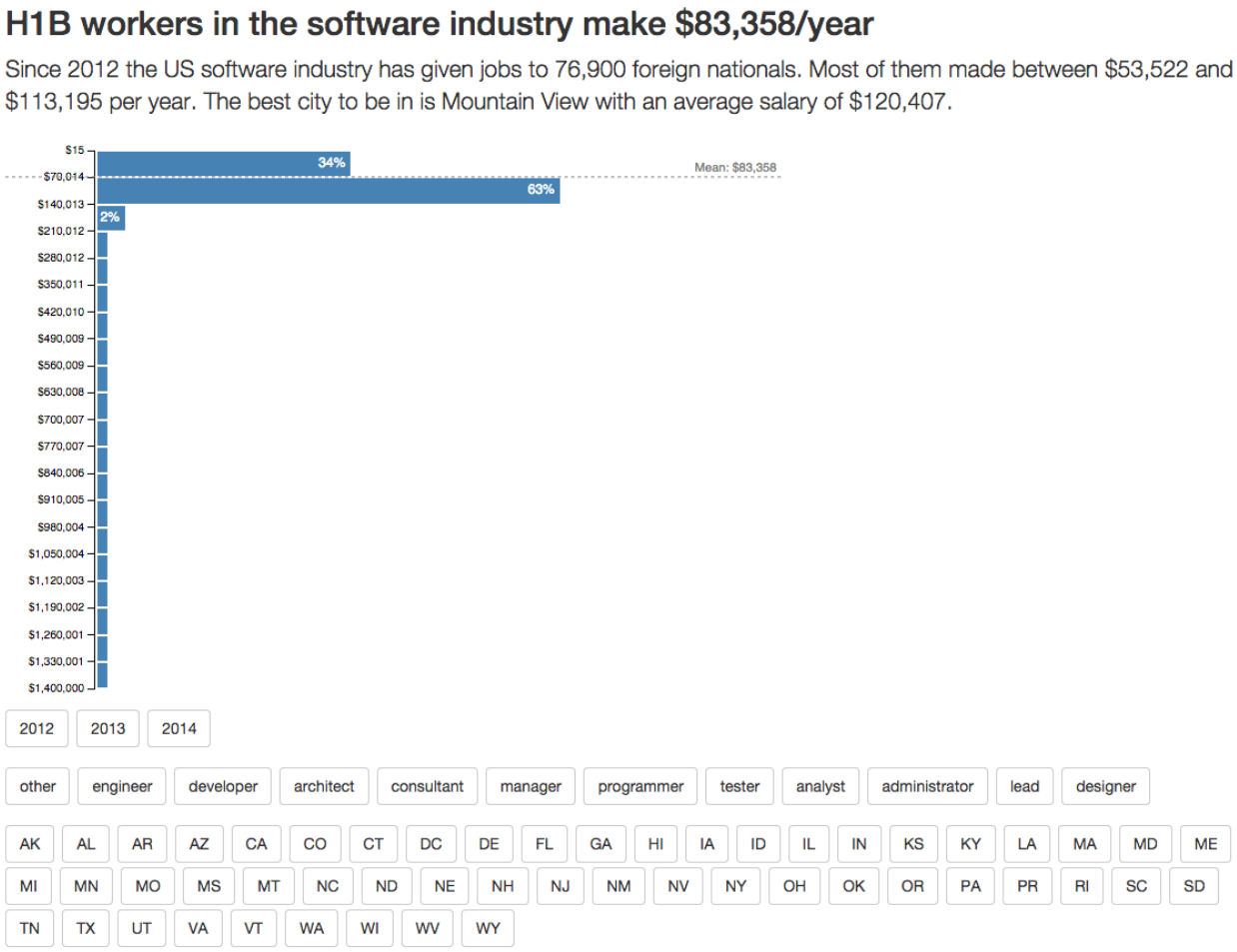
This is where React comes in. For instance, once you've created a histogram component, you can always get a histogram with `<Histogram {...params} />`.

Doesn't that sound like the best? I think it's pretty amazing.

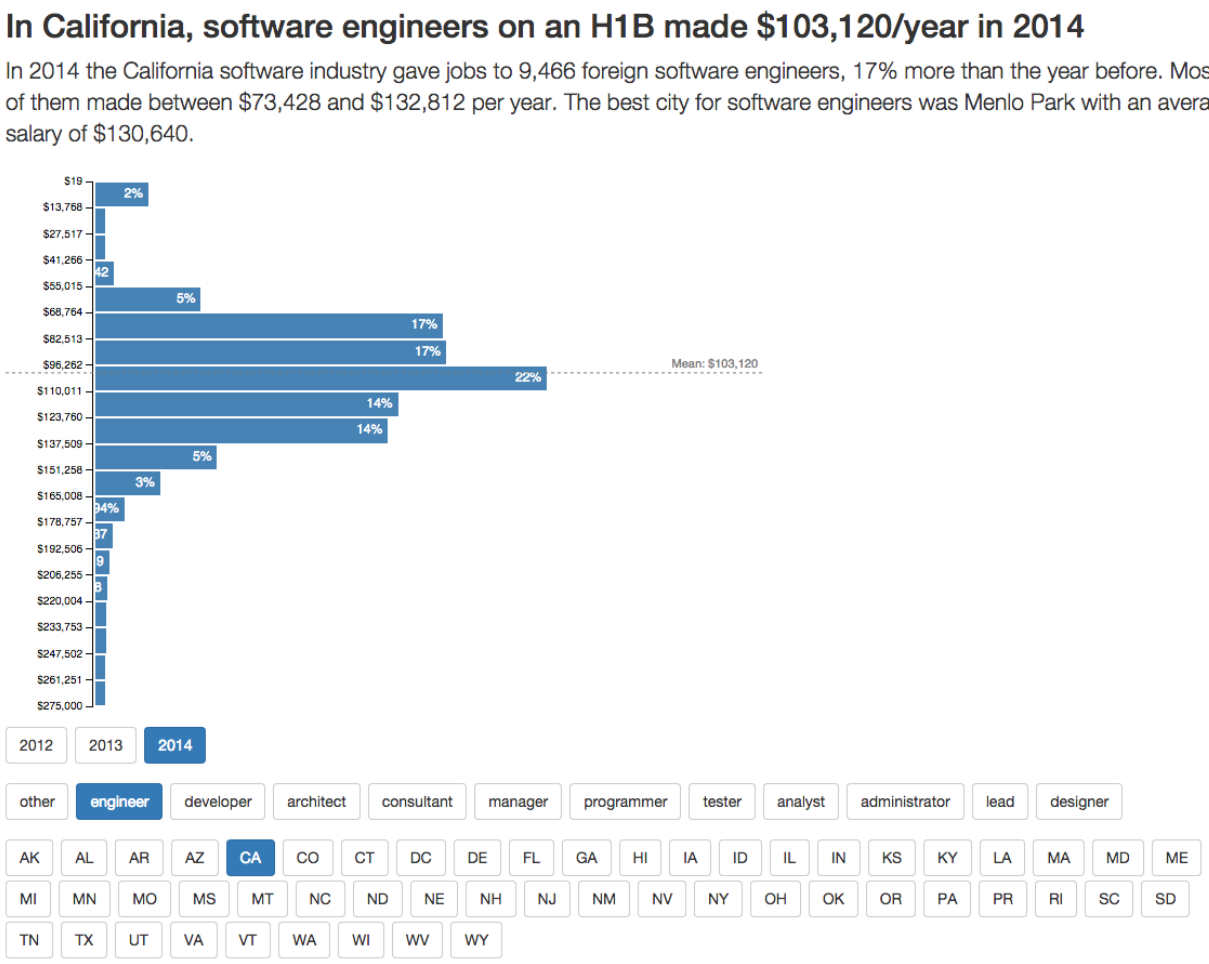
And it gets even better. With React, you can make various graph and chart components build off the same data. This means that when your data changes, the whole visualization reacts.

Your graph changes. The title changes. The description changes. Everything changes. Mind = blown.

Look how this H1B salary visualization changes when the user picks a subset of the data to look at.



Default H1B histogram



Changes after user picks a data subset

React + d3.js. A powerful combination indeed.

A good work environment

Before we begin, we need a good work environment. We're after three things:

- 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

When I first wrote this chapter in April 2015, I suggested using a combination of Browserify, Grunt, NPM, and Bower. This was the wrong approach. It was complicated to set up, it wasn't very extensible, and it took a while to compile everything.

The project on which this book is based takes almost 7 seconds to compile, and Grunt's file change detection isn't as smooth as it could be. On top of that, you also have to wait for `live-server` to detect the compiled files' changes.

When there's an error in your code, the browser has no idea where it happened because it only sees the compiled file.

To make matters worse, your page loses state every time it auto-reloads. That happens twice per file change - first when the server sees a file has changed, and again when Grunt compiles the code and changes the compiled file.

It's a mess. I'm sorry I told you to use it. The old system is included in the appendix for the curious and those stuck in legacy environments.

If you already know how to set up the perfect development environment for modern JavaScript work, go ahead and skip this section.

Bundle with Webpack, compile with Babel

Webpack calls itself a *"flexible unbiased extensible module bundler"*, which sounds like buzzword soup. At its most basic, 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. I can't hope to know even half of them, but some of the coolest I've used are plugins that let you 'require()' Less files *with* magical Less-to-CSS compilation, plugins for require-ing images, and JavaScript minification.

Webpack can solve two more annoyances - losing state when loading new code and accurately reporting errors. We add two more requirements for a total of five:

- 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

Babel

Webpack can't do all this alone though - it needs a compiler.

We're going to use Babel to compile our JSX and ES6 code into the kind of code all browsers understand: ES5. Don't worry if you're not ready to learn ES6, you can read the ES5 version of React+d3.js.

Babel isn't really a compiler because it produces JavaScript, not machine code. That being said, it's still important at this point. According to the JavaScript roadmap, browsers aren't expected to fully support ES6 until some time in 2017. That's a long time to wait, so the community came up with transpilers which let us use some ES6 features *right now*. Yay!

Transpilers are the officially-encouraged stepping stone towards full ES6 support.

To give you a rough idea of what Babel does with your code, here's a fat arrow function with JSX. Don't worry if you don't understand this code yet; we'll go through that later.

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

After Babel transpiles that line into ES5, it looks like this:

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

Babel developers have created a playground that live-compiles[3] code for you. Have fun.

[3] https://babeljs.io/repl/

Quickstart

The quickest way to set this up is using Dan Abramov's react-transform-boilerplate[4] project. It's what I use for new projects these days.

If you know how to do this already, skip ahead to the Visualizing data with React.js chapter. In the rest of this chapter, I'm going to show you how to get started with the boilerplate project. I'll also explain some of the moving parts that make it tick.

NPM for dependencies and tools

NPM is node.js's default package manager. Originally developed as a dependency management tool for node.js projects, it's since taken hold of the JavaScript world as a way to manage the tool belt, and with Webpack's growing popularity, a way to manage client-side dependencies as well.

That's because Webpack automatically recognizes NPM modules, which means we are going to use NPM to install both our toolbelt dependencies (like Webpack) and our client-side dependencies (like React and d3.js).

You can get NPM by installing node.js from nodejs.org[5]. Webpack and our dev server will run in node.

Once you have NPM, you can install Webpack globally with:

```
$ npm install webpack -g
```

If that worked, you're ready to go. If it didn't, Google is your friend.

At this point, there are two ways to proceed:

- You can continue with the step-by-step instructions using a boilerplate
- If you bought the Engineer or Business package, you can use the included stub project that has everything you need

Step-by-step with boilerplate

All it takes to start a project from boilerplate is to clone the boilerplate project, remove a few files, and run the code to make sure everything works.

You will need Git for this step. I assume you have it already because you're a programmer. If you don't, you can get it from Git's homepage[6]. For the un-initiated, Git is a source code versioning tool.

Head to a directory of your choosing, and run:

[4] https://github.com/gaearon/react-transform-boilerplate
[5] http://nodejs.org
[6] https://git-scm.com/

```
$ git clone git@github.com:gaearon/react-transform-boilerplate.git
```

This makes a local copy of the boilerplate project. Now that we've got the base code, we should make it our own.

Our first step is to rename the directory and remove Git's version history and ties to the original project. This will allow us to turn it into our own project.

```
$ mv react-transform-boilerplate react-d3-example
$ rm -rf react-d3-example/.git
$ cd react-d3-example
```

We now have a directory called `react-d3-example` that contains some config files and a bit of code. most importantly, it isn't tied to a Git project, so we can make it all ours.

