Swizec Teller

React + d3js ES6

Reusable dataviz & games
using modern JavaScript

React+d3js ES6

Reusable dataviz & games using modern JavaScript

Swizec Teller

Tweet This Book!

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

The suggested tweet for this book is:

Gonna build me some cool stuff with #reactd3jsES6

The suggested hashtag for this book is #reactd3jsES6.

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=#reactd3jsES6

Contents

Introduction

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 article.

You see, the issue with online articles is that they 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 memes and funny tweets, never to come back.

This hurts my precious writer's ego, but more importantly, it sucks for you. If you're not reading, you're not learning.

So here we both are.

I'm counting on you to take some time away from the internet, to 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 have invested in an article. 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*, but playing with the examples might take some time, too.

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].

[1] https://www.packtpub.com/web-development/data-visualization-d3js

React is a new kid on the block, so I'm going to assume you're not quite as familiar with it. 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.

The examples in React+d3.js ES6 edition are written in ES6 - ECMAScript2015. Familiarity with the new syntax and language constructs will help, but I'm going to explain everything new that we use.

How to read this book

Relax. Breathe. You're here to learn. I'm here to teach. I promise Twitter and Facebook will still be there when we're done.

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.

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';~~
```

Each code sample starts with a commented out file path. That's the file you're editing. Like this:

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

```
// ./src/App.jsx

class App …
```

Commands that you should run in the terminal start with an `$` symbol, like this:

```
$ npm start
```

ES5 and ES6 versions

You are reading React+d3.js ES6 edition. Functionally, it's almost the same as the ES5 version, but it has a better file structure and some additional content.

You can read either version depending on which version of JavaScript you're more comfortable with. Keep in mind, the ES5 version isn't getting further updates.

Both will teach you how to effectively use React and d3.js to make reusable data visualization components.

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 knows 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 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 bar chart"*. 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.

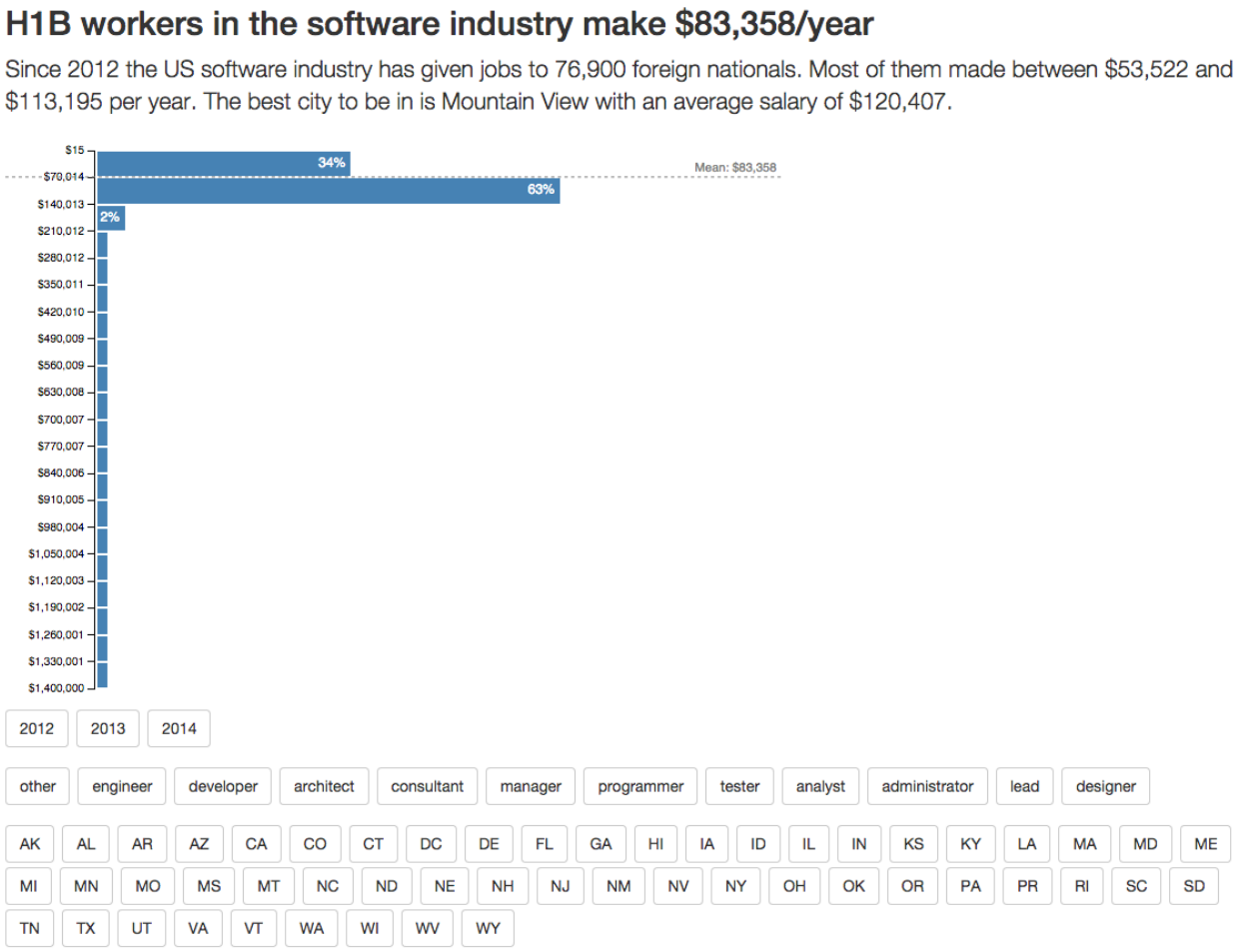
This is where React comes in. For instance, once you've created a histogram component, you can always get a histogram with `<Histogram {...params} />`.

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

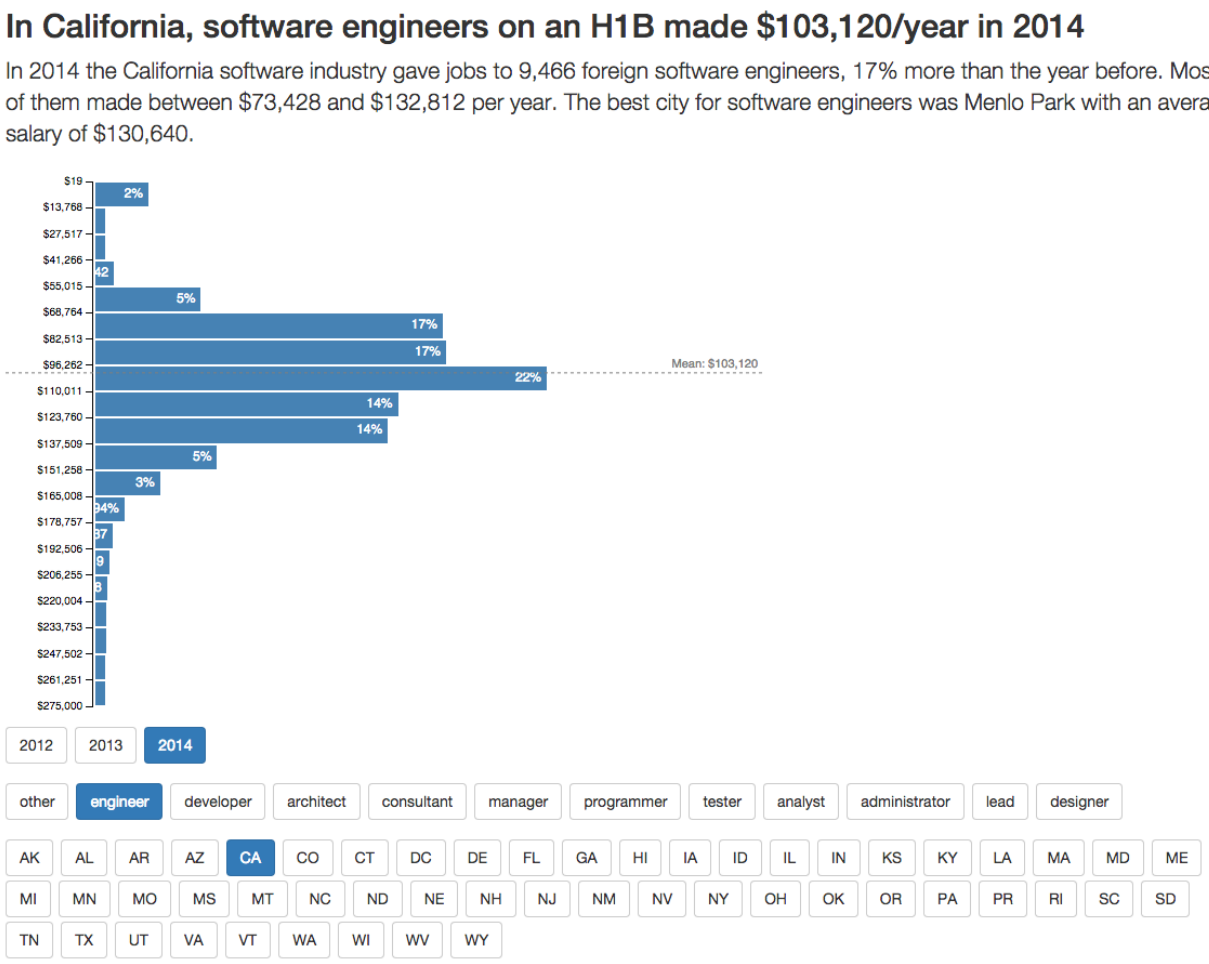
It gets even better. With React, you can make various graph and chart components build off the same data. This means that when your data changes, the whole visualization reacts.

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.

Buzzword soup explained

We're going to use a lot of buzzwords in this book. Hell, we've used some already. Most will get a thorough explanation further on, but let me give you a quick rundown.

- **Babel**, a JavaScript transpiler that makes your code work on all browsers.
- **ECMAScript2016**, official name for the part of ES7 we're getting in 2016
- **ES5**, any JavaScript features that existed before June 2015
- **ES6/ECMAScript2015**, any JavaScript features released as part of the new standard in June 2015 (think => and stuff)
- **ES7**, the standard we were promised for 2016, but also a catch-all for future JavaScript features
- **fat arrows**, a new way to define functions in ES6 (=>)

- **Git**, a version control system. It's pretty popular, you probably know it :)
- **H1B**, a popular type of work visa in the United States
- **JSX**, a language/syntax that lets you use `<a href="/example">` as a native part of JavaScript
- **Mustache**, a popular way to write HTML templates for JavaScript code. Uses `{{ ... }}` syntax.
- **npm**, most popular package manager for JavaScript libraries
- **props**, component properties set when rendering
- **state**, a local dictionary of values available in most components
- **Webpack**, a module packager for JavaScript. Makes it more convenient to organize code into multiple files. Provides cool plugins.

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 think 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

```
ReactDOM.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 something similar. 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 that came with your book.

A good work environment

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

If you don't know, and don't care about this right now: Use the starter project that came with the book. It's what you would get after following this chapter.

A good work environment helps us get the most out of our time. 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 a combination of Browserify, Grunt, NPM, and Bower. This was the wrong approach. It was complicated, it wasn't very extensible, and it was slow.

There were no sourcemaps, which meant your browser's error reporting was useless. There was no hot loading, which meant your code had to process a gigantic 80,000 datapoint file every time you made a change.

It was a mess. I'm sorry I told you to use it. The old system is included in the appendix if you're curious.

The new system is great. I promise. I use it all the time :)

Bundle with Webpack

Instead of using the old setup, I now think the best choice is to use a combination of Webpack and 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 however, 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.

Using Webpack allows us to solve two more annoyances — losing state when loading new code and accurately reporting errors. So we can add two more requirements to our work environment checklist (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 (hot loading)
- browser should report errors accurately in the right source files (sourcemaps)

Compile with 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. If you're not ready to learn ES6, don't worry; 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 to use 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'm also going to explain some of the moving parts that make it tick.

Your book package also contains a starter project. You can use that to get started right away. It's what you would get after following this chapter.

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 toolbelt. With Webpack's growing popularity and its ability to recognize NPM modules, NPM is fast becoming *the* way to manage client-side dependencies as well.

We're 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.
- You can use the stub project included with the book. It has everything ready to go.

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 uninitiated, 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 let us 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 the project our own, we have to change some information inside `package.json`: the name, version, and description.

```
// ./package.json
        {
-         "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:

```
// ./src/package.json
-"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 signed in our name.

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 installs 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 have to install two more libraries:

1. `d3` for drawing
2. `lodash` for some utility functions

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

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 it 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 makes your life easier in bigger projects.

Yes, there's a raging debate about stylesheets versus inline styling. I like stylesheets for now.

Webpack can handle compiling Less to CSS for us. We need 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 css-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

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

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.

Serve static files in development

Our visualization is going to use Ajax to load data. That means the server we use in development can't just route everything to `index.html` – it needs to serve other static files as well.

We have to add a line to `devServer.js`:

Enable static server on ./public

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

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

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

This tells express.js, which is the framework our simple server uses, to route any URL starting with `/public` to local files by matching paths.

Webpack nice-to-haves

Whenever I'm working with React, I like to add two nice-to-haves to `webpack.config.dev.js`. They're not super important, but they make my life a little easier.

First, I 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

```
// ./webpack.config.dev.js
module: {
   loaders: [
      {test: /\.js$/,
      {test: /\.js|\.jsx$/
       loaders: ['babel'],
       include: path.join(__dirname, 'src')
      },
```

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

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

Add resolve to webpack.config.dev.js

```
// ./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.

Optionally enable ES7

Examples in this book are written in ES6, also known as ECMAScript2015. If you're using the boilerplate approach or the stub project you got with the book, all ES6 features work in any browser. The Babel 6 compiler makes sure of that by transpiling ES6 into ES5.

The gap between ES5 (2009/2011) and ES6 (2015) has been long, but now the standard has started moving fast once again. In fact, ES7 is promised to be released some time in 2016.

Even though ES7 hasn't been standardized yet, you can already use some of its features if you enable `stage-0` support in Babel. Everything in `stage-0` is stable enough to use in production, but there is a small chance the feature/syntax will change when the new standard becomes official.

You don't need `stage-0` to follow the examples in this book, but I do use one or two syntax sugar features. Whenever we use something from ES7, I will mention the ES6 alternative.

To enable `stage-0`, you have to first install `babel-preset-stage-0`. Like this:

```
$ npm install --save-dev babel-preset-stage-0
```

Then enable it in `.babelrc`:

Add stage-0 preset to .babelrc

```
// ./.babelrc
{
  "presets": ["react", "es2015"],
  "presets": ["react", "es2015", "stage-0"],
  "env": {
```

That's it. You can use fancy ES7 features in your code and Babel will transpile them into normal ES5 that all browsers support.

Don't worry if you don't understand how `.babelrc` works. You can read more about the environment in the Environment in depth chapter.

Check that everything works

Your environment should be ready. 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 can jump straight to the meat.

The environment in depth

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 any 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

Webpack 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 development version and a production 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.

Both files look alike, so we're going to focus on the dev version.

It comes in four parts:

Webpack config structure

```
// ./webpack.config.dev.js
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 which plugins to use when building our code;
- and **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

```
// ./webpack.config.dev.js
module.exports = {
  devtool: 'eval',
  entry: [
    'webpack-hot-middleware/client',
    './src/index'
  ],
  output: {
```

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. It loads new versions of JavaScript files without reloading the page.

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. If we had 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

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

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

```
// ./webpack.config.dev.js
plugins: [
  new webpack.HotModuleReplacementPlugin(),
  new webpack.NoErrorsPlugin()
],
module: {
```

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 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

```
// ./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 loader 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 one of the most popular frameworks for building websites in node.js.

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.

[7] http://webpack.github.io/docs/

[8] http://survivejs.com

Lines that tell Express to use Webpack

```
// ./devServer.js
app.use(require('webpack-dev-middleware')(compiler, {
  noInfo: true,
  publicPath: config.output.publicPath
}));

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.