Make it your own

To make it our own, we have to change some information inside `package.json`: the name, version, and description.

```
        {
-         "name": "react-transform-boilerplate",
-         "version": "1.0.0",
-         "description": "A new Webpack boilerplate with hot reloading React components\
, and error handling on module and component level.",
          "name": "react-d3-example",
          "version": "0.1.0",
          "description": "An example project to show off React and d3 working together",
          "scripts": {
```

It's also a good idea to update the `author` field:

```
-"author": "Dan Abramov <dan.abramov@me.com> (http://github.com/gaearon)",
        "author": "Swizec <swizec@swizec.com> (http://swizec.com)
```

Use your own name, email, and URL. Not mine :)

If you want to use Git to manage source code versioning, now is a good time to start. You can do that by running:

```
$ git init
$ git add .
$ git commit -a -m "Start project from boilerplate"
```

Great, now we have a project that we've signed.

Our new project comes preconfigured for React and all the other tools and compilers we need to run our code. Install them by running:

```
$ npm install
```

This will install a bunch of dependencies like React, a few Webpack extensions, and a JavaScript transpiler (Babel) with a few bells and whistles. Sometimes, parts of the installation fail. If it happens to you, try re-running `npm install` for the libraries that threw an error. I don't know why this happens, but you're not alone. I've been seeing this behavior for years.

Now that we have all the basic libraries and tools we need to run our code, we have to install three more:

1. `d3` for drawing
2. `lodash` for some utility functions
3. `autobind-decorator`, which I will explain later.

```
$ npm install --save d3 lodash autobind-decorator
```

The `--save` option saves them to `package.json`.

Add LESS compiling

Less is my favorite way to write stylesheets. It looks almost like traditional CSS, but gives you the ability to use variables, nest definitions, and write mixins. We won't need much of this for the H1B graphs project, but nesting will make our style definitions nicer, and LESS will make your life easier in bigger projects.

Webpack can handle compiling LESS to CSS for us. We just have to install a couple of Webpack loaders, and add three lines to the config.

Let's start with the loaders:

```
$ npm install --save style-loader less less-loader
```

Remember, `--save` adds `style-loader` and `less-loader` to package.json. The `style-loader` takes care of transforming `require()` calls into `<link rel="stylesheet"` definitions, and `less-loader` takes care of compiling LESS into CSS.

To add them to our build step, we have to go into `webpack.config.dev.js`, find the `loaders: [` definition, and add a new object like this:

Add LESS loaders

```
19 module: {
20     loaders: [{
21         test: /\.js$/,
22         loaders: ['babel'],
23         include: path.join(__dirname, 'src')
24     },
25     {
26         test: /\.less$/,
27         loader: "style!css!less"
28     }
29     ]
30 }
```

Don't worry if you don't understand what the rest of this file does. We're going to look at that in the next section.

Our addition tells Webpack to load any files that end with `.less` using `style!css!less`. The `test:` part is a regex that describes which files to match, and the `loader` part uses bangs to chain three loaders. The file is first compiled with `less`, then compiled into `css`, and finally loaded as a `style`.

If everything went well, we should now be able to use `require('./style.less')` to load style definitions. This is great because it allows us to have separate style files for each component, and that makes our code more reusable since every module comes with its own styles.

Change a few knickknacks

There's a few more things we have to change to make the rest of this book flow more smoothly.

The first and most important is to make sure we can load our data files while we're running the project through our local server. We have to add a line to `devServer.js`:

Enable static server on ./public

```
app.use(require('webpack-hot-middleware')(compiler));

app.use(express.static('public'));

app.get('*', function(req, res) {
```

Don't worry if you don't understand what this line does. We're going to look at this file in more detail later.

Now we'll add two nice-to-haves for `webpack.config.dev.js`. They aren't super important, but I like to add them to make my life a little easier.

I like to add the `.jsx` extension to the list of files loaded with Babel. This lets me write React code in `.jsx` files. I know what you're thinking: writing files like that is no longer encouraged by the community, but hey, it makes my Emacs behave better.

Add .jsx to Babel file extensions

```
module: {
   loaders: [
      {test: /\.js$/,
      {test: /\.js|\.jsx$/
       loaders: ['babel'],
       include: path.join(__dirname, 'src')
      },
      {test: /\.less$/,
       loader: "style!css!less"
      }
    ]
  }
```

We changed the `test` regex to add `.jsx`. You can read more detail about how these configs work in later parts of this chapter.

Finally, I like to add a `resolve` config to Webpack. This lets me load files without writing their extensions. It's a small detail, but it makes your code cleaner.

Add resolve to webpack.config.dev.js

```
    new webpack.NoErrorsPlugin()
  ],
  resolve: {
    extensions: ['', '.js', '.jsx']
  },
  module: {
```

It's a list of file extensions that Webpack tries to guess when a path you use doesn't match any files

Check that it works

Your environment should be ready to get started now. Let's try it out. First, start the dev server:

```
$ npm start
```

This command runs a small static file server that's written in node.js. The server ensures that Webpack continues to compile your code when it detects a file change. It also puts some magic in place that hot loads code into the browser without refreshing and without losing variable values.

Assuming there were no errors, you can go to `http://localhost:3000` and see a counter doing some counting. That's the sample code that comes with the boilerplate.

If it worked: Awesome, you got the code running! Your development environment works! Hurray!

If it didn't work: Well, um, poop. A number of things could have gone wrong. I would suggest making sure `npm install` ran without issue, and if it didn't, try Googling for any error messages that you get.

Remove sample code

Now that we know our development environment works, we can get rid of the sample code inside `src/`. We're going to put our own code files in there.

We're left with a skeleton project that's full of configuration files, a dev server, and an empty `index.html`. This is a good opportunity for another `git commit`.

Done? Wonderful.

In the rest of this chapter, we're going to take a deeper look into all the config files that came with our boilerplate. If you don't care about that right now, you should jump straight to the meat.

What's in the environment

Boilerplate is great because it lets you get started right away. No setup, no fuss, just `npm install` and away we go.

But you *will* have to change something eventually, and when you do, you'll want to know what to look for. There's no need to know every detail in every config file, but you do have to know enough so that you can Google for help.

Let's take a deeper look at the config files to make future googling easier. We're relying on Dan Abramov's `react-transform-boilerplate`, but many others exist with different levels of bells and whistles. I like Dan's because it's simple.

All modern boilerplates are going to include at least two bits:

- the webpack config
- the dev server

Everything else is optional.

Webpack config

Wepback is where the real magic happens so this is the most important configuration file in your project. It's just a JavaScript file though so there's nothing to fear.

Most projects have two versions of this file: a dev version, and a prod version. The first is geared more towards what we need in development - a compile step that leaves our JavaScript easy to debug - while the second is geared towards what we need in production - compressed and uglified JavaScript that's quick to load.

Since both files are so similar, we're only going to look at the dev version.

It comes in four parts:

Webpack config structure

```
var path = require('path');
var webpack = require('webpack');

module.exports = {
  devtool: 'eval',
  entry: [
  ],
  output: {
  },
  plugins: [
  ],
  module: {
    loaders: []
  }
};
```

- **Entry**, which tells Webpack where to start building our project's dependency tree.
- **Output**, which tells Webpack where to put the result. This is what our index.html file loads.
- **Plugins**, which tells Webpack what plugins to use when building our code.
- **Loaders**, which tells Webpack about the different file loaders we'd like to use.

There's also the `devtool: 'eval'` option, which tells Webpack how to package our files so they're easier to debug. In this case our code will come inside `eval()` statements, which makes it hot loadable.

Let's go through the four sections one by one.

Entry

The entry section of Webpack's config specifies the entry points of our dependency tree. It's an array of files that `require()` all other files.

In our case, it looks like this:

Entry part of webpack.config.dev.js

```
entry: [
  'webpack-hot-middleware/client',
  './src/index'
],
```

We specify that `./src/index` is the main file. In the next section, you'll see that this is the file that requires our app and renders it into the page.

The `webpack-hot-middleware/client` line enables Webpack's hot loading, which can load new versions of JavaScript files without reloading the page.

If we wanted to compile multiple independent apps with a single Webpack config file, we could add more files. A common example is when you have a separate admin dashboard app for some users while providing a friendly end-user app for others.

Output

The output section specifies which files get the output. Our config is going to put all compiled code into a single `bundle.js` file, but it's common to have multiple output files. In the case of an admin dashboard and a user-facing app, this would allow us to avoid loading unnecessary JavaScript for users who don't need every type of functionality.

The config looks like this:

Output part of webpack.config.dev.js

```
output: {
  path: path.join(__dirname, 'dist'),
  filename: 'bundle.js',
  publicPath: '/static/'
},
```

We define a path, `./dist/`, where compiled files live, say the filename for JavaScript is `bundle.js`, and specify `/static/` as the public path. That means the `<script>` tag in our HTML should use `/static/bundle.js` to get our code, but we should use `./dist/bundle.js` to copy the compiled file.

Plugins

There's a plethora of Webpack plugins out there. We're only going to use two of them in our example.

Plugins part of webpack.config.dev.js

```
plugins: [
  new webpack.HotModuleReplacementPlugin(),
  new webpack.NoErrorsPlugin()
],
```

As you might have guessed, this config is just an array of plugin object instances. Both plugins we're using come with Webpack by default. Otherwise, we'd have to `require()` them at the top of the file.

`HotModuleReplacementPlugin` is where the hot loading magic happens. I have no idea how it works, but it's the most magical thing that's ever happened to my coding abilities.

The `NoErrorsPlugin` makes sure that Webpack doesn't error out and die when there's a problem with our code. The internet recommends using it when you rely on hot loading new code.

Loaders

Finally, we come to the loaders section. Much like with plugins, there is a universe of Webpack loaders out there, and I've barely scratched the surface.

If you can think of it, there's likely a loader for it. At my day job, we use a Webpack loader for everything from JavaScript code to images and font files.

For the purposes of this book, we don't need anything that fancy. We just need a loader for JavaScript and styles.

Loaders part of webpack.config.dev.js

```
module: {
 loaders: [
     {test: /\.js|\.jsx$/,
      loaders: ['babel'],
      include: path.join(__dirname, 'src')
     },
     {test: /\.less$/,
      loader: "style!css!less"
     }
  ]
}
```

Each of these definitions comes in three parts:

- **`test`**, which specifies the regex for matching files.

- **`loader` or `loaders`**, which specifies which loader to use for these files. You can compose loader sequences with bangs, `!`.
- optional **`include`**, which specifies the directory to search for files

There might be loaders out there with more options, but this is the most basic I've seen that covers our bases.

That's it for our very basic Webpack config. You can read about all the other options in Webpack's own documentation[7].

My friend Juho Vepsäläinen has also written a marvelous book that dives deeper into Webpack. You can find it at survivejs.com[8].

Dev server

The dev server that comes with Dan's boilerplate is based on the Express framework. It's a popular framework for building websites in node.js. Other boilerplates use different approaches.

Many better and more in-depth books have been written about node.js and its frameworks. In this book, we're only going to take a quick look at some of the key parts.

For example, on line 9, you can see that we tell the server to use Webpack as a middleware. That means the server passes every request through Webpack and lets it change anything it needs.

Lines that tell Express to use Webpack

```
9  app.use(require('webpack-dev-middleware')(compiler, {
10   noInfo: true,
11   publicPath: config.output.publicPath
12 }));
13
14 app.use(require('webpack-hot-middleware')(compiler));
```

The `compiler` variable is an instance of Webpack, and `config` is the config we looked at earlier. `app` is an instance of the Express server.

Another important bit of the `devServer.js` file specifies routes. In our case, we want to serve everything from `public` as a static file, and anything else to serve `index.html` and let JavaScript handle routing.

[7] http://webpack.github.io/docs/
[8] http://survivejs.com

Lines that tell Express how to route requests

```
16 app.use(express.static('public'));
17
18 app.get('*', function(req, res) {
19   res.sendFile(path.join(__dirname, 'index.html'));
20 });
```

This tells Express to use a static file server for everything in `public` and to serve `index.html` for anything else.

At the bottom, there is a line that starts the server:

Line that starts the server

```
22 app.listen(3000, 'localhost', function(err) {
```

I know I didn't explain much, but that's as deep as we can go at this point. You can read more about node.js servers, and Express in particular, in Azat Mardan's books[9]. They're great.

Babel config

Babel works great out of the box. There's no need to configure anything if you just want to get started and don't care about optimizing the compilation process.

But there are a bunch of configuration options[10] if you want to play around. You can configure everything from enabling and disabling ES6 features to sourcemaps and basic code compacting and more. More importantly, you can define custom transforms for your code.

We don't need anything fancy for the purposes of our example project - just the hot module React transform. As you can guess, it enables that hot code loading magic we've mentioned a couple of times.

`.babelrc` is a JSON file that looks like this:

[9] http://azat.co/

[10] http://babeljs.io/docs/usage/options/

.babelrc config

```
{
  "stage": 0,
  "env": {
    "development": {
      "plugins": ["react-transform"],
      "extra": {
        "react-transform": {
          "transforms": [{
            "transform": "react-transform-hmr",
            "imports": ["react"],
            "locals": ["module"]
          }, {
            "transform": "react-transform-catch-errors",
            "imports": ["react", "redbox-react"]
          }]
        }
      }
    }
  }
}
```

I imagine this file is something most people copy paste from the internet, but the basic rundown is that for the `development` environment, we're loading the `react-transform` plugin and enabling two different transforms.

The first one, `react-transform-hmr`, enables hot loading. The second, `react-transform-catch-errors`, uses `redbox-react` to show us errors thrown by the compiler. This makes keeping an eye on the console less important. That gives us one less window with which to convern ourselves.

We don't want either of those in production, so we'll leave the `production` environment without config. Defaults are enough.

A note on Babel 6

Babel 6 came out recently, which changes an important detail: ES6 -> ES5 compilation has to be enabled manually.

The easiest way to do that is to install `babel-preset-es2015`, and add `"es2015"` to the `plugins:` array inside `.babelrc`. This enables all the different transpiles, which now exist as independent projects.

As of late November 2015, `babel-plugin-react-transform`, which our boilerplate relies on, doesn't support Babel 6 yet. That's why we're using Babel 5 and why we don't have to add the `es2015` transform.

Editor config

A great deal has been written about tabs vs. spaces. It's one of the endless debates we programmers like to have. *Obviously* single quotes are better than double quotes … unless … well … it depends, really.

I've been coding since I was a kid, and there's still no consensus. Most people wing it. Even nowadays when editors come with a built-in linter, people still wing it.

But in recent months (years?), a solution has appeared: the `.eslintrc` file. It lets you define project-specific code styles that are programmatically enforced by your editor.

From what I've heard, most modern editors support `.eslintrc` out of the box, so all you have to do is include the file in your project. When you do that, your editor keeps encouraging you to write beautiful code.

The `eslint` config that comes with Dan's boilerplate loads a React linter plugin and defines a few React-specific rules. It also enables JSX linting and modern ES6 modules stuff. By the looks of it, Dan is a fan of single quotes.

.eslintrc for React code

```
Features": {
    "jsx": true,
    "modules": true
  },
  "env": {
    "browser": true,
    "node": true
  },
  "parser": "babel-eslint",
  "rules": {
    "quotes": [2, "single"],
    "strict": [2, "never"],
    "react/jsx-uses-react": 2,
    "react/jsx-uses-vars": 2,
    "react/react-in-jsx-scope": 2
  },
  "plugins": [
    "react"
  ]
}
```

I haven't really had a chance to play around with linting configs like these. Emacs defaults have been good to me so far, but I think these types of configs are a great idea. The biggest problem in a

team is syncing everyone's linter configs, but if you can put a file like this in your Git project, then **BAM!**, everyone's always in sync.

You can find a semi-exhaustive list of options in this helpful gist[11].

That's it. Time to play!

By this point, you not only have a working environment in which to write your code, you also understand how it does what it does, at least from a high-level perspective. The details are far too intricate and, frankly, not that important to your project as a whole.

That's how environments usually are: a combination of cargo culting and rough understanding.

What we care about is that our ES6 code compiles into ES5 so that everyone can run it. We also have it set so that our local version automatically loads new code.

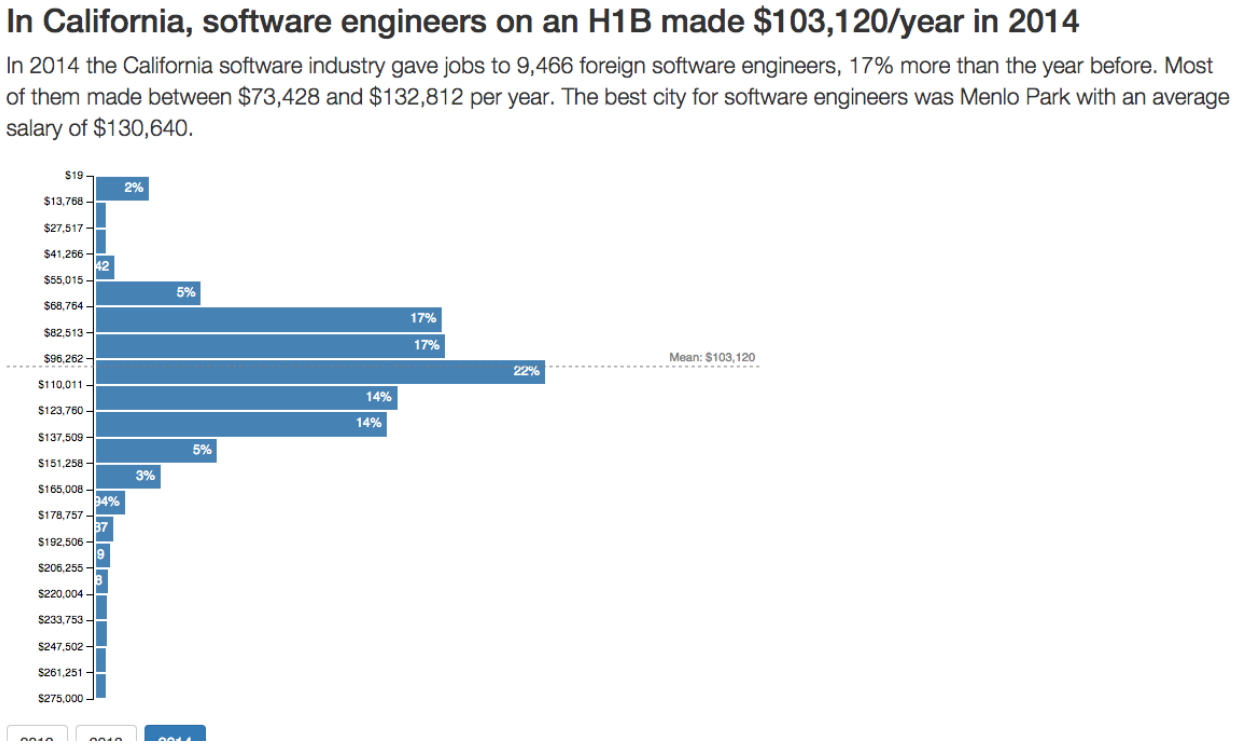
In the next section, we're going to start building a visualization.

[11] https://gist.github.com/cletusw/e01a85e399ab563b1236

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[12] in the United States software industry.



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.

[12] 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:

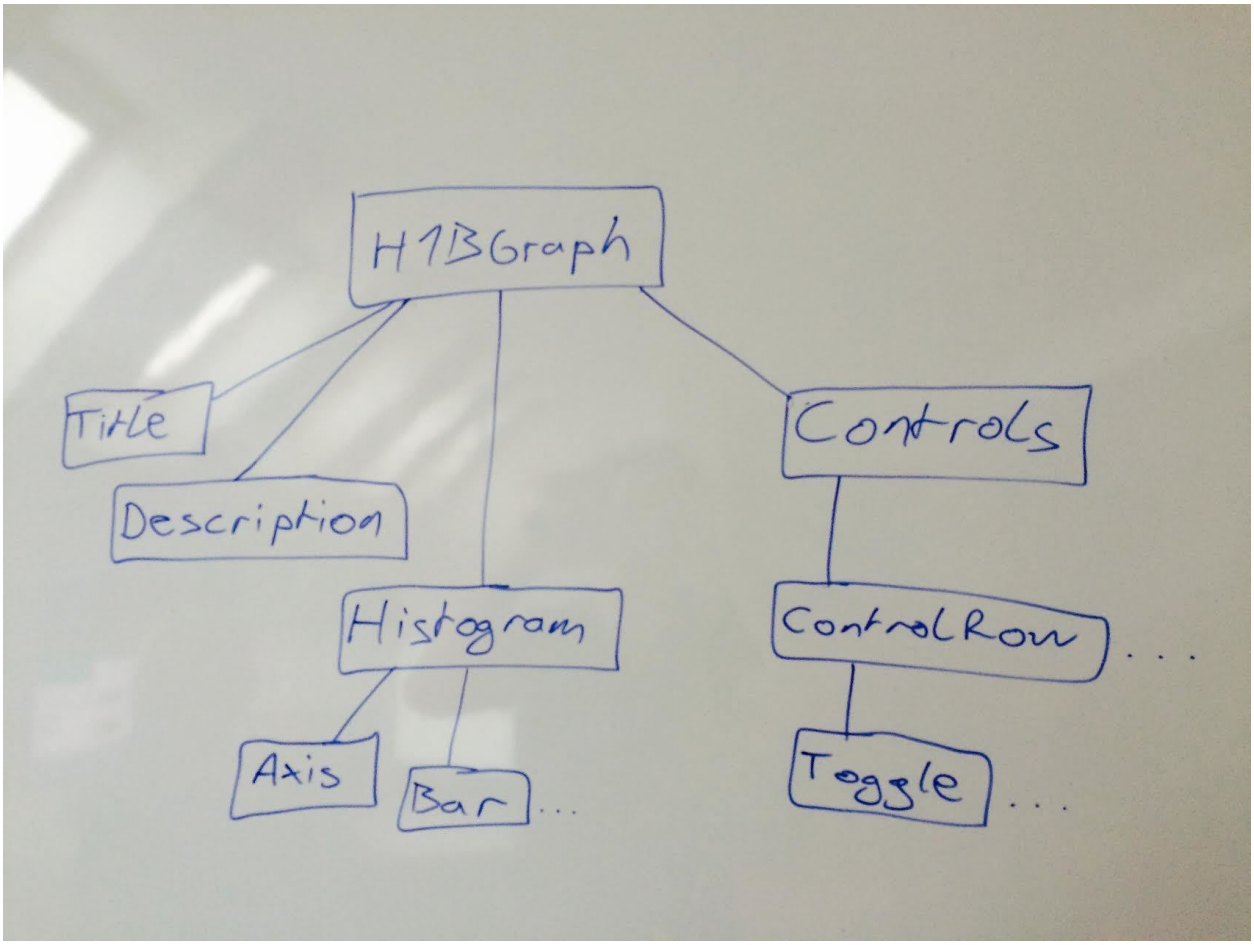
- React owns the DOM
- d3 calculates properties

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.