Lines that tell Express how to route requests

```
// ./devServer.js
app.get('*', function(req, res) {
  res.sendFile(path.join(__dirname, 'index.html'));
});
```

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

```
// ./devServer.js
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.

[9] http://azat.co/

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 source maps 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 a few presets. A preset is a single package that enables a bunch of plugins and code transforms. We use them to make our lives easier, but you can drop into a more specific config if you want to.

The best way to configure Babel is through the `.babelrc` file, which looks like this:

.babelrc config

```
// ./.babelrc
{
  "presets": ["react", "es2015", "stage-0"],
  "env": {
    "development": {
      "presets": ["react-hmre"]
    }
  }
}
```

I imagine this file is something most people copy-paste from the internet, but here's what's happening in our case:

- `react` enables all React and JSX plugins;
- `es2015` enables transpiling ES6 into ES5, including all polyfills for semantic features;
- `stage-0` enables the more experimental ES7 features.

Those are the default presets. For development, we also enable `react-hmre`, which gives us hot loading.

That's it. If you need more granular config, or you want to know what all those presets enable and use, I suggest Googling for them. Be warned, though; the `es2015` preset alone uses 20 different plugins.

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

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. Do that and 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

```
// ./.eslintrc
{
  "ecmaFeatures": {
    "jsx": true,
    "modules": true
  },
  "env": {
    "browser": true,
    "node": true
  },
  "parser": "babel-eslint",
  "rules": {
    "quotes": [2, "single"],
    "strict": [2, "never"],
    "babel/generator-star-spacing": 1,
    "babel/new-cap": 1,
    "babel/object-shorthand": 1,
    "babel/arrow-parens": 1,
    "babel/no-await-in-loop": 1,
    "react/jsx-uses-react": 2,
    "react/jsx-uses-vars": 2,
    "react/react-in-jsx-scope": 2
  },
  "plugins": [
```

```
    "babel",
    "react"
  ]
}
```

I haven't really had a chance to play around with linting configs like these to be honest. Emacs defaults have been good to me for years. But these types of configs are a great idea. The biggest problem in a team is syncing everyone's linters, and if you can put a file like this in your Git project - **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 have a working environment in which to write your code, and you understand how it does what it does. On a high level at least. And you don't need the details until you want to get fancy anyway.

That's how environments 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 hot loads new code.

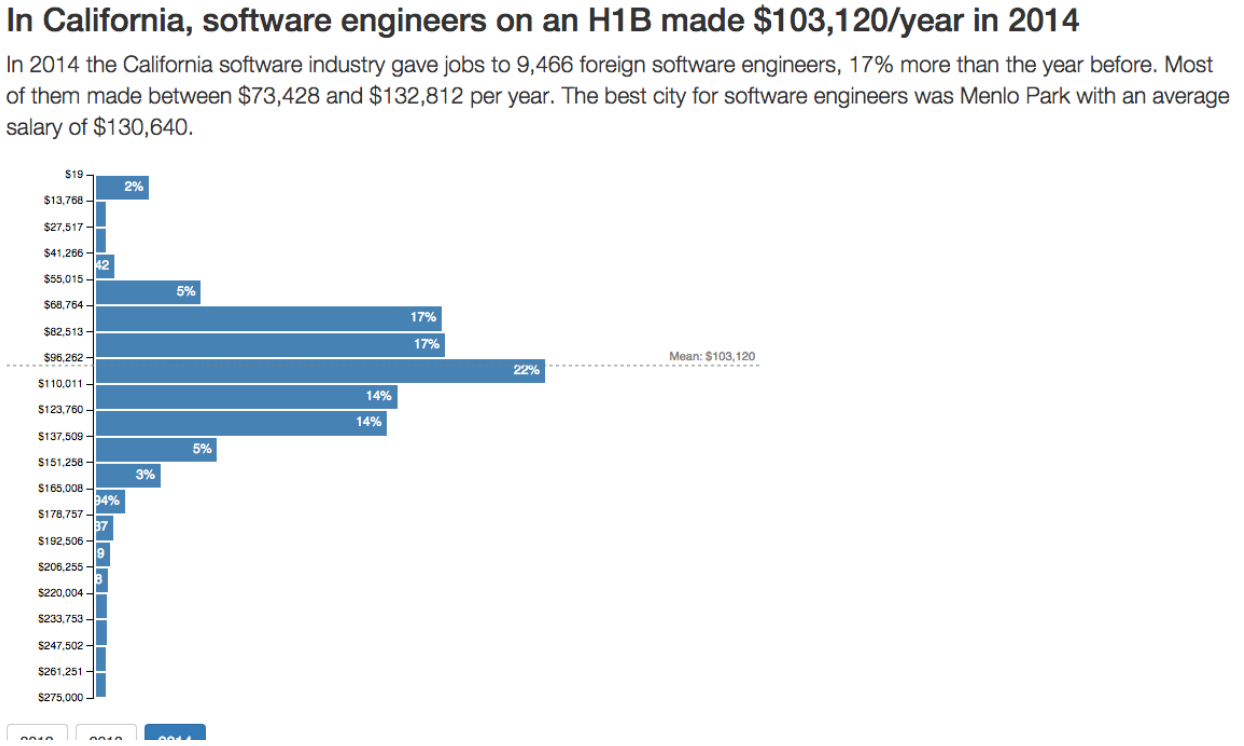
In the next section, we're going to start building a visualization.

[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 have the following dependencies:

- d3.js
- React
- Lodash

You should also have some way of running a static file server, and a way to compile JSX into pure JavaScript. I like having a small node.js server that supports hot loading via Webpack, and I like to compile with Babel.

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

We're going to put all of 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 our dataset from the stub project you got with the book. Put it in the `public/data/` directory of your project.

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() {
    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, it 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 it looks cleaner because it's pure JavaScript. But it's more typing and more function calls for the same result.

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

Despite appearances, 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.

Updating DOM with d3 takes a lot of work: you have to keep track of all the elements you're updating, the ones that are new, what to remove, everything. React does all of that for you. Its

purpose in life is knowing exactly which elements to update, and which to leave unchanged. Update it and forget it. Amazing.

We're going to follow this simple approach:

- React owns the DOM
- d3 calculates properties

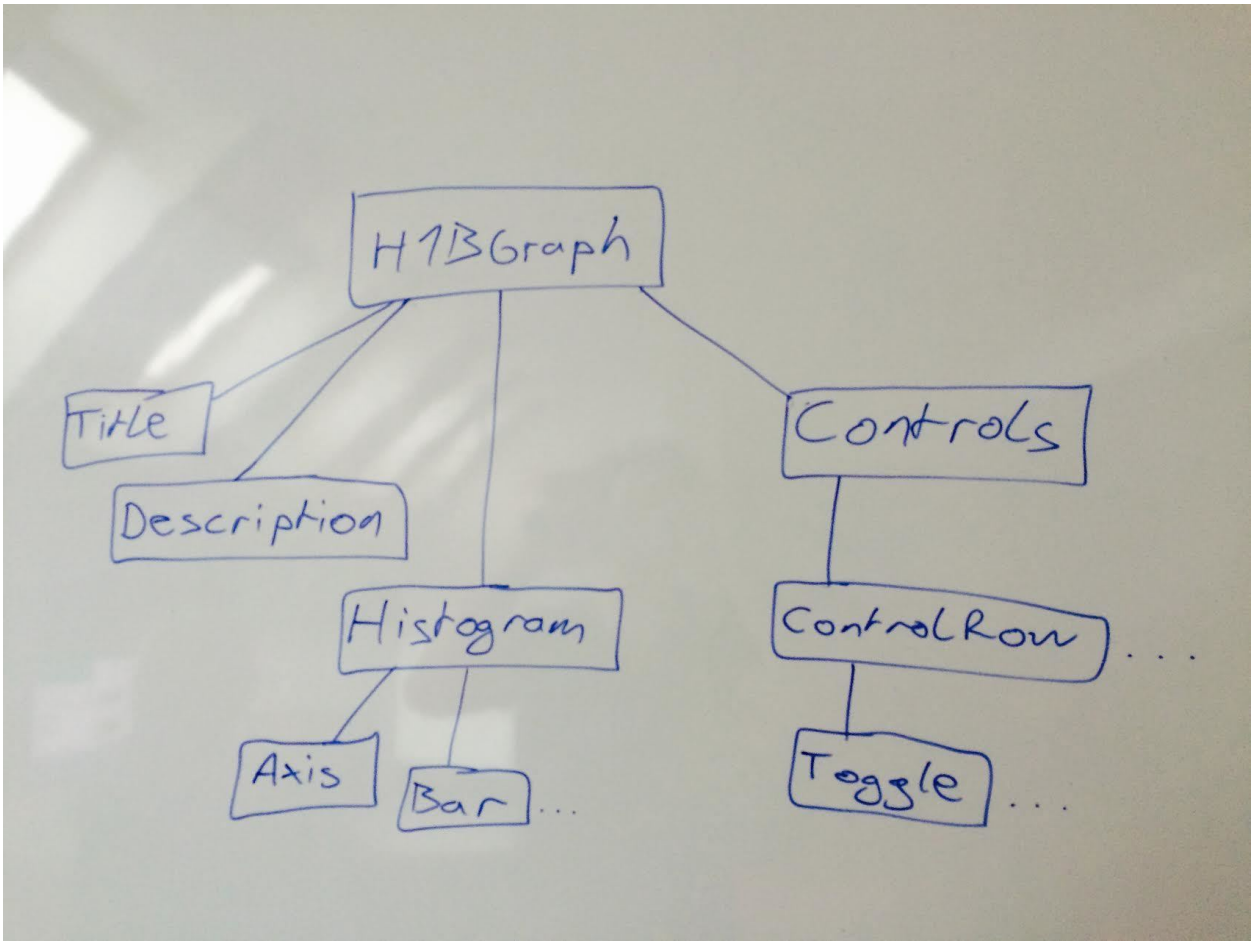
That lets us leverage both React and d3.js for what they're best at.

The Architecture

To make our lives easier, we're going to use a flow-down architecture inspired by Flux. All state is stored in one place - the main component - and data flows down via component properties. Changes travel back up the hierarchy via callbacks.

Our approach differs from Flux (and Redux) in that it doesn't need any extra code, which makes it easier to explain. The downside is that our version doesn't scale as well as Flux would.

If you don't know about Flux or Redux, don't worry; the explanations are self-contained. I only mention them here to make your Googling easier and to give you an idea of how this approach compares. I explain Redux in more detail in the Animating with React, Redux, and d3 chapter.

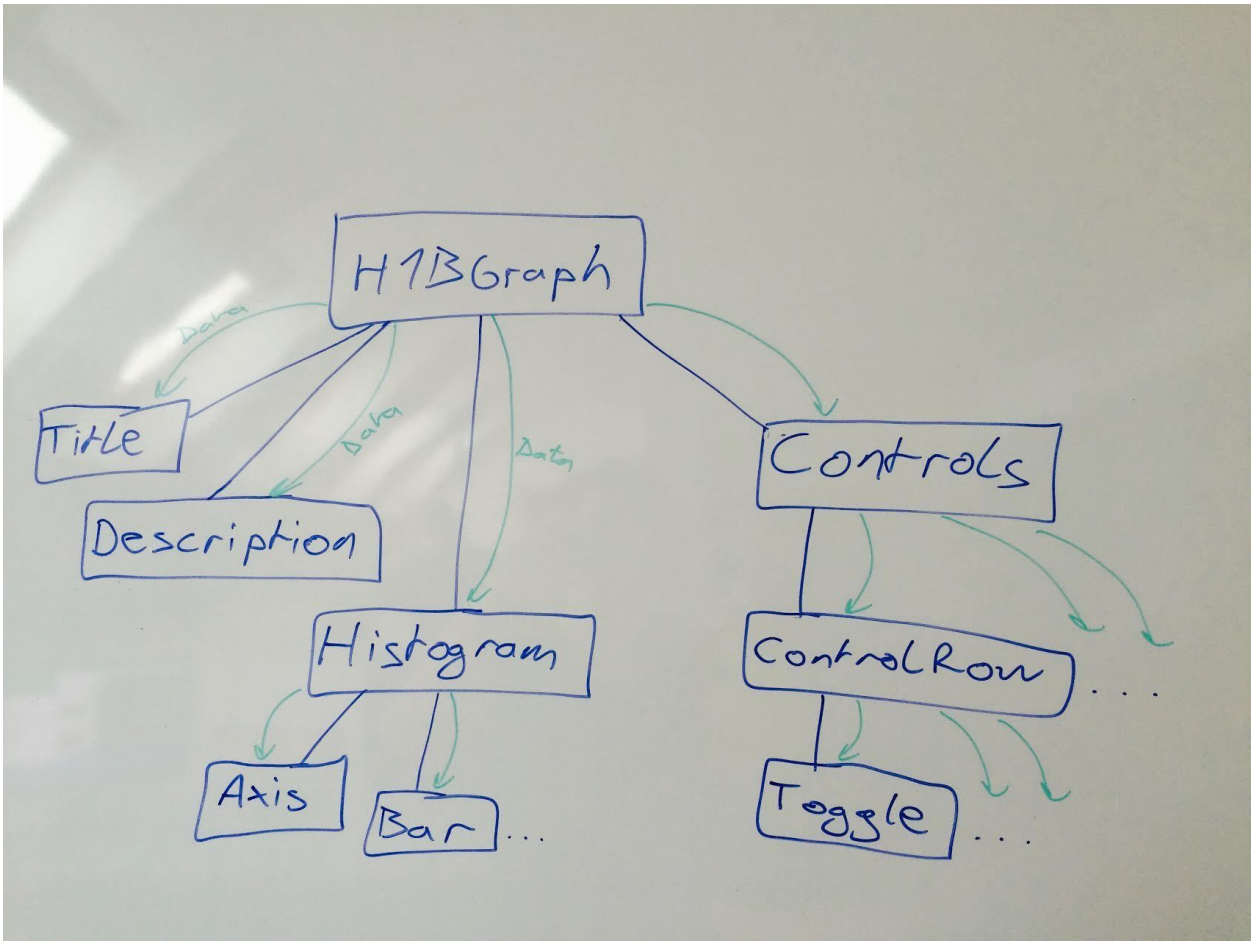


The basic architecture

The idea is this:

- The Main Component is the repository of truth
- Child components react to user events
- They announce changes using 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 better than worrying about parts of the UI going out of date with the rest of the app. I could talk your ear off with debugging horror stories, but I'm nice, so I won't.



Data flows down

Having your components rely just on their properties is like having functions that rely just on their arguments. 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 our interface with React, but we still need 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

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

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

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

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

These 20 lines do everything we need. The `<head>` sets some Bootstrap-recommended meta properties 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 to add 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.

Structuring your React app

As you can guess from the architecture chapter, we're going to structure our app into components. Deciding what to put into one component and what into another is one of the hardest problems in engineering.

Entire books have been written on the topic, so here's a rule of thumb that I like to use: If you have to use the word "and" to describe what your component does, then it should become two components.

Once you have those two components, you can either make them child components of a bigger component, or you can make them separate. The choice depends on their re-usability.

For instance, to build our H1B histogram visualization, we are going to build two top-level components:

- H1BGraph, which handles the entire UI
- Histogram, which is a passive component for rendering labeled histograms

Histogram is going to be a child of H1BGraph in the final hierarchy, but we might use it somewhere else. That makes it a good candidate for a stand-alone component.

Other components such as the mean value dotted line, filter controls, and title/description meta components are specific to this use-case. We'll build them as child components of H1BGraph. Building them as separate components makes them easier to reason about, reusable locally within H1BGraph, and easier to write tests for.

We're not going to write tests here, though. That's a topic for another book. Right now, we're going to focus on React and d3.js.

Each component will have its own folder inside `src/components/`, or its parent component, and at least an `index.jsx` file. Some will have style definitions, child components, and other JavaScript files as well.

In theory, each component should be accessible with `require('./MyComponent')`, and rendered with `<MyComponent {...params} />`. If a parent component has to know details about the implementation of a child component, something is wrong.

You can read more about these ideas by Googling "leaky abstractions"[13], "single responsibility principle"[14], "separation of concerns"[15], and "structured programming"[16]. Books from the late 90's and early 2000's (when object oriented programming was The Future (tm)) are the best source of [curated] info in my experience.

Bootstrap your app into place

Let's start by bootstrapping our app into place. We're going to make a simple `src/index.jsx` file – we set that to be our main entry file in the environment section – and an empty `H1BGraph` component that renders an SVG element.

We start by importing `React`, `ReactDOM`, and `H1BGraph`.

[13] https://en.wikipedia.org/wiki/Leaky_abstraction
[14] https://en.wikipedia.org/wiki/Single_responsibility_principle
[15] https://en.wikipedia.org/wiki/Separation_of_concerns
[16] https://en.wikipedia.org/wiki/Structured_programming

Import main dependencies

```
// ./src/index.jsx
import React from 'react';
import ReactDOM from 'react-dom';

import H1BGraph from './components/H1BGraph';
```

`React` is, well, React. We need it for just about everything. `ReactDOM` is React's DOM renderer, which is new in v0.14. Rendering got split out of base React so that it's easier to build React apps for different render targets like canvas, WebGL, native mobile, etc. `H1BGraph` is going to be our main component.

We're using ES6-style imports instead of `require()` because it gives us greater flexibility to import only the parts we want and it reads like a normal English sentence in most common-use cases. You'll see this play out in future examples.

The second argument must be a string even if you're importing a library.

The main entry point for the app is also a good place to define any convenience helper functions that should be available globally but aren't big enough or important enough to make a new library for. We'll add `capitalize` and `decapitalize` to the `String` class. There are libraries out there that do this, but there's no need to add an entire string manipulation library to our codebase if all we need are two functions.

Define two string helpers

```
// ./src/index.jsx
import React from 'react';
import ReactDOM from 'react-dom';

import H1BGraph from './components/H1BGraph';

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

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

Now we'll be able to capitalize and decapitalize any string in our codebase with a call like `"some thing".capitalize()`. We're going to use this in the `Title` and `Description` meta components. We define them here because they change the global String class, and it would be odd if that happened deep down in a child component... maybe we should've just avoided changing built-in classes.

All we need now is to render `H1BGraph` into the page.

Render H1BGraph onto page

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

This tells React to take control of the HTML element with class `h1bgraph` – a `<div>` inside the main container in our case – and render the `H1BGraph` component. We used `ReactDOM.render` because we're rendering for and in a browser. Other renderers exist, but they're not important right now.

You can think of this function call as a *"Give me a Thing That Does Stuff"*. It doesn't have to stand on its own. You could wrap it in a jQuery plugin, use it inside a Backbone or Angular view, or in whatever else you're already used to. That's how you can make a gradual transition towards React that doesn't force you to throw existing code away.

If you've kept `npm start` running in the background, it should now complain that `components/H1BGraph` doesn't exist. Let's fix that and get an empty `<svg>` element rendering.

Start with a basic component

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

Right now, the main entry file tries to import and render `H1BGraph`, but it doesn't exist yet. Let's start a new file in `src/components/H1BGraph/index.jsx`.

Basic imports

```
// ./src/components/H1BGraph/index.jsx
import React, { Component } from 'react';
import d3 from 'd3';
```

We import React and a base `Component` class. We have to import `React` even if we don't use it explicitly because Webpack throws a "React not defined" error otherwise. We'll use the `Component` class to build our own components off of.

We also import `d3` because we'll use it to load data later.

Empty H1BGraph component

```
// ./src/components/H1BGraph/index.jsx
import React, { Component } from 'react';
import d3 from 'd3';

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

Then we define a component that renders an empty `<svg>` tag inside a `<div>`. A component must have at least a `render()` function; otherwise, React throws an error when it tries to render. With v0.14 we also got stateless functional components, which are essentially just the `render()` function. They don't feature in this Histogram example, but you can see some in the animation chapter.

We're using ES6's concept of classes and class inheritance to extend React's basic `Component`, which we imported earlier. This is widely accepted as more readable than JavaScript's old/standard prototypical inheritance.

You can still use `React.createClass` if you want. It works just fine and is equivalent to `class X extends Component`.

Export H1BGraph

```
// ./src/components/H1BGraph/index.jsx
import React, { Component } from 'react';
import d3 from 'd3';

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

```
}

export default H1BGraph;
```

In the end, we export `H1BGraph` so other parts of the codebase can import it. We define it as the `default` export because this particular file only exports a single class.

If we had multiple things to export, we'd use named exports. They look like this: `export { Thing }`. You can also define the Thing anonymously when exporting, like this: `export function Thing () { // ... }`.

Using default exports makes our components easier to import - `import Thing from 'file'`. To import named exports, you have to specify them inside curly braces: `import { Thing1, Thing2 } from 'file'`.

Hot reloading and continuous compilation should have done their magic by now, and you should see an empty `<svg>` element in your page. You have to keep `npm start` running and a browser window pointing to `localhost:3000` for that.

Congratz! You just made your first React component with ES6 classes. Or at least the first in this book. You're awesome!

Here's what happens next:

1. We'll make `H1BGraph` load our dataset
2. Build a Histogram component
3. Add dynamic meta text descriptions
4. Add some data filtering

Asynchronously loading data

Great, we've got a rendering component. Now we need to load some data so we can make a visualization. If you still don't have `public/data/h1bs.csv`, now is the time to get it. You can find it on Github here[17] or in the stub project included with the book.

We're going to use d3.js's built-in data-loading magic and hook it into React's component lifecycle for a seamless integration. You could implement calls to a REST API in the same way.

We start by adding three methods to `H1BGraph` in `src/components/H1BGraph/index.jsx`:

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

Base methods to load data

```
// ./src/components/H1BGraph/index.jsx
class H1BGraph extends Component {
    constructor() {
        super();

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

    componentWillMount() {
        this.loadRawData();
    }

    loadRawData() {
    }

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

With ES6 classes, we no longer use `getInitialState` to set the initial state of our component. That job goes to the constructor - a function that's called every time our class is instantiated into an object. We use `super()` first to call the parent's constructor, which is React's `Component` constructor in this case.

After calling `super()`, we can use `this.state` as a dictionary that holds our component's current state. It's best to avoid using state as much as possible and rely on props instead.

Next we have `componentWillMount`, which is a component lifecycle function that React calls just before mounting (also known as rendering) our component into the page. This is where you'd want to fit any last minute preparations before rendering.

It's also a great moment to start loading data, especially when you have 12MB of data like we do. We don't want to load that unless we know for sure that the component is being rendered.

We're going to put d3.js's data-loading magic in `loadRawData`. The reason we have a separate function and don't stick everything into `componentWillMount` is flexibility – we can call the function whenever we want without messing with React's built-in functions.

We use `d3.csv` to load our dataset because it's in CSV format. `d3.csv` understands CSV well enough to turn it into an array of dictionaries, using the first row as keys. If our data was JSON, we'd use `d3.json`, `d3.html` for HTML, etc. You can find the full list of data loaders in d3's documentation.

Load data with d3.csv

```
// ./src/components/H1BGraph/index.jsx
loadRawData() {
    d3.csv(this.props.url)
      .get((error, rows) => {
          if (error) {
              console.error(error);
              console.error(error.stack);
          }else{
              this.setState({rawData: rows});
          }
      });
}

render() {
```

This asynchronously loads a CSV file, parses it, and returns the result in the `rows` argument to the callback. The callback is an ES6 fat arrow, which is syntax sugar for `function () { // .. }.bind(this)` – a function bound to current scope. We're going to use these often.

Inside the callback, we check for errors, and if everything went well, we use `this.setState()` to save the loaded data into our component's state. Calling `setState()` triggers a re-render and should generally be avoided because relying on state can lead to inconsistencies in your UI. It's very appropriate to store 12MB of raw data as state, though.

We're now going to be able to access our data with `this.state.rawData`. Hooray!

Data cleanup

Datasets are often messy and annoying to work with. After a naive load like that, our data uses keys with spaces in them and everything is a string.

We can add some cleanup in one fell swoop:

Data cleanup

```
// ./src/components/H1BGraph/index.jsx
loadRawData() {
    let dateFormat = d3.time.format("%m/%d/%Y");

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

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 goes into 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": Date("2013-07-09T22:00:00.000Z"),
    "start_date": 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"
}
```

We cleaned up the keys, parsed dates into `Date()` objects using d3's built-in date formatters, and made sure numbers are numbers. If a row didn't have a `base_salary`, we filtered it out by returning `null`.

Add visual feedback that loading is happening

Loading and parsing 81,000 data points takes some time. Let's tell users what they're waiting for with some explainer text.

Loading indicator

```
    // ./src/components/H1BGraph/index.jsx
    render() {
        if (!this.state.rawData.length) {
            return (
                <h2>Loading data about 81,000 H1B visas in the software industry\
</h2>
            );
        }

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

The `render` method returns different elements based on whether or not we've got some data. We access the data as `this.state.rawData` and rely on the re-render triggered by `this.setState` to get rid of the loading notice.

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

Loading message

Marvelous.

Making your first dataviz component – a Histogram

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

A histogram component isn't unique to our project, so we're going to build it as a standalone component. Create a new directory `src/components/Histogram/` with an empty `index.jsx` file.

We use `index.jsx` as a convenience to make our component easier to import, so we won't put any logic in here. The file looks like this:

src/components/Histogram/index.jsx

```
// ./src/components/Histogram/index.jsx
import Histogram from './Histogram';

export default Histogram;
```

It imports and exports `Histogram`. Now the rest of our project can use `import Histogram from '../Histogram'` without understanding the internal file structure of our component. Users of your components should never have to know how you organized your component internally.

A blank Histogram component

We start with a blank `Histogram` component in `src/components/Histogram/Histogram.jsx`, like this:

Empty histogram

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

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

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

export default Histogram;
```

Just like in `H1BGraph`, we first required some external libraries - React and d3, created a class with a `render()` method, and added it to our exports. This lets other parts of the codebase use it by calling `import Histogram from './Histogram'`.

Right now, the `render()` method contains only an empty grouping, `<g>`, element. These are much like divs in HTML; they hold elements together to make them easier to reference. Unlike divs, however, they don't come with any concept of formatting or sizing.

The grouping element has a class `histogram` (which we'll use later for styling) and uses the `transform` attribute to move into position via the `translate` property.

SVG transforms are a powerful tool, but teaching you all about them goes beyond the scope of this book. We're only going to use the `translate()` property, which specifies a new `x, y` position for the given element.

We used ES6's string templates to create `transform`'s value. These are denoted with backticks and evaluate any JavaScript you put into `${}`. It's a great alternative to concatenating strings, but don't go crazy with these, lest you won't understand what your code is doing six months from now when you have to fix a bug.

In this case, we're using this SVG transformation to give our Histogram some room to breathe. Exactly how much room comes from `this.props.topMargin` – a property specified by whomever renders our `Histogram` component. Something like this: `<Histogram topMargin={100} />`.

c

alculations]d3.js for [dataviz] calculations

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: data binning, calculating positions and dimension of histogram bars, and a few other tidbits. We just have to give it some data and some configuration.

The issue is that d3.js achieves this magic through state. Almost any time you use a d3 function or layout, it has some internal state that it uses to reduce boilerplate.

For instance, all d3 layouts let you define a value accessor. You define it by calling the `.value()` method like this: `.value((d) => d.some.deep.value)`. After that, the layout always knows how to get the correct value from your data.

This is great when you're using d3 alone. It means there's less code to write.

It's less great when you're using React and want to avoid state. The best approach I've found to mitigate this issue is to rely on three methods: `constructor`, `componentWillReceiveProps`, and `update_d3`. Like this:

D3.js management functions

```
// ./src/components/Histogram/Histogram.jsx
class Histogram extends Component {
    constructor(props) {
        super();

        this.histogram = d3.layout.histogram();
        this.widthScale = d3.scale.linear();
        this.yScale = d3.scale.linear();
```

```
        this.update_d3(props);
    }

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

    update_d3(props) {
    }

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

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

In `constructor()`, we create the d3 objects and give them any defaults we know about. Then we call `this.update_d3`. In `componentWillReceiveProps()`, we call `this.update_d3` every time props change. `update_d3()` does the heavy lifting - it updates d3 objects using current component properties.

The other approach to this state problem would be to create new objects on every `render()`, but that feels wasteful. Maybe I'm just old fashioned.

We're going to use the same pattern for all our components that touch d3.js heavily.

This time, `update_d3()` does its heavy lifting like this:

update_d3 function body

```
    // ./src/components/Histogram/Histogram.jsx
    update_d3(props) {
        this.histogram
            .bins(props.bins)
            .value(props.value);

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

        this.widthScale
```

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

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

    render() {
```

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

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

Then, we call `this.histogram()` to ask d3's histogram layout to do its magic. If you print the result in `bars`, you'll see it's an array of arrays that have some properties.

Histogram printout

```
// console.log(bars) in update_d3()
```

[Array[26209], Array[48755], // ...] 0: Array[26209] [0 ... 9999] [10000 ... 19999] [20000 ... 26208] dx: 69999.273 length: 26209 x: 14.54 y: 26209

Each array holds: - data for its respective histogram bin - a `dx` property for bar thickness - a `length`, which is the number of items in this bin - an `x` coordinate - a `y` coordinate

We'll use these to draw our histogram bars.

Before we can do that, we have to update our scales. You can think of scales as 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. If you have a linear scale with a domain `[0, 1]` and a range `[1, 2]` and you call `scale(0.5)`, you will get `1.5`.

To define the correct domains, we first get the `counts` of elements in each bin. Then we use `min`/`max` helpers to find the smallest and largest values. In case of height, we had to add each bar's thickness to its position to find the max.

To define the ranges, we relied on chart dimension properties.

Notice how we use the `x` position to define the domain for our `yScale`? That's because our histogram is going to be horizontal.

We can do that so simply because scales don't care about anything. They map a domain to a range; how you use that is up to you.

Scales are my favorite. You'll see why when we start drawing.

React for the SVG output

Great, our scales are ready to help us draw. Let's add the drawing bit to `render()`.

Add histogram bars to SVG

```
// ./src/components/Histogram/Histogram.jsx
render() {
    let translate = `translate(0, ${this.props.topMargin})`,
        bars = this.histogram(this.props.data);

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

Even though we've already calculated our histogram data in `update_d3()`, we calculate it again in `render()`. This lets us avoid using state and triggering unnecessary re-renders. It seems wasteful at first, but think about it: which is quicker? Re-rendering twice on every prop change, or running a d3 function twice?

It's not immediately obvious which one uses less processing power, but we prefer avoiding state because it saves *our* brain cycles. Those are almost always more expensive than computer cycles.

We also add a new grouping element for the histogram bars. You'll see why we need a group inside a group later.

The coolest part is that we can `.map()` through our histogram data even though we're inside XML. That's the magic of JSX – JavaScript and XML living together as one.

We could have put the entire `makeBar` function in here, but that would make our code hard to read.

One last trick is using ES7's double-colon operator. It's syntax sugar for `this.makeBar.bind(this)`. Remember, even though we're using ES6 classes, we still have to bind callbacks to `this` scope manually.

You can use this syntax sugar if you enabled `stage-0` in the chapter about your dev environment.

Now we need the `makeBar` method. It looks like this:

The makeBar helper method

```
    // ./src/components/Histogram/Histogram.jsx
    makeBar(bar) {
        let percent = bar.y/this.props.data.length*100;

        let 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() {
```

This code uses `Histogram`'s properties and scales to calculate attributes for each bar, then passes the `props` object to a subcomponent called `HistogramBar` using a spread - `{...props}`. You can think of it as shorthand for writing `percent={percent} x={this.props.axisMargin} y={this.yScale(bar.x)}` ….

This is similar to ES6 spreads, but it's better because it supports objects as well as arrays and function arguments.

Now, let's add the `HistogramBar` subcomponent.

HistogramBar component

```
// ./src/components/Histogram/Histogram.jsx
class HistogramBar extends Component {
    render() {
        let 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>
        );
    }
}
```

There's nothing special here: we take some properties and return a grouping element with a rectangle and a text label. Components like these are great candidates for functional stateless components - components defined as functions which render something and don't think too much.

To do that, we would replace `class Foo extends Component { ... render() { return (<Stuff />) } }` with `const Foo () => <Stuff />`. I left it as an exercise for the reader :)

Anyway, in `HistogramBar`, we used an ES6 string template to build the `transform` property, and we added some ad-hoc vertical padding to the bar to make it look better. We also used some ad-hoc calculations to place the `text` element at the end of the bar. This makes our histogram easier to read because every bar has its percentage rendered on the crucial right edge.

Some bars are going to be too small to fit the entire label. Let's avoid rendering it in those cases.

Adjust label for small bars

```
// ./src/components/Histogram/Histogram.jsx
class HistogramBar extends Component {
    render() {
        let translate = `translate(${this.props.x}, ${this.props.y})`,
            label = this.props.percent.toFixed(0)+'%';

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

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

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

```
        return (
            <g transform={translate} className="bar">
                <rect width={this.props.width}
                      height={this.props.height-2}
                      transform="translate(0, 1)">
                </rect>
                <text textAnchor="end"
                      x={this.props.width-5}
                      y={this.props.height/2+3}>
                    {label}
                </text>
            </g>
        );
    }
}
```

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

Adding Histogram to the main component

Despite our magnificent `Histogram` component, the page is still blank. We have to go back to `H1BGraph/index.jsx` and tell `H1BGraph` to render the component we've just made.

First, we have to import our `Histogram` at the top of `H1Bgraph/index.jsx` like this:

Require Histogram

```
export default H1BGraph;

//
// Example 7
//
import React, { Component } from 'react';
```

This works so well because of the `Histogram/index.jsx` file we created, which lets us `import` a directory instead of worrying about the specific files inside.

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

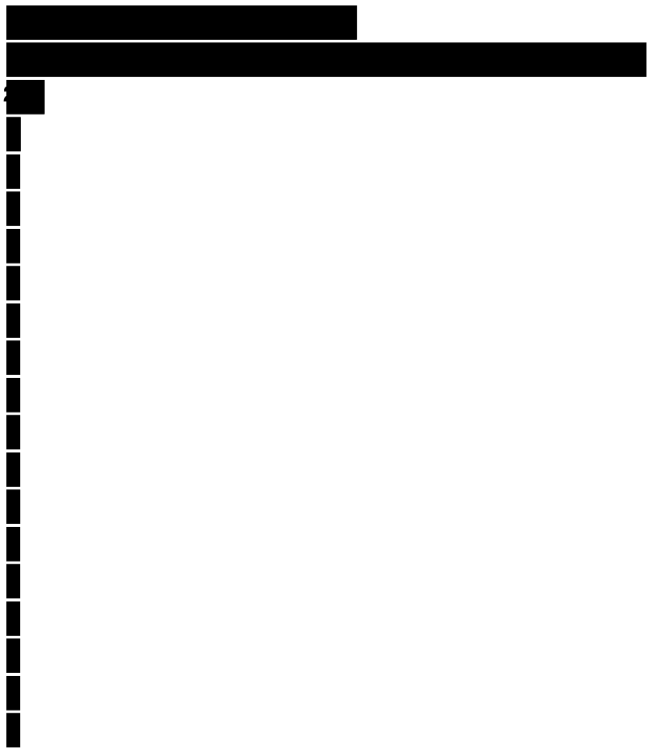
Render the histogram component

```
    // ./src/components/H1BGraph/index.jsx
    render() {
        if (!this.state.rawData.length) {
            return (
                <h2>Loading data about 81,000 H1B visas in the software industry\
</h2>
            );
        }

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

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

We put all our props in a `params` dictionary: dimensions, number of bins, value accessor. It makes our code easier to read.