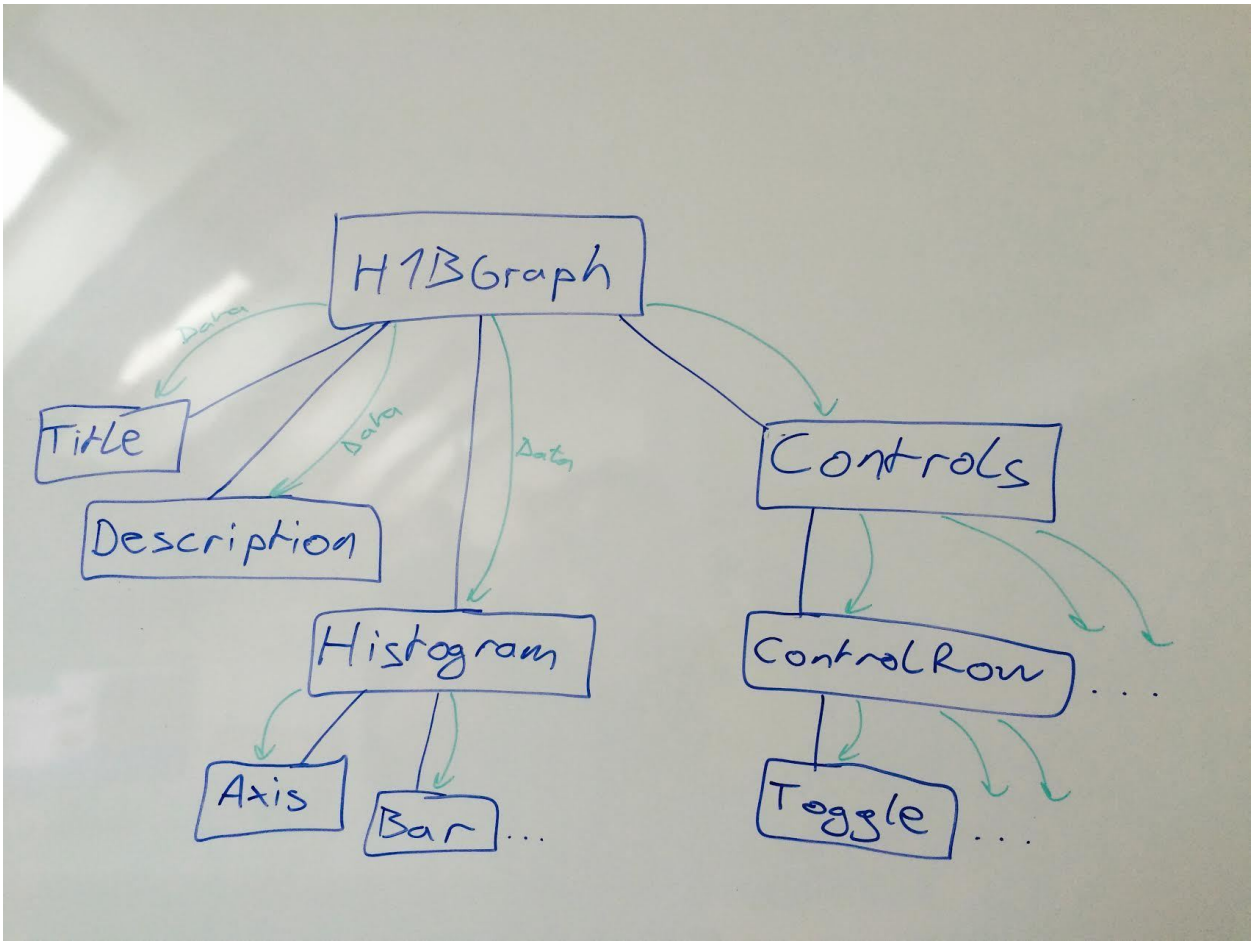
The basic architecture

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

```
<!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.

Loading data

Great, we've got something rendering. Now we need to load some data before we can draw a picture. If you still don't have `public/data/h1bs.csv`, now is the time to get it. You can find it on Github here[13], or in the stub project included with the book.

[13] https://github.com/Swizec/h1b-software-salaries

We're going to use d3.js's built-in data-loading magic and hook it into React's component lifecycle.

We start by adding three short methods to `H1BGraph`:

Base methods to load data

```
var H1BGraph = React.createClass({
    componentWillMount: function () {
        this.loadRawData();
    },

    getInitialState: function () {
        return {rawData: []};
    },

    loadRawData: function () {

    },

    render: function () {
```

React calls `componentWillMount` right before inserting the component into our document. It's a great opportunity to make any last-minute preparations before rendering. In our case, it's a good spot to start loading data.

I'm sure better spots exist, but loading a 12MB data file is always going to take a couple of seconds. You should try to avoid loading data you never intend to render.

`getInitialState` is the React way to set up a default state; in our case, we get an empty `rawData` array.

We're going to put d3.js's data loading magic in `loadRawData`.

We add a call to `d3.csv` because our data is in CSV format. D3.js is smart enough to change every row into an object using values from the title row (first row) as keys.

Load data with d3.csv

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

```
            }
        }.bind(this));
    },
```

This will asynchronously load a CSV file, parse it, and return the result in the `rows` argument to the callback. We then use `this.setState()` to save the data in our component.

Using `setState()` triggers a re-render and should generally be avoided because relying on state can cause inconsistencies in your UI. It's very appropriate to store 12MB of raw data as state, though.

The data we've got now has keys with spaces like `submit date` and `job title`, and everything is a string. This is not very nice to work with.

We can add some cleanup in one fell swoop:

Data cleanup

```
    loadRawData: function () {
        var dateFormat = d3.time.format("%m/%d/%Y");
        d3.csv(this.props.url)
          .row(function (d) {
            if (!d['base salary']) {
                return null;
            }

            return {employer: d.employer,
                    submit_date: dateFormat.parse(d['submit date']),
                    start_date: dateFormat.parse(d['start date']),
                    case_status: d['case status'],
                    job_title: d['job title'],
                    base_salary: Number(d['base salary']),
                    salary_to: d['salary to'] ? Number(d['salary to']) : null,
                    city: d.city,
                    state: d.state};
          }.bind(this))
          .get(function (error, rows) {
            if (error) {
                console.error(error);
                console.error(error.stack);
            }else{
                this.setState({rawData: rows});
            }
          }.bind(this));
    },
```

Using `.row()`, we've given a callback to `d3.csv` that tells it how to change every row that it reads. Each row is fed into the function as a raw object, and whatever the function returns is added to the final result.

We're changing objects that look like this:

Raw CSV rows

```
{
    "employer": "american legalnet inc",
    "submit date": "7/10/2013",
    "start date": "8/1/2013",
    "case status": "certified",
    "job title": "software",
    "base salary": "80000",
    "salary to": "",
    "city": "encino",
    "state": "ca"
}
```

Into objects that look like this:

Cleaned CSV rows

```
{
    "employer": "american legalnet inc",
    "submit_date": "2013-07-09T22:00:00.000Z",
    "start_date": "2013-07-31T22:00:00.000Z",
    "case_status": "certified",
    "job_title": "software",
    "clean_job_title": "other",
    "base_salary": 80000,
    "salary_to": null,
    "city": "encino",
    "state": "ca"
}
```

In our case we cleaned up the keys, parsed dates into `Date()` objects, and made sure numbers are numbers. If a row didn't have a `base_salary`, we filtered it out by returning `null`.

Loading and parsing 81,000 data points takes some time. Let's tell users what they're waiting for by rendering some explainer text when there's no data.

Loading indicator

```
    render: function () {
        if (!this.state.rawData.length) {
            return (
                <h2>Loading data about 81,000 H1B visas in the software industry\
</h2>
            );
        }

        return (
            <div className="row">
                <div className="col-md-12">
                    <svg width="700" height="500">

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

The beauty of our approach is that the `render` method just returns different elements based on whether or not we've got some data. Notice that we access the data as `this.state.rawData` and rely on the re-render when the state updates.

If you've kept `npm start` running in the background, you should see your browser flash the loading text before becoming blank.

Loading message

Marvelous.

Drawing the loaded data

Now that our data is loading, it's time to start drawing. We'll start with a basic histogram, then add an axis, and then add an indicator for the mean value (also known as the "average").

Create a new file in `src/` called `drawers.jsx`. We'll put all our drawing logic here.

Using d3.js for calculations

We start with a blank component called `Histogram`.

Empty histogram

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

var Histogram = React.createClass({
    render: function () {
        var translate = "translate(0, "+this.props.topMargin+")";

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

module.exports = {
    Histogram: Histogram
};
```

Once more, we required external libraries on top and started our component with a simple `render` function. On the bottom, we added it to the exports. This will allow other parts of the codebase to use it as `require('drawers.jsx').Histogram`.

In the `render` method, we created a grouping, `<g>`, element that will hold our histogram together. We used the `transform` property to move it into place. The reason we did some ugly string concatenation is that we can't mix string and variable attribute values.

However, we do want to get `topMargin` via properties given to our component. We access those with `this.props` and they make our components more flexible.

We're going to build our histogram with d3.js's built-in histogram layout. It's smart enough to do everything on its own. We just have to give it some data and some configuration.

We add three methods to our `Histogram` component to manage d3.js:

D3.js management functions

```
var Histogram = React.createClass({
    componentWillMount: function () {
        this.histogram = d3.layout.histogram();
        this.widthScale = d3.scale.linear();
        this.yScale = d3.scale.linear();

        this.update_d3(this.props);
    },

    componentWillReceiveProps: function (newProps) {
        this.update_d3(newProps);
    },

    update_d3: function (props) {

    },

    render: function () {
```

This is the cleanest approach I've found to make d3.js and React work together. We'll use it for all of our d3.js-enabled components.

The problem we're facing is that d3.js achieves its declarative nature is through objects. When you call a d3.js function, it's usually going to have some internal state.

This is great when you want to change how d3.js behaves by daisy-chaining getters and setters, but that approach causes problems with React. It means we have to make sure the internal state is always up-to-date.

We'll use `componentWillMount` to set the defaults, and then `componentWillRecieveProps` to update them when necessary.

Each time, we call `this.update_d3` to do the heavy lifting.

update_d3 function body

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

        var bars = this.histogram(props.data),
            counts = bars.map(function (d) { return d.y; });

        this.setState({bars: bars});

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

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

If you're used to d3.js, this code should look familiar. We updated the number of bins in the histogram with `.bins()` and gave it a new value accessor with `.value()`. This tells the layout how to get the interesting datapoint from our data objects.

We're doing one better by just passing in the function from our properties. This makes the `Histogram` component more reusable. The less we assume about what we're doing, the better.

Then, we called `this.histogram()` to group our data into bins and saved it to our component's state with `this.setState()`.

Finally, we updated our scales. You can think of them as simple mathematical functions that map a domain to a range. Domain tells them the extent of our data; range tells them the extent of our drawing area.

Outputting the result

Now that we've got our histogram in memory, it's time to turn it into SVG elements.

Draw the histogram bars

```
    makeBar: function (bar) {
        var percent = bar.y/this.props.data.length*100;

        var props = {percent: percent,
                     x: this.props.axisMargin,
                     y: this.yScale(bar.x),
                     width: this.widthScale(bar.y),
                     height: this.yScale(bar.dx),
                     key: "histogram-bar-"+bar.x+"-"+bar.y}

        return (
            <HistogramBar {...props} />
        );
    },

    render: function () {
        var translate = "translate(0, "+this.props.topMargin+")";

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

We added another grouping element to our `render` function, which holds all the bars. I'll show you why we need two grouping elements later.

Notice how we can just `.map()` through our histogram data in `this.state.bars`? That's the magic of React - JavaScript and XML living together as one.

We could have put the entire `makeBar` method in here, but I think the code looks cleaner when it's separate. `makeBar` takes data about a single bar, constructs the properties object, and returns a `HistogramBar` component.

We could have returned the whole lump of SVG code right here, but using a subcomponent is better aligned with React principles.

Also, notice how we can pass the entire `props` object with `{...props}`. That's an ES6 trick for passing around complex attributes. React translates it into something like `percent={percent} x={this.props.axisMargin} y={this.yScale(bar.x)}` ….

Let's add the `HistogramBar` component.

HistogramBar component

```
var HistogramBar = React.createClass({
    render: function () {
        var translate = "translate(" + this.props.x + "," + this.props.y + ")",
            label = this.props.percent.toFixed(0)+'%';

        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>
        );
    }
});
```

As you can see, nothing special is happening here. We take some properties and return a grouping element containing a rectangle and a text label.

We've made the rectangle a bit smaller than `this.props.height` to give it breathing room, and we positioned the `text` element at the end of the bar. This will make the histogram easier to read because every bar will have its percentage rendered at the top.

You'll notice that some bars are going to be very small. It's a good idea to avoid rendering the label in those cases.

Adjust label for small bars

```
var HistogramBar = React.createClass({
    render: function () {
        var 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 (
```

This is simple stuff. We add some decimal points if we're showing small numbers, and we completely remove the label when there isn't enough room.

Adding Histogram to the main component

Despite having a great `Histogram` component, our page is still blank. We have to go back to `main.jsx` and tell `H1BGraph` to render the component we've just made.

First, we have to add `drawers.jsx` to our requires.