Inside the `return` statement, we make sure our `<svg>` element has a width and a height, and we add the `<Histogram .. />` component. Once more, we use the spread trick to pass many `params` at once. We keep `data` as a separate attribute to make it more apparent that this component uses data.

If you kept `npm start` running, your browser should show something like this:



Unstyled Histogram

Wow. So much effort went into those labels, and you can't even see them. Let's add some styling to `src/components/Histogram/style.less` to make the Histogram prettier:

style.less

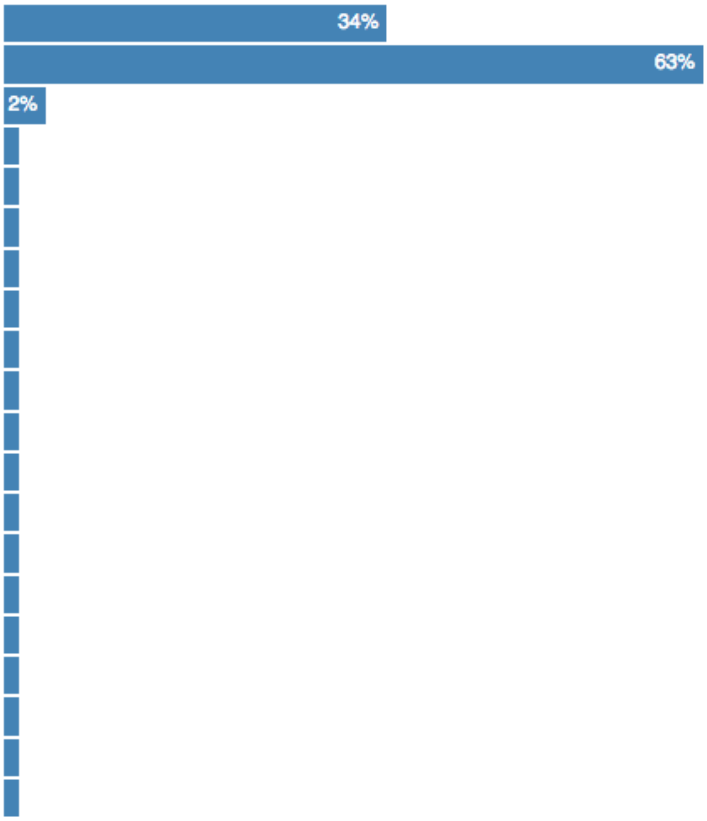
```
// ./src/components/Histogram/style.less
.histogram {
    .bar {
        rect {
            fill: steelblue;
            shape-rendering: crispEdges;
        }
        text {
            fill: #fff;
            font: 12px sans-serif;
        }
    }
}

//
// Example 2
```

Make sure to `require` the styles in `H1Bgraph/index.jsx`:

Require style.less

```
// ./src/components/H1BGraph/index.jsx
import Histogram from '../Histogram';

require('./style.less');
```

Now you should see a histogram like this:



Basic Histogram

Wrapping a pure-d3 element in React - an Axis

Axes are my favorite feat of d3.js magic right after scales. You call a function, set some parameters, and **BAM**, you've got an axis.

The axis automatically adapts to your data, draws the ticks, and labels them. Axes achieve this by using scales for all computation. The nice consequence of this is that you can turn a linear axis into a logarithmic axis by changing the scale it uses.

Despite how easy axes are to use, they're pretty complex to build from scratch. So we're not going to bother and use a dirty little trick instead – give d3.js control of the DOM. Just this once.

The same approach works for wrapping any d3.js visualization in a React component.

Let's start with a blank component for the axis in `src/components/Histogram/Axis.jsx`. It's a good candidate for a general component as well.

Base axis component

```
// ./src/components/Histogram/Axis.jsx
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
import d3 from 'd3';

class Axis extends Component {
    render() {
        let translate = `translate(${this.props.axisMargin-3}, 0)`;
        return (
            <g className="axis" transform={translate}>
            </g>
        );
    }
}

export default Axis;
```

As always, we start with the imports, then define a class - `Axis` – and export it. We add `ReactDOM` to the usual `d3` and `React` imports because we'll need it to give d3.js a reference to our DOM node. It can't take over without a DOM node.

Inside the `Axis` component, we define a `render` method which returns an empty grouping element and uses an SVG transform to emulate margins and padding. We have to do this so d3 has something to play with when it's rendering the axis.

Just like in the `Histogram` component, we integrate d3 into our component using the `constructor`, `componentWillReceiveProps`, and `update_d3` methods.

Axis default properties

```
// ./src/components/Histogram/Axis.jsx
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
import d3 from 'd3';

class Axis extends Component {
    constructor(props) {
        super();

        this.yScale = d3.scale.linear();
        this.axis = d3.svg.axis()
                    .scale(this.yScale)
                    .orient("left")
                    .tickFormat((d) => "$"+this.yScale.tickFormat()(d));

        this.update_d3(props);
    }

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

    update_d3(props) {
    }
```

In the `constructor`, we define a new linear axis for the `y` coordinate, and some default properties for the axis:

- it uses the linear scale
- renders labels on the left
- prepends the scale's tickFormat with a `$` sign

Yes, scales have tick formatters. They take care of rendering numbers nicely. It's awesome.

We let `update_d3` take care of the rest. The same delegation happens in the `componentWillReceiveProps` lifecycle method. The combination of these two methods calling `update_d3` ensures our d3 objects stay up to date.

The logic in `update_d3` looks like this:

Update axis state

```
// ./src/components/Histogram/Axis.jsx
update_d3(props) {
    this.yScale
        .domain([0,
                 d3.max(props.data.map((d) => d.x+d.dx))])
        .range([0, props.height-props.topMargin-props.bottomMargin]);

    this.axis
        .ticks(props.data.length)
        .tickValues(props.data
                        .map((d) => d.x)
                        .concat(props.data[props.data.length-1].x
                               +props.data[props.data.length-1].dx));
}
```

Just like the previous section, we have to tell `yScale` the extent of our data and drawing area. We don't have to tell the axis; it knows because the scale knows. Isn't that neat? I think it's neat.

We *do* have to get around some of the axis's smartness though. By default, an axis only renders a couple of ticks and labels to keep the visualization less cluttered. That would make our histogram look weird.

So we ask for the same number of ticks as there are bars, and we give a list of specific values. The axis tries to find "reasonable" label values otherwise.

Ok, we have our axis in memory. Now here's the dirty trick:

The dirty trick for embedding d3 renders

```
// ./src/components/Histogram/Axis.jsx
componentDidUpdate() { this.renderAxis(); }
componentDidMount() { this.renderAxis(); }

renderAxis() {
    let node = ReactDOM.findDOMNode(this);

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

render() {
```

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. This ensures `renderAxis` gets called every time our component has to re-render.

In `renderAxis`, we use `ReactDOM.findDOMNode()` to find this component's DOM node. We feed the node into `d3.select()`, which is how node selection is done in d3 land, then `.call()` the axis. The axis then adds a bunch of SVG elements inside our node.

As a result, we re-render the axis from scratch on every update. This is inefficient because it goes around React's fancy tree diffing algorithms, but it works well enough.

Before we add our new `Axis` component to the Histogram, we need some styling in `src/components/H1BGraph/style.less`:

Styling the axis

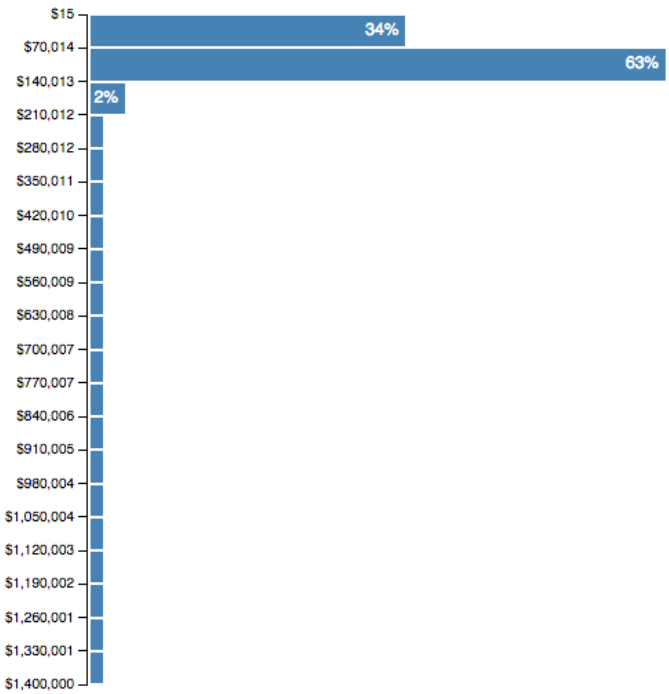
```
// ./src/components/Histogram/style.less
.histogram {
    .bar {
        rect {
            fill: steelblue;
            shape-rendering: crispEdges;
        }
        text {
            fill: #fff;
            font: 12px sans-serif;
        }
    }
    .axis {
        path, line {
            fill: none;
            stroke: #000;
            shape-rendering: crispEdges;
        }
        text {
            font: 10px sans-serif;
        }
    }
}
```

Great. Now we head back to the Histogram file (`src/components/Histogram/Histogram.jsx`) and add Axis to the imports, then add it to the render method.

Add Axis to Histogram imports

```
// ./src/components/Histogram/Histogram.jsx
import Axis from './Axis';

class HistogramBar extends Component {
```

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



Histogram with axis

If it doesn't, you should send me an email, and I'll try to help.

Next up: making the histogram interactive.

Interacting with the user

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

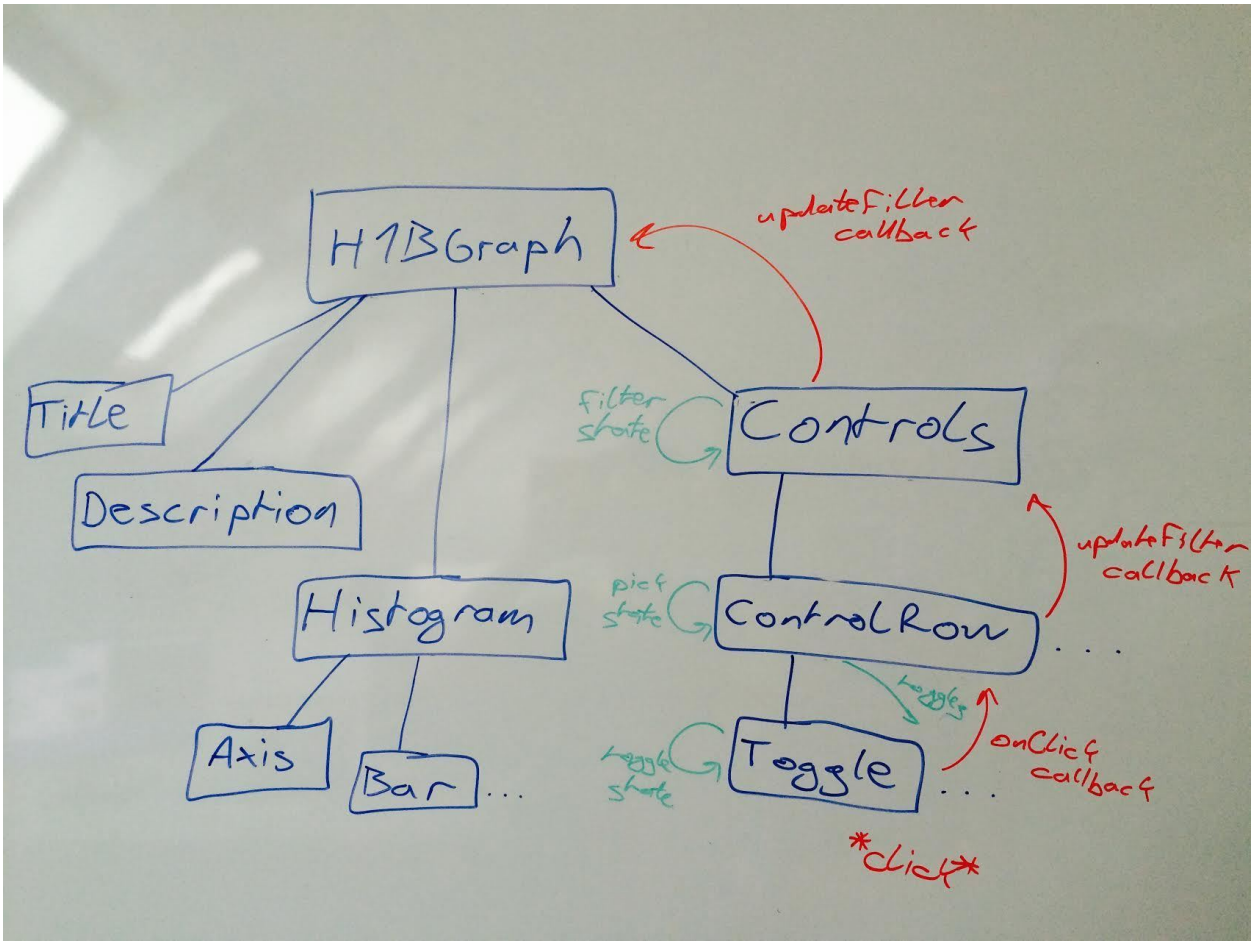
But our histogram looks weird. Most of our data falls into the first three bars, and a lone outlier stretches the data range too far.

We could remove the outlier, but let's be honest: statisticians should worry about statistical anomalies; we're here to draw pretty pictures.

It's better to let users explore the dataset themselves -> filters! Let users limit their view to just the parts they care about. This solves our problem of finding outliers and gives users more freedom. And they might even spend more time on our site!

Win-win-win.

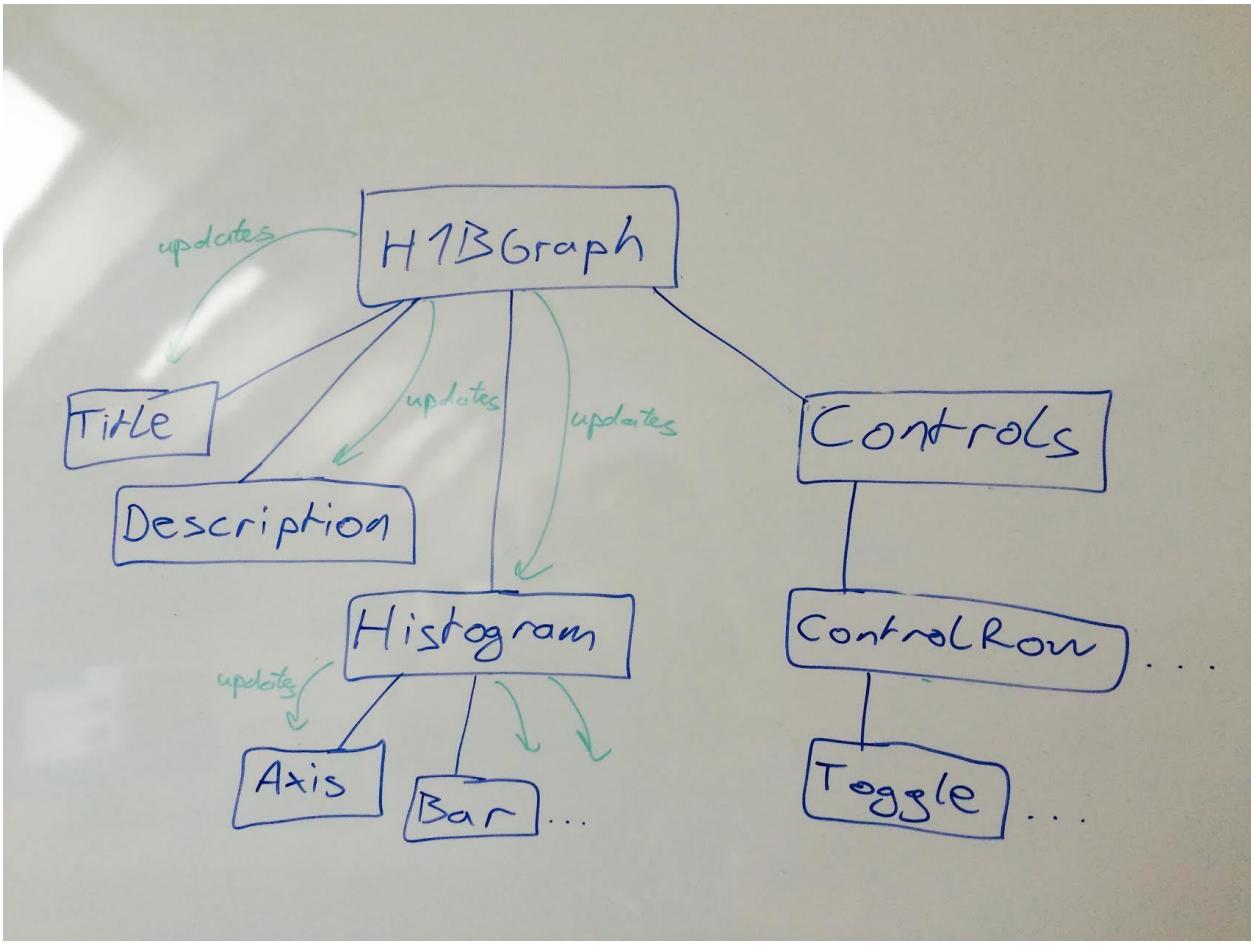
We're going to make controls that let users filter data by year. Then, in order to show you how reusable React components are, we're going to add filtering by US state as well. In the full example, I also added filtering by job title, but that would take too long to explain here. The code is easy, but the dataset is a mess.



Events flow up through callbacks

Remember, we talked about using a Flux-like approach in our architecture. That means:

- user events flow up the component hierarchy via callbacks
- the main component changes its data state
- React propagates data changes back down through component props



Updated data flows back down

The only difference between this bi-directional data-flow and a true Flux architecture is that in a true Flux architecture, the data store is separate from the main component. As a result, everyone has a way to affect changes directly on the data store, which means we don't have to mess about with callbacks.

That turns out to be easier to manage when you have a lot of interacting components. Because our example is so simple using Flux isn't really worth the extra code.

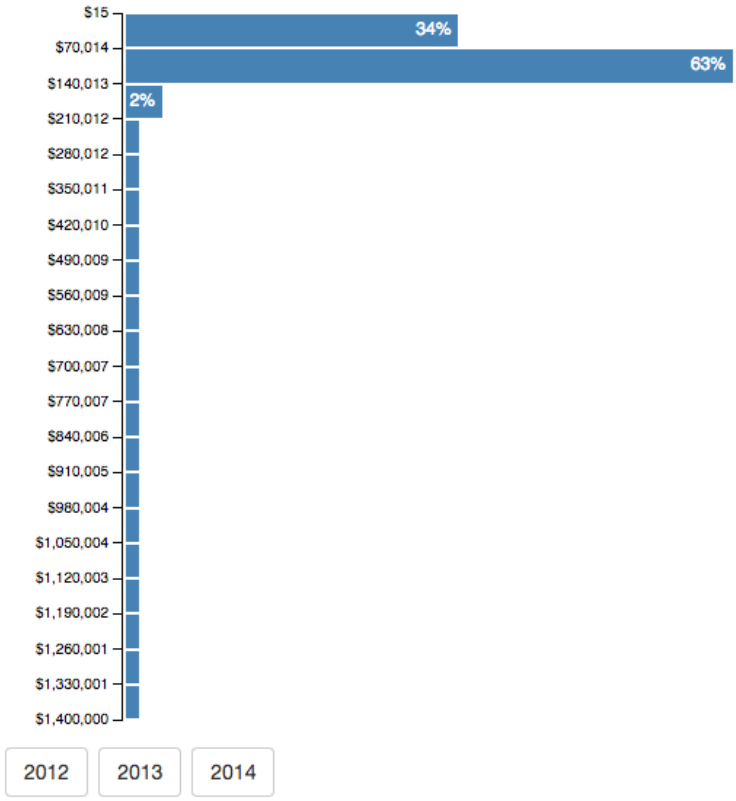
To get a better idea of Flux's pros and cons, you can read this side-by-side comparison of React+Flux (130 lines) and pure jQuery (10 lines) I wrote in October 2015[18]. It's based on a simple form example.

Adding user controls

Our controls are going to look like rows of toggleable buttons. Each row represents a filter – year and US state – and each button represents a value. When the button is "on", our histogram shows only the entries that have that value. Each row can have only one active value.

[18] http://swizec.com/blog/reactflux-can-do-in-just-137-lines-what-jquery-can-do-in-10/swizec/6740

The end result will be something like this:



Histogram with buttons

We'll build the controls out of three components:

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

These components live in a `Controls` subdirectory of `H1BGraph` – `src/components/H1BGraph/Controls`. We could build controls as a standalone component, and in a way we will, but for now it's easier to assume they're a specific solution to a specific problem.

Before diving in, let's render `Controls` in `H1BGraph.render`. It's going to break the build, but you'll see things added to the page in real-time as you code. I love it when that happens.

Add Controls to H1BGraph.render

```
    // ./src/components/H1BGraph/index.jsx
    render() {
        if (!this.state.rawData.length) {
            return (
                <h2>Loading data about 81,000 H1B visas in the software industry\
</h2>
            );
        }

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

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

Notice we added `Controls` outside the `<svg>` element. That's because they aren't a part of the visualization. Not of the graphical part at least.

We used props to give our `Controls` component some data (which it will use to generate the buttons) and a filter update callback. This function doesn't do anything yet. We'll add that in a bit when we talk about propagating events back up the chain. For now, it's here so that we have something to call without throwing an exception.

Don't forget to import controls at the top of the file.

Import Controls

```
// ./src/components/H1BGraph/index.jsx
import Histogram from '../Histogram';
import Controls from './Controls';

class H1BGraph extends Component {
```

Great. If you've kept `npm start` running in the background, your browser should start panicking right now. That's because `Controls/index.jsx` doesn't exist.

Let's make it: start with imports, a class, a render method, and exports. Like this:

Controls component stub

```
// ./components/H1BGraph/Controls/index.jsx
import React, { Component } from 'react';
import _ from 'lodash';

class Controls extends Component {
    render() {
        return (
            <div>

            </div>
        )
    }
}

export default Controls;
```

Your browser should stop panicking now. Instead, it shows a blank `<div>` under your histogram.

Now we add the first `ControlRow` and break the page all over again.

Year ControlRow in Controls

```
// ./components/H1BGraph/Controls/index.jsx
class Controls extends Component {
    render() {
        let getYears = (data) => {
            return _.keys(_.groupBy(data,
                                    (d) => d.submit_date.getFullYear()))
                    .map(Number);
        }

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

We define a function called `getYears` that can go through our dataset and find all of the year values. The `groupBy` creates a dictionary with years as keys, `keys` returns the dictionary's keys, and `map(Number)` makes sure they're all numbers so we won't have to worry about that later.

Then we add a `ControlRow` component to the return statement. As props, we gave it our dataset, the `getYears` function so it can decide which buttons to render, and a dummy update filter callback.

We'll define the filter when we look at propagating events back up the chain.

And, of course, don't forget to import `ControlRow`.

Import ControlRow

```
// ./components/H1BGraph/Controls/index.jsx
import React, { Component } from 'react';
import _ from 'lodash';

import ControlRow from './ControlRow';
```

Perfect. Your browser should start panicking again.

Let's calm your browser down and create `src/components/H1BGraph/Controls/ControlRow.jsx`.

Stubbed ControlRow component

```
// ./src/components/H1BGraph/Controls/ControlRow.jsx
import React, { Component } from 'react';
import _ from 'lodash';

import Toggle from './Toggle';

class ControlRow extends Component {
    render() {
        return (
            <div className="row">
                <div className="col-md-12">

                </div>
            </div>
        );
    }
}

export default ControlRow;
```

Start with imports, define a class, render some empty divs, export class. The usual.

Before your browser stops panicking, we also need the Toggle component in `src/components/H1BGraph/Controls/T`

Stubbed Toggle component

```
// ./src/components/H1BGraph/Controls/Toggle.jsx
import React, { Component } from 'react';

class Toggle extends Component {
    render() {
        return null;
    }
}

export default Toggle;
```

Your browser should stop panicking now, and there should be some divs within divs under the histogram.

To make controls show up, we first have to add toggles to `ControlRow`, then make sure the Toggle component returns a visible button.

A row of buttons

Adding buttons to `ControlRow` isn't too hard.

The `getToggleNames` function in our props gives us a list of button names, and the `Toggle` component renders a button. All we need is a loop that places the buttons.

Let's go back to `src/components/H1BGraph/Controls/ControlRow.jsx` and add it.

Loop through toggle names

```
    // ./src/components/H1BGraph/Controls/ControlRow.jsx
    render() {
        return (
            <div className="row">
                <div className="col-md-12">
                    {this.props
                         .getToggleNames(this.props.data)
                         .map((name) => this._addToggle(name))}
                </div>
            </div>
        );
    }
```

This code calls `getToggleNames` with the data in our props, and it calls `this._addToggle` on every value. Once again, notice the power of JSX – we can embed a JavaScript incantation right inside what looks like HTML code.

Now we need to define the `_addToggle` function, which returns a Toggle component. We do this in a function call so the code is easier to read.

Create a Toggle

```
// ./src/components/H1BGraph/Controls/ControlRow.jsx
class ControlRow extends Component {
    _addToggle(name) {
        let key = `toggle-${name}`,
            label = name;

        if (this.props.capitalize) {
            label = label.toUpperCase();
        }

        return (
            <Toggle label={label}
```

```
                    name={name}
                    key={key}
                    value={this.state.toggleValues[name]}
                    onClick={::this.makePick} />
        );
    }
```

We return a `<Toggle>` component with some properties:

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

These follow conventions for standard HTML form elements. It's the standard I got used to, and there's no need to re-invent the wheel.

We also capitalize the first letter of the label if that flag is set in our props. This is where those `String` class helpers we defined earlier help us out.

Now that we set an `onClick` callback, we have to stub the function lest we break the build. We also have to define some initial state because we used `this.state.toggleValues`. It's easier than checking if the variable exists every time.

makePick and default state

```
// ./src/components/H1BGraph/Controls/ControlRow.jsx
class ControlRow extends Component {
    makePick(picked, newState) {

    }

    componentWillMount() {
        let toggles = this.props.getToggleNames(this.props.data),
            toggleValues = _.zipObject(toggles,
                                       toggles.map(() => false));

        this.state = {toggleValues: toggleValues};
    }
```

We keep `makePick` empty for now because we'll define it later in the propagating events section. It's there just to keep things working.

Using `componentWillMount` to define the default state is an exception. You're supposed to define it in the class constructor, but we don't have access to props there. You can't populate an object property before the property exists. Remember, constructors are called while an object is still being created.

Yes, this tells us we're doing something wrong. Didn't we say all state lies in the top component? We did. But we can create a smoother user experience if we grit our teeth and make some compromises. User experience trumps perfect engineering.

If you're not used to lodash's functional approach, the code in `componentWillMount` might look strange. It loops through the values returned by `getToggleNames` and builds a dictionary – or object – with those names as keys, and `false` as the value.

We're later going to use `this.state.toggleValues` in combination with the `makePick` callback to ensure users can only select one value per row of toggles.

Toggle-able buttons

After all that, we still don't have anything to show to the user. `ControlRow` tries to render a bunch of buttons, but our `Toggle` component returns `null` every time.

Let's go back to `src/components/H1BGraph/Controls/Toggle.jsx` and make it work. Here's what we need to do:

- render a button
- define its class based on toggle-ness
- handle click events
- eagerly update toggle-ness on clicks

Handling toggle-ness is going to be the tricky part. We want the buttons to feel fast while also responding to upstream data changes.

First, let's render the button. We add a few lines to the `render` method:

Toggle renders a button

```
// ./src/components/H1BGraph/Controls/Toggle.jsx
import React, { Component } from 'react';

class Toggle extends Component {
    render() {
        return null;
        let className = "btn btn-default";
```

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

        return (
            <button className={className} onClick={::this.handleClick}>
                {this.props.label}
            </button>
        );
    }
}

export default Toggle;
```

We're using a basic HTML `<button>` element and giving it some React magic properties. `className` is an alias for the DOM `class` attribute. This naming is necessary because in JavaScript, `class` is a reserved symbol. `onClick` defines a click event handler similar to using `$('selector').click(do_thing)`.

React gives us the important event handlers in its `onSomething` magic properties. They might look like DOM Level 2 event handlers from the early 2000's, but their implementation is cleaner. Don't worry if, like me, you've spent the past 10 years of your life avoiding `onclick`. It's back. It's good. It's okay. I promise it has all the merits of using the *"find element, attach listener"* approach.

On line 8 of that example, we change the button's color based on `this.state.value`. (`btn-primary` is the Bootstrap class that makes buttons blue.)

Toggle color switch

```
        let className = "btn btn-default";

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

We only check `this.state.value` for now. Don't worry about props vs. state issues in the `render` method. That comes next.

We need three different functions to make this work smoothly:

- `constructor`, which sets default state
- `componentWillReceiveProps`, which updates internal state when props update
- `handleClick`, which is the click event callback

State and click handling in Toggle

```
// ./src/components/H1BGraph/Controls/Toggle.jsx
class Toggle extends Component {
    constructor() {
        super();

        this.state = {value: false};
    }

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

    handleClick(event) {
       let newValue = !this.state.value;
       this.setState({value: newValue});
    }

    render() {
```

You know the drill by now: `constructor` sets initial state dictionary to `{value: false}`, `componentWillReceiveProps` updates it when a new value comes down from above. This lets us have internal state *and* defer to global application state when needed.

The `handleClick` function toggles that same internal state. If it's `true`, it becomes `false` and vice-versa. Calling `setState` triggers a re-render and our button changes color.

Magic.

Using both state and props lets us change buttons the moment a user clicks them without waiting for the rest of the page to react. This makes user interactions snappier. Snappier interactions make users happy.

You should see a row of buttons under your histogram. Each can be toggled on and off, but nothing else happens yet.

Propagating events through the hierarchy

If we want the histogram to change when users toggle buttons, we have to propagate those click events through the hierarchy. The information being propagated is going to change each step of the way. This will reflect semantic changes in its meaning.

Toggles say something happened.

ControlRow says what happened.

Controls explains how to filter.

H1BGraph filters.

Our `Toggle` component is pretty much wired up already. It responds to user events in the `handleClick` method. Telling `ControlRow` that something happened is as easy as adding a function call.

Call event callback in handleClick

```
    // ./src/components/H1BGraph/Controls/Toggle.jsx
    handleClick(event) {
        let newValue = !this.state.value;
        this.setState({value: newValue});
        this.props.onClick(this.props.name, newValue);
    }
```

Notice how we can just call functions given in props? I love that. It's the simplest approach to inter-component interaction that I've ever seen.

I know we have a leaky abstraction right here. The parent component gives us a callback via props, and the first argument is also a prop. Why does the toggle have to tell its name to the callback, when the callback comes from the same place that the name does?

Because it makes our code simpler.

Let me show you. We implement the `makePick` function back in `src/components/H1BGraph/Controls/ControlRow.j` like this:

makePick function in ControlRow component

```
// ./src/components/H1BGraph/Controls/ControlRow.jsx
class ControlRow extends Component {
    makePick(picked, newState) {
        let toggleValues = this.state.toggleValues;

        toggleValues = _.mapValues(toggleValues,
                                   (value, key) => newState && key == picked);

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

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

Two things happen:

1. We make sure only one value can be turned on at a time
2. We call `updateDataFilter` with the new value

For both of these, we need the clicked toggle's name. When we use `_.mapValues` to reconstruct the `toggleValues` dictionary, the name tells us which field should be set to `true`. When we call `updateDataFilter`, it tells us the value we're filtering by.

Remember, we collected toggle names out of possible values for each filter. In the case of filtering by year, each year value became a toggle button.

If you click on a button now, you'll see it disables the others. When `this.state.toggleValues` changes, it triggers a re-render, which pushes new props down to `Toggle` components, which then re-render themselves with the new `className`.

Neat.

Now we can go back to `src/components/H1BGraph/Controls/index.jsx` and implement that `updateDataFilter` callback we stubbed earlier.

We have to change the callback definition in `render`.

Change ControlRow callback

```
    // ./src/components/H1BGraph/Controls/index.jsx
    render() {
        let getYears = (data) => {
            return _.keys(_.groupBy(data,
                                    (d) => d.submit_date.getFullYear()))
                    .map(Number);
        }

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

It's the same sort of code as always. If you don't like the `::` syntax sugar from ES7, you can use `this.updateDataFilter.bind(this)` instead.

Now here's the interesting part: we'll have to hold the current year filter function and value in memory, the component's state. We have to do this so that we can build complex filters on multiple properties.