Require drawers.jsx

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

var H1BGraph = React.createClass({
```

Then, we can add the histogram component to our `render` method.

Render the histogram component

```
    render: function () {
        if (!this.state.rawData.length) {
            return (
                <h2>Loading data about 81,000 H1B visas in the software industry\
</h2>
            );
        }

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

        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>
        );
    }
```

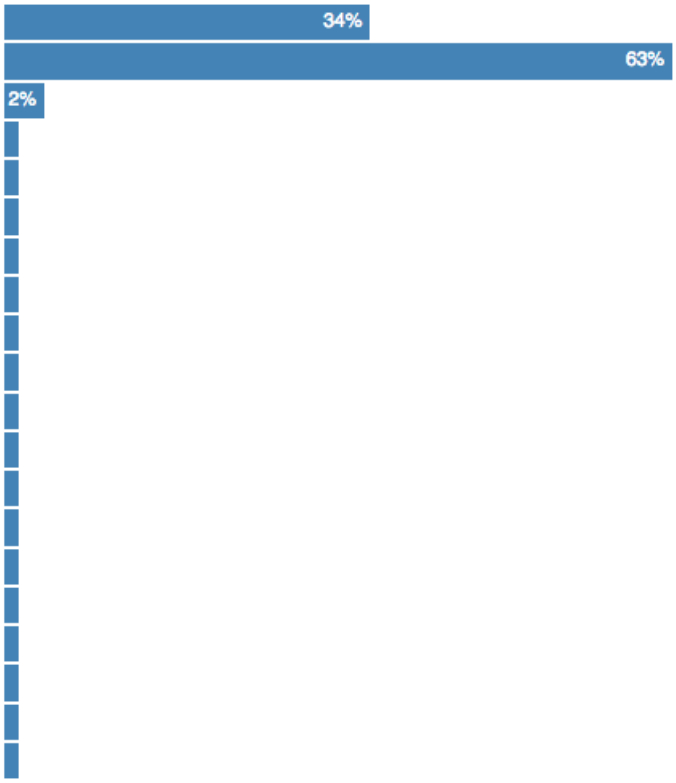
That's it. You can render as many histograms as you want just by adding `<drawers.Histogram />` to your output and setting the properties.

Let's add some styling to `src/style.less` to make the Histogram prettier:

style.less

```
@import (inline) "../bower_components/bootstrap/dist/css/bootstrap.min.css";

.histogram {
    .bar {
        rect {
            fill: steelblue;
            shape-rendering: crispEdges;
        }
        text {
            fill: #fff;
            font: 12px sans-serif;
        }
    }
}
```

You should now see a histogram like this:



Basic Histogram

Adding an axis

Axes are one of my favourite feats of d3.js magic. You call a function, set some parameters, and **BAM**, you've got an axis.

The axis automatically adapts to your data, draws the ticks it needs, and labels them. Axes are even smart enough to work with different types of scales. You don't have to do anything special to turn a linear axis into a logarithmic axis.

These are complex objects, and recreating them in React would be silly. We're going to use a dirty trick and give d3.js control of the DOM just this once.

Let's start with a blank component that's got function stubs for d3.js integration. Put it in `src/drawers.jsx`.

Axis component with function stubs

```
var Axis = React.createClass({
    componentWillMount: function () {
        this.update_d3(this.props);
    },

    componentWillReceiveProps: function (newProps) {
        this.update_d3(newProps);
    },

    update_d3: function (props) {

    },

    render: function () {
        var translate = "translate("+(this.props.axisMargin-3)+", 0)";
        return (
            <g className="axis" transform={translate}>
            </g>
        );
    }
});
```

Even though we're going to let d3.js handle DOM changes, we still have to return *something* in `render`. A grouping element that moves the resulting axis into position is the perfect candidate.

Now, let's set some axis defaults in `componentWillMount`:

Axis default properties

```
var Axis = React.createClass({
    componentWillMount: function () {
        this.yScale = d3.scale.linear();
        this.axis = d3.svg.axis()
                      .scale(this.yScale)
                      .orient("left")
                      .tickFormat(function (d) {
                          return "$"+this.yScale.tickFormat()(d);
                      }.bind(this));

        this.update_d3(this.props);
    },
```

We created a scale and an axis. The axis will use our linear scale, will be oriented to the left, and is going to prepend a dollar sign to the scale's default tick formatter.

Yes, scales have tick formatters. It's awesome.

In `update_d3`, we have to make sure things stay up-to-date:

Update axis state

```
    update_d3: function (props) {
        this.yScale
            .domain([0,
                     d3.max(props.data.map(
                         function (d) { return d.x+d.dx; }))])
            .range([0, props.height-props.topMargin-props.bottomMargin]);

        this.axis
            .ticks(props.data.length)
            .tickValues(props.data
                            .map(function (d) { return d.x; })
                            .concat(props.data[props.data.length-1].x
                                   +props.data[props.data.length-1].dx));
    },
```

Just like the previous section, we had to tell `yScale` the extent of our data and drawing area. Conversely, we didn't have to tell any of this to the axis. It already knows because we updated the scale.

We *did* have to get around a great deal of the axis's smartness. We gave it the exact number of ticks we want and the exact values they should show. This is because we wanted every bar in the histogram to have a corresponding label on the axis.

Finally, here comes the dirty trick:

The dirty trick

```
    componentDidUpdate: function () { this.renderAxis(); },
    componentDidMount: function () { this.renderAxis(); },

    renderAxis: function () {
        var node = this.getDOMNode();

        d3.select(node).call(this.axis);
    },

    render: function () {
```

I'm sure this goes against everything React designers fought for, but it works. We hook into the `componentDidUpdate` and `componentDidMount` callbacks with a `renderAxis` method. Then, we use `this.getDOMNode()` to get a reference to the rendered element, feed it into `d3.select()`, and `.call()` the axis on it.

What we're doing is basically throwing away the whole element and creating it from scratch on every render. It's rather inefficient, but it's okay when used sparingly.

Now we have to add some styling and make sure the axis is included somewhere.

The styling goes into `src/style.less`:

Styling the axis

```
.histogram {
    .bar {
        rect {
            fill: steelblue;
            shape-rendering: crispEdges;
        }
        text {
            fill: #fff;
            font: 12px sans-serif;
        }
    }
    .axis {
        path, line {
```

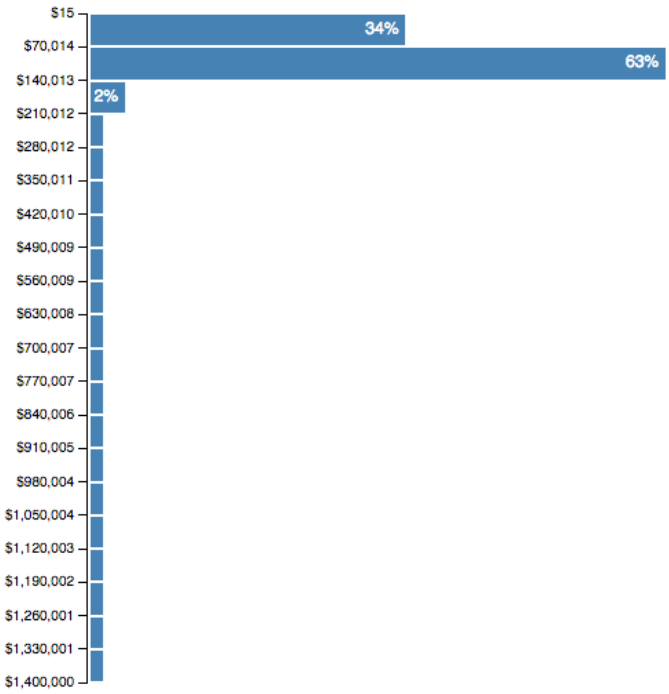
```
            fill: none;
            stroke: #000;
            shape-rendering: crispEdges;
        }
        text {
            font: 10px sans-serif;
        }
    }
}
```

And it makes to render our axis in `Histogram.render`:

Add Axis to Histogram.render

```
    render: function () {
        var translate = "translate(0, "+this.props.topMargin+")";

        return (
            <g className="histogram" transform={translate}>
                <g className="bars">
                    {this.state.bars.map(this.makeBar)}
                </g>
                <Axis {...this.props} data={this.state.bars}  />
            </g>
        );
    }
```

That's it. Your histogram should look like this:



Histogram with axis

Reacting to data changes

We've got a histogram with an axis and the building blocks to make as many as we want. Awesome!

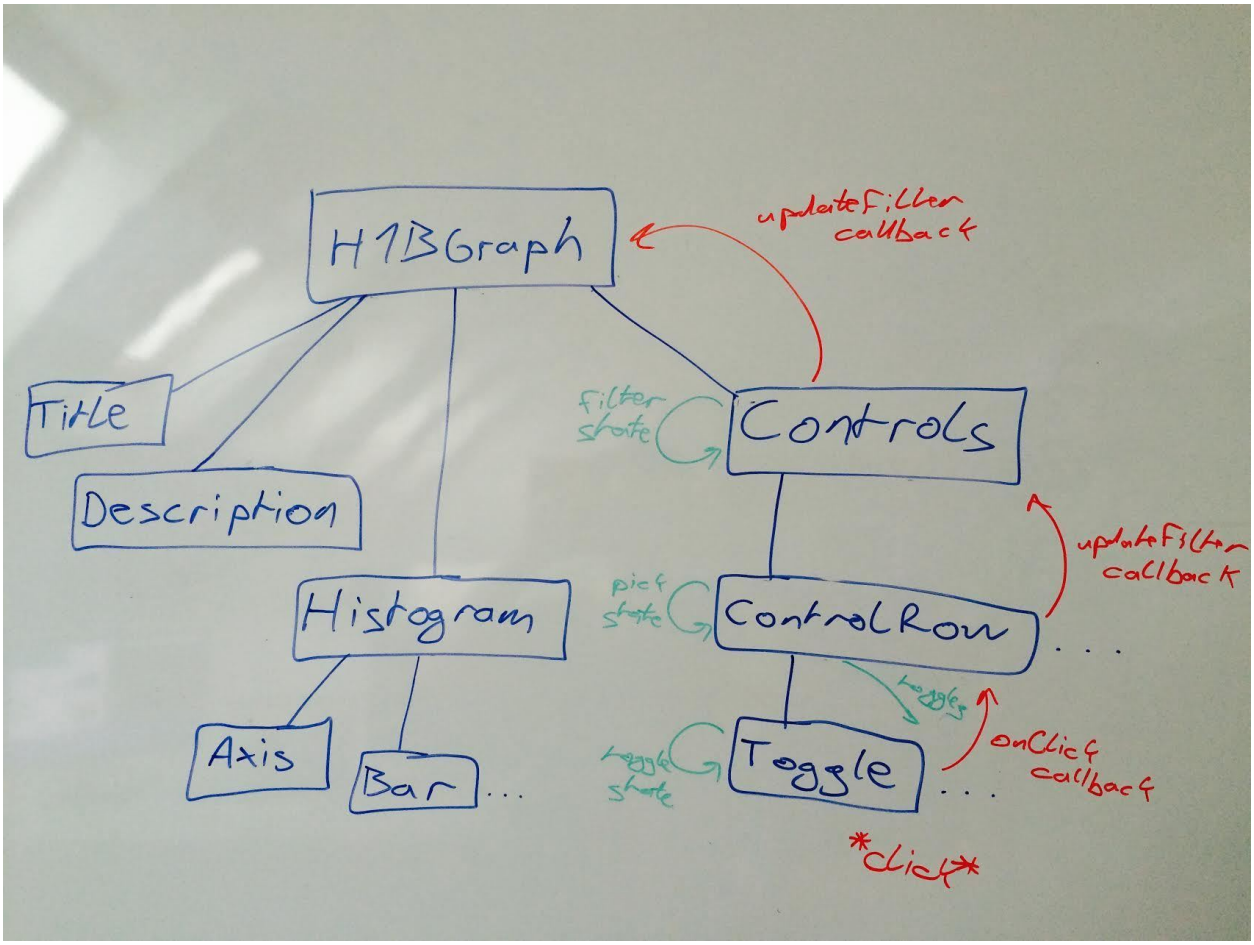
But our histogram looks weird. Most of our data falls into the first three bars and a lone outlier stretches the data range far too much.

Sure, we could take care of this by removing the outlier, but let's be honest: statisticians should worry about statistical anomalies. We're here to draw pictures.

The next best thing is letting users filter data and take a closer look. This solves our problem of finding outliers, and it gives users more freedom. And maybe, just maybe, they're going to spend more time on our site.

Win-win-win.

We're going to make controls that let users filter data by year. In the full example, I added filtering by state and job title as well, but they follow the same principles and talking about them here would take too long.

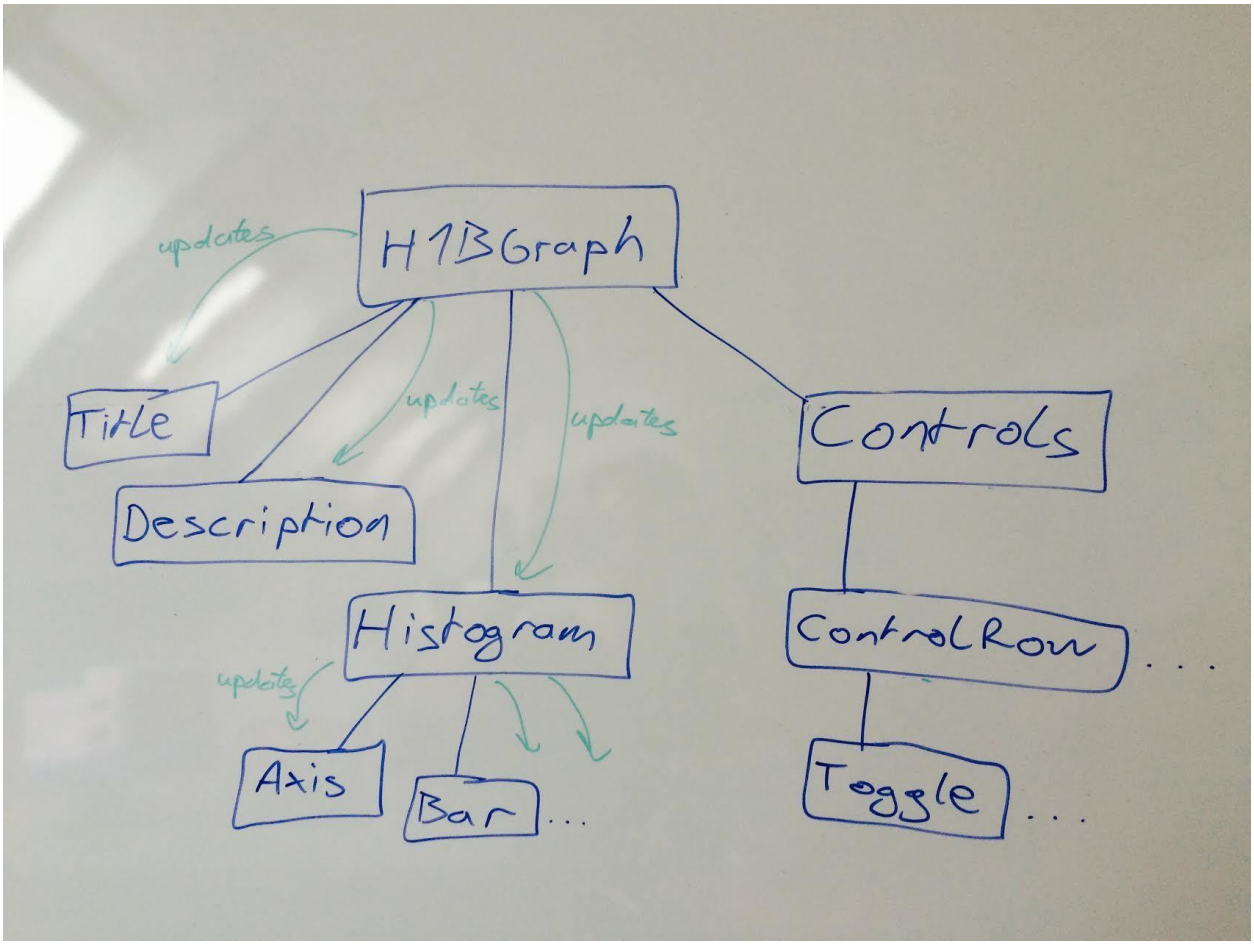


Events flow up through callbacks

Our approach will follow the approach of having events flow up the hierarchy and having data updates flow down. When a user clicks a button, it calls a function on its parent. This in turn calls a function on its parent and so on until we hit `H1BGraph`, which is our holder of truth.

We could also call functions directly on `H1BGraph` if we had a reference to them, but that would break the separation of concerns in our case.

Once `H1BGraph` updates its state, the changed data flows back down to the relevant components via `props`.



Updated data flows back down

Creating user controls

We're going to organise our controls into three components:

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

Let's start a new file called `src/controls.jsx` with a `Controls` component stub:

Controls component stub

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

var Controls = React.createClass({
    render: function () {

        return (
            <div>

            </div>
        )
    }
});

module.exports = Controls;
```