Each time a user clicks a toggle button, we only know the current selection and property. We don't want to delete the filtering on, say, US state.

We need some default state in the constructor.

Default filter state

```
// ./src/components/H1BGraph/Controls/index.jsx
class Controls extends Component {
    constructor() {
        super();

        this.state = {
            yearFilter: () => true,
            year: '*',
        };
    }
```

`yearFilter` is a dummy function, and `year` is an asterisk. It could be anything, but a few months from now, you will still recognize an asterisk as an "anything goes" value.

With that done, we need the `updateYearFilter` function itself. It looks like this:

updateYearFilter function

```
    // ./src/components/H1BGraph/Controls/index.jsx
    updateYearFilter(year, reset) {
        let filter = (d) => d.submit_date.getFullYear() == year;

        if (reset || !year) {
            filter = () => true;
            year = '*';
        }

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

Our filter is a function that checks for equality between the year in a datum and the year argument to `updateYearFilter` function itself. This works because of JavaScript's wonderful scoping, which means that functions carry their entire scope around. No matter where we pass the filter function to, the local variables that existed where it was defined are always there.

This might lead to memory leaks. This may be why Chrome takes almost 10 gigabytes of memory on my laptop. Or maybe I keep too many tabs open. Who knows …

If the year is not defined, or reset is explicitly set to `true`, we reset the filter back to default values.

Almost there! We're handling the in-state representation of the year filter. Now we have to tell the parent component, `H1BGraph`.

We do that in the `componentDidUpdate` lifecycle method. As you can guess, React calls it on every re-render; triggered every time we use `this.setState`.

Propagate filter changes up the hierarchy

```
// ./src/components/H1BGraph/Controls/index.jsx
componentDidUpdate() {
    this.props.updateDataFilter(
        ((filters) => {
            return (d) =>  filters.yearFilter(d);
        })(this.state)
    );
}
```

For every update, we call `this.props.updateDataFilter`, which is the change callback defined by `H1BGraph` when it renders `Controls`. As the sole argument, we give it our filter wrapped in another function.

This looks (and is) far too complicated. It makes more sense when we add another filter. Hint: this is where we combine multiple filters into a single function.

Whatever you do, *DON'T* click a button yet. Your code will go into an infinite loop and crash your browser tab.

That's because when React decides whether to update your component or not, it only performs a shallow state and props comparison. Because we rely on function changes, we need a better comparison.

Prevent infinite loops

```
    // ./src/components/H1BGraph/Controls/index.jsx
    shouldComponentUpdate(nextProps, nextState) {
        return !_.isEqual(this.state, nextState);
    }

    render() {
```

With `shouldComponentUpdate`, we can define a custom comparison function. In our case, using Lodash's `isEqual` between current and future state is good enough.

Crisis averted! Click away.

Just one thing left to do before the graph starts changing for every click: use the new filters to filter data before rendering.

Go to `src/components/H1BGraph/index.jsx` and add this function:

Filter update callback

```
    // ./src/components/H1BGraph/index.jsx
    updateDataFilter(filter) {
        this.setState({dataFilter: filter});
    }

    // ./src/components/H1BGraph/index.jsx
```

Guess what it does? It stores the new filter in `this.state`.

Of course, this means we need a default filter in the constructor as well.

Default dataFilter

```
// ./src/components/H1BGraph/index.jsx
class H1BGraph extends Component {
    constructor() {
        super();

        this.state = {
            rawData: [],
            dataFilter: () => true
        };
    }
```

The default is a function that always says *"Yup, leave it in."*.

Now let's add all the heavy lifting to the `render` method. It looks like this:

Rendering filter data

```
    // ./src/components/H1BGraph/index.jsx
    render(){
        if (!this.state.rawData.length) {
            return (
                <h2>Loading data about 81,000 H1B visas in the software industry\
</h2>
            );
        }

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

        let filteredData = this.state.rawData
                               .filter(this.state.dataFilter);

        return (
            <div>
                <svg width={fullWidth} height={params.height}>
                    <Histogram {...params} data={this.state.rawData} />
                    <Histogram {...params} data={filteredData} />
                </svg>
                <Controls data={this.state.rawData} updateDataFilter={() => true\
} />
                <Controls data={this.state.rawData} updateDataFilter={::this.upd\
ateDataFilter} />
            </div>
        );
    }
```

A few things changed:

1. We pass our data through a filter

2. We feed filtered data into the `Histogram`
3. We give `Controls` the correct callback

Yes, this means we filter the data from scratch every time the page re-renders for whatever reason. It turns out this is pretty fast and works well enough, even with tens of thousands of data points.

There you go. A histogram that changes its shape when you click a button.

Amazing.

Now, let me show you just how reusable React components can be.

Component reusability

All right, we have a filter for years. It looks like a row of buttons and users can select a specific year to focus the histogram.

Let's look at how reusable we've made the `ControlRow` component by adding another filter - US states.

Here's a quick rundown of what we're going to do:

1. Add another `ControlRow` to `Controls`
2. Build a list of US states from the data
3. Add an `updateUSStateFilter` function to `Controls`
4. Add `USstateFilter` to component state
5. Include `USstateFilter` in `componentDidUpdate`

That's all we need: a few lines of copy-pasted code in `src/components/H1BGraph/Controls/index.jsx`. There are no changes necessary to either `ControlRow` or `Toggle`, nor are there any changes for the graph itself.

Adding a new `ControlRow` and building a list of US states might look familiar by now.

Add another ControlRow

```
    // ./src/components/H1BGraph/Controls/index.jsx
    render() {
        let getYears = (data) => {
            return _.keys(_.groupBy(data,
                                    (d) => d.submit_date.getFullYear()))
                    .map(Number);
        }

        let getUSStates = (data) => _.sortBy(_.keys(_.groupBy(data, (d) => d.sta\
```

```
te)));

        return (
            <div>
                <ControlRow data={this.props.data}
                            getToggleNames={getYears}
                            updateDataFilter={::this.updateYearFilter} />
                <ControlRow data={this.props.data}
                            getToggleNames={getUSStates}
                            updateDataFilter={::this.updateUSStateFilter}
                            capitalize="true" />
            </div>
        )
    }
```

We add a new `ControlRow` to render, give it a function that extracts US states from the data, and create a new update callback, which we'll define next.

Add a US state filter update

```
    // ./src/components/H1BGraph/Controls/index.jsx
    updateUSStateFilter(USstate, reset) {
        var filter = (d) => d.state == USstate;

        if (reset || !state) {
            filter = () => true;
            USstate = '*';
        }

        this.setState({USstateFilter: filter,
                       USstate: state});
    }
```

The new update filter callback is a copy-paste of `updateYearFilter` with a few keywords changed. We *could* have been clever about it and implemented dynamic magic that would let us get away with a single function, but it's unnecessary. It also makes the code both harder to explain and harder to understand a few months later.

Avoid cleverness when you code. It's too hard to debug.

Now let's add the new US state filter to the main callback in `componentDidUpdate`.

Include the new filter in the main callback

```
    // ./src/components/H1BGraph/Controls/index.jsx
    componentDidUpdate() {
        this.props.updateDataFilter(
            ((filters) => {
                return (d) =>  filters.yearFilter(d)
                    && filters.USstateFilter(d)
            })(this.state)
        );
    }
```

See, all that trouble of wrapping filters in extra functions pays off. We can compose many filters into a single expression.

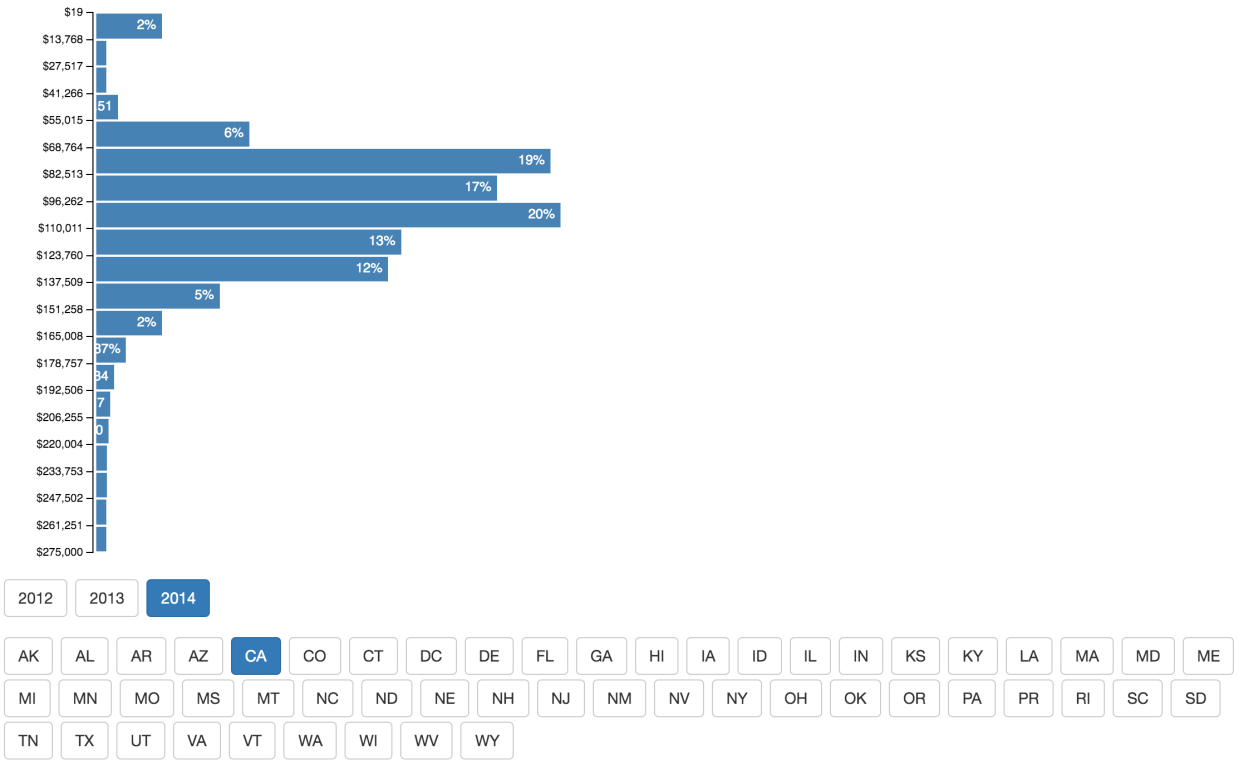
Finally, we need to define a default state for the US state filter. We do that in the class constructor.

Add default value for US state filter

```
// ./src/components/H1BGraph/Controls/index.jsx
class Controls extends Component {
    constructor() {
        super();

        this.state = {
            yearFilter: () => true,
            year: '*',
            USstateFilter: () => true,
            USstate: '*'
        };
    }
```

Same as the year filter, different key names.

If all went well, you should have a histogram with two rows of filters underneath. Users can choose a year *and* a US state to filter by.



Histogram with two filters

Making disparate components act together

If you came this far, you have a nice histogram of H1B salaries in the software industry. It's blue and there's an axis. Everything renders on the front-end and changes when the user picks a specific year or US state.

But we're missing something. 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.

That data flow-down architecture we built is the key to making disparate components act together. Because there is only one repository of truth, and because every component renders from props, we can make every component on the page change when data changes.

We can make the title and description change together with the histogram, which means they're always accurate and specific to what users see. They never have to think hard about the story behind the data; we just tell them.

Marvelous.

We're going to build two components - a Title and a Description. They're going to share some common methods, which we'll put in a base component.

Let's start with the base component. It goes in `src/components/H1BGraph/Meta/BaseComponent.jsx`.

Stubbed base component

```
// ./src/components/H1BGraph/Meta/BaseComponent.jsx
import React, { Component } from 'react';
import d3 from 'd3';
import _ from 'lodash';

export default class Meta extends Component {
    getYears(data) {

    }

    getUSStates(data) {

    }

    getFormatter(data) {

    }
}
```

The `BaseComponent` exports a `Meta` class which is all about common getters that both `Title` and `Description` are going to use. Extending from `Meta` will put functions from here into local scope so we can do stuff like `this.getYears()` and it calls the function from `Meta`.

Neat, huh? I think it is.

The function bodies themselves aren't too hard. They go through the dataset and return lists of values. `getYears` returns all year values, `getUSStates` returns all US states, and `getFormatter` returns a salary value formatter.

The entire Meta base component

```
// ./src/components/H1BGraph/Meta/BaseComponent.jsx
export default class Meta extends Component {
    getYears(data) {
        data || (data = this.props.data);

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

    getUSStates(data) {
        data || (data = this.props.data);

        return _.keys(_.groupBy(this.props.data,
                                (d) => d.state)
        );
    }

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

        return d3.scale.linear()
                 .domain(d3.extent(this.props.data,
                                   (d) => d.base_salary))
                 .tickFormat();
    }
}
```

This is nothing we haven't done before. Sprinkle some lodash functions for traversals and groupings, and use some d3 linear scale magic for the formatter.

Let's stub out the `Title` and `Description` components so we can add them to `H1BGraph` and let hot loading do its magic. We want to see what we're doing in real time.

`Title` goes in `src/components/H1BGraph/Meta/Title.jsx`.

Stubbed Title component

```
// ./src/components/H1BGraph/Meta/Title.jsx
import React, { Component } from 'react';
import d3 from 'd3';

import Meta from './BaseComponent';
import StatesMap from './StatesMap';

class Title extends Meta {
    render() {
        let title = (<h2>This is a title</h2>);


        return title;
    }
}

export default Title;
```

For now we return a dummy title so we get something rendering. Notice that we extended from `Meta` instead of `Component` like before.

`Description` goes in `src/components/H1BGraph/Meta/Description.jsx`.

Stubbed Description component

```
// ./src/components/H1BGraph/Meta/Description.jsx
import React, { Component } from 'react';
import d3 from 'd3';

import Meta from './BaseComponent';
import StatesMap from './StatesMap';


class Description extends Meta {
    render() {
        return (
            <p className="lead">This is a description</p>
        )
    }
}

export default Description;
```

Same as before – a dummy description and a few basic imports.

Both of those files imported a `StatesMap` file alongside `BaseComponent`. That's a mapping of US state shortcodes to full names.

You can get it from my github here[19]. It was a pain in the ass to make, so I don't suggest doing it yourself. Put it in `src/components/H1BGraph/Meta/StatesMap.jsx`. If you're using the stub project, it's already there.

One last bit we need is a `src/components/H1BGraph/Meta/index.jsx` file to make importing easier.

Meta/index.jsx

```
// ./src/components/H1BGraph/Meta/index.jsx
import * as T from './Title';
import * as D from './Description';

export class Title extends T.default {};
export class Description extends D.default {};
```

This looks weird. We have to do some roundabout hand waving so that we can both import and export classes with the same name. These are the so-called pass-through imports.

I don't know if that's the best way to do it, but it's what I came to after hours of banging my head against a wall. You can read the background behind these four lines of code on my blog[20].

Add Title and Description to the main view

Now that everything's stubbed out, adding the Title and Description components to the main view doesn't take much. We have to add three lines of code.

The import:

Import Title and Description

```
// ./src/components/H1BGraph/index.jsx
import Histogram from '../Histogram';
import Controls from './Controls';
import { Title, Description } from './Meta';
```

The render:

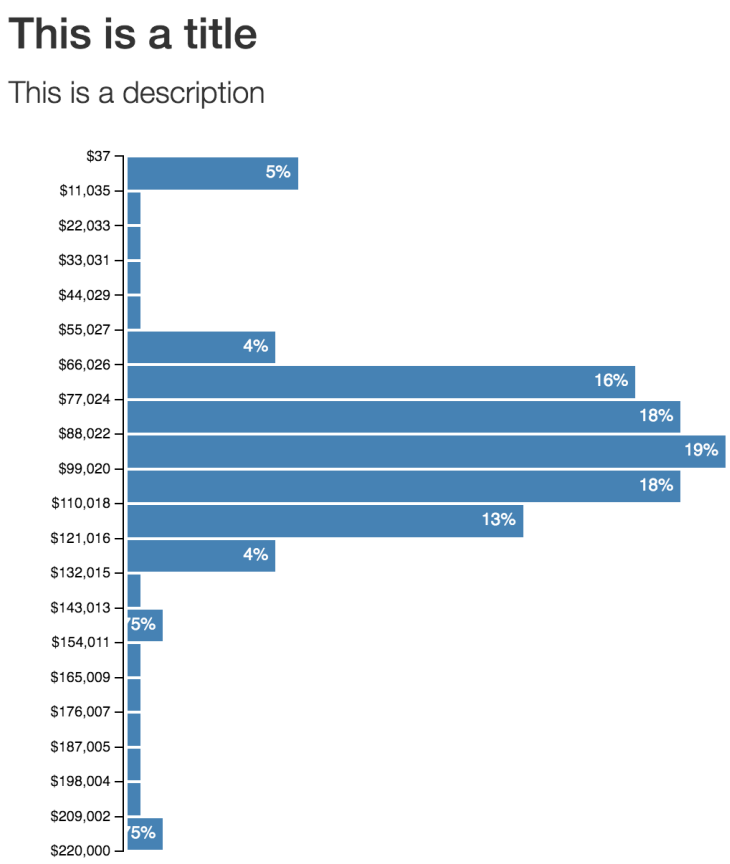
[19] https://github.com/Swizec/h1b-software-salaries/blob/e032f5b131af4659c5f9a0981b9d85054d2b5a17/src/components/H1BGraph/Meta/StatesMap.jsx

[20] http://swizec.com/blog/theres-a-bug-in-es6-modules/swizec/6753

Add Title and Description to render method

```
        // ./src/components/H1BGraph/index.jsx
        return (
            <div>
                <Title data={filteredData} />
                <Description data={filteredData} allData={this.state.rawData} />
                <svg width={fullWidth} height={params.height}>
                    <Histogram {...params} data={this.state.rawData} />
                    <Histogram {...params} data={filteredData} />
                </svg>
                <Controls data={this.state.rawData} updateDataFilter={() => true\
} />
                <Controls data={this.state.rawData} updateDataFilter={::this.upd\
ateDataFilter} />
            </div>
        );
    }
```

That's it. If you have `npm start` running in the background, you should see something like this now:



Histogram with dummy title and description

The title

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

We start with the year fragment:

getYearsFragment function

```
// ./src/components/H1BGraph/Meta/Title.jsx
class Title extends Meta {
    getYearsFragment() {
        let years = this.getYears(),
            title;

        if (years.length > 1) {
            title = "";
        }else{
```

```
            title = `in ${years[0]}`;
        }

        return title;
    }
```

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, we assume it's been filtered by year.

We make the same assumption for the US state fragment.

getUSStateFragment function

```
    // ./src/components/H1BGraph/Meta/Title.jsx
    getUSStateFragment() {
        var states = this.getUSStates(),
            title;


        if (states.length > 1) {
            title = "";
        }else{
            title = `in ${StatesMap[states[0].toUpperCase()]}`;
        }

        return title;
    }

    render() {
```

Great, that's the two dynamic pieces we need. Let's put them together in the `render` method.

Title render method

```
    // ./src/components/H1BGraph/Meta/Title.jsx
    render() {
        let title = (<h2>This is a title</h2>);
        let mean = d3.mean(this.props.data, (d) => d.base_salary),
            format = this.getFormatter();

        let
            yearsFragment = this.getYearsFragment(),
            USstateFragment = this.getUSStateFragment(),
```

```
            title;

        if (yearsFragment && USstateFragment) {
            title = (
                <h2>{USstateFragment}, H1B workers in the software industry made\
 ${format(mean)}/year {yearsFragment}</h2>
            );
        }else{
            title = (
                <h2>H1B workers in the software industry {yearsFragment.length ?\
 "made" : "make"} ${format(mean)}/year {USstateFragment} {yearsFragment}</h2>
            );
        }

        return title;
    }
```

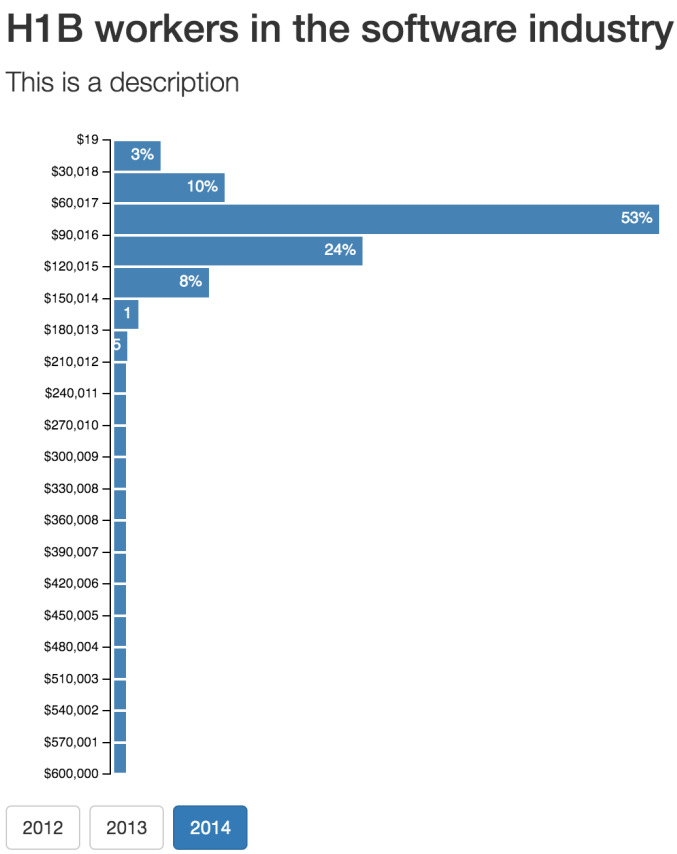
There's plenty going on here, but the gist of it is that big `if` statement:

- if both the years and US state fragments have content, our title is "In <State>, blahblah made $X/year in <Year>"
- if one or neither have content, our title is "Blahblah <made/make> $X/year <in State / in Year>"

The `$X/year` comes from combining the average salary in our dataset, calculated with `d3.mean`, and the number formatter. The rest is about plugging both fragments in the right parts of the string.

Note that in the second case, we still have to render both fragments because we don't know which one has content. They're syntactically interchangeable at this point. The sentence works the same if it ends "in California" or "in 2014".

Your histogram should look like this:



Histogram with title

Let's fix that description.

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.

We need two helper functions to get all data for a specific year or US state. This will help us do the last year look-backs.

Description's getAllData functions

```
// ./src/components/H1BGraph/Meta/Description.jsx
class Description extends Meta {
    getAllDataByYear(year, data) {
        data || (data = this.props.allData);

        return data.filter((d) => d.submit_date.getFullYear() == year);
    }

    getAllDataByUSState(USstate, data) {
        data || (data = this.props.allData);

        return data.filter((d) => d.state == USstate);
    }
```

Not much happens here. They're just filter functions.

The `getPreviousYearFragment` function is more interesting.

Description getPreviousYearFragment function

```
    // ./src/components/H1BGraph/Meta/Description.jsx
    getPreviousYearFragment() {
        let years = this.getYears().map(Number),
            fragment;

        if (years.length > 1) {
            fragment = "";
        }else if (years[0] == 2012) {
            fragment = "";
        }else{
            let year = years[0],
                lastYear = this.getAllDataByYear(year-1),
                USstates = this.getUSStates();


            if (USstates.length == 1) {
                lastYear = this.getAllDataByState(USstates[0], lastYear);
            }

            if (this.props.data.length/lastYear.length > 2) {
                let times_more = (this.props.data.length/lastYear.length).toFixe\
```

```
d();

                fragment = `, ${times_more} times more than the year before`;
            }else{
                let percent = ((1-lastYear.length/this.props.data.length)*100).t\
oFixed();

                fragment = `, ${Math.abs(percent)}% ${percent > 0 ? "more" : "le\
ss"} than the year before`;
            }
        }

        return fragment;
    }
```

Whoa, there's so much going on. First, we get all the years in filtered data, then:

- if there's more than one, we stop
- if it's the first year in our dataset, we stop
- otherwise:
  - we get all datapoints for last year
  - get US states in filtered data
  - filter `lastYear` by US state if user filtered by that as well
  - if the factor between this and last year is big, return "X times more" fragment
  - otherwise return "X% more/less" fragment

That wasn't too bad, was it? I'm sure better ways exist to generate human-readable text based on parameters, but there's no need to be fancy.

We also need the year and US state fragments, same as the title. Their functions look like this:

Year and US state fragment functions

```
    // ./src/components/H1BGraph/Meta/Description.jsx
    getYearFragment() {
        let years = this.getYears(),
            fragment;

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

```
        }

        return fragment;
    }

    getUSStateFragment() {
        let states = this.getUSStates(),
            fragment;

        if (states.length > 1) {
            fragment = "US";
        }else{
            fragment = StatesMap[states[0].toUpperCase()];
        }

        return fragment;
    }
```

You already know this code from the `Title` component. The only difference is in the strings produced. This is also why we can't throw these two functions into `BaseComponent`.

Now that we have all of the helper functions, it's time to put them together. We do that in the `render` method, like this:

Description render function

```
    // ./src/components/H1BGraph/Meta/Description.jsx
    render() {
        let formatter = this.getFormatter(),
            mean = d3.mean(this.props.data,
                           (d) => d.base_salary),
            deviation = d3.deviation(this.props.data,
                                     (d) => d.base_salary);

        let yearFragment = this.getYearFragment(),
            USStateFragment = this.getUSStateFragment(),
            previousYearFragment = this.getPreviousYearFragment(),
            N = formatter(this.props.data.length),
            min_salary = formatter(mean-deviation),
            max_salary = formatter(mean+deviation);

        return (
            <p className="lead">This is a description</p>
```

```
            <p className="lead">{yearFragment.length ? yearFragment : "Since 201\
2"} the {USStateFragment} software industry {yearFragment.length ? "gave" : "has\
 given"} jobs to {N} foreign nationals{previousYearFragment}. Most of them made \
between ${min_salary} and ${max_salary} per year.</p>
        );
    }
```

That is one beast of a return statement right there.

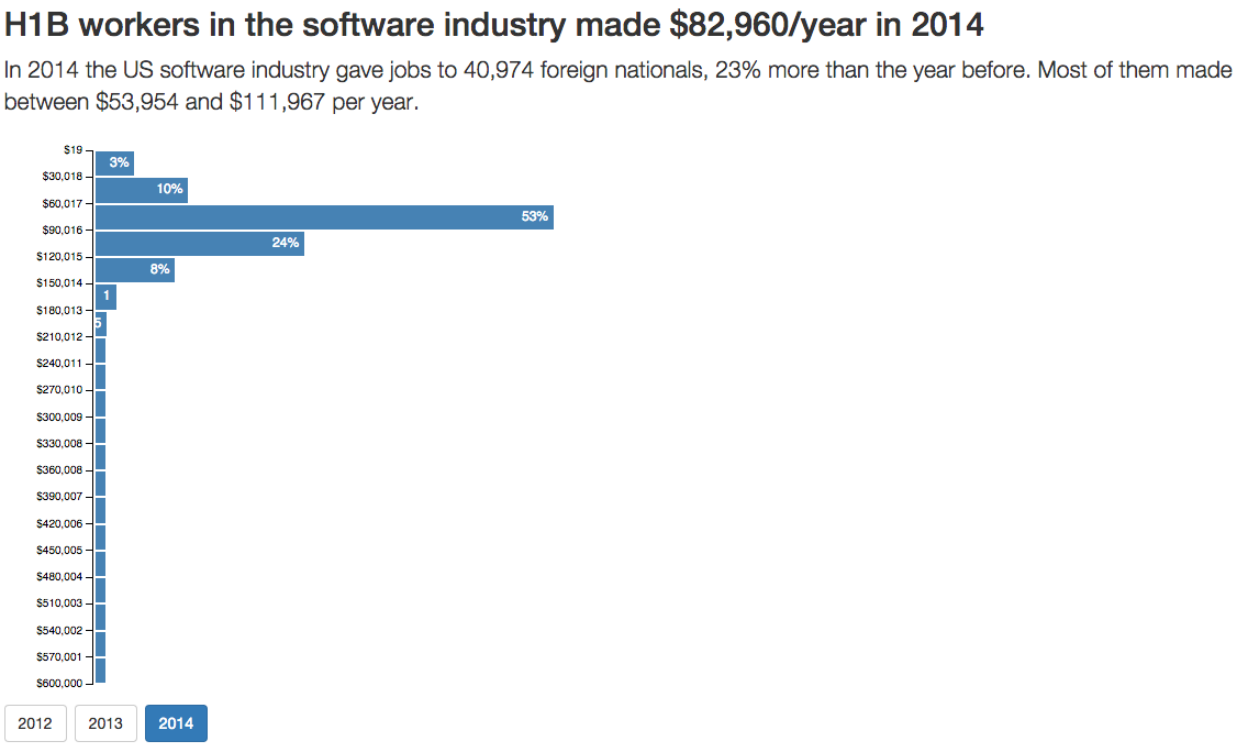
It's not too bad when you take a closer look. The description can start with either "Since 2012" or "In <Year>". Then it adds the selected US state, the correct tense of "to give", the number `N` of H1Bs in our dataset, and the previous year fragment. In the end, it adds info about the one standard deviation spread of salaries for selected year and US state.

We got those numbers using d3's default math functions. Assuming salaries follow a normal distribution, 68% of them fall within this 1-standard-deviation range.

This is the sort of assumption that would make a statistics professor cry. But I think it's good enough. Given enough datapoints, everything becomes normally distributed according to the central limit theorem[21] anyway.

You should have a histogram with a nice description now:

[21] https://en.wikipedia.org/wiki/Central_limit_theorem



Histogram with description

Woohoo, you've made a histogram of H1B salaries with a title and description and everything changes when the visualization changes. You're amazing!

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

In the next section, we're going to look at something fancier - advanced animations, using Redux, and a potential way to make simple games.

Animating with React, Redux, and d3

And now for something completely different. No more histogram, no more salaries and serious stuff. Just nerdy fun.

This chapter builds a particle generator. It makes tiny circles fly out of where you click. Hold down your mouse and move around. The particles keep flying out of your cursor.

On mobile and only have a finger? That works, too.

I'm a nerd, so this is what I consider fun. Your mileage may vary.

To see the particle generator in action, go here[22].

To see the entire code, go to the repository on GitHub[23]. Code samples in this chapter are less complete than before, but you will see enough to understand what's going on.

We're going to use Redux as our state handler, and approach animation and dataviz like it was a game. A giant state tree (stored in a Redux store) holds information about *everything*, the game loop makes changes on every 60th of a second, things magically move.

It's going to be great.

Here's how it works

The whole thing is built with React, Redux, and d3. No tricks for animation; just a bit of cleverness.

Here's the general approach:

We use **React to render everything**: the page, the SVG element, the particles inside. All of it is built with React components that take some props and return some DOM. This lets us tap into React's algorithms that decide which nodes to update and when to garbage collect old nodes.

Then we use some **d3 calculations and event detection**. D3 has great random generators, so we take advantage of that. D3's mouse and touch event handlers calculate coordinates relative to our SVG. We need those, and React can't do it. React's click handlers are based on DOM nodes, which don't correspond to `(x, y)` coordinates. D3 looks at real cursor position on screen.

All **particle coordinates are in a Redux store**. Each particle also has a movement vector. The store holds some useful flags and general parameters, too. This lets us treat animation as data transformations. I'll show you what I mean in a bit.

[22] http://swizec.github.io/react-particles-experiment/

[23] https://github.com/Swizec/react-particles-experiment

We use **actions to communicate user events** like creating particles, starting the animation, changing mouse position, and so on. On each requestAnimationFrame, we **dispatch an "advance animation" action**.

On each action, the **reducer calculates a new state** for the whole app. This includes **new particle positions** for each step of the animation.

When the store updates, **React flushes changes** via props and because **coordinates are state, the particles move**.

The result is smooth animation.

Keep reading to learn the details. The code is also on GitHub[24].

A version of this article will be featured as a chapter in my upcoming React+d3js ES6 book[25].

3 presentation components

We'll start with the presentation components because they're the least complicated. To render a collection of particles, we need:

- a stateless `Particle`
- a stateless `Particles`
- a proper `App`

None of them contain state, but `App` has to be a proper component so that we can use `componentDidMount`. We need it to attach d3 event listeners.

The `Particle` component is a circle. It looks like this:

Particle component

```
// src/components/Particles/Particle.jsx
import React, { PropTypes } from 'react';

const Particle = ({ x, y }) => (
    <circle cx={x} cy={y} r="1.8" />
);

Particle.propTypes = {
    x: PropTypes.number.isRequired,
    y: PropTypes.number.isRequired
};
```

[24] https://github.com/Swizec/react-particles-experiment

[25] http://swizec.com/reactd3js/

```
export default Particle;
```

It takes `x` and `y` coordinates and returns an SVG circle.

The `Particles` component isn't much smarter – it returns a list of circles wrapped in a grouping element, like this:

Particles list

```
// src/components/Particles/index.jsx
import React, { PropTypes } from 'react';

import Particle from './Particle';

const Particles = ({ particles }) => (
    <g>{particles.map(particle =>
        <Particle key={particle.id}
                  {...particle} />
        )}
    </g>
);

Particles.propTypes = {
    particles: PropTypes.arrayOf(PropTypes.shape({
        id: PropTypes.number.isRequired,
        x: PropTypes.number.isRequired,
        y: PropTypes.number.isRequired
    }).isRequired).isRequired
};

export default Particles;
```

Note that `key={particle.id}` part. React complains endlessly without it. I think it's used to tell similar components apart, which makes the fancy algorithms work.