We're going to need React and Lodash in this file, and Controls is the only component we'll expose to the outside.

Let's add all of this to the main `H1BGraph` component. It's just three lines inside `render`:

Add Controls to H1BGraph.render

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

We had to wrap everything in another `<div>` because React throws a hissy fit if we try to return multiple elements side-by-side. We used props to give the `Controls` component some data and the filter update callback.

Don't forget to add `Controls = require('./controls.jsx')` somewhere near the top of the file.

You should have some empty components showing up on the screen. Let's add the row of buttons to the `Controls.render` method:

Render a ControlRow

```
var Controls = React.createClass({
    render: function () {
        var getYears = function (data) {
            return _.keys(_.groupBy(data,
                                    function (d) {
                                        return d.submit_date.getFullYear()
                                    }))
                    .map(Number);
        };

        return (
            <div>
                <ControlRow data={this.props.data}
                            getToggleNames={getYears}
                            updateDataFilter={this.updateYearFilter} />
            </div>
        )
    }
});
```

This will break in your browser because we don't have a `ControlRow` component yet. The idea is to have a component that takes data, a function that generates the button labels, and an `updateDataFilter` callback.

We'll define the callback later. Let's make the `ControlRow` component first.

ControlRow component

```
var ControlRow = React.createClass({
    render: function () {
        return (
            <div className="row">
                <div className="col-md-12">
            {this.props.getToggleNames(this.props.data).map(function (name) {
                var key = "toggle-"+name,
                    label = name;
```

```
                return (
                    <Toggle label={label}
                            name={name}
                            key={key}
                            value={this.state.toggleValues[name]}
                            onClick={this.makePick} />
                );
            }.bind(this))}
                </div>
            </div>
        );
    }
});
```

There's nothing special going on here. We've got a `render` method that returns a Bootstrap row and fills it with a bunch of `Toggle` components. It does so by looping over the result of `this.props.getToggleNames`, which is the function we defined in `Controls.render`.

Normally, `onClick` is a click event handler. React gives us this sort of magic prop for every valid user event. It's the same as jQuery's `$('foo').on('click')`.

They only work on HTML elements, not React components, so this one is just a function callback that we'll use manually.

We had to define the `key` property to help React identify specific elements. I assume it uses these to identify which elements to re-render if they're created in a loop.

We've also got a `value` property that comes out of `this.state.toggleValues`. It tells our toggles whether to render as turned on or off. We track this state in `ControlRow` instead of just the `Toggle` itself, so we can make sure only one turns on at the time.

Let's add the state handling.

Exclusive toggling

```
var ControlRow = React.createClass({
    makePick: function (picked, newState) {
        var toggleValues = this.state.toggleValues;

        toggleValues = _.mapValues(toggleValues,
                           function (value, key) {
                               return newState && key == picked;
                           });

        this.setState({toggleValues: toggleValues});
    },
```

```
    getInitialState: function () {
        var toggles = this.props.getToggleNames(this.props.data),
            toggleValues = _.zipObject(toggles,
                                       toggles.map(function () { return false; }));

        return {toggleValues: toggleValues};
    },

    render: function () {
        return (
```

The `makePick` function is called when the user clicks on a `Toggle` element. It goes through the `toggleValues` dictionary and sets them to `false` if they aren't the one the user clicked on. The one they did click is set to `true`.

Then it saves the dictionary with `this.setState`, which triggers a re-render and updates all the buttons.

We used the trusty `getInitialState` to create the `toggleValues` dictionary with everything set to `false`.

Now we need the `Toggle` component.

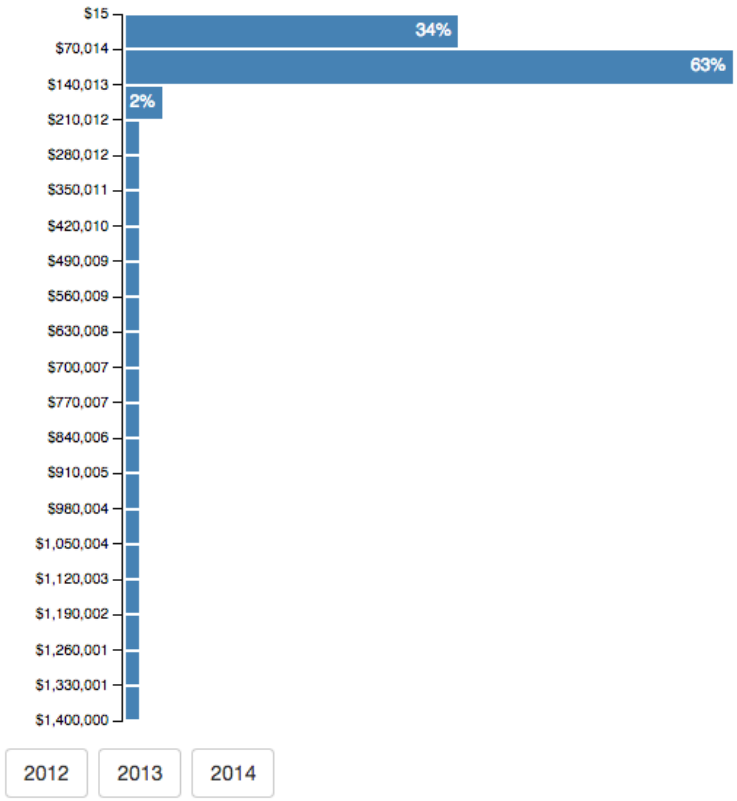
Toggle component

```
var Toggle = React.createClass({
    getInitialState: function () {
        return {value: false};
    },

    componentWillReceiveProps: function (newProps) {
        this.setState({value: newProps.value});
    },

    render: function () {
        var className = "btn btn-default";

        if (this.state.value) {
            className += " btn-primary";
        }

        return (
            <button className={className} onClick={this.handleClick}>
```

```
                {this.props.label}
            </button>
        );
    }
});
```

Toggle renders a button. If `this.state.value` is `true`, then the button is highlighted. We set it to off by default and update it when `componentWillReceiveProps` triggers. That happens when the component's properties change from above.

Now your page shows three buttons under the histogram:



Histogram with buttons

But clicking on a button will throw an error.

Propagating events up the hierarchy

To make the buttons work, we have to implement the click event handler - `this.handleClick`. Then we have to call every callback up the chain until we get to the histogram component, which is our holder of truth.

Let's start by adding a click handler to the `Toggle` component:

Toggle click handler

```
var Toggle = React.createClass({
    getInitialState: function () {
        return {value: false};
    },
    handleClick: function (event) {
       var newState = !this.state.value;
       this.setState({value: newState});
       this.props.onClick(this.props.name, newState);
    },

    componentWillReceiveProps: function (newProps) {
```

It's a simple function. All it cares about is toggling the `value` state and calling the `this.props.onClick` callback.

The reason we're using state to highlight the button is that making this change optimistically instead of waiting for propagation looks better to the user. It's a tiny bit more instant, which makes our app feel faster.

We still override the `value` in `componentWillReceiveProps`, if the parent component wants us to.

If you click around now, the buttons are going to toggle on and off. `ControlRow` will continue making sure you can only choose one at a time.

2013 toggled on

To make these changes propagate further than `ControlRow`, we have to call `ControlRow`'s callback in the `makePick` method:

Callback in ControlRow.makePick

```
var ControlRow = React.createClass({
    makePick: function (picked, newState) {
        var toggleValues = this.state.toggleValues;

        toggleValues = _.mapValues(toggleValues,
                           function (value, key) {
                               return newState && key == picked;
                           });
```

```
        // if newState is false, we want to reset
        this.props.updateDataFilter(picked, !newState);

        this.setState({toggleValues: toggleValues});
    },
```

Nothing fancy, just a call to `this.props.updateDataFilter` with information on what was picked and whether we want to reset the filter or not. The reset is used when the user clicks on a button that's already turned on.

Now comes the fun part. We get to construct the data filter in our `Controls` component, then tell the main `H1BGraph` component.

We gave `this.updateYearFilter` as the update callback to `ControlRow`. It's a simple function that creates a filter function based on the chosen year and saves it with `setState()` like this:

Controls.updateYearFilter function

```
var Controls = React.createClass({
    updateYearFilter: function (year, reset) {
        var filter = function (d) {
            return d.submit_date.getFullYear() == year;
        };

        if (reset || !year) {
            filter = function () { return true; };
        }

        this.setState({yearFilter: filter});
    },

    getInitialState: function () {
        return {yearFilter: function () { return true; }};
    },

    render: function () {
```

We also added the `getInitialState` function to make sure the filter is set to an "accept everything" version by default.

Our next step is to tell the parent component about our new filter.

Propagating filter up the chain

```
    getInitialState: function () {
        return {yearFilter: function () { return true; }};
    },

    componentDidUpdate: function () {
        this.props.updateDataFilter(
            (function (filters) {
                return function (d) {
                    return filters.yearFilter(d)
                };
            })(this.state)
        );
    },

    shouldComponentUpdate: function (nextProps, nextState) {
        return !_.isEqual(this.state, nextState);
    },

    render: function () {
```

React calls `componentDidUpdate` when we change the state. This gives us a great opportunity to call our filter callback - `this.props.updateDataFilter`. We feed it a new filter function that combines all of the potential filters in our state. In our trimmed-down example, our only filter is `yearFilter`.