Cool. Given an array of `{id, x, y}` objects, we can render our SVG circles. Now comes our first fun component: the `App`.

`App` takes care of rendering the scene and attaching d3 event listeners. The rendering part looks like this:

App component

```
// src/components/index.jsx

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

import Particles from './Particles';
import Footer from './Footer';
import Header from './Header';

class App extends Component {
    render() {
        return (
            <div onMouseDown={e => this.props.startTicker()} style={{overflow:\
 'hidden'}}>
                <Header />
                <svg width={this.props.svgWidth}
                     height={this.props.svgHeight}
                     ref="svg"
                     style={{background: 'rgba(124, 224, 249, .3)'}}>
                    <Particles particles={this.props.particles} />
                </svg>
                <Footer N={this.props.particles.length} />
            </div>
        );
    }
}

App.propTypes = {
    svgWidth: PropTypes.number.isRequired,
    svgHeight: PropTypes.number.isRequired,
    startTicker: PropTypes.func.isRequired,
    startParticles: PropTypes.func.isRequired,
    stopParticles: PropTypes.func.isRequired,
    updateMousePos: PropTypes.func.isRequired
};

export default App;
```

There's more going on, but the gist is that we return a `<div>` with a `Header`, a `Footer`, and an `<svg>`.

Inside `<svg>`, we use `Particles` to render many circles. Don't worry about the Header and Footer; they're text.

Notice that the core of our rendering function only says *"Put all Particles here, please"*. There's nothing about what's moved, what's new, or what's no longer needed. We don't have to worry about that.

We get a list of coordinates and naively render circles. React takes care of the rest. If you ask me, that's the real magic here.

Oh, and we call `startTicker()` when a user clicks on our scene. No reason to have the clock running *before* any particles exist.

D3 event listeners

To let users generate particles, we have to wire up those functions we mentioned in `propTypes`. It looks like this:

Event listeners

```
// src/components/index.jsx

class App extends Component {
    componentDidMount() {
        let svg = d3.select(this.refs.svg);

        svg.on('mousedown', () => {
            this.updateMousePos();
            this.props.startParticles();
        });
        svg.on('touchstart', () => {
            this.updateTouchPos();
            this.props.startParticles();
        });
        svg.on('mousemove', () => {
            this.updateMousePos();
        });
        svg.on('touchmove', () => {
            this.updateTouchPos();
        });
        svg.on('mouseup', () => {
            this.props.stopParticles();
        });
        svg.on('touchend', () => {
            this.props.stopParticles();
```

```
        });
        svg.on('mouseleave', () => {
            this.props.stopParticles();
        });

    }

    updateMousePos() {
        let [x, y] = d3.mouse(this.refs.svg);
        this.props.updateMousePos(x, y);
    }

    updateTouchPos() {
        let [x, y] = d3.touches(this.refs.svg)[0];
        this.props.updateMousePos(x, y);
    }
```

There are some events to think about:

- `mousedown` and `touchstart` turn on particle generation
- `mousemove` and `touchmove` update the mouse location
- `mouseup`, `touchend`, and `mouseleave` turn off particle generation

Then you have `updateMousePos` and `updateTouchPos`, which use d3's magic to calculate new `(x, y)` coordinates relative to our SVG element. The particle generation step uses this data as each particle's initial position.

Yes, it's complicated.

Remember, React isn't smart enough to figure out mouse position relative to our drawing area. React knows that we clicked a DOM node. D3 does the magic[26] to find exact coordinates.

For touch events, we only consider the first touch's position. We *could* let users shoot particles out of multiple fingers at once, but there's enough going on as it is.

That's it for the rendering and the user events. 107 lines of code[27].

6 Actions

Redux actions are a fancy way of saying *"Yo, a thing happened!"*. They're functions you call to get structured metadata about what's up.

We have 6 actions. The most complicated one looks like this:

[26] https://github.com/mbostock/d3/blob/master/src/event/mouse.js

[27] https://github.com/Swizec/react-particles-experiment/blob/master/src/components/index.jsx

Actions

```
// src/actions/index.jsx
export const CREATE_PARTICLES = 'CREATE_PARTICLES';

export function createParticles(N, x, y) {
    return {
        type: CREATE_PARTICLES,
        x: x,
        y: y,
        N: N
    };
}
```

It tells the system to create `N` particles at `(x, y)` coordinates. You'll see how that works when we look at the Reducer, and you'll see how it triggers when we look at the Container.

Actions *must* have a `type`. Reducers use the type to decide what to do.

Our other actions[28] are `tickTime`, `tickerStarted`, `startParticles`, `stopParticles`, and `updateMousePos`. You can guess what they mean :)

1 Container component

Containers are React components much like the presentation bits. Unlike presentation components, containers talk to the redux data store.

I'm not sure this separation is strictly necessary, but it does make your code look nice. You have functionally clean presentation components that turn properties into elements, and dirty, dirty containers that talk to the outside world.

You can think of them as data store monads if it helps.

The gist of our `AppContainer` looks like this:

[28] https://github.com/Swizec/react-particles-experiment/blob/master/src/actions/index.js

Main container component

```
        // src/containers/AppContainer.jsx
        import { connect } from 'react-redux';
        import React, { Component } from 'react';

        import App from '../components';
        import { tickTime, tickerStarted, startParticles, stopParticles, updateMousePos\
, createParticles } from '../actions';

        class AppContainer extends Component {
            componentDidMount() {
                const { store } = this.context;
                this.unsubscribe = store.subscribe(() =>
                    this.forceUpdate()
                );
            }

            componentWillUnmount() {
                this.unsubscribe();
            }

           // ...

            render() {
                const { store } = this.context;
                const state = store.getState();

                return (
                    <App {...state}
                         startTicker={::this.startTicker}
                         startParticles={::this.startParticles}
                         stopParticles={::this.stopParticles}
                         updateMousePos={::this.updateMousePos}
                    />
                );
            }
        };

        AppContainer.contextTypes = {
            store: React.PropTypes.object
        };
```

```
export default AppContainer;
```

We import stuff, then we define `AppContainer` as a proper React `Component`. We need to use lifecycle methods, which aren't available in stateless components.

There are three important parts in this code:

1. We wire up the store in `componentDidMount` and `componentWillUnmount`. Subscribe to data changes on mount, unsubscribe on unmount.
2. When rendering, we assume the store is our context, use `getState()`, then render the component we're wrapping. In this case, we render the `App` component.
3. To get the store as our context, we *have to* define `contextTypes`. It won't work otherwise. This is deep React magic.

Contexts are nice because they let us implicitly pass properties. The context can be anything, but Redux prefers it be the store. That contains *everything* about application state – both UI and business data.

That's where the monad comparison comes from … my foray into Haskell may have broken me. I see monads everywhere.

In case you were wondering, that `{::this.startTicker}` syntax comes from ES2016. It's equivalent to `{this.startTicker.bind(this)}`. Enable `stage-0` in your Babel config to use it.

AppContainer talks to the store

Great, you know the basics. Now we need to define those callbacks so `App` can trigger actions. Most are boilerplate-y action wrappers. Like this:

Container-store communnication

```
// src/containers/AppContainer.jsx
    startParticles() {
        const { store } = this.context;
        store.dispatch(startParticles());
    }

    stopParticles() {
        const { store } = this.context;
        store.dispatch(stopParticles());
    }

    updateMousePos(x, y) {
```

```
            const { store } = this.context;
            store.dispatch(updateMousePos(x, y));
        }
```

That's boilerplate. The action function gives us that `{type: ..}` object, and we dispatch it on the store.

When that happens, Redux uses our reducer to create a new instance of the state tree. We'll talk more about that in the next section.

First, we have to look at the `startTicker` callback. It's where our magic begins.

startTicker

```
    startTicker() {
        const { store } = this.context;

        let ticker = () => {
            if (store.getState().tickerStarted) {
                this.maybeCreateParticles();
                store.dispatch(tickTime());

                window.requestAnimationFrame(ticker);
            }
        };

        if (!store.getState().tickerStarted) {
            console.log("Starting ticker");
            store.dispatch(tickerStarted());
            ticker();
        }
    }
```

Oof. Don't worry if you don't "get" this immediately. It took me a few hours to create.

This drives our animation loop. Some might call it the game loop.

It dispatches the `tickTime` action on every `requestAnimationFrame`. Every time the browser is ready to render, we get a chance to update our Redux data store. In theory, that's 60 times a second, but it depends on many factors. Look it up.

`startTicker` updates the store in two steps:

1. Check `tickerStarted` flag and only start the ticker if it hasn't been started yet. This way we don't try to run multiple animation frames per render frame. As a result, we can be naive about binding `startTicker` to `onMouseDown`.

2. Create a `ticker` function that generates particles, dispatches the `tickTime` action, and recursively calls itself on every `requestAnimationFrame`. We check the `tickerStarted` flag every time so we can potentially stop time.

Yes, that means we are asynchronously dispatching redux actions. It's okay; that sort of Just Working (tm) is one of the main benefits of immutable data.

The `maybeCreateParticles` function itself isn't too interesting. It gets `(x, y)` coordinates from `store.mousePos`, checks the `generateParticles` flag, and dispatches the `createParticles` action.

That's the container. 83 lines of code[29].

1 Reducer

Sweet. With the actions firing and the drawing done, it's time to look at the entire logic of our particle generator. We'll get it done in just 33 lines of code and some change.

Ok. Honestly? It's a bunch of change. But the 33 lines that make up `CREATE_PARTICLES` and `TIME_TICK` changes are the most interesting.

All of our logic goes in the reducer. Dan Abramov says[30] to think of reducers as the function you'd put in `.reduce()`. Given a state and a set of changes, how do I create the new state?

A simplistic example would look like this:

Reducer concept

```
let sum = [1,2,3,4].reduce((sum, i) => sum+i, 0);
```

For every number, take the previous sum and add the number.

Our particle generator is a more complicated version of that. It takes the current application state, incorporates an action, and returns the new application state. To keep things simple, we'll put everything into the same reducer and use a big `switch` statement to decide what to do based on `action.type`.

In bigger applications, we'd split this into multiple reducers, but the base principles stay the same.

Let's start with the basics:

[29] https://github.com/Swizec/react-particles-experiment/blob/master/src/containers/AppContainer.jsx
[30] http://redux.js.org/docs/basics/Reducers.html

Redux reducer basic state

```
// src/reducers/index.js
const Gravity = 0.5,
      randNormal = d3.random.normal(0.3, 2),
      randNormal2 = d3.random.normal(0.5, 1.8);

const initialState = {
    particles: [],
    particleIndex: 0,
    particlesPerTick: 5,
    svgWidth: 800,
    svgHeight: 600,
    tickerStarted: false,
    generateParticles: false,
    mousePos: [null, null]
};

function particlesApp(state = initialState, action) {
    switch (action.type) {
    default:
            return state;
    }
}

export default particlesApp;
```

This is our reducer.

We started with the gravity constant and two random generators. Then we defined the default state:

- an empty list of particles
- a particle index, which I'll explain in a bit
- the number of particles we want to generate on each tick
- default svg sizing
- and the two flags and `mousePos` for the generator

Our reducer doesn't change anything yet. It's important to always return at least an unchanged state.

Update the state to animate

For most actions, our reducer updates a single value. Like this:

Reducer big switch

```
        // src/reducers/index.js
            switch (action.type) {
                case 'TICKER_STARTED':
                    return Object.assign({}, state, {
                        tickerStarted: true
                    });
                case 'START_PARTICLES':
                    return Object.assign({}, state, {
                        generateParticles: true
                    });
                case 'STOP_PARTICLES':
                    return Object.assign({}, state, {
                        generateParticles: false
                    });
                case 'UPDATE_MOUSE_POS':
                    return Object.assign({}, state, {
                        mousePos: [action.x, action.y]
                    });
```

Even though we're only changing values of boolean flags and two-digit arrays, *we have to create a new state*. Always create a new state. Redux relies on application state being immutable.

That's why we use `Object.assign({}, ...` every time. It creates a new empty object, fills it with the current state, then overwrites specific values with new ones.

Either do that every time, or use a library for immutable data structures. That might work better, but I haven't tried it yet.

The two most important state updates – animation tick and create particles – look like this:

The particles magic

```
    case 'CREATE_PARTICLES':
        let newParticles = state.particles.slice(0),
            i;

        for (i = 0; i < action.N; i++) {
            let particle = {id: state.particleIndex+i,
                            x: action.x,
                            y: action.y};

            particle.vector = [particle.id%2 ? -randNormal() : randNormal(),
```

```
                                -randNormal2()*3.3];

            newParticles.unshift(particle);
        }

        return Object.assign({}, state, {
            particles: newParticles,
            particleIndex: state.particleIndex+i+1
        });
    case 'TIME_TICK':
        let {svgWidth, svgHeight} = state,
            movedParticles = state.particles
                                .filter((p) =>
                                    !(p.y > svgHeight || p.x < 0 || p.x > svgW\
idth))
                                .map((p) => {
                                    let [vx, vy] = p.vector;
                                    p.x += vx;
                                    p.y += vy;
                                    p.vector[1] += Gravity;
                                    return p;
                                });

        return Object.assign({}, state, {
            particles: movedParticles
        });
```

That looks like a bunch of code. Sort of. It's spread out.

The first part – `CREATE_PARTICLES` – copies all current articles into a new array and adds `action.N` new particles to the beginning. In my tests, this proved smoother than adding particles at the end. Each particle starts life at `(action.x, action.y)` and gets a random movement vector.

This is bad from Redux's perspective. Reducers are supposed to be pure functions, and randomness is inherently impure. I think it's fine in this case.

The other possible approach would shove this logic into the action. That has benefits, but makes it harder to see all logic in one place. Anyway …

The second part – `TIME_TICK` – doesn't copy the particles (although maybe it should). Arrays are passed by reference, so we're mutating existing data either way. This is bad, but it's not *too* bad. It definitely works faster :)

We filter out any particles that have left the visible area. For the rest, we add the movement vector to their position. Then we change the `y` part of the vector using our `Gravity` constant.

That's an easy way to implement acceleration.

That's it. Our reducer is done. Our particle generator works. Our thing animates smoothly. \o/

What we learned

Building this particle generator in React and Redux, I made three important discoveries:

1. **Redux is much faster than I thought**. You'd think creating a new copy of the state tree on each animation loop was crazy, but it works well. We're probably creating only a shallow copy for the most part, which explains the speed.
2. **Adding to JavaScript arrays is slow**. Once we hit about 300 particles, adding new ones becomes visibly slow. Stop adding particles and you get smooth animation. This indicates that something about creating particles is slow: either adding to the array, or creating React component instances, or creating SVG DOM nodes.
3. **SVG is also slow**. To test the above hypothesis, I made the generator create 3000 particles on first click. The animation speed is *terrible* at first and becomes okayish at around 1000 particles. This suggests that shallow copying big arrays and moving existing SVG nodes around is faster than adding new stuff. Here's a gif[31]

There you go. Animations done with React, Redux, and d3. New superpower? Maybe.

Here's the recap:

- React handles rendering
- d3 calculates stuff
- Redux handles state
- element coordinates are state
- change coordinates on every `requestAnimationFrame`
- magic

:)

[31] http://i.imgur.com/ug478Me.gif

Conclusion

You finished the book. That makes you my favorite person!

If you followed the examples, you've created your first two visualizations with React and d3.js - something useful, and something fun. You're amazing!

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

You understand why the declarative nature of React works 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. You know how to configure Webpack and Babel to create an awesome environment to work with. You understand when using Redux is better than doing yourself, and when it's not.

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, poke me on twitter. I'm @Swizec[32]. Or send me an email at swizec@swizec.com.

Got consulting time with your package? Schedule a call and I'll help you fix whatever's going wrong.

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

Appendix - Browserify-based environment

When I first wrote this book in the 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 I think Webpack is 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 toolbelt.

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

You can get it by installing node.js from nodejs.org[33]. 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, it has 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.

It's also great if you want to share the code with others.

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

[33] http://nodejs.org

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.

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 our code 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. It 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 it makes our site slower. React will also throw a warning and ask you never to use `JSXTransform` in production.

Finally, we concatenate all of 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[34], 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:

[34] http://gruntjs.com

```
$ 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[35] 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 people have suggested using Webpack[36] instead, but I haven't tried it yet. Apparently it's the best thing since bacon because it can even `require()` images.

Reactify[37] will take care of making our JSX files work with Browserify.

Less[38] will compile Less files to CSS, `watch` will automatically run our tasks when files change, and `jit-grunt` will load Grunt plugins automatically so we don't have to deal with that.

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:

[35] http://browserify.org
[36] http://webpack.github.io/
[37] https://github.com/andreypopp/reactify
[38] https://github.com/gruntjs/grunt-contrib-less

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 of our JavaScript plugins into some sort of `vendor/` directory or by having a `plugins.js` file and manually copy-pasting code in there.

That approach works fine up 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 it 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.

This is awesome, but it 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 also 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.