You'd think we could do all of this straight in the `updateYearFilter` function, but we don't have access to the updated state yet. Because we might add more types of filters later, it's better to rely on the already saved filters.

Ok, we've just created an infinite loop. When `componentDidUpdate` is called we call something on the parent component. This potentially changes our `props`, which again triggers `componentDidUpdate`.

We have to make sure the component updates *only* when properties have actually changed. The easiest way to do that is using `shouldComponentUpdate`. If it returns `false`, the component doesn't update.

Anyway, it's time to add the filtering functionality to `H1BGraph`.

Filter update callback in H1BGraph

```
    getInitialState: function () {
        return {rawData: [],
                dataFilter: function () { return true; }};
    },

    updateDataFilter: function (filter) {
        this.setState({dataFilter: filter});
    },

    render: function () {
```

Nothing fancy. We initially set the `dataFilter` to accept everything and update it when something calls `updateDataFilter` with a new filter function.

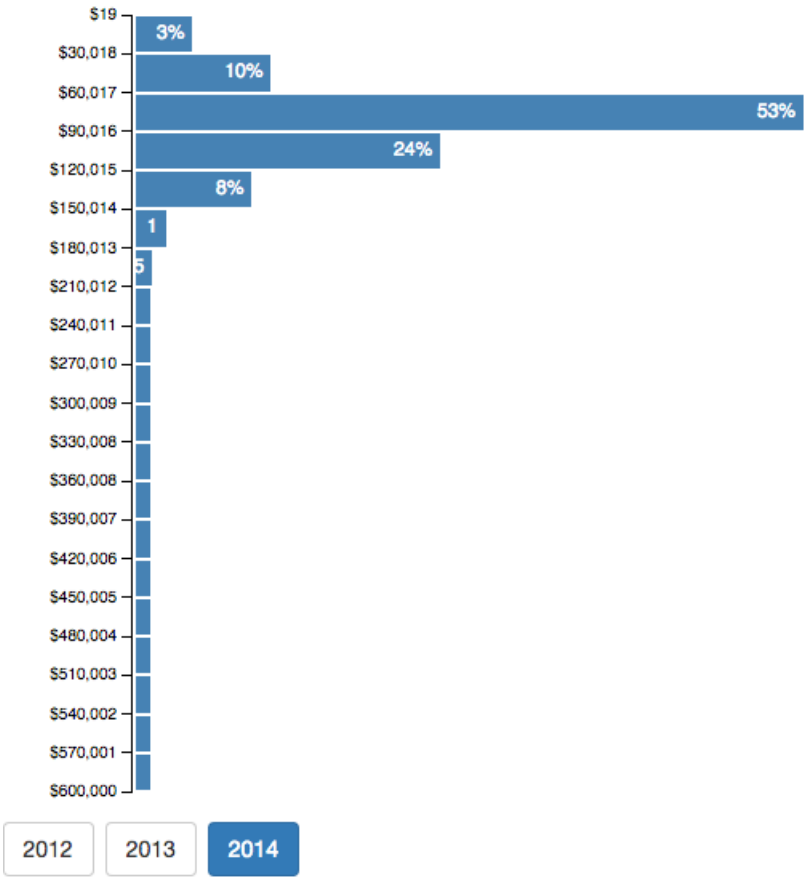
The last step is filtering the data in our `H1BGraph.render` method.

Filter data in H1BGraph.render

```
        var filteredData = this.state.rawData.filter(this.state.dataFilter);

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

As you can see, we didn't have to do much. We passed the `rawData` through our filter before feeding it to the `Histogram`.

You should now see a histogram like this:



Histogram with only 2014 salaries

Hooray! We have a histogram with user filtering.

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.

Adding some meta data

If you've followed along this far, you've got a nice histogram of H1B salaries in the software industry. It renders fully on the front-end and changes when the user picks a specific year. You've even got a couple of building blocks to make it better.

But a good visualization needs a title and a description. The title tells users what they're looking at, and the description gives them the story.

As great as people are at understanding pictures, it goes much better when you flat-out *tell* them what they're looking at.

We start with a stubbed out `src/meta.jsx` file:

Stubbed meta.jsx file

```
var React = require('react'),
    d3 = require('d3'),
    _ = require('lodash'),
    States = require('./states.js');

var MetaMixin = {
};

var Title = React.createClass({
    mixins: [MetaMixin],

    render: function () {
        return null;
    }
});

var Description = React.createClass({
    mixins: [MetaMixin],

    render: function () {
        return null;
    }
});

module.exports = {
    Title: Title,
    Description: Description
}
```

We're going to make a mixin called `MetaMixin` because both the `Title` and `Description` components are going to use some of the same functions. It will have a function to get a list of years out of the data, a function to give us a label formatter, and a function to give us data filtered by a specific year.

Let's add them.

MetaMixin functions

```
var MetaMixin = {
    getYears: function (data) {
        data || (data = this.props.data);

        return _.keys(
            _.groupBy(this.props.data,
                      function (d) { return d.submit_date.getFullYear(); })
        );
    },

    getFormatter: function (data) {
        data || (data = this.props.data);

        return d3.scale.linear()
                 .domain(d3.extent(this.props.data,
                                   function (d) { return d.base_salary; }))
                 .tickFormat();
    }

    getAllDataByYear: function (year, data) {
        data || (data = this.props.allData);

        return data.filter(function (d) {
            return d.submit_date.getFullYear() == year;
        });
    }
};
```

There's nothing special going on here. `getYears` uses `_.groupBy` to group the data by year, then returns the keys of the array. We could make this more efficient, but it works well enough and is easy to read.

`getFormatter` relies on the fact that d3.js scales have formatters. It creates a linear scale with a domain and returns the `tickFormat()` function. Formatters don't work well without a defined domain.

`getAllDataByYear` is a simple filter function. We'll use it to access data for specific years when making the description.

To keep our meta data code clean, we've also got a file that maps state abbreviations to names. It's called `src/states.jsx` and exports an object that looks like this:

module.exports = { "AL": "Alabama", "AK": "Alaska", // ...

You can build your own, or you can get it from Github[14].

Let's start with the title.

The title

Our titles are going to follow a formula like: "H1B workers in the software industry made $x/year in <year>".

We start with the year fragment:

getYearsFragment function

```
var Title = React.createClass({
    mixins: [MetaMixin],

    getYearsFragment: function () {
        var years = this.getYears(),
            fragment;

        if (years.length > 1) {
            fragment = "";
        }else{
            fragment = "in "+years[0];
        }

        return fragment;
    },
```

We get the list of years in the current data and return either an empty string or `in Year`. If there's only one year in the data, we assume it's been filtered by year.

The `render` function is going to be simple as well:

[14] https://github.com/Swizec/h1b-software-salaries/blob/master/src/states.js

Title.render function

```
    render: function () {
        return null

        var mean = d3.mean(this.props.data,
                           function (d) { return d.base_salary; }),
            format = this.getFormatter();

        var
            yearsFragment = this.getYearsFragment(),
            title;

        return (
            <h2>H1B workers in the software industry {yearsFragment.length ? "ma\
de" : "make"} ${format(mean)}/year {yearsFragment}</h2>
        );
    }
});
```

We're returning an `<h2>` element with a title that includes some dynamic tags. We change `made` to `make` based on whether or not we're adding `in years`, and we used `d3.mean` to get the average.

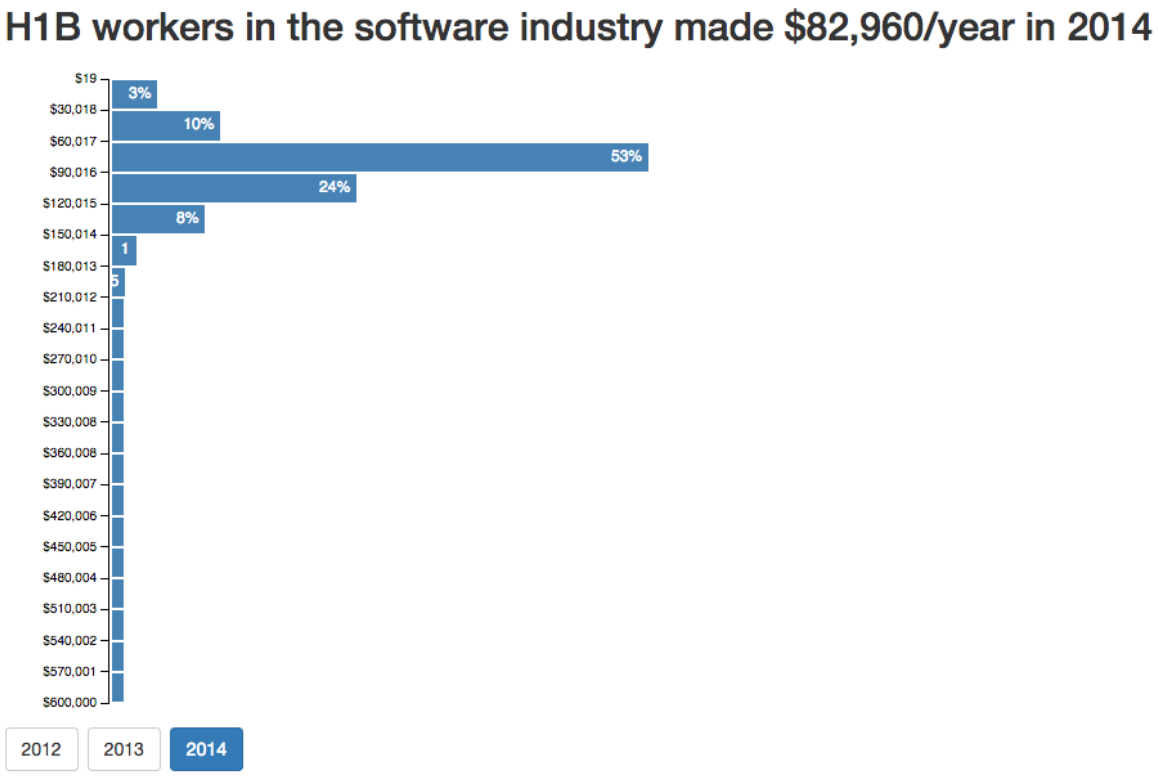
Now we can add the title into H1BGraph's render method. You can use as many as you want, wherever you want, but I think putting it above the graph works best.

Add title to H1BGraph.render

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

Don't forget to add a `meta = require('./meta.jsx');` at the top of the `src/main.jsx`.

Your histogram should now look like this:



Histogram with title

Now let's add a paragraph of description to make the story clearer.

The description

Our descriptions won't be much more complicated than the titles. We'll need a sentence or two explaining what the histogram is showing and comparing it to the previous year.

First, we add the `getYearFragment` function:

Description.getYearFragment function

```
var Description = React.createClass({
    mixins: [MetaMixin],

    getYearFragment: function () {
        var years = this.getYears(),
            fragment;

        if (years.length > 1) {
            fragment = "";
        }else{
            fragment = "In "+years[0];
        }

        return fragment;
    },
```

It's the same as in `Title`, but with a capital first letter. Yes, this might fit better in `MetaMixin`.

But the `getPreviousYearFragment` function is more complex:

Description.getPreviousYearFragment

```
getPreviousYearFragment: function () {
    var years = this.getYears().map(Number),
        fragment;

    if (years.length > 1) {
        fragment = "";
    }else if (years[0] == 2012) {
        fragment = "";
    }else{
        var year = years[0],
            lastYear = this.getAllDataByYear(year-1),
            percent = ((1-lastYear.length/this.props.data.length)*100).toFixed();

        fragment = ", "+Math.abs(percent)+"% "+(percent > 0 ? "more" : "less")+"\
 than the year before";
    }

    return fragment;
},
```

First, we get the list of years. If it's empty or the current year is 2012 (the first in our dataset), we return an empty fragment.

In any other case, we get a look at last year's data with `this.getAllDataByYear()`. We can then compare the number of entries with this year and return a fragment that's either "X% **more** than the year before" or "X% **less** than the year before".

Now we've just got to use these two fragments in the render method.

Description.render function

```
    render: function () {
        return null;
        var formatter = this.getFormatter(),
            mean = d3.mean(this.props.data,
                           function (d) { return d.base_salary; }),
            deviation = d3.deviation(this.props.data,
                                     function (d) { return d.base_salary; });

        var yearFragment = this.getYearFragment(),


        return (
            <p className="lead">{yearFragment.length ? yearFragment : "Since 201\
2"} the US software industry {yearFragment.length ? "gave" : "has given"} jobs t\
o {formatter(this.props.data.length)} foreign nationals{this.getPreviousYearFrag\
ment()}. Most of them made between ${formatter(mean-deviation)} and ${formatter(\
mean+deviation)} per year.</p>
        );
    }
```

We've crammed a bunch into that return statement, but it's not as hairy as it looks. We start the sentence with either "Since 2012" or "In Year", then decide what tense to use for the verb "give", followed by the count of H1Bs in the current data and "foreign nationals".

Right after that, with no space, we add the previous year fragment. Then we add a sentence describing the range that the majority of salaries fall into. To get the numbers, we used d3.js's built-in standard deviation calculator and assumed the distribution is roughly normal. That means 68.2% of our data falls within a standard deviation to each side of the mean.

Yes, my statistics professor would be very angry, but it's a good-enough approximation.

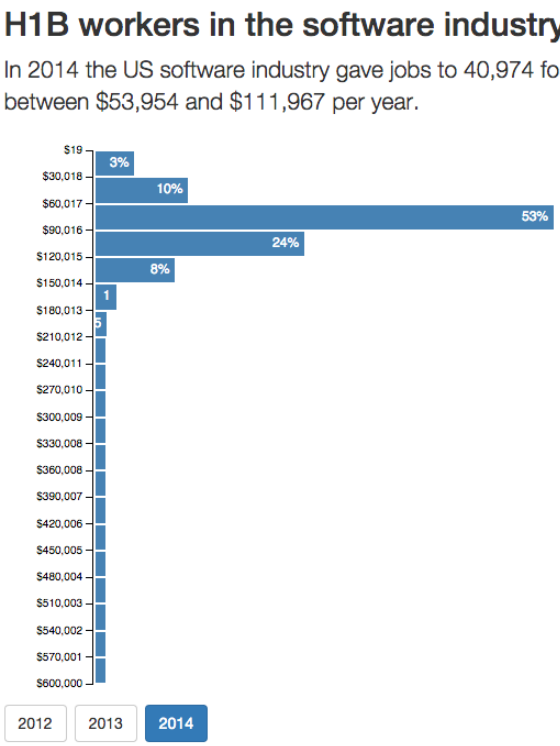
Let's put the description next to our title.

Add description to H1BGraph.render

```
        return (
            <div>
                <meta.Title data={filteredData} />
                <meta.Description data={filteredData} allData={this.state.rawDat\
a} />
                <div className="row">
                    <div className="col-md-12">
                        <svg width={fullWidth} height={params.height}>
                            <drawers.Histogram {...params} data={filteredData} />
                        </svg>
                    </div>
                </div>
                <Controls data={this.state.rawData} updateDataFilter={this.updat\
eDataFilter} />
            </div>
        );
```

Unlike the title, our description needs the full data just like the controls do. Without this, it wouldn't be able to get data for previous years.

Your histogram should now have a nice description:



Histogram with description

Hooray, you've made a histogram of H1B salaries with a title and description that both change when the visualization changes. You're awesome!

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

Conclusion

If you've followed the examples, you've just created your first visualization with React and d3.js. You're amazing!

If you've only read through the book, that's awesome, too. You know much more about making reusable visualizations than you did an hour ago.

You understand how the declarative nature of React works really well with d3.js and vice-versa. You know what it takes to write reusable components and how much easier they make your life once you've built them. And you know how to configure Webpack, Babel, and everything else to create an awesome environment to work with.

In short, you know everything you need to start making your own visualization components. At the very least, you know whether React and d3.js are a good tech stack for you.

And that's awesome!

If you have any questions, just poke me on twitter. I'm @Swizec[15]. Or send me an email at swizec@swizec.com.

[15] https://twitter.com/Swizec

Appendix

When I first wrote this book in Spring of 2015, I came up with a build and run system based on Grunt and Browserify. I also suggested using Bower for client-side dependencies.

I now consider that to have been a mistake, and Webpack to be a much better option. I also suggest using one of the numerous boilerplate projects to get started quickly.

I'm leaving the old chapter here as a curiosity, and to help those stuck in legacy systems. With a bit of tweaking, you *can* use Grunt with Webpack, and Webpack *can* support Bower as a package manager.

NPM for server-side tools

NPM is node.js's default package manager. Originally developed as a dependency management tool for node.js projects, it's since taken hold of the JavaScript world as a way to manage the tool belt.

We'll use NPM to install the other tools we need.

You can get it by installing node.js from nodejs.org[16]. Grunt, Bower, and our development server will run in node as well.

Once you've got it, create a working directory, navigate to it, and run:

```
$ npm init .
```

This will ask you a few questions and create `package.json` file. It contains some meta data about the project and more importantly, the list of dependencies. This is useful when you return to a project months or years later, and can't remember how to get it running.

Also great, if you want to share the code with others.

And remember, the stub project included with the book already has all this set up.

The development server

Production servers are beyond the scope of this book, but we do need a server running locally. You could work on a static website without one, but we're loading data into the visualization dynamically and that makes browser security models panic.

[16] http://nodejs.org

We're going to use `live-server`, which is a great static server written in JavaScript. Its biggest advantage is that the page refreshes automatically when CSS, HTML, or JavaScript files in the current directory change.

To install `live-server`, run:

```
$ npm install -g live-server
```

If all went well, you should be able to start a server by running `live-server` in the command line. It's even going to open a browser tab pointing at `http://localhost:8080` for you.

Compiling our code with Grunt

Strictly speaking, we're writing JavaScript and some CSS. We don't *really* have to compile our code, but it's easier to work with, if we do.

Our compilation process is going to do three things:

- compile Less to CSS
- compile JSX to pure JavaScript
- concatenate source files

We have to compile Less because browsers don't support it natively. We're not going to use it for anything super fancy, but I prefer having some extra power in my stylesheets. Makes them easier to write.

You can use whatever you want for styling, even plain CSS, but the samples in this book will assume you're using Less.

Compiling JSX is far more important.

JSX is React's new file format that lets us embed HTML snippets straight in our JavaScript code. You'll often see render methods doing something like this:

A basic Render

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

See, we're treating HTML - in this case an H1BGraph component - just like a normal part of our code. I haven't decided yet if this is cleaner than other templating approaches like Mustache, but it's definitely much better than manually concatenating strings.

As you'll see later, it's also very powerful.

But browsers don't support this format, so we have to compile it into pure JavaScript. The above code ends up looking like this:

JSX compile result

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

We could avoid this compilation step by using `JSXTransform`. It can compile JSX to JavaScript in the browser, but makes our site slower. React will also throw a warning and ask you never to use `JSXTransform` in production.

Finally, we concatenate all our code into a single file because that makes it quicker to download. Instead of starting a gazillion requests for each and every file, the client only makes a single request.

Install Grunt

We're going to power all of this with Grunt[17], which lets us write glorified bash scripts in JavaScript. Its main benefits are a large community that's created plugins for every imaginable thing, and simple JavaScript-based configuration.

To install Grunt and the plugins we need, run:

```
$ npm install -g grunt-cli
$ npm install --save-dev grunt
$ npm install --save-dev grunt-browserify
$ npm install --save-dev grunt-contrib-less
$ npm install --save-dev grunt-contrib-watch
$ npm install --save-dev jit-grunt
$ npm install --save-dev reactify
```

Browserify[18] will allow us to write our code in modules that we can use with `require('foo.js')`. Just like we would in node.js. It's also going to concatenate the resulting module hierarchy into a single file.

Some test readers have suggested using Webpack[19] instead, but I haven't tried it yet. Apparently it's the best thing since bacon because it can even `require()` images.

Reactify[20] will take care of making our JSX files work with Browserify.

Less[21] will compile Less files to CSS, `watch` will automatically run our tasks when files change, and `jit-grunt` loads Grunt plugins automatically so we don't have to deal with that.

[17] http://gruntjs.com
[18] http://browserify.org
[19] http://webpack.github.io/
[20] https://github.com/andreypopp/reactify
[21] https://github.com/gruntjs/grunt-contrib-less

Grunt Config

Now that our tools are installed, we need to configure Grunt in `Gruntfile.js`. If you're starting with the stub project, you've already got this.

We'll define three tasks:

- `less`, for compiling stylesheets
- `browserify`, for compiling JSX files
- `watch`, for making sure Grunt keeps running in the background

The basic file with no configs should look like this:

Base Gruntconfig.js

```
module.exports = function (grunt) {
    require('jit-grunt')(grunt);

    grunt.initConfig({ /* ... */ });

    grunt.registerTask('default',
                       ['less', 'browserify:dev', 'watch']);
);
```

We add the three tasks inside `initConfig`:

Less task config

```
        less: {
            development: {
                options: {
                    compress: true,
                    yuicompress: true,
                    optimization: 2
                },
                files: {
                    "build/style.css": "src/style.less"
                }
            }
        },
```

This sets a couple of options for the less compiler and tells it which file we're interested in.

Browserify task config

```
        browserify: {
            options: {
                transform: ['reactify', 'debowerify']
            },
            dev: {
                options: {
                    debug: true
                },
                src: 'src/main.jsx',
                dest: 'build/bundle.js'
            },
            production: {
                options: {
                    debug: false
                },
                src: '<%= browserify.dev.src %>',
                dest: 'build/bundle.js'
            }
        },
```

The `reactify` transform is going to transform JSX files into plain JavaScript. The rest just tells `browserify` what our main file is going to be and where to put the compiled result.

I'm going to explain `debowerify` when we talk about client-side package management in the next section.

Watch task config

```
        watch: {
            styles: {
                files: ['src/*.less'],
                tasks: ['less'],
                options: {
                    nospawn: true
                }
            },

            browserify: {
                files: 'src/*.jsx',
                tasks: ['browserify:dev']
            }
        }
```

This tells `watch`, which files it needs to watch for changes, and what to do with them.

You should now be able to start compiling your code by running `grunt` in the command line. If you didn't start with the stub project, it will complain about missing files. Just create empty files with the names it complains about.

Managing client-side dependencies with Bower

Client-side dependency management is the final piece in the puzzle.

Traditionally this is done by dumping all JavaScript plugins into some sort of `vendor/` directory. Or having a `plugins.js` file and manually copy-pasting code in there.

That approach works fine right until the day you want to update one of the plugins. Then you can't remember exactly which of the ten plugins with a similar name and purpose you used, or you can no longer find the Github repository.

It's even worse if the plugin's got some dependencies that also need to be updated. Then you're in for a ride.

This is where Bower comes in. Instead of worrying about any of that, you can just run:

```
$ bower install <something>
```

You could use NPM for this, but Bower can play with any source anywhere. It understands several package repositories, and can even download code straight from Github.

To begin using Bower, install it and init the project:

```
$ npm install -g bower
$ bower init
```

This will create a `bower.json` file with some basic configuration.

When that's done, install the four dependencies we need:

```
$ bower install -S d3
$ bower install -S react
$ bower install -S bootstrap
$ bower install -S lodash
```

We're going to rely heavily on d3 and React. Bootstrap is there to give us some basic styling, and lodash will make it easier to play around with the data.

All of these were installed in the `bower_components/` directory.

Which is awesome, but creates a small problem. If you want to use Browserify to include d3, you have to write something like `require('../bower_components/d3/d3.js');`, which not only looks ugly, but means you have to understand the internal structure of every package.

We can solve this with `debowerify`, which knows how to translate `require()` statements into their full path within `bower_components/`.

You should install it with:

```
$ npm install --save-dev debowerify
```

We already configured Debowerify in the Grunt config section under Browserify. Now we'll be able to include d3.js with just `require('d3');`. Much better.

Final check

Congratulations! You should now have a sane work environment.

Running `grunt` will compile your code and keep it compiling. Running `live-server` will start a static file server that auto-updates every time some code changes.

Check that your work directory has at least these files:

- package.json
- Gruntfile.js
- bower.json
- node_modules/
- bower_components/
- src/

I'd suggest adding a `.gitignore` as well. Something like this:

.gitignore

```
bower_components
build/.*
node_modules
```

And you might want to set up your text editor to understand JSX files. I'm using Emacs and `web-mode` is perfect for this type of work.

If `grunt` complains about missing files, that's normal. We're going to create them in the next section. But if it's bugging you too much, just create them as empty files.

You can also refer to the stub project included with the book if something went wrong. If that doesn't help, Google is your friend. You can also poke me on Twitter (@Swizec) or send me an email at swizec@swizec.com.